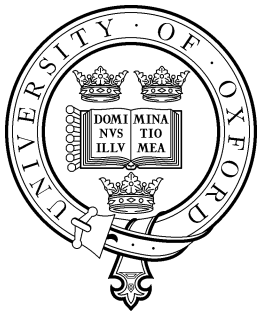
SPECIFICATION, IMPLEMENTATION AND VERIFICATION OF REFACTORINGS

Max Schäfer
Wolfson College

Trinity Term 2010

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy



Oxford University Computing Laboratory
Programming Tools Group

SPECIFICATION, IMPLEMENTATION AND VERIFICATION OF REFACTORINGS

Max Schäfer
Wolfson College

D.Phil. Thesis
Trinity Term 2010

Abstract

Refactoring is the process of reorganising or restructuring code by means of behaviour-preserving program transformations, themselves called refactorings. Most modern development environments come with built-in support for refactoring in the form of automated refactorings that the user can perform at the push of a button. Implementing refactorings is notoriously complex, however, and even state-of-the-art implementations have very low standards of correctness and can introduce subtle changes of behaviour into refactored programs.

In this thesis, we develop concepts and techniques that make it possible to give concise, modular specifications of refactorings. These specifications are precise enough to cover all details of the object language, and thus give rise to full featured, high-quality refactoring implementations. Their modularity, on the other hand, makes them amenable to formal proof, and hence opens the door to the rigorous verification of refactorings.

We discuss a disciplined approach to maintaining name bindings and avoiding name capture by treating the binding from a name to the declaration it refers to as a dependency that the refactoring has to preserve. This approach readily generalises to other types of dependencies for capturing control flow, data flow and synchronisation behaviour.

To implement complex refactorings, it is often helpful for the refactoring to internally work on a richer language with language extensions that make the transformation easier to express. We show how this allows the decomposition of refactorings into small microrefactorings that can be specified, implemented and verified in isolation.

We evaluate our approach by giving specifications and implementations of many commonly used refactorings that are concise, yet match the implementations in the popular Java development environment Eclipse in terms of features, and outperform them in terms of correctness.

We give detailed informal correctness proofs for some of our specifications, which are greatly aided by their modular structure. Finally, we discuss a rigorous formalisation of the central name binding framework used by most of our specifications in the theorem prover Coq, and show how its correctness can be established mechanically.

Acknowledgements

I could not have wished for a better supervisor than Oege. He made it possible for me to come to Oxford and stood by me throughout my doctorate with expert guidance, boundless enthusiasm and unflagging support.

Jeremy and Todd very kindly agreed to examine my thesis. Their thoughtful comments and helpful suggestions on content and form considerably improved this dissertation.

Tyng-Ruey and Shin-Cheng encouraged me to apply to Oxford in the first place. They and Horst Reichel wrote the recommendation letters that paved my way.

The Programming Tools Group provided a very pleasant and congenial work environment. Discussions with its members helped shape this work more than they can imagine. Damien, possibly the only person in the world who actually enjoys proofreading, was always at hand with helpful suggestions. Mathieu helped me get off to a good start and generously shared his broad knowledge about refactorings. Torbjörn showed me that real-world languages are nothing to be afraid of; without him (and JastAdd!) I would most likely have ended up refactoring the lambda calculus.

Frank invited me to T.J. Watson Research Center for a brief but fruitful internship. He and his group, in particular Manu, Emina and Julian, proved to me that if minds move fast enough, time dilates and even a few weeks are enough to accomplish significant research.

The anonymous reviewers who refereed the papers on which this thesis is based contributed many valuable suggestions and corrections, which had a significant impact on my work. Many people in the refactoring community shared their experience and advice with me, and JetBrains even allowed me to use their internal refactoring test suite before IntelliJ IDEA became open source.

Bas made the final few months of my doctorate an unexpectedly cheerful time. Anyone writing up a thesis should try to have someone like him around.

Yet none of this would ever have happened without my wife Tracy. She put up with another year of painful long-distance relationship even after it seemed that we had finally found a place to live together. She boldly moved to a foreign country, leaving everything behind to be with me. Through all these years, she has been a constant source of love, trust and support.

沒有妳的愛、妳的支持、妳的耐心，就沒有現在的我。

Thank you all.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Software refactoring is the process of improving the structure of existing code through behaviour-preserving transformations, themselves called refactorings. It can be used to clean up legacy code, uncovering its structure in the process and preparing for performance enhancements or bug fixes. It is also a mainstay of agile software development and other flavours of iterative software development that emphasise the concurrent development of, and cross-fertilisation between, requirements and solutions.

Refactoring is often performed in an informal manner, but over the years many particularly useful and frequently recurring individual refactoring operations have been identified. The first catalogue of refactorings for C++ was given by Bill Opdyke in his thesis [100]. More recently, two very influential textbooks on software refactoring by Martin Fowler [45] and Joshua Kerievsky [67] have categorised popular refactorings for Java, with informal descriptions and examples of their use.

Programmers who refactor a piece of software are often not the original authors and hence may have an incomplete understanding of its structure, so applying refactorings can be an error-prone task, because it may be difficult for the programmer to anticipate the global effect of a refactoring.

In agile development, this problem is mitigated by a strict testing regimen based on a comprehensive suite of unit tests that can be run after each refactoring step to ensure that the refactored program's behaviour has not changed.

Of course, such a test suite is not always available, so it is desirable to have a way of ensuring that a given refactoring does not change program behaviour. To this end, Opdyke pioneered the use of *preconditions* in refactoring, which are conditions formulated for every refactoring that the program has to fulfil in order for the refactoring to be safely applicable. By checking these conditions, the programmer can gain confidence that a refactoring will not break their code, although they may not be completely familiar with its every detail.

But even if the preconditions are sufficient to guarantee behaviour preservation, there remains the factor of human error both in checking the conditions and in actually performing the transformation. This problem has led to the development of *automated refactoring tools*, which offer support for checking preconditions to determine the applicability of a refactoring, and automating the code transformation involved.

The first example of such a system was the Smalltalk Refactoring Browser by John Brant and Don Roberts [112], and since then support for automated refactoring has become *de rigueur* in interactive development environments (IDEs), especially for object-oriented languages. Recent versions of IDEs for Java such as NetBeans [92], Eclipse JDT [44] and IntelliJ IDEA [59] all come with built-in refactoring engines

supporting a large number of refactorings, as do IDEs for languages like C#.[1]

Given the widespread interest in the process of refactoring and the need for reliable automated tools to perform refactorings, it may come as a surprise that even state-of-the-art refactoring tools for languages like Java, where refactoring is very popular among developers, still have fairly low standards of correctness. While they work well enough on most code, they are not very good at handling more advanced language features and will often produce code that either fails to compile, or, worse yet, still compiles, but has a subtly different semantics [118].

If a bug is identified and fixed for one particular refactoring, it is, of course, entirely possible that a similar bug occurs with another refactoring. Unfortunately, most refactorings have never been rigorously specified: beyond the rather informal description usually given in textbooks, the implementation itself is the only specification. This makes it hard to judge which, if any, refactorings are potentially affected by a given bug, and how to go about fixing them.

To counter this trend, this thesis develops concepts and techniques that make it possible to give concise, modular specifications of refactorings utilising a common framework encapsulating common functionality. These specifications are precise enough to cover all details of the refactored language, and thus give rise to full featured, high-quality refactoring implementations. Their modularity, on the other hand, encourages code reuse and makes them amenable to formal proof, opening the door to the verification of realistic refactorings.

1.1 Examples of refactorings

To make the discussion more concrete, we will introduce three well-known refactorings for Java, automated implementations of which are offered by most modern IDEs: RENAME VARIABLE, INLINE TEMP, and EXTRACT METHOD. We will briefly describe their purpose and the code transformation they perform, and point out challenges that have to be met in order to provide a behaviour-preserving implementation. Subsequent chapters will concentrate on addressing these issues in a systematic fashion. By the end of this thesis, we will be able to give precise specifications of these three refactorings, and many others as well, that provide the basis for robust implementations and are accessible to formal reasoning.

1.1.1 RENAME VARIABLE

The RENAME VARIABLE refactoring changes the name of a variable, for example to more clearly reflect its intended purpose.[2] This entails changing all the references to this variable, as well as ensuring that no other variable references become captured in the process.

A simple example program to be refactored is shown in Figure 1.1 on the left. There are two variable declarations in this program, one for the field `x`, which is an instance member field of class `A`, and one for the parameter `y` of the (only) constructor of `A`.

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

[1] When discussing these IDEs below, we will, unless indicated otherwise, base our discussion on NetBeans IDE 6.8, Eclipse 3.5 and IntelliJ IDEA Community Edition 9.0.1.

[2] We follow standard Java terminology [48] in using *variable* as an umbrella term to denote both local variables (including parameters) and fields (either static or instance fields).

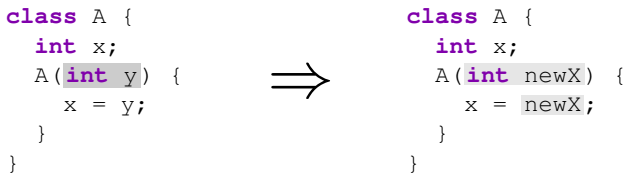


Figure 1.1: A simple example of RENAME VARIABLE

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

The example program is arguably not in very good style, since the name of the constructor parameter `y` bears no relation to the name of the field it is used to initialise, which may be a bit confusing. We could, for example, rename the parameter to `newX`; this is easily accomplished either by hand or by using an implementation of the RENAME VARIABLE refactoring in an IDE such as Eclipse, and will yield the refactored code on the right in which the parameter declaration and all references to it use the new name. Obviously, the refactored program has the same behaviour as the original program.

In future examples of the application of a refactoring, we will use the same conventions as in Figure 1.1: the original input program is given on the left (or top if it is very wide), and the refactored output program on the right (or bottom, respectively). To indicate the correct application of a refactoring, the two are connected by a heavy arrow ($\Rightarrow$). Sometimes we wish to give examples of faulty refactorings that change the behaviour of the input program; we will indicate this by a struck-through connecting arrow ($\not\Rightarrow$). If there is no way to correctly perform a proposed refactoring, a lightning symbol ($\lightning$) takes the place of the refactored program.

To more clearly highlight the effect of the refactoring, the part of the original program where the refactoring is applied (in this case, the parameter `y`) is highlighted in dark grey, whereas all changed code in the refactored program (in this case both the parameter and its use) is highlighted in lighter grey.

This example of RENAME VARIABLE is quite well-behaved: we are renaming a parameter, which can only be accessed from within its lexical scope (the constructor body, in this example), and there are not even any variable accesses that might be captured accidentally. Other cases are not so trivial. If we were to rename a field, for instance, we would, in general, have to search the whole program for accesses, and the name change could provoke name capture not only within the type where the field is declared, but also within any type that extends it, or even within a nested type. To make matters worse, there are certain situations in Java, discussed in more detail in Section 2.1 in the next chapter, where it is not clear whether a name refers to a variable, a type or a package. Such ambiguities are resolved by favouring variables over types, and types over packages, so when renaming a variable we have to be very careful not to change the outcome of this disambiguation process.

The same kind of global analysis may be required for the related RENAME TYPE refactoring that changes the name of a class, interface or type variable. In the case of RENAME METHOD we additionally need to be careful not to change overloading resolution or dynamic dispatch behaviour.

Traditionally, refactoring implementations have tried to avoid these complications by imposing preconditions that prevent them from occurring. As we shall see, however, the preconditions used in state-of-the-art refactoring tools are often too weak, in that they are insufficient to guarantee behaviour preservation in all cases. Sometimes, on the other hand, they can be too strong, in that they prevent refactorings that would not

```
double basePrice = anOrder.basePrice();
return basePrice > 1000;
```

```
return anOrder.basePrice() > 1000;
```

Figure 1.2: A simple example of INLINE TEMP

change the behaviour of the program. We shall see that an alternative approach that is based on dependency preservation instead of preconditions can fix both of these problems.

1.1.2 INLINE TEMP

Let us now look at a more local refactoring that only affects a single method. The INLINE TEMP refactoring eliminates a local variable (but never a parameter) by inlining its initialisation expression into all its uses. A very simple example, taken from Fowler's textbook [45], is shown in Figure 1.2, where the local variable `basePrice` is inlined into its single use.

Based on this example, the refactoring is easy to describe: replace all uses of the inlined variable with a copy of its initialising expression, then delete the declaration of the variable. But of course things are not always this easy: since Java allows variable declarations to be intermingled freely with other statements, name resolution may be different at the point where the expression is inlined, which could lead to names binding to different declarations and hence change the behaviour.

Since the inlined expression may read the values of fields or local variables, the refactoring needs to ensure that these reads see the same values at the point of inlining as at their original position. In particular, there should not be an intervening write that might change the value of one of these variables. Given a control flow graph that describes all possible execution paths, this is easy to ascertain for local variables or parameters; for fields, on the other hand, some form of alias analysis has to be performed to determine whether an intervening assignment to a field could actually influence a read, and if there are method calls their effect on field values has to be determined as well.

Furthermore, evaluating an expression in Java may incur side effects such as assignments to variables or interaction with the user. Inlining an expression makes these side effects occur at a different place in the program, or even at many different places; the refactoring has to ensure that this does not change the observable behaviour of the program. For sequential programs, this could mean, for instance, ensuring that the same methods are called in the same order with the same arguments, and that fields are left with the same values after the method finishes executing. For concurrent programs, the implementation must additionally guarantee that interaction between threads is not affected: synchronisation protocols should work the same as before, and no deadlocks or livelocks may be introduced.

This raises rather formidable challenges for an implementation, which current implementations are, as we shall see, ill prepared to meet. Most refactoring engines rely on very crude checks to identify potentially problematic data flow changes, and almost no attempt is made to deal with the complexities arising from concurrent program execution. We will show that reasoning in terms of static semantic dependencies that capture both data flow and synchronisation constraints offers a simple way of understanding INLINE TEMP and similar refactorings, and shows the way to implementing them in a principled and safe fashion.

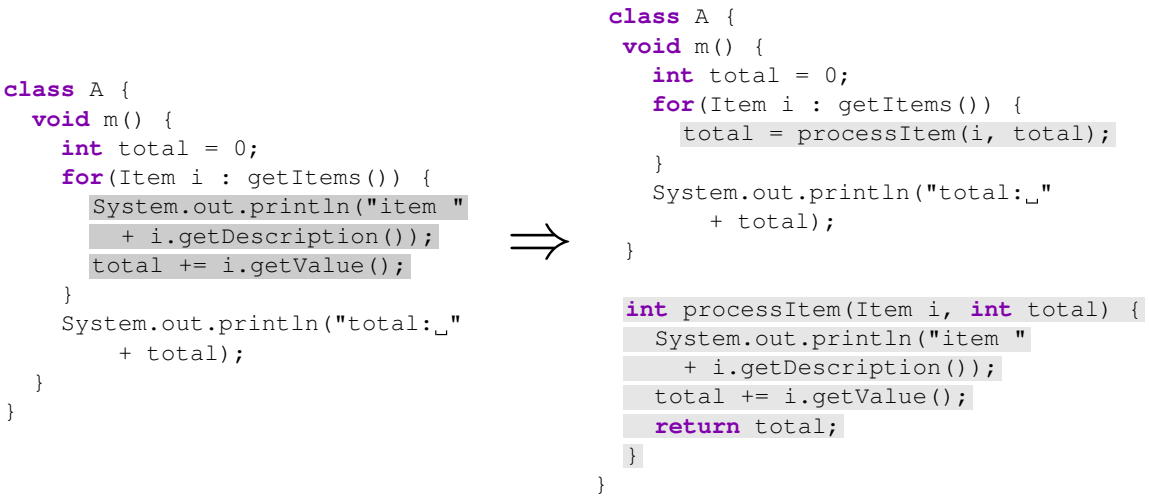


Figure 1.3: A simple example of EXTRACT METHOD

1.1.3 EXTRACT METHOD

As our final example, let us consider EXTRACT METHOD. This refactoring has garnered some attention in the literature, in particular since it was dubbed "Refactoring's Rubicon" in an influential article by Fowler [46], where it was described as a typical example of a non-trivial refactoring that requires a certain amount of analysis to be performed correctly. An implementation of EXTRACT METHOD, according to Fowler, is the hallmark of a "serious" refactoring tool.

Again, the refactoring is deceptively easy to describe: Given a sequence of statements, extract these statements into the body of a newly created method, and replace the original statements by a call to that method. As a simple example, consider the method `m` in Figure 1.3 on the left, and assume we want to extract the body of the **`for`** loop into a new method `processItem`.

This is relatively easy; all we need to do is provide `i` and `total` as parameters to the new method, and return the value of `total` to update the original variable after the method returns. Thus the resulting program should look like the one shown in the same figure on the right.

But a little thought should convince the reader that the simplicity of this example is misleading. What if the code to be extracted is not straight-line? What if it contains unstructured control flow such as **`break`**, **`continue`** or **`return`**? What if it throws exceptions? How do we determine which parameters to pass to the newly created methods? How do we decide which values to return to the calling method, and what do we do if more than one value would need to be returned?

Although EXTRACT METHOD is a fairly well-studied refactoring, giving preconditions that are sufficient to guarantee behaviour preservation is a daunting task, and understanding them can be no less challenging. While the previous two refactorings RENAME VARIABLE and INLINE TEMP are simple enough to be thought of as one-step transformations, in the case of EXTRACT METHOD it would be beneficial if we could split the transformation into several steps that each address a subset of the problematic issues in turn. We will show that this is indeed possible, but in order to achieve a natural decomposition we have to view the refactoring as working on an enriched language that contains features not found in plain Java.

1.2 Challenges in specifying and implementing refactorings

As the examples in the previous section have shown, specifying and implementing even conceptually very simple refactorings for a language like Java is a formidable challenge, which is not adequately met by current state-of-the-art implementations. In this section we will give an impressionistic overview of some of the main issues, before we outline how to handle them in the next section.

Name binding Almost any refactoring is susceptible to name binding problems. This is especially poignant for the RENAME refactorings but every refactoring that moves named program entities (fields, methods, types) might likewise introduce unwanted shadowing or name capture. Conversely, if a refactoring introduces a new reference to a named entity into the program, it is often not easy to ensure that it really binds to the intended declaration, and is not captured by another declaration of the same name.

The name binding rules of Java are quite complex: variable and type lookup follow a lexical scoping discipline, enriched with inheritance of member fields and member types; methods can be overridden and overloaded; accessibility rules put further constraints on lookup results; names can be qualified in many different ways, but these qualified names introduce syntactic ambiguities where a name has to undergo a process of disambiguation to determine whether it binds to a package, a type or a variable.

Control and data flow Refactorings that restructure code within methods have to be very careful about changing control and data flow. For example, they usually need to ensure that side effects occur in the same order and that variable reads still see the same writes as before the refactoring.

Control flow analysis in Java is made difficult by unstructured control flow constructs such as `break` and `continue`, as well as the pervasive use of exceptions. Determining which method a call may dispatch to at runtime is non-trivial, because instance methods are invoked by virtual dispatch, where the runtime type of the receiver object decides which method is dispatched to. Data flow analysis has to deal with reference aliasing, where two references may refer to the same object at runtime. Performing inter-procedural data flow analysis to determine the data flow behaviour of method calls can be very costly; for native methods and methods invoked by reflection it may even be impossible to obtain precise results.

Concurrency To preserve the behaviour of concurrent programs, refactorings have to make sure that synchronisation occurs on the same monitor objects, and that access to state that is shared between threads is synchronised in the same way as before the refactoring. Failure to do so may introduce data races or deadlocks, or enable new concurrent behaviour that was not possible in the original program.

The behaviour of concurrent Java programs is described by the Java Memory Model, which provides very weak guarantees about the order in which actions taken by one thread are observed to happen by another thread. This allows a compiler greater freedom in optimising threads individually, but it greatly complicates reasoning about possible interactions between threads.

Language idiosyncrasies Most real-world languages have evolved a fairly non-orthogonal syntax with many constructs providing "syntactic sugar" to abbreviate commonly used idioms. Violating syntactic rules will usually result in uncompilable code, which can be very frustrating to the user, while syntactic sugar may obscure crucial dependencies that a refactoring should preserve.

In Java, there are different kinds of expression that can only occur in specific syntactic positions; for instance, array initialisers are only allowed in variable initialisations. Depending on its syntactic position, an

expression may be subject to different kinds of implicit conversions, which would need to be made explicit in other positions.

1.3 Our approach

We propose in this thesis a novel approach to understanding refactorings that we believe is superior to a purely precondition-based approach for the purposes of specifying, implementing and verifying refactorings. It is based on three main ingredients.

The first is the notion of *dependencies*, which capture static semantic properties of the program to be refactored. For instance, there is a name binding dependency from every name to the declaration it accesses, and a flow dependency from every use of a variable to each of its reaching definitions. We provide a general framework for tracking such dependencies over the course of a refactoring and for making sure that they are preserved in the output program. This framework is independent of any individual refactoring, reflecting the generic nature of the problems it addresses.

The second ingredient is the concept of *language restrictions and extensions*. Real-world programming languages often offer constructs that abbreviate commonly seen idioms in an implicit form. Such constructs can present difficulties to a refactoring, since they may obscure semantic dependencies that the refactoring needs to take care of. The implementation of many refactorings can be simplified by expanding such constructs into a more explicit form so that the refactoring can work on a simpler, more restricted language. Conversely, the language may be missing certain constructs that would streamline the formulation of a refactoring. It may then be helpful to allow intermediate refactoring steps to work on an enriched language that supports these constructs, as long as they are eliminated or translated away in the output program; we call such ephemeral constructs *lightweight language extensions*. If the same language restrictions or extensions are used by several refactorings, their handling can be centralised, thus increasing modularity and code reuse.

The third important feature of our approach is the decomposition of refactorings into smaller *microrefactorings*, which are themselves behaviour-preserving refactorings that accomplish one step of the overall refactoring. Often, language extensions are needed to reveal the fine structure of a refactoring in order for it to be decomposed in this way. These microrefactorings can be understood, specified, implemented and tested in isolation. Once a refactoring is decomposed, it often turns out that some of its constituent microrefactorings can be reused as components of other refactorings.

The dependency framework allows us to abstract away from many of the gory details of name binding or control and data flow, which are handled transparently. In a similar manner, the language restrictions and extensions allow us to pretend that we are working on a more orthogonal language. The decomposition into microrefactorings gives individual refactorings a modular structure, and promotes reuse of similar functionality between different refactorings.

Taken together, these concepts allow us to give concise, high-level specifications of refactorings that abstract away from confusing details. Those details are, however, not ignored, but simply handled by a common framework. Thus the specifications are precise enough to serve as the basis of high-quality refactoring implementations. At the same time, their high level of abstraction and modularity makes them easier to reason about and to verify their correctness.

1.4 Contributions

The main contributions of this thesis can be briefly summarised as follows:

- We propose a new approach to the specification and implementation of refactorings, based on dependency preservation, language restrictions and extensions, and microrefactorings.
- We offer a novel explanation of renaming in terms of access construction, a partial right inverse of name lookup, yielding a general name binding framework that can be used to ensure preservation of name binding and to avoid name capture.
- We show how control and data flow dependencies can be used to ensure the correctness of refactorings that rearrange or move code.
- We give the first principled treatment of refactoring for concurrent Java programs, based on the concept of synchronisation dependencies, a static approximation to concurrent program behaviour.
- We introduce the novel concept of lightweight language extensions for simplifying and modularising the description of refactorings.
- We report on the results of an extensive case study to show that the proposed techniques apply to a wide range of refactorings, yielding both high-level descriptions and high-quality implementations.
- We formalise our name binding framework for a core calculus of Java in the theorem prover Coq [30], including a formalisation of name lookup and access construction.
- We describe a mechanical proof of the correctness of access construction in our formalisation.

1.5 Outline

We begin by discussing the RENAME refactoring in Chapter 2. We explain how it can be seen as one application of a general naming framework that can construct names that are guaranteed to bind to a certain declaration and will automatically add qualifiers to avoid name capture where necessary.

In Chapter 3 we show that the relationship between a name and the declaration it should bind to is just one example of a more general class of dependencies that refactorings need to preserve. As further examples of such dependencies, we discuss control and data flow dependencies, and then show that the concept carries over to the case of concurrent programs via the technique of synchronisation dependency preservation.

Building on the dependency preservation framework, Chapter 4 shows how refactorings can be decomposed into smaller microrefactorings if we allow intermediate steps to work on a richer language, making the description of refactorings both simpler and more modular.

The techniques and concepts we have introduced are put to work in Chapter 5, where we report on an effort to give high-level specifications of the majority of refactorings offered in recent versions of the Eclipse IDE. We show that these specifications easily yield high-quality refactoring implementations.

By emphasising the role of statically computable dependencies and breaking down large refactorings into more manageable components, we also gain a new approach to the verification of refactorings, which is the topic of Chapter 6. As a first result in this area, we present a verified implementation of the name binding framework for a subset of Java in the theorem prover Coq.

Throughout the thesis, our discussion focusses on Java as a typical example of a mainstream object-oriented programming language with an established community of refactoring users. In Chapter 7, we consider the wider applicability of the techniques and concepts we have introduced beyond the scope of Java. We also evaluate the potential of other approaches to the specification and implementation of refactorings, and conclude by outlining avenues for future work.

The exposition is complemented by two appendices, the first presenting the complete implementation of name lookup and the naming framework for the subset of Java considered in Chapter 2 and Chapter 6, the second detailing conventions and definitions used in the specifications in Chapter 5.

1.6 Previous work by others

To put this thesis into a broader context, we will now give a brief overview of important previous work on the specification, implementation and verification of refactorings. Additionally, each thesis chapter will include a more detailed discussion of related work that positions the presented material with respect to the literature.

The refactoring literature has generally followed a precondition-based approach to specifying and implementing refactorings. This approach, pioneered by Opdyke [100], relies on finding, for every refactoring, a set of *preconditions* that need to hold in order for the refactoring to be behaviour preserving. Once the preconditions have been established, the refactoring amounts to nothing more than a simple syntactic transformation.

This approach was complemented by Roberts [111] with *postconditions* that can be expected to hold after the refactoring has been performed, provided, of course, the preconditions hold initially. Postconditions are particularly useful when two refactorings are performed in sequence, as some of the preconditions of the second refactoring may be implied by the postconditions of the first, and hence do not have to be checked.

Preconditions similar to those given by Opdyke and Roberts have since been used in popular textbooks on refactoring [45, 67], and they also lie at the heart of the refactoring implementations of Eclipse and other IDEs. Many of these preconditions are of a quite syntactic nature, so they are easy to check without any deep program analysis. At the same time, however, this makes it very hard to show that the conditions are sufficient to ensure behaviour preservation. Opdyke, for instance, only offers informal arguments for the correctness of his preconditions, and only considers a simplified subset of C++.

A very different approach was proposed by Griswold [49]: he studies refactorings in terms of their effect on the static semantic dependencies of a program as represented by its program dependence graph (PDG). He then specifies several simple refactorings in such a way that they become isomorphisms on PDGs, thus obtaining strong behaviour preservation results.

While it is unclear whether this approach scales to more complicated refactorings and more complicated languages than the first-order subset of Scheme he considers, the idea of using static semantic information to formulate refactorings and the emphasis on dependency preservation also figure prominently in our approach.

Much of the more recent literature on the specification and implementation of refactorings is based on the Opdyke-Roberts approach, often refined by formulating preconditions in terms of static semantic dependencies. Our approach, on the other hand, emphasises the *preservation* of static semantic dependencies, either by checking their preservation after a refactoring has been performed, or by specifying a refactoring in such a way that it preserves certain dependencies by construction.

Up to a point, this is an implementation detail: instead of checking that a dependency is preserved after the refactoring, we might as well introduce a precondition to ensure its preservation beforehand. Often, however,

preservation conditions are more easily expressed with respect to the refactored program (as opposed to the original program), and at least for name bindings we shall see that the preservation-oriented approach can be generalised to fix broken dependencies.

The composition of simple refactorings into more complicated ones was already considered by Opdyke, and in more detail by Kniesel and Koch [71]. Both use a precondition-based approach in which preconditions and postconditions of constituent refactorings are composed into conditions for the composite refactoring.

We have found that when considering all the complexities of real-world languages, even supposedly simple refactorings are not so simple after all, and can benefit from further decomposition into microrefactorings. The concept of lightweight language extensions, which we have found indispensable for achieving this decomposition, does not seem to have appeared in the refactoring literature before.

The verification of refactorings has attracted considerable interest in the literature. Beyond the informal correctness arguments presented in early papers on refactoring, many authors have worked on formally proving the preconditions of certain refactorings sufficient with respect to a formal semantics [10, 31, 41, 83, 127].

Invariably, these papers only consider toy languages or simplified subsets of real-world languages. The results presented in this thesis suggest that by working with static semantic dependencies one can obtain meaningful correctness results even for larger languages without the need for a full formal semantics.

1.7 Previous publications

Most of the material presented in this dissertation is based on the following five publications by the author:

Sound and Extensible Renaming for Java [116] This paper, published at OOPSLA 2008, proposes name binding preservation as a correctness criterion for renaming, and details the design and implementation of a name binding framework for Java. It forms the basis of Chapter 2.

Formalising and Verifying Reference Attribute Grammars in Coq [117] This paper, published at ESOP 2009, introduces a shallow embedding of reference attribute grammars into the type theory of the theorem prover Coq, and uses it to formalise and verify a name binding framework for a subset of Java. The second half of Chapter 6 expands on this work.

Stepping Stones over the Refactoring Rubicon [119] This paper, published at ECOOP 2009, demonstrates that refactorings become easier to implement and more modular if intermediate steps can work on an enriched language. It underlies Chapter 4.

Correct Refactoring of Concurrent Java Code [115] This paper, published at ECOOP 2010, advocates a dependency-based approach to refactoring concurrent programs, and relates it to the Java Memory Model. The second half of Chapter 3 is based on this work.

Specifying and Implementing Refactorings [114] This paper, published at SPLASH 2010, shows that our dependency-based presentation of refactorings and their modular decomposition, aided by the use of language extensions, allows us to give high-level specifications of many refactorings that directly lead to high-quality implementations. Chapter 5 elaborates on this work.

Each chapter of this thesis will further detail the relationship of the presented material with these papers. Since all of them are joint work with other researchers, we will also clarify which parts of the development are the author's own work, and what has been achieved as the result of collaboration with others.

Chapter 2

Name Binding

We start our exposition by investigating what is arguably the most basic refactoring of all, the RENAME refactoring. To be precise, there is actually a whole family of RENAME refactorings, since different named entities in Java have to be renamed in different ways. Perhaps the most intricate of these is RENAME VARIABLE, which was briefly discussed in the introduction; other related refactorings are RENAME TYPE, RENAME PACKAGE and RENAME METHOD.

Figure 1.1 has shown a very simple example of renaming a constructor parameter that is handled satisfactorily by current refactoring implementations. Continuing with this example, assume we want to rename the constructor parameter to have the same name as the field it initialises, which is a commonly seen idiom. The proposed transformation is shown in Figure 2.1.

Since the access to field `x` would be captured by the renaming, we need to qualify it with `this`. While IntelliJ can perform this simple adjustment, both NetBeans and Eclipse refrain from doing so, and instead reject the whole refactoring. This is symptomatic of a more general problem with the precondition-based approach to refactoring correctness: preconditions have to be formulated from scratch by reverse-engineering the language specification, and it is very hard to ensure that they cover all problematic cases without forbidding too many useful refactorings.

When we tried to rename `newX` to `x` above, the preconditions were strong enough and warned about the name collision. In fact, they were perhaps a little too strong, since the refactoring can easily be carried out with only slightly more invasive changes to the user's code. However, in Figure 2.2 on the left we have a program, not much more complex, where Eclipse's preconditions are not strong enough.

This little program constructs a thread and runs it; the thread will print the value of the local variable `y` of method `main`, namely `23`. Similar to before, we have a field `x` and a local variable `y`, but this time their scopes are nested in the opposite way. If we tell Eclipse to rename the local variable to `x`, it obliges, but

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

Figure 2.1: A slightly more complex example of RENAME VARIABLE

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

⇏

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

Figure 2.2: A yet more complex example of RENAME VARIABLE

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

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

⇒

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

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

Figure 2.3: Renaming variables in the presence of static imports

creates the output program seen on the right, whose behaviour is different (as indicated by the struck-through arrow): it will print `42`, since the access `x` now refers to the field instead of the local variable.

Now one might well argue that the above example is quite artificial. After all, who would use an anonymous class in this way? But remember that refactoring is a process for improving code, which is likely to start with a very ill-structured and chaotic program. It is hence dangerous to make assumptions that certain corner cases do not appear in "normal" code. Automated refactoring tools are most useful for dealing with convoluted, obscure code that the developer does not yet fully understand. In this example, it would be much preferable for the refactoring tool to indicate the imminent shadowing and to abort the refactoring, since there is no way to qualify the access to `x` as we did for the field access in the first example.

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

To pick just one example, let us look at the static import feature. It allows the programmer to import static fields and methods of classes so that they do not have to be explicitly qualified, which is very useful for frequently used constants like `Math.PI`. But the introduction of static imports also had a significant impact on the way name lookup is performed: Previously, only types could be imported, so **`import`** statements never affected the visibility of variables. Static imports do, however, and what affects lookup also affects renaming.

Take, for example, the program in Figure 2.3 on the left. The class `Indiana` is located in a compilation unit that statically imports the field `Math.PI` so that it can be accessed using the simple name `PI`; at the same time, the class also defines a static field named `myPI`.

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

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

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

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

Figure 2.4: A variation of the static import example

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

Yet both NetBeans and IntelliJ (before version 8.0) fail to take this into account and incorrectly refactor this example. Eclipse detects the name clash in this case, but fails on slightly more complicated examples [118].

While the above example works in recent versions of IntelliJ, a slightly more devious program shown in Figure 2.4 still is not handled correctly. When renaming `myPI` to `PI`, IntelliJ realises that the reference to `java.lang.Math.PI` in method `circleArea` is about to become shadowed, so it qualifies it. Since the referenced field is static, the access is qualified with an access to its declaring class, `java.lang.Math`, and because there is no other class `Math` that could hide this class, IntelliJ decides that it is safe to replace the endangered access with `Math.PI`. It is not.

Syntactically speaking, the name `Math` in the access `Math.PI` could either refer to a variable (with `PI` being a member field of that variable's type), or to a type. In this example, of course, we mean it to be the latter, but according to the Java language specification the interpretation as a variable is preferred: only if there is no visible variable `Math` should we try to interpret it as a type. Since there *is* a field `Math` in the surrounding class, `Math` will be interpreted as an access to that field, and `PI` will be resolved as a field of type `int`, the type of the field: field `Math` is said to *obscure* class `Math`.

Of course, the type `int` has no fields, so the resulting program will not compile, but it would be easy to slightly change the program so that it still compiles but behaves differently. In this example, we should instead replace the simple access `PI` with the qualified access `java.lang.Math.PI`—but even that access would not be safe if a variable or class `java` were in scope, since packages can be obscured by both variables and types.

These examples should be sufficient to show that the name lookup rules in Java are quite intricate, making it very difficult to come up with sufficient preconditions to avoid accidental name capture in all cases.

We instead propose to look at this problem from a different angle: when performing a renaming, what we really want to achieve is that in the refactored program all names still bind to the same declaration as in the original program. Of course, all uses of the declaration being renamed should be updated such that they still bind to this declaration, while all names that bind to different declarations should continue binding to those declarations, which may make it necessary to qualify them in some cases. Nothing else should change in the program.

This goal can be realised by *access construction*. Assume we have a procedure for access construction that, given a position in the program and a declaration, constructs an access (i.e., a possibly qualified name) that binds to the declaration if inserted into the program at the given position. Then renaming is easy to implement: First, compute for every name which declaration it binds to. Then change the name of the declaration to be renamed. Finally, go over all names and compute an access that binds to the same declaration it bound to originally, and replace the name with this new access. If the access construction function is correct, this will yield a program with the same binding structure as the original program, as desired.

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

$$lookup_p : access \rightharpoonup decl$$

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

$$access_p : decl \rightharpoonup access$$

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

$$lookup_p(access_p(d)) = d \tag{2.1}$$

for every declaration d and program position p, if $access_p(d)$ is defined.

Access construction is widely useful beyond renaming: for instance, many refactorings need to create code containing contains names that should bind to certain known declarations, or move around code without upsetting name binding. With an access computation function at hand this is easily done, since we can always use it to construct an access that is guaranteed to bind to the declaration we want it to bind to. All these refactorings can make use of the common access construction functionality, which acts as a general naming framework that shoulders the burden of dealing with Java's complex naming rules.

This chapter shows how to systematically derive an implementation of the access computation function from an implementation of the lookup function for Java, and evaluates it empirically. Chapter 6 will revisit the problem of formally establishing the correctness of an access function with respect to the lookup function.

2.1 Name lookup in Java

Before we delve into the details of how to construct accesses to declarations, it is worthwhile to review the most important concepts of name lookup in Java on some simple examples. The authoritative specification of name lookup, as of every other aspect of the Java language, is the Java Language Specification [48], on which we base this brief exposition.

There are five kinds of named entities in Java: packages, types, variables, methods, and labels. The former four appear in the program in Figure 2.5, which shows a compilation unit in package `p`. That compilation unit declares five types, namely the classes `Super`, `Outer`, `Inner` and two classes named `A`. It also uses the built-in type `int`. Observe that in Java classes can be either toplevel classes like `Super` and `Outer`, or member classes like `Inner` and the two `A`.

```
package p;

class Super {
  int f; /*①*/
  class A { }
  int m(int i) { return 42; }
}

class Outer {
  int f; /*②*/
  int x;
  class A { }
  class Inner extends Super {
    int f; /*③*/
    int y;
    int m(int f /*④*/) {
      A a1;
      Outer.A a2;
      p.Outer.A a3;
      return x + y
           + f                  // → ④
           + this.f             // → ③
           + super.f            // → ①
           + Inner.this.f       // → ③
           + Inner.super.f      // → ①
           + Outer.this.f       // → ②
           + ((Super)this).f    // → ①
           + super.m(23);
    }
  }
}
```

Figure 2.5: Example for name lookup in Java

Class `Super` declares a field `f`, a class `A`, and an instance method `m`; 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`. Besides its local members, `Inner` also inherits the member class `A` from `Super`. It does not inherit field `f`, since the local field *shadows* it, and it does not inherit method `m`, since the local method *overrides* it. Also note that the field `f` of class `Outer` is *hidden* inside `Inner` by the local field `f` in `Inner`.

Method `m` has a parameter `f` that hides 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 accesses. A type access can be a simple name, as in the case of the declaration of `a1`. Type names are looked up by proceeding through the enclosing scopes until the first matching type is found. In this case, class `Inner` inherits type `A` from `Super`, so the type access in the declaration of `a1` binds to that `A`, not the `A` from `Outer`. To access the latter type, we have to qualify it with an access to its enclosing type, which can be the simple name `Outer` as in the declaration of `a2`. We can also qualify an access to a toplevel type like `Outer` with an access to the package it belongs to, as in the declaration of `a3`.

To illustrate variable lookup, we demonstrate several cases in the **`return`** statement of method `m`. First of all, note that `x` and `y` can be accessed inside `m` without any qualification, since they are declared in enclosing classes and are not hidden by another declaration. This would still work if `y` were declared in class `Super`,

```
class A {
  static class Weird {
    static double PI = 22.0/7;
  }

  static class java {
    static class lang {
      static class Math {
        static double PI = 3.0;
      }
    }
  }

  Weird Math;
  Math m;

  double pi1 = m.PI;                    // 3.141592653589793
  double pi2 = Math.PI;                 // 22.0/7
  double pi3 = java.lang.Math.PI;       // 3.0
}
```

Figure 2.6: Example for ambiguous names

or `x` in a superclass of `Outer`, but not if `y` were declared in an enclosing class of `Super`. The simple name `f` in the next line, of course, refers to the parameter `f`, as indicated by the comment.

The following lines show different ways of qualifying field accesses. We can access the field `f` of class `Inner`, which is hidden by the parameter `f`, by qualifying it with **`this`**. The field `f` from `Super`, although shadowed by the field `f` in `Inner`, is accessible through a qualification with **`super`**. We can access the same two fields through qualification with `Inner.`**`this`** and `Inner.`**`super`**, although such qualified **`this`** accesses are more usually employed to access hidden fields of enclosing classes, as with the access `Outer.`**`this`**`.f`. Note that for fields, the access **`super`**`.f` is equivalent to `((Super)`**`this`**`).f` in the sense that both bind to the same field, and at runtime will refer to the same memory location.

For methods, the situation is slightly different. Method accesses can be qualified in the same way as field accesses, and are bound to method declarations in a very similar way. In particular, the access **`super`**`.m(23)` in the final line of method `m` binds to the method `m` in class `Super`, as would the access `((Super)`**`this`**`).m(23)`. However, at runtime the former is dispatched as a static invocation, that is, no matter what the dynamic type of **`this`** turns out to be, it is always the method in `Super` that is invoked. The latter, on the other hand, is dynamically dispatched, hence it would dispatch to method `m` in `Inner` if the dynamic type of **`this`** is `Inner`, resulting in an infinite loop.

Note that names referring to methods or labels can only occur in very restricted syntactic positions (method invocation expressions for the former, **`break`** and **`continue`** statements for the latter), so it is always clear whether a name refers to a method, to a label, or to some other entity. For packages, types, and variables, however, there is an ambiguity arising from the use of qualified names.

As an example, consider the program in Figure 2.6, which bears some similarity to an example from the introduction. Here we have a class `A` with two member classes `Weird` and `java`, as well as some instance fields.

First, note that the name `Math` appearing in the declaration of `m` is unambiguously a type name. Given its syntactic position, it cannot possibly refer to a variable, so there is no ambiguity with the field `Math`. Instead,

it binds to the class `java.lang.Math` from the standard library, and `pi1` is initialised to the value of the static field `java.lang.Math.PI`.

Things are not so clear in the case of the name `Math.PI` appearing in the initialiser of field `pi2`. The name `Math` could either refer to a type, with `PI` a static field of that type, or it could refer to a variable, with `PI` a member field of the type of that variable. In such a situation, the language specification dictates that `Math` is first looked up as a variable. In the example, this succeeds, since there is indeed a variable `Math` in scope, so we proceed to look up `PI` as a member field of `Weird`, the type of field `Math`. If there were no variable `Math` in scope, we would instead try to look up `Math` as a type, and `PI` as a static field of that type. Note, however, that the decision of whether to interpret `Math` as a type access or as a variable access is made *before* looking up `PI`: for instance, if there were no field `PI` in class `Weird`, the decision to interpret `Math` as a variable access would not be revised; instead, the compiler would report an unbound name.

Similarly, in the initialisation of `pi3`, the names `java` and `lang` could conceivably either refer to packages or types or variables, and `Math` could either refer to a type or a variable, though definitely not a package. Again, we start by disambiguating `java`, giving preference to the type `java` over the package `java`: we say that the type *obscures* the package. This, of course, eliminates the possibility that `lang` could be a package: it must refer either to a member field or to a member type of class `java`, and since the class has no such member field it is determined to be a member type. Based on this decision, we now determine that `Math` must bind to a member type of `lang`, there being no member field of the right name, and `PI` is looked up as a field in that type.

The precise rules of how to disambiguate names based on their syntactic position and the names in scope are a bit involved; the interested reader is referred to the language specification [48, §6].

2.2 Implementation of name lookup in JastAddJ

As outlined above, our implementation of renaming will rely on a general naming framework that implements a process of access construction, which can be seen as a right inverse of name lookup. In order to implement access construction, then, we need an implementation of name lookup to base it on.

Our refactoring engine is implemented on top of the JastAddJ extensible Java compiler by Ekman and others [39]. We use the compiler's frontend to parse the program to be refactored and build an abstract syntax tree representation of it. The frontend also performs semantic analyses and error checking, among which name resolution is the most important for the purposes of this chapter. In this section, we will give a brief overview of how name lookup is implemented in JastAddJ; for a more in-depth treatment, the reader is referred to the literature [38].

JastAddJ is a full-featured Java 5 compiler implemented using JastAdd [40], a Java-based compiler compiler system. JastAdd extends Java with rewritable circular reference attribute grammars [37, 86] as a declarative specification language, and a number of aspect-oriented concepts, notably inter-type declarations, to improve modularity.

From the programmer's perspective, JastAdd extends Java with two new kinds of modules: abstract grammar modules for defining node types of the abstract syntax tree, and aspect modules for defining attributes and inter-type methods on these node types. These new modules are combined and translated into plain Java by a preprocessor and then compiled by a normal Java compiler, hence attribute definitions can be used from and integrated with arbitrary Java code.

```
Program ::= ClassDecl*;

ClassDecl ::= ⟨Name:String⟩ [Super:Access] BodyDecl*;

abstract BodyDecl;
Field : BodyDecl ::= VarDecl [Init:Expr];
Initializer : BodyDecl ::= Block;
MemberClass : BodyDecl ::= ClassDecl;

VarDecl ::= Type:Access ⟨Name:String⟩;

abstract Stmt;
Block : Stmt ::= Stmt*;
LocalVar : Stmt ::= VarDecl [Init:Expr];

abstract Expr;

abstract Access : Expr;
SimpleAccess : Access ::= ⟨Name:String⟩;                    // x
QualAccess : Access ::= Left:Expr Right:SimpleAccess;       // e.f

This : Expr;                                                // this
QualThis : Expr ::= Type:Access;                            // A.this
CastExpr : Expr ::= Type:Access Expr;                       // (C)e
```

Figure 2.7: An abstract grammar for a small subset of Java

2.2.1 Abstract grammars

To make our exposition more concrete, we will explain how to implement name lookup for a miniature subset of Java in JastAdd. The most interesting bits of code will be discussed in the text; the full definition can be found in Appendix A.

The subset of Java which we consider, henceforth called *NameJava*, is designed to model some of the most essential features of name lookup for variables and types in Java, to the exclusion of most other language features. NameJava has nested blocks with local variables, nested class declarations with single inheritance, and qualified names for both types and variables. There are no statements except local variable declarations and blocks, and no methods or constructors, but only initialisers. To simplify presentation, we also omit packages, compilation units, import declarations and access modifiers. We will briefly explain how these can be handled below.

An abstract grammar module to describe the abstract syntax of NameJava in JastAdd is given in Figure 2.7. The grammar consists of a set of productions, with each production describing a nonterminal, which represents a type of nodes in the abstract syntax tree (AST). Instead of the traditional approach of providing multiple productions for the same nonterminal to indicate alternative ways to construct or derive it, JastAdd favours a more object-oriented approach, where node types can inherit from each other. The supertype a node type inherits from, if any, is given after a colon; for example, `Field` inherits from `BodyDecl`. This means that a node of type `Field` can occur anywhere a node of type `BodyDecl` is expected. JastAdd will generate one Java class for every AST node type, with the inheritance relation between node types straightforwardly translated into inheritance between the generated Java classes. Node types that are declared **`abstract`** will become abstract classes, so for instance there will never be an AST node that actually has type `BodyDecl` or `Expr`.

Node types can declare child elements, given after the `::=` symbol. A child element can either be terminal, representing a value stored in the node, or nonterminal, representing a child node. Terminal children are declared between angle brackets $\langle\cdot\rangle$, giving a name followed by a colon and a type. Terminal children can have arbitrary Java types; for instance, node type `ClassDecl` has a terminal child `Name` of type `String`, i.e., the standard library type `java.lang.String`. The name of the terminal child can be omitted, in which case it defaults to the name of its type.

Nonterminal children come in three flavours: the simplest just represent one child node. For instance, type `VarDecl` has a child node `Type`, which is an `Access`. Note that in contrast to terminal children the type of a nonterminal child must be a node type. As with terminal children, the name of a nonterminal child can be omitted, and will then default to the name of its type; for instance, the single child node of type `Initializer` has name `Block`. Nonterminal children can further be declared to be optional by enclosing them within brackets $[\cdot]$; for instance, the child `Super` of type `ClassDecl`, which holds the name of the superclass, is optional to indicate that not every `ClassDecl` needs to have a `Super` child. Finally, node children can represent a list of zero or more child nodes of the same type if their declaration is postfixed with a star. For instance, a `ClassDecl` node may have zero or more `BodyDecl` child nodes.

The grammar should now be mostly straightforward to understand, but several node types deserve some explanation: notice that both `Field` and `LocalVar` have a child node of type `VarDecl`, which in turn has a name and a type. We use `VarDecl` nodes to encapsulate code that is common to fields and local variables and only depends on them having a type and a name; for instance, the result of a variable lookup will be a `VarDecl`.

In NameJava, there are only four kinds of expressions: accesses, cast expressions, and unqualified and qualified **`this`** expressions; the comments in the grammar provide an example of each. There are no **`super`** expressions, which can be simulated by casts and **`this`** expressions. Accesses are either simple names, or qualified accesses consisting of a qualifying expression and a simple name; depending on syntactic context, an access refers either to a class or to a variable. For instance, in a position where a variable access is expected, the access `C.f` will be interpreted as "the field `f` of variable `C`", unless there is no such variable, in which case it will be interpreted to mean "the field `f` of class `C`"; in the former case access `C` binds to a variable, in the latter to a class.

As mentioned above, JastAdd will generate one Java class per nonterminal of the grammar. These classes have fields corresponding to the nonterminal's children, and getter and setter methods to access them. For example, part of the public interface of the class generated for `ClassDecl` is shown in Figure 2.8. It has a getter method `getName()` to access the class name, and a method `getSuper()` to retrieve the superclass access. The child `Super` was declared optional in the abstract grammar definition, so there is another method `hasSuper()` that can be invoked to find out whether a particular class declaration does, in fact, have a superclass access. In a similar vein, `getNumBodyDecl()` returns the number of `BodyDecl` children of the `ClassDecl`, `getBodyDecl(int)` provides access to individual body declarations, whereas `getBodyDecls()` provides a way to iterate over all body declarations.

Since node type `ClassDecl` is not declared to inherit from any other node type, its class directly extends `ASTNode`, which is the common superclass of all node types. Essentially, `ASTNode` behaves as if it were defined as

```
ASTNode ::= Child:ASTNode*;
```

In particular, it provides a method `getChild` to retrieve a child by index, and a method `getNumChild` to determine the total number of child nodes.

```
class ClassDecl extends ASTNode {
  public String getName() { ... }

  public boolean hasSuper() { ... }
  public Access getSuper() { ... }

  public int getNumBodyDecl() { ... }
  public BodyDecl getBodyDecl(int i) { ... }
  public Iterable<BodyDecl> getBodyDecls() { ... }
}
```

Figure 2.8: Skeleton of the node class generated for nonterminal `ClassDecl`

```
syn VarDecl Stmt.localVariable(String name) {
  return prev() == null ? null : prev().localVariable(name);
}

eq LocalVar.localVariable(String name) {
  if(name.equals(getVarDecl().getName()))
    return getVarDecl();
  return super.localVariable(name);
}

inh lazy Stmt Stmt.prev();
eq Block.getStmt(int i).prev() = i > 0 ? getStmt(i-1) : null;
eq Initializer.getBlock().prev() = null;
```

Figure 2.9: Attributes for looking up local variables in blocks

2.2.2 The AST: the ultimate symbol table

Traditionally, name lookup is performed by traversing the program's abstract syntax tree and creating symbol tables with information about visible declarations for various locations in the tree. While this works well enough for purely block-structured languages, it is less well-suited for an object-oriented language like Java: Multiple AST traversals may be necessary to ensure that both the visible declarations and their type information are entered into the symbol table before they are needed for looking up variable names. Scheduling such traversals by hand can be quite tricky in the presence of inheritance and name disambiguation, where name lookup and type analysis are mutually dependent. This problem is discussed in more detail in the literature [5].

JastAddJ takes a different approach in which name lookup is implemented as a family of *reference attributes*, that is, attributes whose value is a reference to another node. For instance, the value of the lookup attribute on a given node for a given name is a reference to the node of the declaration which the name binds to. In this way, the AST itself is used as a symbol table without any need for additional data structures. Of course, attributes that perform different parts of the name lookup still exhibit a high degree of mutual dependency, but due to their declarative nature their evaluation can be scheduled automatically by JastAdd.

One of the simplest parts of the name analysis for NameJava is shown in Figure 2.9: the *synthesised attribute* `localVariable` defined on node type `Stmt` accepts a name as its argument and looks it up on the statement itself and any other statements that come before it in the same block. Its default implementation for statements that do not declare a local variable is to recursively invoke itself on the previous statement if

```
interface Declaration { }
VarDecl implements Declaration;
ClassDecl implements Declaration;

VarDecl ASTNode.lookupVariable(String name);
ClassDecl ASTNode.lookupClass(String name);

Declaration Access.decl();

ClassDecl Expr.type();
```

Figure 2.10: The API for name lookup and type analysis

there is one, and to return `null` otherwise. This behaviour is overridden for local variables, where we check whether this is actually the variable to be looked up, and return it if so.

Note that definitions of attributes use normal Java method syntax, and indeed synthesised attributes will become instance methods of their respective node classes in the generated Java code. While attribute definitions can utilise arbitrary Java code, they are side-effect free by convention to preserve declarativity.

To compute the previous statement, the above code uses an attribute `prev`, which is implemented as an *inherited attribute*. In contrast to synthesised attributes, whose value on a given node is defined based on attributes defined on either that node or its children, inherited attributes are defined in terms of a node's siblings and parents, and in particular in terms of the position of the node relative to its siblings and parents. For instance, when computing the previous statement of a statement *s* we need to know whether *s* occurs in a block, and if so at which position. If *s* is not the first statement, then the previous statement is its immediate left sibling; if *s* is the first statement in the block, or if it is not contained in a block at all, it does not have a previous statement.

This kind of pattern matching on the position of a node is reflected in the syntax JastAdd offers for defining inherited attributes. Among the two defining clauses for attribute `prev`, the former defines it for any statement that occurs as the `i`th statement within a block; instead of the method-like syntax we used for defining `localVariable`, we here use an equational syntax, which is simply a short-hand for a method body consisting of a single `return` statement. Note that although this clause *defines* the value for the child node, it is *evaluated* on the parent node: hence the method call `getStmt(i-1)` is performed on the block, not its child statement.

In NameJava, not all statements are contained in a block. For instance, the body of an initialiser is a block, and hence a statement. Thus the first clause alone is not a sufficient definition, and we need the second clause to cover this case. The `inh` declaration preceding the two defining clauses exposes the attribute on statements, so that it can be called as a method on class `Stmt`. We furthermore declare the attribute to be `lazy`, instructing JastAdd to generate code for automatically caching computed attribute values.

The attribute `localVariable` is just a utility attribute that does not form part of the name lookup API shown in Figure 2.10. That API defines one interface `Declaration` as the common supertype of variable declarations and class declarations; note how JastAdd's aspect- oriented *declare parent* mechanism is used to make `VarDecl` and `ClassDecl` implement it *a posteriori*.

Furthermore, we have two attributes `lookupVariable` and `lookupClass` for looking up variables and classes by their simple name, respectively; these two attributes perform most of the heavy lifting of name resolution. One additional attribute `decl` determines the declaration an access binds to, taking into account

```
inh VarDecl Access.lookupVariable(String name);
inh VarDecl Block.lookupVariable(String name);

eq Block.getStmt(int i).lookupVariable(String name) {
  VarDecl v = getStmt(i).localVariable(name);
  if(v != null)
    return v;
  return lookupVariable(name);
}
```

Figure 2.11: Block-structured lookup

its syntactic position to determine whether the access refers to a class or a variable. For this reason, it returns a `Declaration`, which is either a `VarDecl` or a `ClassDecl`.

Finally, attribute `type` computes the type of an expression, returning a reference to the corresponding class declaration. The following subsections will show how these attributes are implemented for NameJava.

2.2.3 Block-structured lookup

Let us first tackle block-structured lookup. We have seen above how to look up a local declaration inside a given block, but now we want to implement more general lookup attributes that determine, for a given location in the program as represented by a syntax tree node, what class or variable a name refers to.

Inherited attributes are a perfect match for implementing block-structured lookup: for instance, when looking up a name starting from a statement, we need to know whether that statement occurs as a child of another block whose local variables we then have to search, or if it is a block that forms the body of an initialiser, so that we can proceed to examine the fields of the surrounding class.

The first defining clause for `lookupVariable`, shown in Figure 2.11, defines the attribute for a `Stmt` that occurs as the `i`th statement in a `Block`. It makes use of the previously defined attribute `localVariable` to search the statement itself and any previous statements in the same block for a declaration of the variable. If one is found, it is returned; otherwise the attribute is evaluated recursively.

Remember that this recursive invocation takes place on the `Block` node, *not* its `Stmt` child, so what may look like an infinite recursion is in fact a well-founded recursion towards the root of the AST. In order to evaluate `lookupVariable` on the `Block`, then, JastAdd has to find a defining equation in its parent node, which might be another block or an initialiser. In the former case, the same equation is evaluated again, with the erstwhile parent `Block` now taking the place of the `Stmt` child. We have not specified any equation for the latter case, i.e., a block occurring as the child of an initialiser; for such cases, JastAdd provides a default copy rule that keeps walking up the tree until a defining clause is found.

2.2.4 Variable lookup with inheritance

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

```
syn VarDecl BodyDecl.localField(String name) {
  return next() == null ? null : next().localField(name);
}
eq Field.localField(String name) {
  if(name.equals(getVarDecl().getName()))
    return getVarDecl();
  return super.localField(name);
}

inh lazy BodyDecl BodyDecl.next();
eq ClassDecl.getBodyDecl(int i).next() {
  return i < getNumBodyDecl()-1 ? getBodyDecl(i+1) : null;
}

syn VarDecl ClassDecl.localField(String name) {
  return getNumBodyDecl() == 0 ? null : getBodyDecl(0).localField(name);
}

syn VarDecl ClassDecl.memberField(String name) {
  VarDecl f = localField(name);
  if(f != null)
    return f;
  if(hasSuper())
    return getSuper().type().memberField(name);
  return null;
}

eq ClassDecl.getBodyDecl(int i).lookupVariable(String name) {
  VarDecl f = memberField(name);
  if(f != null)
    return f;
  return lookupVariable(name);
}

eq Program.getChild().lookupVariable(String name) = null;
```

Figure 2.12: Variable lookup with inheritance

Consider the implementation in Figure 2.12. Similar to the case of local variables we have an attribute `localField` on node type `BodyDecl` that looks up a field on a body declaration or any body declaration to the right of it, as determined by the inherited attribute `next`. For convenience, we define a corresponding wrapper attribute on `ClassDecl` that simply invokes local field lookup on the first body declaration. The synthesised attribute `memberFields` searches through both local declarations and declarations inherited from superclasses. We include superclasses transitively by recursively invoking `memberFields` on the type of the specified superclass name (the definition of the attribute `type` will be given below). In the last line, we provide a "catch all" definition of `lookupVariable`: if evaluation propagates all the way to the program root, it simply returns `null`.

Class lookup works in essentially the same way as variable lookup, but is a bit simpler. It employs synthesised attributes `localClass` and `memberClass` to look up member classes, `toplevelClass` for toplevel classes, and an inherited attribute `lookupClass` for block-scoped lookup. The complete definition is given in Appendix A.

Due to the use of `type` in the definition of `memberField` there is a dependency between name lookup

```
class A { }

class X {
 A a;
}

class Y {
 A b;
 class Z extends X {
   A c;
   Z z;
   {
     A a1 = z.c; // OK
     A a2 = z.a; // OK
     A a3 = z.b; // ERROR
   }
 }
}
```

Figure 2.13: Qualified field access

and class lookup, but the attribute evaluation scheme will schedule the computations automatically. More serious is the dependency of type analysis on itself, through a use of `type` in the definition of `memberClass`. If the class hierarchy is cyclic, this can lead to an infinite loop. We will get back to this issue below.

2.2.5 Qualified names

A `ClassDecl` plays a dual role from a name lookup perspective. As we have seen in Figure 2.12, when looking up a simple name from inside a class declaration, we first try to look it up as a, possibly inherited, member. If no such member is found, we proceed outwards, for example to look it up in an enclosing class. If, however, we want to access a field of a class under a qualifier, only members of the class itself should be considered, not members of enclosing classes. This is illustrated in Figure 2.13: while fields `c` and `a` can be accessed under the qualifier `z` since they are member fields of class `Z`, which is the type of the qualifier, we cannot access `b` in the same way, since it is declared in an enclosing class of `Z`, but is not one of its member fields.

With this in mind, it is easy to define the rules for qualified name lookup, shown in Figure 2.14: when looking up a variable from a qualified position, look it up as a member field of the type of its qualifier; when looking up a type from a qualified position, ensure that its qualifier resolves to a class, and then look it up as a member class of that class. The additional check for qualified type lookup is needed, since Java does not allow types to be accessed as members of arbitrary expressions.

2.2.6 Name disambiguation

The attribute `type` is now quite easy to define, as seen in Figure 2.15: the type of **`this`** is the enclosing class, determined by the inherited attribute `hostClass`, the type of a qualified **`this`** is the qualifying type, the type of a cast expression is the target type. For accesses, we make use of an auxiliary attribute `type` for declarations: we follow the JastAddJ convention of letting a class declaration be its own type, and for variables we simply need to determine the declared type.

```
eq QualAccess.getRight().lookupVariable(String name)
  = getLeft().qualifiedLookupVar(name);
eq QualAccess.getRight().lookupClass(String name)
  = getLeft().qualifiedLookupClass(name);

syn VarDecl Expr.qualifiedLookupVar(String name) = type().memberField(name);

syn ClassDecl Expr.qualifiedLookupClass(String name) = null;
eq Access.qualifiedLookupClass(String name) = decl().qualifiedLookupClass(name);

syn ClassDecl Declaration.qualifiedLookupClass(String name);
eq VarDecl.qualifiedLookupClass(String name) = null;
eq ClassDecl.qualifiedLookupClass(String name) = memberClass(name);
```

Figure 2.14: Qualified name lookup

```
syn ClassDecl Expr.type();
eq Access.type() = decl().type();
eq This.type() = hostClass();
eq QualThis.type() = getType().type();
eq CastExpr.type() = getType().type();

syn ClassDecl Declaration.type();
eq VarDecl.type() = getType().type();
eq ClassDecl.type() = this;

inh ClassDecl Expr.hostClass();
eq ClassDecl.getChild().hostClass() = this;
```

Figure 2.15: Typing in NameJava

The `decl` attribute is slightly more involved, since it needs to deal with ambiguous names. To resolve such names, the first step is to examine the name's syntactic position to determine whether it could refer to a type, to a variable, or to either. This information is represented by the enumeration type `NameKind`, as seen in Figure 2.16, with its three values `VAR`, `TYPE` and `EITHER`. The inherited attribute `nameKind` computes the name kind of a node; for instance, the access appearing in the **`extends`** clause of a class must clearly refer to a type, whereas an access used as a field initialiser must refer to a variable. Some similarly simple rules have been omitted; they can be found in Appendix A.

The most complicated case arises for accesses to the left of a dot: if the name it is qualifying has name kind `TYPE`, then the access itself also has name kind `TYPE`, since in NameJava any access that qualifies a type access must itself be a type access; in any other case, it could either be a type or a variable. This analysis is performed by the method `qualKind` of type `NameKind`.

Once we know the name kind of an access, it is easy to compute its declaration: if it could refer to a variable, try looking it up as one; if that fails and the access could potentially refer to a type, look it up as a type instead. Otherwise, return **`null`** to indicate that the access is unbound. Since qualified lookup is already handled by the `lookup` attributes, the declaration a qualified access binds to is simply the declaration its right hand child binds to.

We have now introduced most of the techniques used in JastAddJ to implement name lookup for the full Java language, with two exceptions. As mentioned above, there is a circularity in the definition of

```
enum NameKind {
  VAR    { NameKind qualKind() { return EITHER; } },
  TYPE   { NameKind qualKind() { return TYPE; } },
  EITHER { NameKind qualKind() { return EITHER; } };

  abstract NameKind qualKind();
}

inh NameKind SimpleAccess.nameKind();
eq QualAccess.getLeft().nameKind() = getRight().nameKind().qualKind();
eq ClassDecl.getSuper().nameKind() = NameKind.TYPE;
eq Field.getInit().nameKind() = NameKind.VAR;
// some rules omitted

syn Declaration Access.decl();
eq SimpleAccess.decl() {
  if(nameKind() == NameKind.VAR || nameKind() == NameKind.EITHER) {
    VarDecl vd = lookupVariable(getName());
    if(vd != null)
      return vd;
  }
  if(nameKind() == NameKind.TYPE || nameKind() == NameKind.EITHER) {
    return lookupClass(getName());
  }
  return null;
}
eq QualAccess.decl() = getRight().decl();
```

Figure 2.16: Name disambiguation and lookup

type lookup that may lead to non-termination when looking up members of classes with a circular class hierarchy. For this reason, Java forbids circular class hierarchies, and JastAddJ checks to make sure that no such circularity arises. However, in order to determine whether a class hierarchy is cyclic, we first need to resolve supertypes, which already could lead to non-termination. For this reason, `lookupClass` cannot assume that it is operating on acyclic hierarchies only.

In JastAddJ, this is solved by using *circular attributes*. Such attributes, declared with the keyword **`circular`**, are computed by fixpoint iteration, starting from a given initial value. This prevents non-termination and can be used to detect type hierarchy cycles. For the purposes of refactoring, however, this is a minor issue, since we assume that the input program has passed all semantic checks, and in particular has an acyclic type hierarchy.

Another feature of JastAdd that is used in JastAddJ for name analysis are *rewrite rules*. These rules specify conditions under which AST nodes should be replaced, with the new node computed by a snippet of Java code. Rewrite rules are applied implicitly whenever a node is accessed, and they are applied exhaustively until all conditions of all applicable rules are false.

Rewrite rules allow a rather elegant solution to the problem of ambiguous names, in which nodes representing names are rewritten in several stages until it is clear whether they bind to packages, types or variables. While this solution beautifully parallels the informal description of name disambiguation in the Java language specification, it is very hard to reason about due to the implicit way in which rewrite rules are applied. Instead, one can use a more laborious formulation as in Figure 2.16, where the resolving of ambiguities is made part of name lookup.

The full Java language provides a number of naming-related features that are not found in NameJava, but can be handled by an essentially straightforward application of the techniques used in this section. Packages introduce a new source of ambiguity, necessitating the introduction of additional name kinds. Together with access modifiers, they also introduce additional constraints on the accessibility of variables and types, which can be dealt with by augmenting the lookup rules with filters that discard inaccessible declarations. For instance, the attribute `memberField` on `ClassDecl` defined in Figure 2.12 should, after looking up a member field in the superclass, check that the field, if found, is not private before returning it.

2.3 Inverting lookup and constructing accesses

We will now show how the implementation of name lookup just discussed can be turned around to yield an implementation of access construction which is right-inverse to it. Essentially, we invert every lookup equation in isolation, and again rely on the attribute evaluation mechanism to piece these equations together in the same way as for lookup.

Note that there are many different versions of access construction satisfying Equation 2.1. In particular, a partial function that is not defined anywhere would fit the bill. In Subsection 2.3.1 we first show an implementation that tries just a little harder: it returns a simple name for the declaration if it can be directly accessed, and **`null`** if it cannot. The correctness criterion for the attribute `accessVariable` then simply states that for every AST node `p` and every variable `v`, if `p.accessVariable(v)` is not **`null`**, then

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

We will show concretely how to systematically derive the definition of `accessVariable` for NameJava from the definition of `lookupVariable` presented in the last section, and briefly discuss how this extends to full Java.

Of course, this version of `accessVariable` is overly simplistic. We would not even have to go through all the trouble of inverting lookup equations: instead, we could simply use lookup directly to determine whether the variable is visible. But we can without too much effort extend this version to produce qualified names, as we show in Subsection 2.3.2. Now instead of just returning a name, attribute `accessVariable` returns a potentially qualified `Access`.

This makes the formulation of our correctness criterion more complicated, however, since there is no single `lookup` function for resolving qualified accesses. This is no accident, as it is far from clear what it should mean to look up a qualified name at an arbitrary node, especially if that node is itself qualified.

As a brief illustration, consider the program in Figure 2.17. Assume we want to build an access to field `A.f` from the position marked with a bullet. This position is qualified with expression `b`, so if we were to look up a name from this position, it would be resolved as a member field of `B`. Analogously, we could try to construct an access to `A.f` as a member field of `B`. Since this field is shadowed by `B.f`, we might construct the qualified access `((A)this).f`. Clearly, however, it makes no sense to insert this access into the marked position: the result would not be syntactically correct.

Instead, we should merge the qualifier `b` into the qualified access `((A)this).f` yielding `((A)b).f`, and replace the whole qualified access with this new access. Assuming that this merging is performed by a method `splice`, we now propose the following correctness criterion: if `p.accessVariable(v)` does not return **`null`**, then

```
p.splice(p.accessVariable(v)).decl() == v
```

```
class A {
  int f;
}

class B extends A {
  int f;
}

class C {
  int m(B b) {
    return b.•;
  }
}
```

Figure 2.17: The need for merging accesses

```
syn String Stmt.accessLocal(VarDecl v) {
  return prev() == null ? null : prev().accessLocal(v);
}
eq LocalVar.accessLocal(VarDecl v) {
  return getVarDecl() == v ? v.getName() : super.accessLocal(v);
}

syn String BodyDecl.accessLocalField(VarDecl v) {
  return next() == null ? null : next().accessLocalField(v);
}
eq Field.accessLocalField(VarDecl v) {
  return getVarDecl() == v ? v.getName() : super.accessLocal(v);
}
```

Figure 2.18: Inverting local lookup in blocks and classes

Together, the functions `accessVariable` and `splice` achieve what we want, namely to construct an access that binds to a given declaration. As for the simple version of `accessVariable`, we will show how to implement this more complicated version and the `splice` method for NameJava, and briefly discuss its extension to full Java.

Since access construction for classes is very similar to access construction for variables, we do not explain it in any detail, but its implementation for NameJava can be found in Appendix A. Access construction for methods has to deal with method overloading, which we will briefly discuss below.

2.3.1 Non-intrusive access construction

The simple access computation we introduce at first will only tell us whether a variable can be accessed through its simple name. Given a variable `v`, it will thus either return the variable's name if it is directly visible, or `null` otherwise.

As in our discussion of name lookup, we start by considering lookup of local variables in blocks, and, since the code is very similar, lookup of local fields in classes. Both equations are easy to invert: instead of traversing the block or class scanning for a variable of the name to look up, we traverse it scanning for the variable to be accessed, as shown in Figure 2.18. Since in a valid Java program there can be at most one local variable or field of a given name, this really does give us correct inverses.

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

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

Figure 2.19: A failed attempt at inverting a block-structured lookup rule

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

Figure 2.20: A right inverse of a block-structured lookup rule

Next up are the rules for block-structured lookup. Recall that these rules invariably try to first look up a name in the current scope, and then dispatch to the enclosing scope if it cannot be found. It would then be obvious to do the same for access construction, leading to code as in Figure 2.19 on the left.

But this is not correct: consider the code snippet in the same figure on the right, and assume we want to construct an access to the field `x` in `A` from the position marked •. Since that position is the second statement within a block, the equation will be evaluated with parameter `i` set to `1` (indexing is zero-based). The auxiliary method `accessLocal` will not find the declaration we are looking for: although there is a declaration for a local variable `x`, this declaration is not equal to the declaration we are trying to access.

Hence we will invoke `accessVariable` again, this time on the parent node. Eventually it will return the access `x`, which does in fact refer to the field if seen *outside* the body of method `m`. Inside that body, however, it does not, but will instead refer to the like-named local variable of `m`.

Generally speaking, we cannot always directly return the access computed recursively higher up in the syntax tree, but we may need to adjust it to make sure it is still valid. As we are not attempting to construct qualified accesses just yet, all we do is to check for shadowing, and fail (i.e., return `null`) if that occurs. The resulting code is in Figure 2.20.

It is now easy to see that this equation is indeed right-inverse to the corresponding equation in Figure 2.11, assuming that `accessLocal` is right-inverse to `localVariable`, which we have established above, and that `accessVariable` on the parent node is right-inverse to `lookupVariable` on the parent node, which acts as an induction hypothesis.[1]

The remaining equations can all be inverted in the same fashion.

[1] This intuitive argument will be made precise in Chapter 6.

```
interface Scope { }
ClassDecl implements Scope;
Block implements Scope;

syn boolean Scope.declaresVariable(String name);
eq ClassDecl.declaresVariable(String name)
  = memberField(name) != null;
eq Block.declaresVariable(String name)
  = getNumStmt() > 0 && getStmt(getNumStmt()-1).localVariable(name) != null;

interface SymbolicVarAccess {
  SymbolicVarAccess moveInto(Scope s);
  SymbolicVarAccess moveDownTo(ClassDecl c);
}
```

Figure 2.21: Symbolic accesses and scopes

2.3.2 Adding qualifiers

We now aim to extend the above approach to not just give up when an impending name capture is detected, but to add qualifiers to evade it. The crucial problem to solve is when to add a qualifier, and what qualifier to add. Note that at the point where we actually find the declaration an access is to be constructed for, the declaration is always directly visible without any need for qualification; it is rather at some later point, when the constructed access is moved into another scope, that we may need to add a qualifier.

Given the bewildering array of qualifications possible in Java, it is more convenient to work with *symbolic accesses* during the access construction, which carry enough information to build a concrete access, but are easier to manipulate. In particular, we often need to move a symbolic access from an outer scope to an inner scope and from a parent class to a child class. These two operations form the interface `SymbolicVarAccess` in Figure 2.21. We abstract all different kinds of scopes into the common interface `Scope`: in our small language, only classes and blocks introduce scopes; in full Java, so do methods, constructors, interfaces, **`catch`** clauses, and **`for`** loops. The most important thing we need to know about a scope is whether it declares a variable with a given name; this check is easy to implement for both classes and blocks.

Accesses to fields have to be handled differently from accesses to local variables, so the two are represented by different classes implementing interface `SymbolicVarAccess`. The case of local variables is particularly simple, since an access to a local variable can never be qualified, so we can in fact simply let `LocalVar` itself implement `SymbolicVarAccess`, as shown in Figure 2.22: whenever we try to move a symbolic access to a local variable into a scope where it would be captured by another declaration, we return **`null`**, otherwise we leave it untouched. We never need to move a symbolic access to a local variable from a superclass to a subclass, so we have the corresponding equation simply return **`null`** as well.

Symbolic accesses to fields are more complicated. The key insight for determining what information we need is that when we look up a field in Java, we always find it in an ancestor class `A` of a class `B` lexically surrounding the point of lookup. We call the class `A` the *source* and the class `B` the *bend* of the field lookup.

To illustrate all possible cases, Table 2.1 indicates for every field in the program in Figure 2.23 what its source and bend are when looked up from the position •. The table also gives a *safely qualified* access for each field, i.e., an access that would refer to the field even if it were shadowed or hidden in some way.

The table suggests, for example, that a field can always be safely accessed by qualifying with **`this`** if its source and bend class are both equal to the host class lexically surrounding the point of lookup. Similar

```
LocalVar implements SymbolicVarAccess;

public SymbolicVarAccess LocalVar.moveInto(Scope s) {
  return s.declaresVariable(getName()) ? null : this;
}
public SymbolicVarAccess LocalVar.moveDownTo(ClassDecl c) {
  return null;
}
```

Figure 2.22: Implementing symbolic accesses

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

Figure 2.23: Qualified accesses in Java

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

Table 2.1: Safely qualified accesses

```
class SymbolicFieldAccess implements SymbolicVarAccess {
  private VarDecl target;
  private ClassDecl source;
  private ClassDecl bend;
  private boolean needsQualifier;

  // constructor omitted

  public SymbolicVarAccess moveInto(Scope s) {
    if(s.declaresVariable(target.getName()))
      needsQualifier = true;
    return this;
  }

  public SymbolicVarAccess moveDownTo(ClassDecl c) {
    if(c.localField(target.getName())!=null)
      needsQualifier = true;
    bend = c;
    return this;
  }
}
```

Figure 2.24: Symbolic field accesses

```
eq ClassDecl.getBodyDecl(int i).accessVariable(VarDecl v) {
  SymbolicVarAccess acc = accessMember(v);
  if(acc != null) return acc;
  acc = accessVariable(v);
  return acc == null ? null : acc.moveInto(this);
}

syn SymbolicVarAccess ClassDecl.accessMember(VarDecl v) {
  SymbolicVarAccess acc = accessLocalField(v);
  if(acc != null) return acc;
  if(hasSuper()) {
    acc = getSuper().type().accessMember(v);
    return acc == null ? null : acc.moveDownTo(this);
  }
  return null;
}
```

Figure 2.25: Access construction with qualification

rules exist for the other qualifications, so in order to determine how a field can be accessed it is enough to compute its source and bend, and to keep track of the host class and the qualifier under which the field should be accessed, if any. These considerations lead us to define class `SymbolicFieldAccess` as shown in Figure 2.24, where we omit the constructor, which is standard. The other methods perform straightforward bookkeeping.

With these definitions in place, it is now easy to give inverses of `memberField` and `lookupVariable` when invoked on a body declaration inside a class, as shown in Figure 2.25. In the first case, after having computed an access to variable `v` from outside the class (for example, from its enclosing class), we need to move this access into the class; in the second case, an access constructed from the super class needs to be moved down to the child class to account for possible shadowing.

Once we have constructed a symbolic access to a field or variable, we need to splice it into a given position in the syntax tree, possibly merging it with a qualifier. This is handled by the method `splice` shown in Figure 2.26, omitting some error handling code that is included in the full version found in Appendix A. This method in turn dispatches to two methods both named `toAccess`, that build an actual access out of a symbolic one, either with or without an additional qualifying expression. In the latter case, the host class, i.e., the type of **`this`**, needs to be passed as an additional argument in order to be able to build qualified field accesses.

To determine whether the position into which the access should be spliced is qualified, we use the auxiliary method `getQualifyingDot`: for the right-hand child of a qualified access, this method returns the qualified access itself; for all other accesses it returns **`null`**. The method `replaceWith` performs node replacement, returning the new node.

If no qualifying expression is given, we pass the host class as argument to the access construction. This argument is ignored when constructing accesses to local variables: such accesses are always just simple names. For fields, we can also just use a simple name if the field is directly visible, i.e., if the `needsQualifier` field of its symbolic access is false. Otherwise, there are three cases: if the target field `f` is a visible member of the host class, we construct an access of the form **`this`**`.f`; if it is declared in an ancestor `S` of the host class, but shadowed by another field, we construct an access of the form `((S)`**`this`**`).f`; finally, if it is declared in an ancestor `S` of a class `B` surrounding the host class, we construct an access `((S)B.`**`this`**`).f`. The attribute `accessClass`, discussed below, constructs accesses to the requisite classes.

Constructing an access involving a qualifying expression is very similar: since local variable accesses cannot be qualified, their `toAccess` attribute evaluates to **`null`** in this case. For fields, we either directly qualify them with the given expression if the field is visible, or cast it to the appropriate ancestor class as described above.

Correctness of this enhanced implementation is no longer as easy to see as for the simple implementation; we will discuss a formal correctness proof in Chapter 6.

Access construction for classes works in essentially the same way: again, we construct symbolic accesses by inverting lookup rules, adding qualifiers where necessary. One additional finesse arises from name ambiguity: when constructing a concrete access to a class from a symbolic one, we need to ensure that there is no variable in scope that could obscure it. Details of the implementation can again be found in Appendix A.

For methods, we need to account for overloading resolution. A simple way of doing this is to first construct a possibly qualified access to the desired method in a similar way as for variables, and then use `lookupMethod` to check whether the constructed method access resolves to the desired declaration. If it does not, we insert casts on all the arguments (using `accessClass` to construct accesses to the requisite classes), casting them to the parameter types of the method we want to bind to.

2.4 Locked names

The previous section has discussed how to implement access construction attributes that, for any position p in the syntax tree and any declaration d of a type, variable, or method, produce, if possible, a symbolic access n, which can be spliced into the syntax tree at position p to yield an actual access that binds to d.

Sometimes, however, we want to create an access without knowing where it should be spliced into the AST. For example, in the `toAccess` methods of Figure 2.26 we are constructing a qualified access, as part of

```
public Access Access.splice(SymbolicVarAccess sacc) {
  QualAccess qacc = getQualifyingDot();
  if(qacc == null)
    return replaceWith(sacc.toAccess(hostClass()));
  else
    return qacc.replaceWith(sacc.toAccess(qacc.getLeft()));
}

inh QualAccess Access.getQualifyingDot();
eq ClassDecl.getChild().getQualifyingDot() = null;
eq QualAccess.getRight().getQualifyingDot() = this;

syn Access SymbolicVarAccess.toAccess(ClassDecl host);
eq LocalVar.toAccess(ClassDecl host) = new SimpleAccess(getName());
eq SymbolicFieldAccess.toAccess(ClassDecl host) {
  Access acc = new SimpleAccess(target.getName());
  if(needsQualifier) {
    if(source == bend && source == host)    // this.f
      return new QualAccess(new This(), acc);
    else if(bend == host)                   // ((S)this).f
      return new QualAccess(new Cast(accessClass(source), new This()), acc);
    else                                    // ((S)B.this).f
      return new QualAccess(new Cast(accessClass(source),
                            new QualThis(accessClass(bend))), acc);
  } else {
    return acc;
  }
}

syn Access SymbolicVarAccess.toAccess(Expr qual) = null;
eq SymbolicFieldAccess.toAccess(Expr qual) {
  Access acc = new SimpleAccess(target.getName());
  if(needsQualifier) {
    if(bend != qual.type())
      return null;
    return new QualAccess(new Cast(accessClass(source), qual), acc);
  } else {
    return new QualAccess(qual, acc);
  }
}
```

Figure 2.26: Splicing symbolic accesses into the AST

```
ast LockedClassAccess : SimpleAccess ::= <Target:ClassDecl>;

eq LockedClassAccess.decl() = getTarget();

public void SimpleAccess.lock() {
  Declaration decl = decl();
  if(decl instanceof ClassDecl)
    replaceWith(new LockedClassAccess((ClassDecl)decl));
  else
    ...
}

public void LockedClassAccess.unlock() throws RefactoringException {
  SymbolicClassAccess sacc = accessClass(getTarget());
  if(sacc == null || splice(sacc) == null)
    throw new RefactoringException("cannot␣unlock");
}

private LockedClassAccess SymbolicFieldAccess.accessClass(ClassDecl target) {
  return new LockedClassAccess(target);
}
```

Figure 2.27: Locked class accesses

which we need to construct a class access. At the point where this access is constructed the node into which it should be spliced does not exist yet.

An elegant solution is to extend our abstract grammar with new node types representing *locked names*. Such locked names refer to a fixed variable, type, package or method, and bypass the normal lookup mechanism to directly bind to their target declaration. When we need to construct an access to a specific declaration, we then simply construct a locked access. When we need to ensure that a name stays bound to its declaration, we lock it, i.e., replace it by a locked name with that declaration as its target.

Of course, locked names should not appear in the output program after the refactoring is finished: they should be unlocked, replacing them by (potentially qualified) names. This unlocking is easy to implement using the access construction algorithm presented above, and splicing the resulting symbolic access into the AST at the position where the locked access was located. Unlocking a name may not be possible, in which case the operation has to be aborted and any changes to the AST have to be undone.

Implementing locking and unlocking of type and variable accesses is quite straightforward: Figure 2.27 shows the declaration of the node type of locked class accesses, its definition of attribute `decl` that overrides normal lookup rules, and the methods `lock` and `unlock` to convert to and from the locked form. The latter method may throw a `RefactoringException` if unlocking cannot be performed.

By no means do we suggest that locked names should be added to the Java language as a feature for the programmer to use. They solely exist to facilitate the implementation of refactorings, and they will always be eliminated before the end of the refactoring operation. Locked names are thus an example of a lightweight language extension, which will be the subject of Chapter 4.

2.5 Specifying RENAME

Given the framework of access construction and locked names introduced so far, it is easy to implement the RENAME refactorings. As two representative examples, this section will discuss the implementation of RENAME VARIABLE and RENAME METHOD; the others are similar.

For RENAME VARIABLE, we are given a variable *v* (which may be a field, a local variable, or a parameter) and a new name *n* that it should be renamed to. The refactoring proceeds in four steps:

1. Ensure that the renaming is possible syntactically: for fields, the host class cannot already contain a field of name *n*; for local variables or parameters, the scope of the renamed variable has to be disjoint from the scope of any other local variable or parameter named *n*.
2. Identify all declarations that are endangered by the renaming and lock all names binding to them.
3. Set the name of *v* to *n*.
4. Unlock all locked names in the program.

The first step checks some preconditions to ensure that the refactoring makes sense at all: if one of these conditions is violated, the refactoring would not be able to produce a well-formed Java program. Notice, however, that these preconditions alone do *not* guarantee that the refactoring will be successful: the last step may still fail to unlock a name that cannot be accessed even with qualifiers.

The most interesting aspect from an implementation point of view is how to determine all names that might be endangered, i.e., that might be accidentally captured due to the renaming. The easiest choice is to simply consider all names that are either the same as the old name of *v* or are the same as *n* to be endangered. Note that this does not only include variable names, but also type names and package names that occur in an ambiguous syntactic position.

Obviously, this is safe since no other name's binding could change, but it will in general be an overapproximation in the sense that some of those names could, in fact, not possibly change their binding, so locking and unlocking them is unnecessary. It is certainly possible to improve this approximation, but as we shall see in Section 2.7 the rough approximation works well in practice and is not a serious performance bottleneck.

No matter how we compute the set of endangered names, as long as it is a conservative approximation the above algorithm will ensure that all names refer to the same declaration before and after the refactoring, and that nothing in the program changes except for the names. For full Java, this does not mean that behaviour is preserved: programs could use Java's built-in reflection mechanism to indirectly access fields, and there is in general no way to reliably detect and adjust such accesses. We will get back to this issue in Chapter 7.

Let us now consider RENAME METHOD, say, the problem of renaming a method declaration *m* to *n*. When renaming methods, we have to be a little more careful than with variables. Although we again want to preserve the bindings between method calls and their target methods as for RENAME VARIABLE, this is not enough.

First, we should not only rename *m* itself, but also any method that it overrides or that is overridden by it, and any method that is overridden by/overrides any of *those* methods, and so on. Otherwise the overriding relationship between methods would be lost, which, in the benign case, could lead to an uncompilable output program, and in the worst case could subtly change the dynamic dispatch behaviour of method calls.

By the same token, we also have to be careful not to pick up any additional overriding relationships. Consider the program in Figure 2.28 on the left; we have a class `B` extending another class `A`, where `B`

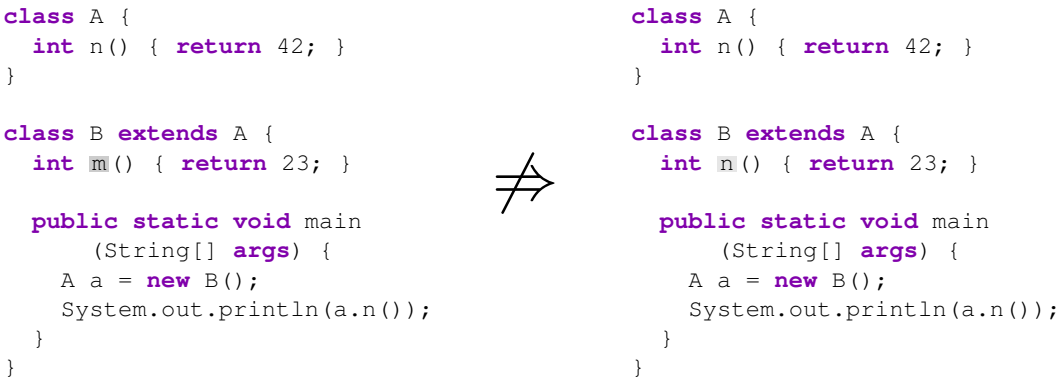


Figure 2.28: Change of overriding due to renaming of methods

declares a method `m` that returns the value `23`, and `A` a method `n` that returns the value `42`. The main program constructs an object of type `B`, stores it in a variable of type `A`, and prints the value of invoking method `n` on it, namely `42`.

If we now rename method `m` to `n`, we do not want the refactoring engine to produce the program on the right. Although method bindings have not changed (the invocation of `n` still binds to method `n` in type `A`), the call is now dynamically dispatched to the renamed method, so the program prints `23`.

One might argue that this could be accommodated by expanding the definition of binding preservation for method calls: not only should every method call still bind to the same static invocation target, but indeed the whole set of methods that it could possible dispatch to at runtime should not change. This, however, is not correct. Recall that in a language such as Java it is undecidable which methods a call could dispatch to, so we would have to make do with a conservative over-approximation. But then the fact that we have preserved the approximate set of invocation targets does not guarantee that runtime dispatch behaviour does not change, since an overly coarse approximation could hide a change in dynamic behaviour.

Instead we require that the RENAME METHOD refactoring, besides preserving the binding structure, also preserves the overriding relationship.[2] We capture this by *overriding dependencies*, which indicate that one method must override another. Like the locked names, these dependencies are computed as the first step of the renaming. When we unlock names in the final step, we also check at the same time that overriding dependencies and actual overriding relationships match, i.e., that methods have neither picked up nor lost any overriding relationships. If we detect a violation, the refactoring is aborted with an explanatory error message for the user.

Thus, to rename a method m to n, we proceed as follows:

1. Identify the set of endangered methods, i.e., methods whose overriding behaviour may be changed, or where a call of the method could change its binding. For every endangered method, lock all accesses to it, and compute the set of methods it overrides.

2. Consider every method m' such that m and m' have a common ancestor method m'' which they both reflexive-transitively override.

[2] Obviously, none of the other RENAME refactorings could change overriding, so we do not need to take any precautions there.

(a) Ensure that renaming m' to n is possible at all: the host type T should not already contain a method with the same signature; if there is a method of name n such that m' would hide it or be hidden by it after the renaming, make sure that this is possible.

(b) Change the name of m' to n.

3. Unlock all locked method accesses; ensure preservation of all overriding dependencies.

We face the same choices as before on how to compute endangered methods. The syntactic checks are a bit more complicated this time, but again simply serve to ensure that the output program still compiles without trying to guarantee that the refactoring will succeed as a whole.

There is a certain similarity between overriding dependencies and locked names; indeed, locked names can be seen as *name binding dependencies* that state that some name should bind to a given declaration. We will see in the next chapter that many other aspects of the dynamic behaviour of programs can similarly be captured by static constraints that the refactoring implementation must aim to preserve. Name binding dependencies are special, however, in the sense that we can actually fix a broken dependency by constructing an appropriate access. This is not easily possible for overriding dependencies.

2.6 Extensibility and reusability

One of the main advantages of modelling access construction closely after the implementation of name lookup is extensibility: to support new language features, all we need to know is how they affect lookup. Given an implementation of lookup for the new feature, we can systematically invert every equation to extend our implementation of access construction, and hence of renaming.

2.6.1 Extending access construction to inter-type declarations

As an example to show this extensibility, let us consider the problem of providing support for inter-type declarations. Inter-type declarations [4, Chapter 2] are a powerful concept to separate concerns that cross-cut the static class hierarchy by providing support for adding new members to already declared classes. Such declarations are extensively used in JastAdd, and have figured prominently in our examples, but now we want to examine how to provide support for them in our naming framework.

More specifically, we will show how to handle inter-type method declarations. In contrast to a regular method, which is declared in an interface or a class, an inter-type method is declared in an *aspect*, which syntactically looks like a new variant of a type declaration. The aspect in which it is declared is the inter-type method's *host aspect*, while the type on which it is declared is its *introduced type*. The inter-type method behaves like a member method of its introduced type, except that it can also access members of its host aspect.

Take for example the program in Figure 2.29. It contains both an aspect `X` and a class `B` with the aspect declaring an inter-type method `m` on `B`, i.e., the host aspect of `m` is `X`, while `B` is its introduced type. The method `m` can access both members of `B`, like `y` in the example, and members of `X`, like `x`.

The implementation of inter-type declarations introduces new node types to represent aspects, inter-type methods, and inter-type fields. In particular, an inter-type method is represented by this node type:

```
IntertypeMethodDecl : BodyDecl ::= Target:TypeAccess Method:MethodDecl;
```

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

class B {
  int y;
}
```

Figure 2.29: A simple program using inter-type declarations

```
eq IntertypeMethodDecl.getMethod().lookupVariable(String name) {
  VarDecl v = getTarget().type().memberFields(name);
  if(v != null) return v;
  return lookupVariable(name);
}
```

Figure 2.30: Variable lookup on inter-type methods

```
eq IntertypeMethodDecl.getMethod().accessVariable(VarDecl decl) {
  SymbolicVarAccess acc = getTarget().type().accessMemberField(decl);
  if(acc != null) return acc;
  acc = accessVariable(decl);
  if(acc != null)
    return acc.moveInto(this);
  return null;
}
```

Figure 2.31: Access computation on inter-type methods

Variable lookup for these methods is then easily implemented as shown in Figure 2.30: inside the `Method` child, lookup will proceed as before. The `IntertypeMethodDecl` interrupts lexically scoped lookup to check whether the variable can be resolved as a member field of the target type; if not, we dispatch to the parent for further lookup.

This lookup rule has the same basic shape as other rules for block structured lookup, so we can invert it in the same way, obtaining the code in Figure 2.31. We assume that `IntertypeMethodDecl` implements interface `Scope` and provides an appropriate definition of `declaresVariable`.

Other lookup rules for methods and for fields follow the same schema, and are likewise easily inverted.

2.6.2 Reusing the naming framework

As we shall see in the sequel, the problem of constructing names that bind to an intended declaration and of maintaining existing name bindings is pervasive, occurring in almost every refactoring. Our naming framework provides a convenient tool for addressing these problems: to introduce a name that should bind to a particular declaration, create a locked name with that declaration as its target, the generic unlocking mechanism will then take care of replacing that locked name with a normal name; to preserve the binding of an

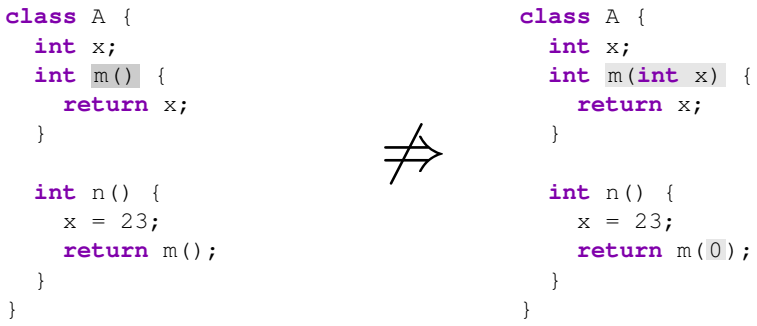


Figure 2.32: Naming problem with INTRODUCE PARAMETER

existing name, lock it onto its current declaration, again the unlocking mechanism will ensure that the binding is maintained, inserting a qualifier where necessary.

The same level of genericity is hard to achieve with precondition-based refactorings, and indeed the industrial-strength refactoring engines we are aware of perform rather poorly in this respect. Recall, for example, from the introduction that Eclipse does not permit the renaming of a parameter if the parameter could capture a field reference. This precaution is, however, not taken in the INTRODUCE PARAMETER refactoring. The purpose of that refactoring is to extend a method with a new parameter of a given type and name; all calls of the method are updated to pass some default value as argument. For instance, assume that in the program in Figure 2.32 on the left we want to give `m` a new parameter `x` of type `int`. Eclipse's implementation of INTRODUCE PARAMETER produces the program on the right, which is wrong, since the access to `x` inside the body of `m` has been captured by the newly introduced parameter. Whereas in the original program method `n` would return `23`, it will now return `0`. Arguably, this is exactly the same problem as with renaming an existing parameter so that it captures variable accesses within the method or constructor body.

Using our naming framework, both situations are handled in the same way: we lock endangered names inside the method, and the unlocking mechanism ensures that they bind to the same declaration after the refactoring as before. This commonality is hard to identify and easy to forget about in a purely precondition-based approach.

2.7 Evaluation

We will now evaluate our implementation of the RENAME refactorings in terms of correctness, code size, and performance.

2.7.1 Test suite

The source distribution of Eclipse comes with an extensive test suite, which contains many hundreds of test cases for the different refactorings Eclipse implements. We adapted these test cases to run through our implementation instead, so that we could gain some confidence in the correctness of our implementation. The results of evaluating our implementation of the RENAME refactorings in this way are shown in Table 2.2.

The table lists the four considered refactorings in the first column. For every refactoring, the column labelled "Total" indicates how many test cases are provided by Eclipse. The remaining four columns classify these test cases into four disjoint categories.

Refactoring	Total	Inapplicable	Missing Feature	Rejected by Eclipse	Same Result
RENAME PACKAGE	35	17	4	0	14
RENAME TYPE	247	16	35	68	128
RENAME METHOD	235	26	18	15	176
RENAME VARIABLE	215	30	18	33	134

Table 2.2: Evaluation of RENAME implementations on Eclipse's test suite

Category "Inapplicable" comprises those test cases that we could not run through our implementation, most of them because the input program does not compile: remember that our implementation can only refactor correct Java programs, whereas Eclipse's refactoring engine also handles incorrect programs. Some test cases that exercise details of Eclipse's precondition checking algorithm are also included in this category.

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.

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

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

In summary, our implementation does quite well: while we do not implement all the additional features that Eclipse provides, our refactoring engine is very reliable, and can refactor many programs on which the Eclipse implementation has to give up.

Along with our implementation, we have also developed our own test suite with some 200 renaming test cases. These tests systematically explore both common and exotic cases for all the RENAME refactorings, and have been very helpful not only for validating our own implementation, but also for finding bugs in other IDEs. A commented list of examples that we found to be refactored incorrectly by recent versions of popular refactoring engines can be found online [118].

2.7.2 Code size

As a very coarse measure of the complexity of our implementation, we counted the number of lines of source code. The naming framework (i.e., access construction and locked names) comprises around 1400 lines of code, not including utility code, whereas the four RENAME refactorings together take less than 250 lines of code to implement.[3] In absolute terms, these numbers show that our implementation is quite compact, even though it covers all of Java 5.

To provide a rough point of comparison, the combined size of the main implementation files of Eclipse's RENAME refactorings is a bit more than 5000 lines of code.[4] While such a comparison can, of course, never be fair, the numbers at least strongly suggest that implementing our refactorings in JastAdd and reusing the existing infrastructure of JastAddJ gives us quite a lot of leverage.

[3] These measurements, as well as all other code size measurements for Java in this thesis, were generated using David A. Wheeler's 'SLOCCount' [139].

[4] More precisely, this is the combined size of the `Rename*Processor.java` files in package `org.eclipse.jdt.internal.corext.refactoring.rename` of the Eclipse 3.5 source distribution.

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

Table 2.3: Performance of RENAME implementation on Jigsaw; EA stands for "endangered accesses"

2.7.3 Performance

Another important question to evaluate is if our implementation is practical on large input programs. To address this, we performed a number of renamings on the source code of release 2.2.5 of the Jigsaw web-server [138], which comprises about 100,000 lines of Java 1.4 code.

The results of our experiments, as measured on an AMD Athlon 64 X2 machine running Linux 2.6.22, are put together in Table 2.3. We will briefly explain the data relating to type renaming. Interesting types to rename for evaluation purposes are on the one hand those which are referenced a lot, and on the other hand those which are rarely referenced. Hence we chose three of the most heavily used types (the classes `Attribute` and `Request` as well as the interface `HTTP`) and some rarely used classes to rename; the number of types referencing each of them is given in the first column.

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

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

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

The situation for field and local variable renaming is not much different. For the latter, we have chosen a variable with perhaps the most common name of all. Here, the discrepancy between actual references

and endangered accesses as determined by our algorithm is very marked, but still not big enough to cause a substantial decrease in performance.

In conclusion, the numbers show that our implementation of the RENAME refactorings compares favourably to industrial-strength refactoring engines in terms of performance, while often outstripping them in terms of correctness.

2.8 Related work

This chapter is based on the author's paper "Sound and Extensible Renaming for Java" [116] published at OOPSLA 2008, which is joint work with Torbjörn Ekman and Oege de Moor. All key ideas and concepts are the author's, as is the implementation; both have benefited greatly from discussions with the co-authors.

The problem of avoiding name capture arises whenever syntactic manipulations of languages that include binding constructs are made precise. Hilbert and Ackermann famously neglected to state a side condition concerning bound occurrences of substituted variables when defining the substitution rule for first-order logic [51], which was only pointed out many years later by Church [26]. In recent years, there has been a resurgence of interest in the precise treatment of languages with binding [107, 108], mostly with the goal of handling binding issues once and for all in a modular fashion so that it does not impede further meta-theoretic study of the object language. In approach and scope, this area of research is only tangentially related to renaming as we understand it here.

In general, the refactoring literature has not paid a lot of attention to the RENAME refactorings, which is perhaps surprising given their popularity among programmers. Opdyke [100] considers RENAME CLASS, RENAME VARIABLE and RENAME METHOD, but his object language (a subset of C++) is quite simple from a naming perspective, and does not present many of the intricacies that make renaming for Java so difficult. In particular, classes cannot be nested, and names are never ambiguous. His specification of RENAME VARIABLE does not attempt to resolve name conflicts by adding qualification, instead forbidding this situation to arise through the use of preconditions. On the other hand, his version of RENAME METHOD is more flexible than ours in that he allows additional overriding relations to occur due to the refactoring, if it can be proved that the dynamic target of method invocations does not change.

The same three refactorings are again considered by Roberts [111], this time in the context of Smalltalk. He also uses a precondition-based approach with very similar conditions as those posed by Opdyke, but he additionally provides postconditions that can be assumed to hold after the refactoring. This, in turn, may eliminate the need to check preconditions of a subsequently performed refactoring, if they can be proved to be implied by the postconditions. While Smalltalk is very different from Java, it is interesting to see that Roberts' specification of RENAME CLASS provides a precondition to guard against a class and a global variable having the same name, which is somewhat reminiscent of the problem with obscuring in Java. However, Roberts makes no provisions to introduce additional qualifiers to evade name capture, either.

Mens *et al.* [91] specify refactorings as transformations on a largely language-independent graph representation of programs that concentrates on those aspects of the source code to be preserved by the refactoring. Their definition of preservation properties is very close to the binding preservation invariant we use here. However, their graph representation does not seem suitable for specifying renamings, since name lookup is not explicitly represented in the program graph.

Vorthmann's work on the specification of name lookup [137] introduces the concept of visibility networks to describe the name visibility and binding semantics of programs. Such networks are composed of

sub-networks corresponding to scopes or other constructs influencing name binding; the lookup rules of a language can then be described by specifying what kinds of networks exist, how they arise from concrete programs, and how they can be composed. This leads to a high-level description of name lookup, which might make a good starting point for implementing RENAME refactorings.

Another approach to the specification of refactorings is introduced by Garrido and Meseguer [47]. Based on a formalisation of Java in rewriting logic, they give executable specifications of several refactorings in Maude, among them RENAME TEMPORARY VARIABLE. Their specifications are quite concise, but this is perhaps partly due to the fairly local nature of the refactorings they describe. They base their work on an early version of Java with much simpler name lookup rules than Java 5.

A very elegant specification of the essence of name lookup in Java is given by Verbaere [134], who formulates name lookup as a stream of visible declarations, with earlier declarations taking precedence over later declarations. This specification forms the basis of a simple implementation of RENAME VARIABLE that is formulated in terms of such declaration streams, and hence generic over the concrete lookup rules of the language. However, such a generic rename cannot introduce additional qualifications. Verbaere also gives a specification of a more powerful RENAME refactoring that constructs qualified accesses where needed, directly based on an analysis of the possible forms of qualification in Java.

Chapter 3

Control and Data Flow

The previous chapter has introduced a name binding framework that enables us to track name binding dependencies and to ensure that names bind to the right declarations. This framework not only makes the important family of RENAME refactorings very easy to implement, it is also widely useful in the implementation of many other refactorings. In general, however, it is not by itself sufficient to ensure behaviour preservation.

In this chapter, we will apply ourselves to the study of a different class of refactorings: local restructurings that reorganise code within the scope of a single method. Arguably the two most widely used local restructurings are INLINE TEMP and EXTRACT TEMP, which will be our running examples in this chapter: the former, briefly introduced in the introduction, eliminates a local variable by replacing all its uses with its definition; the latter, its inverse, introduces a new local variable, assigns it a certain expression as its initial value, and replaces one or more occurrences of that expression with an access to the local variable.

For both of these, correctly handling naming issues is important, as shown in Figure 3.1: when inlining an expression that contains an access to a field, such as `x` in the program on the left, it could be that this field is hidden by a newly declared local variable at a point where the expression is inlined. Of course, we do not want the field access to become captured by this new variable, so we use the naming framework to lock all names in the expression to be inlined and unlock them again after inlining. In the example, a qualifying `this` will be added to the field access, as shown in the program on the right. Similar precautions have to be taken when extracting an expression into a local variable.

But this by itself is not enough. INLINE TEMP can obviously only be behaviour-preserving if the expression to be inlined has the same value at the point where it is inlined as at the point where it was evaluated in the original program. In particular, there cannot be any intervening redefinitions of variables used in the

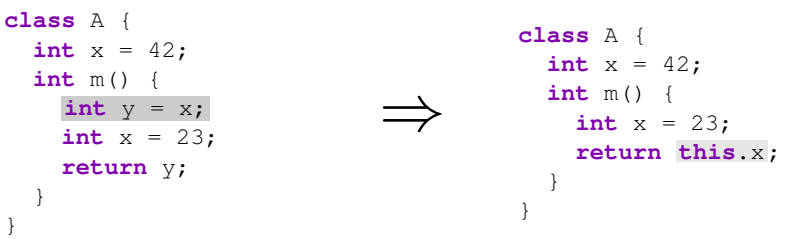


Figure 3.1: Example of naming issue with INLINE TEMP

expression, as this may change the computed value. Simply looking at the name bindings will not tell us whether such a redefinition may occur; we need something more.

To handle this kind of problem, we generalise the idea of name binding preservation that underlies renaming to the preservation of other kinds of dependencies. In the first part of this chapter we will study several classes of control and data flow dependencies, collectively referred to as *flow dependencies*, that statically capture certain aspects of the dynamic execution behaviour of the program. For instance, a so-called data dependency connects an assignment to a variable to all reads of the same variable that possibly see the value written by the assignment. To capture the interaction between control and data flow, we will furthermore enrich data dependencies with information about the paths along which the value might propagate from the write to the read.

Preserving these flow dependencies is enough to guarantee behaviour preservation for INLINE TEMP and EXTRACT TEMP on sequential programs. For concurrent code, the situation is more complicated, as explained in the second part of the chapter: in addition to control and data flow, we also have to account for the interaction between reads and writes to shared memory and synchronisation constructs.

As it turns out, dependency preservation is again the key to success: we define a class of synchronisation dependencies that statically capture this interaction, and whose preservation makes a sequentially correct refactoring behaviour-preserving even on concurrent programs, provided these programs do not contain data races.

At the end of this chapter, we will have all the basic tools we need to describe and implement many useful refactorings. Subsequent chapters will develop additional techniques to make these tools usable on a real-world language like Java with its many idiosyncrasies and quirks.

3.1 Sequential programs

Our running example for introducing the concepts and techniques needed to deal with local restructurings will be INLINE TEMP, with occasional references to EXTRACT TEMP which, being its inverse, is handled very similarly.

As explained in the introduction, the purpose of the INLINE TEMP refactoring is to eliminate a local variable by replacing its every use with a copy of the expression the variable is initialised to. While this is a conceptually very simple operation, it will be convenient to further divide it up into three steps, which themselves are refactorings:

1. We first split off the initialiser of the local variable into a stand-alone assignment.

2. Then we take the assignment created in the first step and inline it into all its uses.

3. Finally, we remove the variable declaration if it is not referenced anymore.

Obviously, the second step, which we call INLINE ASSIGNMENT, performs most of the real work, and it will be the focus of our attention. Splitting off a variable's initialiser into an assignment is made surprisingly tricky by a number of syntactic peculiarities of Java, but this step does not pose any deep problems and we will defer its discussion along with that of the final step to Chapter 5. Dividing up the refactoring in this way also makes it more flexible: if the variable is assigned more than once, the refactoring will still go ahead, but only uses of the initial assignment will be substituted, and the variable will not be removed.

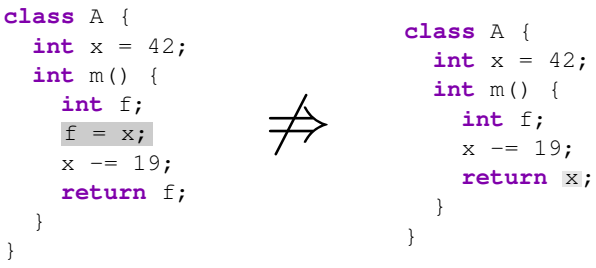


Figure 3.2: Problematic example of INLINE ASSIGNMENT (I)

In an analogous way, we can divide EXTRACT TEMP into three steps that are the respective inverses of the steps for INLINE TEMP above:

1. We introduce an unused local variable.

2. Then we create an assignment of the expression to be extracted to that local variable and replace all copies of the expression with a reference to the local variable (EXTRACT ASSIGNMENT).

3. Finally, we merge the assignment with the declaration, turning it into an initialiser.

For simplicity, we will require the expression that is extracted or inlined to be pure in the sense that its evaluation cannot cause any side effects or write variables. We will, however, allow evaluation to terminate abruptly by throwing an exception; this is crucial, since many if not most kinds of expressions in Java can in principle throw exceptions, and a fairly sophisticated analysis is needed to prove that an expression never gives rise to an exception.

3.1.1 Control and data flow dependencies

When inlining an assignment, we need to ensure that the inlined expression has the same value at the point where it is inlined and at the point where it was located originally. For instance, the refactoring in Figure 3.2 should not be allowed to go ahead, since the inlining moves the read of field `x` from the second to the fourth statement of method *m*, over the assignment in the third statement. It may come as a bit of a surprise that none of the current state-of-the-art refactoring engines check for this kind of problem: the most recent versions of Eclipse, IntelliJ and NetBeans are all happy to perform the flawed refactoring.

Abstracting from the concrete example, we could describe the problem by saying that the variable read `x`, which forms part of the inlined expression, "sees" a different write before and after the refactoring: in the original program, it does not see any write within `m`, i.e., it will read whatever value was last written to the field before the method was invoked; in the refactored program, on the other hand, it sees the assignment `x -= 19`. It seems reasonable to require that this cannot happen: the refactoring should not alter the writes a variable read inside the inlined expression sees. For an expression that cannot throw an exception this is already a sufficient criterion: since the expression is pure, its value is entirely determined by the values that are seen by the variable reads contained in it, and the expression cannot itself change the value read by any other variable reads.[1]

[1] We are here ignoring possible non-determinism arising from interaction with the operating system or the hardware.

In order to make the concept of which variable read sees which variable write more precise, we need a way to reason about all possible executions of a method; this is usually done by constructing a *control flow graph*, or CFG for short, that statically captures this information.

Traditionally, CFGs are often defined over a linearised intermediate representation, with the program code given as a sequence of machine code-like instructions and control flow encoded explicitly by conditional branching instructions. From a refactoring perspective, it makes more sense to view the CFG as an additional graph structure superimposed on top of the AST. Its vertices are nodes representing expressions and statements, collectively referred to as *control flow nodes*. There is an edge from a control flow node n_1 to a control flow node n_2 if the statement or expression represented by n_2 could be evaluated immediately after the one represented by n_1. We then say that n_2 is a *control flow successor* of n_1, or equivalently that n_1 is a *control flow predecessor* of n_2, and write $n_1 \rightarrow_s n_2$.

We call an edge in the CFG that leads from a statement within a loop body back to the loop header a *back edge*. In Java, back edges originate from the last statement of a loop body, as well as from **`continue`** statements. For a path p, we define $loops(p)$ to be the set of all nodes that appear as the target of a back edge within p, i.e., the set of all loop headers that p traverses using a back edge.

Control flow graphs are often classified as intra-procedural or inter-procedural, depending on whether method calls are treated as opaque instructions or are connected to a CFG of the called method. Obviously, an inter-procedural CFG can be much more expensive to construct, in particular in a language with virtual dispatch like Java, where it may be hard or even impossible to determine which method a given call will dispatch to at runtime. For our purposes it is enough to construct an intra-procedural CFG on a per-method basis that does not descend into method calls. We do, however, require that the CFG correctly models exceptional behaviour of methods: that is, if a called method declares a thrown exception there should be an edge in the CFG from this call to any catch clause that might handle this exception, and additional edges to model possible unchecked exceptions such as `NullPointerException`s that need not be declared.

We refer to any CFG edge that models exception handling as an *exceptional edge*. If there is an exceptional edge between nodes n_1 and n_2, we will sometimes write this as $n_1 \rightarrow_e n_2$. Exceptional edges can originate at method or constructor calls, **`throw`** statements, or any other kind of expression that may throw an exception, such as integer division expressions, which can throw an `ArithmeticException` if the denominator is zero; member access expressions, which can throw a `NullPointerException` if the qualifier is **`null`**; or arithmetic expressions, which can throw a `NullPointerException` if one of their arguments is a **`null`** reference of boxed primitive type.

Finally, we assume that there is a designated *entry node* for every method, that is a control flow node without control flow predecessors such that every control flow node in the method is reachable from it. Symmetrically, there are one or more *exit nodes* without control flow successors, such that every control flow node in the method can reach one of them. The rationale for allowing more than one exit node is to allow the control flow graph to have one exit node representing normal completion of the method, and one or more other nodes for representing exceptional termination; we do not require that these nodes are distinct, though. In any case, the exit nodes serve as targets of exceptional edges whose exception type is not handled within the method.

When reasoning about data flow, we are mostly interested in those CFG nodes that read or write memory locations, where a memory location is either a local variable, a parameter, a field of an object, or an array element. A simple way to represent locations statically is to equate them with their declarations in the AST, introducing special nodes to represent all elements of all arrays of a given type. It is then easy to determine

for any variable access which locations it reads and writes: if the variable access occurs in an lvalue position, it writes the location represented by the declaration it binds to; if it occurs in an rvalue position, it reads that same location. Of course, the same variable access may both read and write the same location (consider, for instance, `x++`).

This representation has the important property of being *conservative*: if two variable accesses read or write the same location at runtime, we can predict this statically. It is also quite imprecise in that, for instance, two accesses `o.f` and `p.f` to the same field are always considered to potentially refer to the same location, even if it is obvious from context that `o` and `p` must always be different objects. A more sophisticated representation of memory locations could be employed to improve on this.

Our development makes no assumptions about how the locations read or written are determined; we simply assume that for every node n in the CFG we can determine a set $W(n)$ of written and $R(n)$ of read locations, and that this is done in a conservative way. In particular, the memory locations read and written by invoked methods or constructors must be approximated conservatively. A simple, but again imprecise, choice is to assume that a method or constructor may both write and read every field and array element, but not local variables or parameters of the calling method.

In a similar manner, a purely intra-procedural analysis needs to make conservative assumptions about which locations have been written before the analysed method is invoked, and which locations may be read after it returns control to its caller. In the absence of a more precise analysis, we can conservatively assume that the entry node of a method writes all fields and array elements, and that every exit node reads all fields and array elements.

Given this information, we can now define our central data flow concepts.

Definition 1. *We say that there exists a* data dependency *between CFG nodes w and r if there is some location $l \in W(w) \cap R(r)$, and there is a (non-empty)* witnessing path *between w and r in the CFG such that no vertex on the path except for w writes l. We denote this as $w \rightarrow_d r$.*

Sometimes it will be necessary to keep track of the loop headers traversed by the witnessing path; we will then write $w \rightarrow_d^L r$ to mean that $L = loops(p)$ for a witnessing path p.

If $w \rightarrow_d r$, we also say that w is a reaching definition *of r, and that r is a* reached use *of w. We say that w is a reaching definition of r* through *l to mean $w \rightarrow_d^L r$ for some $L \ni l$, and likewise for reached uses. On the other hand, we call w a* direct reaching definition *of r if $w \rightarrow_d^{\emptyset} r$.*

Note that the restriction to non-empty witnessing paths is essential: a variable access that is both a read and a write, such as the `x` in `x++`, performs the write *after* the read, so it should not have a data dependency on itself, except if there is another non-empty witnessing path to establish this dependency.

For example, in the original program in Figure 3.2, the access of `x` that we want to inline has no reaching definition inside the method, so the entry node of the method becomes its only reaching definition by default. After the erroneous inlining, it has picked up a new reaching definition, namely the assignment to `x`. Thus the concept of reaching definitions formalises the informal notion of a read "seeing" a write. We would certainly want any implementation of INLINE ASSIGNMENT to preserve the reaching definitions of any variable read within the inlined expression.

In Figure 3.2, the reaching definitions of the read of `x` were all direct. Figure 3.3 shows an example of a data dependency through a loop: in the program on the left, the read of `y` that is to be inlined has *both* writes of `y` as its reaching definitions: the first is direct, the second is through the while loop. In the program on the right, it still has these same two reaching definitions, but now both of them are direct.

Figure 3.3: Problematic example of INLINE ASSIGNMENT (II)

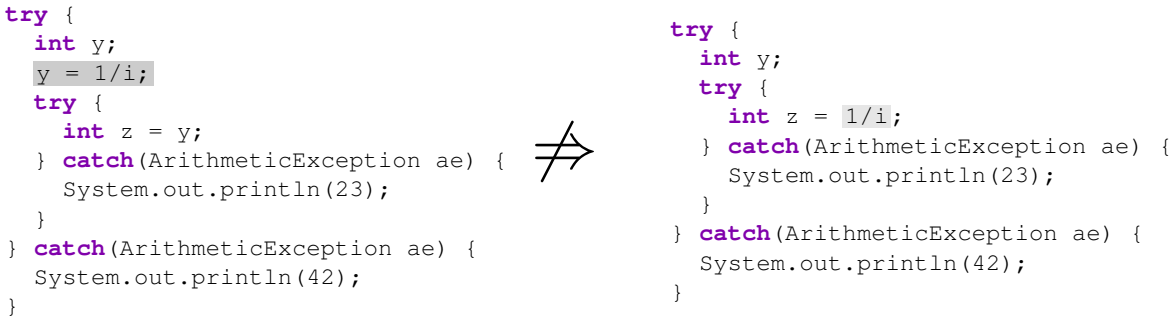


Figure 3.4: Problematic example of INLINE ASSIGNMENT (III)

Of course, the transformation in this example is not in general behaviour preserving: in the program on the left, method `m` is called with the argument `42` in the first iteration of the loop and with argument `23` on every subsequent iteration, whereas in the program on the right the argument is always `23`. The example thus shows that we have to ensure not only that reaching definitions are preserved, but that they go through the same loops as before.

To make it easy to enforce this preservation, we extend the dependency locking metaphor, which plays such a crucial role in the formulation of RENAME in the preceding chapter, to data flow: before performing the inlining, we lock all reaching definitions of the expression e to be inlined, which means computing and caching the set of reaching definitions (along with their kind) of any read within e. After the inlining, we "unlock" these dependencies by recomputing reaching definitions and checking that the recomputed set is identical to the cached set. If this is not the case, unlocking fails, causing the entire refactoring to be aborted.

So far our examples have been about inadvertent changes in data flow, i.e., in the way values propagate from variable writes to variable reads, but control flow, i.e., the order in which expressions and statements are executed, can also be influenced by inlining assignments. To see how, consider the program in Figure 3.4 on the left. Here we inline a variable initialised to the expression `1/i`, where we assume that `i` is an integer variable from an enclosing scope, for instance a method parameter. If `i` happens to have value 0, an `ArithmeticException` will be thrown, and the program will print `42` to the console. In the incorrectly refactored program on the right, the same execution will print `23`, since the thrown exception is caught by a different **`catch`** clause.

To prevent this kind of situation, it is clearly sensible to require that exceptional control flow edges originating in the expression to be inlined must be preserved by the refactoring. That is, if there is an edge $n \to_e c$ from a node n within the inlined expression in the original program, then there must be an edge $n' \to_e c$ for every inlined copy n' of n, and vice versa. This can again be understood as locking and unlocking of dependencies.

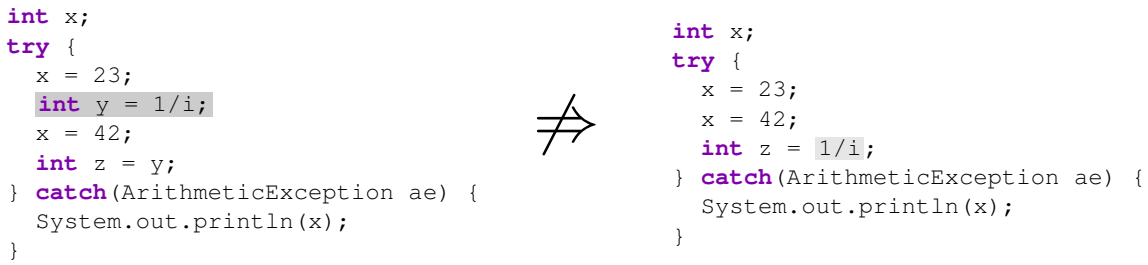


Figure 3.5: Problematic example of INLINE ASSIGNMENT (IV)

Locking exceptional control flow edges is by itself not sufficient, however, as can be seen on the example in Figure 3.5. Here, the exception thrown by the inlined expression is handled in the same way in the original program and the refactored program, but the read of `x` in the **`catch`** block now sees a different write that was exposed by the inlining. In terms of data flow dependencies, we can see that the set of reached uses of the assignment `x = 42` has changed: before the refactoring, it did not reach any use in the displayed snippet; after the refactoring, however, it reaches the use of `x` in the print statement.

To describe this in data flow terms, we need one more concept:

Definition 2. *We say that a location l is* live *at a node n_1 if there is a node n_2 with $l \in R(n_2)$ and a (possibly empty) path from n_1 to n_2 such that no node on that path writes l, except maybe n_2.*

In the above example, `x` is live at the beginning of the **`catch`** clause, since it is passed as an argument to the print method without first being redefined. In general, then, we want to avoid any write between the point where the expression was evaluated originally and the point where it is evaluated in the refactored program if the written location is live at any target of an exceptional edge originating at e or one of its subexpressions.

3.1.2 INLINE ASSIGNMENT

In the previous subsection, we have motivated some conditions that must be fulfilled in order for INLINE ASSIGNMENT to be behaviour preserving, and have shown how to describe them in terms of control and data flow. We will now collect these into a more systematic description of the refactoring, and give a brief intuitive argument as to why they are sufficient to guarantee behaviour preservation. A more detailed argument will be presented in Chapter 6.

Assume we are given an assignment statement d of the form `x = e;`, where `x` refers to a local variable of the enclosing method, constructor or initialiser m, and e is a pure expression. Let U be the set of all accesses to `x` that have d as their reaching definition. The purpose of the refactoring is to replace every $u \in U$ with a copy of e and remove d from m.

We first need to ensure that the following three preconditions hold:

1. None of the $u \in U$ is a write.
2. Every $u \in U$ has only a single reaching definition.
3. If e can throw exceptions, then every path from d to an exit node of m must contain at least one $u \in U$.

The first condition is straightforward: it makes no sense to inline into an expression like `x--`, since what is decremented here is `x`, not its value. The second condition makes sure that we only inline into variable

Figure 3.6: Inlining into non-uniquely reached use

accesses that definitely get their value from the assignment we are inlining. To see why this is necessary, consider the program in Figure 3.6 on the left. The assignment we are inlining has a single reached use within the displayed snippet; however, that use only gets its value from the assignment if `b` is false, otherwise it will read the value written by the other assignment to `x`, namely `42`; hence it is not behaviour preserving to perform this inlining. The third condition ensures that the refactoring does not eliminate exception throws.

Now we iterate over all uses $u \in U$. For every such u, we create a copy e_u of u with locked names, locked reaching definitions, and locked exceptional edges. Let C be the set of all targets of these exceptional edges. If e can throw exceptions, we check there is no node n with $d \to_s^* n \to_s^* u$ such that n causes any side effects or writes a location that is live at some $c \in C$. Then we replace u with e_u.

After all uses have been replaced, we turn d into an empty statement and unlock all locked names, control and data flow edges. If the unlocking fails, the whole refactoring is aborted and changes are rolled back. If it succeeds, an additional step can eliminate the empty statement.

We will now give a brief intuitive argument why the refactored method m' has the same behaviour as the original method m, which will be made more precise in Chapter 6.

First of all, we observe that preservation of reaching definitions means that the value seen by a read in the original expression e cannot be overwritten by a write occurring between d and the position u where it is inlined, so the inlined reads see the same value as the original reads.

Lemma 1. *Let r be a variable read in e, and r_u one of its inlined copies in some e_u. Assume that r_u has the same data flow dependencies as r. Then no control flow path from d to r_u that passes through d at most once contains a write w with $W(w) \cap R(r) \neq \emptyset$.*

Proof. Let L be the set of loops in which r (and hence d) is nested, and L' the set of loops for u (and hence r_u). Let b be the innermost block around d; then u is part of a statement s that comes after d in b (otherwise u would not be uniquely reached by d).

Assume now for contradiction that there is a path p from d to r_u not passing through d more than once, but containing a w with $W(w) \cap R(r) \neq \emptyset$. We choose the last such w to occur on p.

If $loops(p) = \emptyset$, then r_u has a direct data dependency of w; but then r would also have a direct data dependency on w, which is impossible, since any witnessing path for such a dependency would go through at least one $l \in L$.

Otherwise, r_u has a data dependency on w through some loop $l' \in L'$. Since p does not pass through d more than once, we must have $l' \notin L$, so again r cannot have a dependency on w through l'.

We conclude that such a w cannot exist. □

Now consider an execution of m. If it never executes the assignment d, it cannot execute any of the reads in U either, so the execution will play out the same way in m'.

If it does execute d, it needs to evaluate e. The evaluation of e might result in an exception being thrown, which will be caught inside m or lead to abrupt termination of the method. If we consider an execution of m'

starting in the same state, that execution will of course not evaluate the assignment, which is not part of m' anymore. No matter which path the execution follows afterwards, however, it will eventually reach a copy e_u of e, and start evaluating that copy. By the lemma above variable reads in that copy will see the same values as in the original, so the evaluation of e_u will throw the same exception as the evaluation of e in m, which will be handled in the same way. From that point on, execution of m and m' will proceed in the same way: any additional writes in m' do not affect further reads, since they cannot write locations that are live at the point where the exception is handled.

Finally, consider the case where the execution of m evaluates e without an exception being thrown. If it does not encounter any $u \in U$ later on, its behaviour does not differ from the corresponding execution in m' that simply skips the assignment; after all, the evaluation of e cannot have any side effects or change any variable values. If it does encounter some such u, the execution of m' will at this point proceed to evaluate e_u. As above, we argue that since reaching definitions have been preserved each read sees the same value as the corresponding read in e, so the expression evaluates to the very value that is obtained from the read u in m, guaranteeing that the further execution proceeds in the same way.

A similar argument shows the reverse direction, that every execution of m' corresponds to an execution of m exhibiting the same behaviour.

3.1.3 EXTRACT ASSIGNMENT

When considering EXTRACT TEMP, or rather its main constituent subrefactoring EXTRACT ASSIGNMENT, the work we have done on INLINE ASSIGNMENT above immediately pays off: since EXTRACT ASSIGNMENT is the exact inverse of INLINE ASSIGNMENT, we only have to ensure that after extracting the conditions for inlining are fulfilled and the transformations cancel each other out.

To extract an assignment, we start out with nodes $e_1, \ldots, e_n$ that represent n copies of the same expression e. We check that corresponding names bind to the same declaration in all copies, that reads have the same reaching definitions, and that exceptional control flow edges have the same target. We create a master copy e_0 of the expression, with names, reaching definitions and exceptional control flow locked to the same targets, and insert an assignment d of e_0 to `x` in the required position. If e could throw an exception, we check that there is one e_i on every path from d to an exit node, and that there are no side effects or writes to locations that are live at targets of exceptional control flow edges between d and any e_i. Finally, we replace all the e_i by accesses to `x`, ensuring that none of them is a variable write, that all have d as their only reaching definition, and that there are no other reached uses of d.

To see that this gives us a precise inverse, consider what would happen if we were to perform an inlining immediately afterwards. We have made sure that the reaching definitions of d are precisely the positions from where we extracted, so the transformations cancel each other out. Since we have checked name bindings, reaching definitions and exceptional flow edges to be the same, there will not be any problem with unlocking these dependencies. All the other conditions of INLINE ASSIGNMENT have been checked verbatim, so they hold as well. This shows that our above definition of EXTRACT ASSIGNMENT is behaviour preserving if INLINE ASSIGNMENT is.

3.1.4 Implementation considerations

We briefly explain how our refactoring engine implements the necessary control and data flow analyses underlying the INLINE TEMP and EXTRACT TEMP refactorings.

```
interface CFGNode {
  Collection<CFGNode> succ();
  Collection<CFGNode> pred();
}

Stmt implements CFGNode;
Expr implements CFGNode;
ParameterDeclaration implements CFGNode;

ast EntryNode : EmptyStmt;
ast ExitNode : EmptyStmt;
```

Figure 3.7: Interface of the control flow analysis module

To perform control flow analysis, we rely on a revised implementation of the control flow module for JastAddJ originally implemented by Nilsson-Nyman and others [98]. In this module, control flow is implemented as an attribute `succ` that computes the set of control flow successors for every control flow node, and a dual attribute `pred` for computing predecessors. Both statements and expressions are control flow nodes, as are parameters, which represent the assignment of concrete arguments to formal parameters. Every method has one entry node, one exit node for normal method return, one additional node for every declared exception it may throw, and one node for unchecked exceptions. These nodes serve as the control flow successors of **`throw`** statements where the thrown exception is not caught within the method. Since the analysis is intraprocedural, control flow does not "step into" method calls or constructor invocations, but the analysis does take possible exceptions thrown by such calls into account.

For a more detailed discussion of the implementation the reader is referred to the literature; the API is summarised in Figure 3.7: we have a common interface `CFGNode` for all node types that appear as vertices of the control flow graph; this interface exposes the attributes for successor and predecessor computation. Two new node types `EntryNode` and `ExitNode` are declared, which represent entry and exit nodes of methods, constructors and initialisers; since they inherit from `EmptyStmt`, they are in particular `CFGNode`s.

The necessary data flow analysis is easily implemented on top of this module. Locations corresponding to fields, parameter declarations and local variables are represented by their declarations, whereas array elements are represented by a new class `ArrayElementLocation`, which holds the type of the array element. Additionally, there is a singleton class `AnyHeapLocation` to represent an unknown heap location. The location read or written by a variable access is then simply the declaration it binds to, for an array element access it is the appropriate `ArrayElementLocation` object, and for method or constructor calls it is the `AnyHeapLocation` instance. The corresponding code is shown in Figure 3.8.

For every two `Location` objects, we need to be able to tell whether they may *alias*, i.e., whether they could represent the same location. The rules for determining aliasing are quite straightforward, given our simple representation of locations: fields can alias other fields with the same declaration, local variables and parameters can only alias themselves, the unknown heap location can alias any other heap location; an array element can alias any other array element whose type it has a casting conversion to, which roughly means that one type is a subtype of the other.[2] The special rule for array elements accounts for the fact that array types are covariant in Java.

[2]The precise definition of casting conversion can be found in the JLS [48, §5.5], we reuse the implementation in JastAddJ.

```
// common interface for representing memory locations
interface Location {
  boolean isHeapLocation();
  boolean mayAlias(Location l);
  boolean mustAlias(Location l);
}

// locations corresponding to variables are represented by their declaration
Variable extends Location;

// common representation of all array elements of a given type
class ArrayElementLocation implements Location {
  private TypeDecl type;
  // constructor omitted
}
// lazily allocate one ArrayElementLocation for every array type
syn lazy ArrayElementLocation ArrayDecl.getElementLocation()
  = new ArrayElementLocation(componentType());

// representation of an arbitrary heap location (singleton class)
class AnyHeapLocation implements Location {
  private AnyHeapLocation() { }
  public static final AnyHeapLocation instance = new AnyHeapLocation();
}

syn boolean Location.isHeapLocation() = true;
eq ParameterDeclaration.isHeapLocation() = false;
eq VariableDeclaration.isHeapLocation() = false;

// locations may alias themselves; heap locations also alias AnyHeapLocation
syn boolean Location.mayAlias(Location l)
  = this == l || isHeapLocation() && l instanceof AnyHeapLocation;

// additionally, array elements may alias other array elements of compatible type
eq ArrayElementLocation.mayAlias(Location l)
  = super(l)  || l instanceof ArrayElementLocation &&
                 ((ArrayElementLocation)l).type.castingConversionTo(type);

// AnyHeapLocation, of course, may alias any heap location
eq AnyHeapLocation.mayAlias(Location l) = l.isHeapLocation();

// alias information is precise for non-heap locations
syn boolean Location.mustAlias(Location l) = !isHeapLocation() && mayAlias(l);

syn lazy Location Access.getLocation();
eq VarAccess.getLocation() = decl();
eq ArrayAccess.getLocation() = type().arrayType().getElementLocation();
eq MethodAccess.getLocation() = AnyHeapLocation.instance;
// same for constructor invocations
```

Figure 3.8: Locations and aliasing

```
syn lazy Collection<CFGNode> Access.reachingDefinitions() {
  Collection<CFGNode> res = new HashSet<CFGNode>();
  if(!isRead())
    return res;
  for(CFGNode p : pred())
    res.addAll(p.reachingDefinitionsFor(getLocation()));
  return res;
}

syn lazy Collection<CFGNode> CFGNode.reachingDefinitionsFor(Location l)
    circular [new HashSet<CFGNode>()] {
  Collection<CFGNode> res = new HashSet<CFGNode>();
  if(mayWrite(l)) {
    res.add(this);
    if(mustWrite(l))
      return res;
  }
  for(CFGNode p : pred())
    res.addAll(p.reachingDefinitionsFor(l));
  return res;
}

syn boolean CFGNode.mayWrite(Location l) = false;
eq EntryStmt.mayWrite(Location l) = true;
eq VariableDeclaration.mayWrite(Location l) = this == l;
eq ParameterDeclaration.mayWrite(Location l) = this == l;
eq Access.mayWrite(Location l) = isWrite() && l.mayAlias(getLocation());

syn boolean CFGNode.mustWrite(Location l) = false;
eq VariableDeclaration.mustWrite(Location l) = this == l;
eq ParameterDeclaration.mustWrite(Location l) = this == l;
eq Access.mustWrite(Location l) = isWrite() && l.mustAlias(getLocation());
```

Figure 3.9: Reaching definitions analysis in JastAdd

Sometimes it is also useful to know whether two `Location`s *must* alias, i.e., whether they definitely represent the same location. We can rarely answer this in the positive, except for local variables and parameter declarations.

The chosen representation of locations makes the analysis lightweight but imprecise. Efficient algorithms for control and data flow analysis are a staple of the program analysis literature, so it should not be too hard to improve upon this implementation. An interesting empirical question would be to determine how much of an impact the very conservative treatment of method invocations has in practice.

As an example of the implementation of data flow analyses, Figure 3.9 shows how to determine the reaching definitions of a variable read. The analysis is defined as an attribute `reachingDefinitions` on interface `CFGNode`. When compiling to Java, JastAdd inserts a copy of the definition of this attribute into every class implementing `CFGNode`.

The attribute `reachingDefinitions` is just a thin wrapper around the auxiliary attribute `reachingDefinitionsFor`, which determines the reaching definitions for a given location at some CFG node. Observe that it invokes that attribute on all its control flow predecessors and unions the results instead of directly invoking it on the access whose reaching definitions we compute: this avoids the above-mentioned problem with witnessing paths of length zero.

The attribute `reachingDefinitionsFor` is declared to be **`circular`**, so its value will be determined by fixpoint iteration, starting from the initial value provided between brackets, namely the empty set. The computation itself is fairly straightforward: if the current node could write to the location in question, it is a reaching definition; if it definitely writes to it, we can stop the computation; otherwise we simply take the union over reaching definitions as determined at the control flow predecessor nodes. The entry node is special in that it is a reaching definition for any heap location, and local variable declarations are reaching definitions for themselves.

The implementation is easily extended to also determine which loops the reaching definitions go through, by keeping track of traversed back edges when performing the recursive invocation on the predecessors.

To implement locking and unlocking of reaching definitions, we furnish class `Access` with an additional field `lockedReachingDefs` to hold locked reaching definitions. When locking reaching definitions, we compute them using attribute `reachingDefinitions` and store the returned value into the field. Upon unlocking, we recompute `reachingDefinitions` and check whether its value is the same as the stored value. If so, the stored value is cleared, otherwise we abort the refactoring, indicating a violation of dependency preservation. The same approach is used to lock and unlock exceptional control flow edges.

At the moment, our implementation treats a violation of a data or control flow dependency in the same way as a name binding violation: it aborts the refactoring with an error message. Arguably, however, it would be better to instead present such violations to the user as warnings and let them decide whether the refactoring should go ahead or not. If a name binding changes where it should not, this is almost certainly an error. If a data flow dependency is violated, on the other hand, this may well be since the conditions are too coarse, or our underlying data flow analysis is too conservative, especially where method calls are involved. Given the very local character of refactorings like INLINE ASSIGNMENT, it is likely that the user can decide whether a violation looks serious enough to abort the refactoring, or whether it should go ahead anyway.

The control and data flow analyses together with the machinery for locking and unlocking dependencies make it essentially straightforward to implement the informal specification of INLINE ASSIGNMENT and hence INLINE TEMP. We will discuss their implementation in more detail in Chapter 5.

3.1.5 Related work

The material in this section has not previously been published, except for a very cursory discussion of INLINE TEMP and EXTRACT TEMP in our ECOOP 2009 paper [119], which introduced the decomposition of these two refactorings into three steps. That paper did not, however, specify sufficient conditions for behaviour preservation, nor did it discuss the implementation in any detail.

As we have shown on some examples, current industrial-strength refactoring engines do not reason about control and data flow at all. Although Opdyke's thesis [100] contains a brief high-level discussion of data flow analysis, he proposes to use such an analysis for an entirely different purpose, namely for reasoning about class invariants when specialising classes.

Control and data flow analysis plays a central role in Griswold's thesis [49]. He views refactorings in terms of their effects on the program dependency graph (PDG), an abstract representation of programs based on flow dependencies. He derives a number of behaviour preserving transformations on PDGs, and then investigates how to express refactorings using these atomic transformations, thereby guaranteeing their behaviour preservation. While this approach is very elegant and ties in well with our emphasis on dependency-based reasoning as opposed to reasoning with preconditions, it is unclear how well it works in practice.

Griswold's proposed refactorings work on a first-order subset of Scheme, which is a very simple language compared to Java.

The refactorings we have discussed in this section are, of course, very similar to code transformations commonly performed by optimising compilers, that have been studied in the literature for decades: EXTRACT TEMP is essentially just another name for common subexpression elimination [29], and INLINE TEMP has been studied under the name of forward substitution [74]. It thus seems like an obvious idea to try and reuse some of the techniques developed for compilers in the context of refactorings. Indeed, concepts like reaching definitions and liveness come straight from the compiler literature.

There are, however, important differences between the two application areas. Most notably, optimising compilers operate on a simplified intermediate representation of the program that is easy to analyse and manipulate, whereas refactoring tools work entirely on the source level, so that both analysis and transformation have to be done on program source code. Many classic algorithms in the compiler construction literature, however, are specifically geared towards this kind of representation, and are not directly reusable at the source level. For instance, common subexpression elimination is usually only done for expressions that consist of a single operator applied to arguments that are either constants or symbolic registers [94]. Adapting an algorithm designed with these assumptions in mind to work on arbitrary source level expressions is not easy.

On the other hand, refactoring tools have it easier than compilers in two respects: first of all, while a compiler has to decide by itself whether or not to apply a transformation, for a refactoring tool this decision is always made by the user. No cost metrics are needed to determine whether a transformation is worthwhile, instead the tool simply has to do its best to perform the requested refactoring.

Secondly, an optimising compiler can only perform a certain transformation if it can prove beyond doubt that it will not change program semantics. This is also the goal of a refactoring engine, and the tool should never silently perform a transformation that it cannot prove to be behaviour preserving. But in such a case it could still be worthwhile to give the user a preview of the transformation about to be performed by the refactoring, with a warning indicating potential sources of behaviour change. If the transformation is local enough and the user is sufficiently familiar with the program to be refactored, they may be able to convince themselves that the transformation is indeed safe, and choose to go ahead anyway. From a usability perspective, this is much preferable.

3.2 Concurrent programs

The previous section has shown how well-known concepts of control and data flow analysis can be put to work to formulate and reason about refactorings that rearrange code. So far our discussion has been limited to sequential programs, and we have only considered control and data flow for a single thread executing in isolation.

Since its very inception, however, Java has placed a great deal of emphasis on concurrent programming: it comes with a built-in model of multi-threading that is exposed at the library level with threads reified as objects of class `java.lang.Thread`, and a synchronisation model based on monitors that is directly embedded into the language through the **`synchronized`** keyword. Furthermore, the `java.lang.concurrent` library, introduced in Java 5, provides more fine-grained control over every aspect of inter-thread communication and synchronisation. The theoretical underpinnings of concurrency in Java are provided by the Java Memory Model [87], which is a formalisation of the behaviour of multi-threaded Java programs.

```
import static java.lang.System.out;

class A {
  int f;
  public void m1() {
    int g = 0;
    synchronized (this) {
      g = f;
    }
    if(g % 2 != 0)
      out.println("how odd!");
    synchronized (this) { g = f; }
  }
  public synchronized void m2() {
    ++f; ++f;
  }
}
```

⇏

```
import static java.lang.System.out;

class A {
  int f;
  public void m1() {
    int g = 0;
    int n = f;
    synchronized (this) {
      g = n;
    }
    if(g % 2 != 0)
      out.println("how odd!");
    synchronized (this) { g = n; }
  }
  public synchronized void m2() {
    ++f; ++f;
  }
}
```

Figure 3.10: EXTRACT TEMP on concurrent code

In this section we will examine the impact of concurrency on refactoring in Java. We will see that even refactorings that are behaviour preserving on sequential code can change the behaviour of concurrent code by enabling new interleavings that were not originally possible, or by introducing deadlocks or livelocks. We will introduce a concept of synchronisation dependencies that statically capture concurrent execution constraints, and relate these dependencies to the formal specification of the memory model and prove that preserving these dependencies gives us strong guarantees about behaviour preservation in the presence of concurrency. Finally, we discuss how to integrate these additional dependencies into our implementation.

3.2.1 Motivating examples

As before, our running examples will be the refactorings EXTRACT TEMP and INLINE TEMP.

Let us first consider the program in Figure 3.10 on the left. We have a method `m2` that increments the field `f` twice. Note that `m2` is declared to be **`synchronized`**, i.e., whenever the method is called on an object `a` of type `A`, the built-in monitor lock of `a` is acquired, and released upon completion. Method `m1` reads the value of field `f` and stores it into a local variable `g`. This read happens within a **`synchronized`** block that synchronises on the receiver object's monitor lock, so `m2` executes atomically from the point of view of the read, and it will never see an odd value for `f`. This invariant is checked in the next line. Finally, we perform another read of `f`, although the result is not used.

Assume now that we apply EXTRACT TEMP to both reads of `f` in method `m1`, extracting them into a new local variable `n`. In a sequential setting, the program on the right is an entirely valid result of this refactoring, since `f` is not modified in between the two reads.

Note, however, that as a byproduct of our refactoring, the reads of `f` have been moved out of the **`synchronized`** blocks in order for the local variable to be visible at both uses (the only reason for including the extra read of `f` in the input program is to provoke this behaviour). Thus, `m2` no longer executes atomically with respect to the single remaining read of `f`, which may now see an odd value: the behaviour of the program has changed.

```
import static System.out;

class A {
  volatile boolean a = false;
  volatile boolean b = false;
  public void m1() {
    boolean x = (a = true);
    while (!b);
    if (x);
    out.println("m1 finished");
  }
  public void m2() {
    while (!a);
    b = true;
    out.println("m2 finished");
  }
}
```

⇏

```
import static System.out;

class A {
  volatile boolean a = false;
  volatile boolean b = false;
  public void m1() {

    while (!b);
    if ((a = true));
    out.println("m1 finished");
  }
  public void m2() {
    while (!a);
    b = true;
    out.println("m2 finished");
  }
}
```

Figure 3.11: INLINE TEMP on concurrent code

As another example, consider the program in Figure 3.11 on the left. This time, there are no `synchronized` blocks: the program instead relies only on two `volatile` fields `a` and `b` to synchronise between threads. The Java Memory Model guarantees that after a write of a `volatile` field by one thread, any other thread that reads the field later sees all memory updates performed by the first thread, up to and including the volatile write.[3] Thus a volatile field can be used as a flag by which one thread notifies another about the completion of a task or the availability of data.

Assume now that `m1` and `m2` are executed concurrently. The thread executing `m1` assigns `true` to `a`, and then waits in a tight loop until `b` becomes `true`. The only way for this to happen is for the other thread executing `m2` to perform the assignment to `b`. Meanwhile, the thread executing `m2` begins by spinning until `a` becomes `true`, which happens when the other thread assigns `true` to `a`. Now `b` is set to `true`, and the thread terminates after printing "`m2 finished`". After `b` becomes `true`, the `if`-statement in `m1` is executed and the thread terminates after printing "`m1 finished`". In summary, the execution of the actions by the two threads proceeds in a mostly deterministic order (except that the order in which the print statements execute is not constrained), and always terminates.

On the right, we see an incorrectly refactored version, where `x` has been inlined. Note that refactoring has moved the write of the `volatile` field `a` after the spin-loop in method `m1`. This means that both threads are now executing their spin-loop until the other sets the `volatile` field used in its condition to `true`. Neither thread can make progress, resulting in a livelock, which was not possible in the original version of the program. Incidentally, our description of INLINE TEMP in the preceding section would reject this refactoring because the inlined expression is not pure; it is not hard to see, however, that the proposed refactoring would be behaviour preserving in a sequential setting.

3.2.2 Synchronisation dependencies

The preceding two examples, while artificial for the sake of brevity, show that the sequential concepts of data and control flow are not enough to guarantee behaviour preservation for concurrent code. Even moving

[3] In contrast, writes to non-volatile fields without other synchronisation may appear to happen out-of-order from the perspective of other threads.

expressions without any apparent dependencies can lead to a behaviour change due to interference with another thread. One possible way to avoid this would be to extend the definition of reachable definitions to take inter-thread dependencies into account, but the sheer number of possible interactions that would need to be tracked makes this approach seem impractical.

There is, in both examples above, a more immediate reason for the behaviour change: in both cases, the refactoring reorders concurrency-related constructs. In the first case, a field read is moved out of a `synchronized` block, in the second case a write and a read of two volatile fields are permuted. As we have seen, such reorderings enable new concurrent behaviour and should hence be avoided by the refactoring. We will now introduce *synchronisation dependencies*, which statically capture the possible dynamic ordering of actions relevant to thread synchronisation. Preserving these constraints will ensure that undesirable reorderings cannot happen.

For example, an access to a field is synchronisation dependent on every `synchronized` block in which it is nested or which precedes it, so it will lose a dependency when it is moved out of or in front of one of these blocks. The refactoring engine will compute all synchronisation dependencies of expressions it moves and checks that dependencies are preserved, thereby avoiding bugs like the one above.

Determining what synchronisation dependencies must be modelled and how exactly they must be preserved requires consulting the memory model defining possible concurrent behaviours, in our case the JMM. While the detailed specification of the model is very technical, its main consequences in terms of permissible instruction reorderings are neatly summarised in Lea's "JSR-133 Cookbook" [77], from which we take the matrix in Table 3.1 that specifies forbidden reorderings.

The JMM classifies instructions into several categories, five of which figure in the reordering matrix:

1. A *normal access* is a read or write of a non-volatile shared memory location, i.e., an array element or a field that is not declared `volatile`.
2. A *volatile read* is a read of, and a *volatile write* a write of, a field declared `volatile`.[4]
3. A *monitor enter* is an instruction that acquires a lock; it corresponds to the start of a `synchronized` block or method.
4. A *monitor exit* is an instruction that releases a lock; it corresponds to the end of a `synchronized` block or method.

An instruction from any of these categories that occurs in a particular execution of the program is called a (memory) *action*. Many other instructions, such as reads or writes of local variables or arithmetic operations, are not relevant to the memory model and do not give rise to actions.

The matrix specifies under which conditions an action can be reordered with an action that follows it in some execution. Each cell corresponds to a situation where an action of the kind indicated by the row label is followed (not necessarily immediately) by an action of the kind indicated by the column label. If the cell is labelled $\times$, these two instructions cannot in general be reordered.[5]

For example, the $\times$ in the first column of the third row indicates that a monitor enter cannot be permuted with a subsequent normal access, which at source level would correspond to moving a normal access out of a `synchronized` block: precisely the kind of situation that led to the behaviour change in the first example

[4] Array elements cannot be declared volatile in Java.

[5] These restrictions are chosen from a pragmatic perspective, presupposing only moderate analysis capabilities, and hence are slightly conservative. For instance, a very sophisticated global analysis may be able to prove that a volatile field is only accessible from a single thread, and can hence be treated like a normal field [77].

	Normal Access	Volatile Read	Monitor Enter	Volatile Write	Monitor Exit
Normal Access				×	×
Volatile Read	×	×	×	×	×
Monitor Enter	×	×	×	×	×
Volatile Write		×	×	×	×
Monitor Exit		×	×	×	×

Table 3.1: JMM Reordering Matrix: each cell corresponds to an action of the kind given by the row label being followed by an action of the kind given by the column label; × indicates a forbidden reordering.

above. On the other hand, the blank cell in the third column of the first row indicates that the reverse permutation is acceptable: normal accesses can be moved *into* `synchronized` blocks, provided that no other constraints such as data dependencies prohibit it.

Table 3.1 shows that as far as reordering is concerned, volatile reads behave the same as monitor enters, and volatile writes the same as monitor exits. The former two kinds of actions are collectively referred to as *acquire actions*, and the latter two as *release actions*. Both acquire actions and release actions are *synchronisation actions*, but normal accesses are not. Hence, the matrix can be summarised as stating that

1. a normal access cannot be moved after a release action;
2. a normal access cannot be moved before an acquire action;
3. synchronisation actions are not to be reordered.

To model these requirements, we define two kinds of synchronisation dependencies for nodes in the program's control flow graph:

- A CFG node b has an *acquire dependency* on a node a if a corresponds to an acquire action and there is a path from a to b in the CFG.
- A CFG node a has a *release dependency* on a node b if b corresponds to a release action and there is a path from a to b in the CFG.

In terms of these dependencies, Table 3.1 implies that during the reorderings performed by a refactoring,

1. a node corresponding to a normal access may never lose acquire dependencies,
2. a node corresponding to a normal access may never lose release dependencies,
3. a node corresponding to a synchronisation action may never gain acquire or release dependencies.

Note that on the other hand the matrix *allows*, e.g., a normal access to be moved into a `synchronized` block, so normal accesses are generally allowed to gain synchronisation dependencies.

The matrix does not mention two other kinds of actions defined by the JMM: external actions and thread management actions. The former category comprises actions that interact with the program's environment (such as input/output), whereas the latter represents the thread management methods from the Java standard library's `Thread` class. External actions do not require any special treatment. To ensure that no action is ever reordered with a thread management action, we introduce a third kind of synchronisation dependency: a node a has a *thread management dependency* on any node b that corresponds to a thread management action and is reachable from it in the CFG. We require that

4. a node corresponding to any memory action may never gain or lose a thread management dependency.

Synchronisation dependencies are easy to compute once we have a control flow graph of the program to be refactored, in particular since they do not require any form of alias analysis. For example, a normal access has an acquire dependency on *any* preceding volatile read, no matter which field the read refers to.

Roughly speaking, any sequential refactoring can be made safe for concurrent programs by tracking synchronisation dependencies: the refactoring simply needs to compute all synchronisation dependencies before and after the transformation, and ensure that they have been preserved as described above. If any violation is detected, the refactoring should be aborted. Implementations of EXTRACT TEMP and INLINE TEMP updated in this way will then reject the two example refactorings seen earlier: in Figure 3.10, the read of `f` loses an acquire dependency; in Figure 3.11, the read of `b` gains a release dependency on the write of `a`.

But note that our refactorings do more than just reordering code: EXTRACT TEMP replaces multiple identical expressions with an access to the same new local, so it can possibly decrease the number of field accesses. Similarly, applying INLINE TEMP may duplicate an expression, thereby replacing one field access with many. While the reordering matrix suggests that this is not a problem as long as dependencies are preserved, we want to examine the meaning of our dependencies in the more formal setting of the memory model, to arrive at strong correctness guarantee even for refactorings that go beyond code permutations.

3.2.3 Java Memory Model basics

Before formalising our refactorings, we first recall some central concepts of the Java Memory Model [48, §17]. The JMM abstracts away from the concrete syntactic structure of programs, instead considering a program to be given as the (possibly infinite) set of its threads, and each thread as a set of memory traces representing possible executions. A memory trace is a list of actions paired up with their value, i.e., the value read or written by a normal or volatile access. These traces are required to obey *intra-thread semantics* in the sense that they correspond to executions of threads in isolation, except that reads of shared locations (fields or array elements) may yield arbitrary values to account for interaction between threads.

The set of memory traces for a thread is an over-approximation of the behaviour it may actually exhibit when run in parallel with other threads. The JMM defines the notion of an *execution*, which chooses a particular trace for every thread and relates their actions in three ways. The first and simplest relation is the *program order* $\leq_{\text{po}}$, which reflects the intra-thread ordering of actions, and is hence determined by the choice of traces. The program order never relates actions from different threads. Second, the execution defines a global total order $\leq_{\text{so}}$ on all synchronisation actions in the traces, known as the *synchronisation order*. For synchronisation actions occurring within the same thread, this order has to be consistent with $\leq_{\text{po}}$. Finally, the execution assigns to every read action r a corresponding write action $W(r)$ on the same field or array element, requiring that the value seen by read r is the value written by write $W(r)$.

Based on the program order $\leq_{\text{po}}$ and the synchronisation order $\leq_{\text{so}}$ of an execution, two additional orders are defined. The *synchronises-with* order $\leq_{\text{sw}}$ relates a release action r to an acquire action q if they correspond (i.e., either r is a write of a volatile field v which is read in q, or r exits a monitor m which q enters) and $r \leq_{\text{so}} q$. The *happens-before* order $\leq_{\text{hb}}$ is defined as the transitive closure of $\leq_{\text{po}} \cup \leq_{\text{sw}}$. This means that $a \leq_{\text{hb}} b$ if either (1) $a \leq_{\text{po}} b$, or (2) there is a release action r and an acquire action q such that $a \leq_{\text{po}} r \leq_{\text{sw}} q \leq_{\text{hb}} b$. A *data race* is a pair of accesses to the same shared location, at least one of which is a write, such that these accesses are not ordered by $\leq_{\text{hb}}$.

Finally, the JMM defines a set of *legal executions* for a program, i.e., those behaviours that may actually occur when executing the program. To determine these legal executions, the JMM starts with *well-behaved executions*, which are executions in which every read r sees a most recent write $W(r)$ to the same variable in the happens-before order. To derive a legal execution from a well-behaved execution, one then proceeds to commit data races, i.e., one decides for each race whether the read involved (if any) sees the value written by the write or not. This process can proceed one race at a time or can involve multiple races, and may even be restarted, although committed choices cannot be undone.

In a *correctly synchronised* program, i.e., a program with no data races, all legal executions are well-behaved, and the most recent write occurring before a given read is always uniquely defined. It is perhaps worth pointing out that correctly synchronised programs in this terminology are only required to be free of the low-level data races defined by the JMM. They may still contain higher-level races.

3.2.4 Correctness proofs

The JMM deals with programs in a very abstract and low-level representation that is quite far removed from the Java source code a refactoring actually manipulates. Yet it is this high level of abstraction that allows us to easily establish our first correctness result:

Theorem 1. *Any refactoring that is trace-preserving, i.e., does not alter the set of memory traces of a program, preserves the behaviour of arbitrary concurrent programs: every possible behaviour of the original program is a behaviour of the refactored program and vice versa. This holds even in the presence of data races.*

Proof. This is just a reformulation of a result in [58]. □

Perhaps surprisingly, a great many refactorings are trace-preserving if they are correct for sequential programs, since many source code constructs do not correspond to JMM actions. For example, the memory model has no concept of classes or methods, so refactorings that reorganise the program at this level are trace-preserving, among them PULL UP METHOD, PUSH DOWN METHOD, MOVE METHOD, and type-based refactorings such as INFER GENERIC TYPE ARGUMENTS. The model does not deal with names either, so none of the RENAME refactorings becomes any more complicated in a concurrent setting.

The JMM also does not model method calls (in a sense, method calls are always inlined in traces), so the refactorings EXTRACT METHOD, INLINE METHOD, and ENCAPSULATE FIELD are all trace-preserving.

Two important refactorings that are not trace-preserving in general are INLINE TEMP and EXTRACT TEMP, since they may reorder field accesses, as seen above. Note, however, that if these two refactorings are applied to expressions that do not involve field accesses or method calls (e.g., arithmetic expressions on local variables), they again become "invisible" to the memory model, and Theorem 1 guarantees their correctness on all programs.

Thus the JMM concept of traces and memory actions gives us a convenient criterion to approximate whether a refactoring is affected by concurrency at all.

For non-trace-preserving refactorings, we can pursue two directions: we can identify further subclasses of refactorings for which general results can be proved, or we can tackle the refactorings one by one to prove that their sequential implementation can be updated to preserve behaviour on concurrent programs.

Exploring the former approach first, we note that among those refactorings that do in fact alter the set of memory traces a program yields, most do not actually remove any code from the refactored program (at

least not code that corresponds to memory actions), but merely rearrange it. This might entail reordering statements or expressions, or merging pieces of code that do the same thing.

On the level of the JMM, we describe such transformations as follows:

Definition 3. *A* restructuring transformation *is a partial function* ρ *on programs such that for every program* $P \in \mathrm{dom}(\rho)$ *and every execution* E' *of* $\rho(P)$ *there is an execution* E *of* P *and a mapping* f *from actions in* E *to actions of the same kind in* E'*. Also, this mapping does not map actions belonging to the same thread in* E *to different threads in* E'*.*

Intuitively, for every execution of the transformed program $\rho(P)$ we can find a corresponding execution of the original program P. We do not require that this execution has the same behaviour in any sense, but just that there is a mapping between their actions which shows that no actions of the old program have been lost, even though new actions may have been introduced.

Most importantly, however, the kinds of all actions need to be preserved. That means, in particular, that field accesses have to refer to the same fields and read or write the same values, and monitor operations have to handle the same locks.

Given this very liberal specification, it is impossible to prove that such a transformation preserves behaviour. Instead, we will show that a restructuring transformation cannot introduce new data races or new deadlocks between existing actions if it respects the synchronisation dependencies introduced in the previous section.

Definition 4. *A restructuring transformation is said to* respect synchronisation dependencies *if its mapping* f *fulfils the following three conditions for all actions* a, b*.*

1. *If* $a \leq_{\mathrm{so}} b$*, then also* $f(a) \leq'_{\mathrm{so}} f(b)$*.*
2. *If* a *is an acquire action and* $a \leq_{\mathrm{po}} b$*, then also* $f(a) \leq'_{\mathrm{po}} f(b)$*.*
3. *If* b *is a release action and* $a \leq_{\mathrm{po}} b$*, then also* $f(a) \leq'_{\mathrm{po}} f(b)$*.*

Since $\leq_{\mathrm{so}}$ is a total order, the first requirement says that f cannot swap the order of synchronisation actions, whereas the second and third requirements prohibit reordering normal accesses to appear before acquire actions or after release actions. Note that this is just a formalisation of the synchronisation dependencies introduced before.[6]

We first establish a slightly technical result.

Lemma 2. *Let a synchronisation dependency respecting restructuring be given, and let* a *and* b *be actions. If* $a \leq_{\mathrm{hb}} b$*, then either* $f(b) \leq'_{\mathrm{hb}} f(a)$ *or* $f(a) \leq'_{\mathrm{hb}} f(b)$*.*

Proof. We first treat the case where a is an acquire action. In this case, we can actually prove that $a \leq_{\mathrm{hb}} b$ implies $f(a) \leq'_{\mathrm{hb}} f(b)$ by induction on $\leq_{\mathrm{hb}}$:

If $a \leq_{\mathrm{po}} b$, then $f(a) \leq'_{\mathrm{po}} f(b)$, and hence $f(a) \leq'_{\mathrm{hb}} f(b)$, by Definition 4. Otherwise we have a release action l and an acquire action q such that $a \leq_{\mathrm{po}} l \leq_{\mathrm{sw}} q \leq_{\mathrm{hb}} b$. As before, this means that $f(a) \leq'_{\mathrm{po}} f(l)$; since f preserves action kinds and $\leq_{\mathrm{so}}$ we have $f(l) \leq'_{\mathrm{sw}} f(q)$, and finally $f(q) \leq'_{\mathrm{hb}} f(b)$ by induction hypothesis. Together this again shows $f(a) \leq'_{\mathrm{hb}} f(b)$.

Now consider the general case where a is not necessarily an acquire action. If $a \leq_{\mathrm{po}} b$, then $f(a) \leq'_{\mathrm{po}} f(b)$ or $f(b) \leq'_{\mathrm{po}} f(a)$, since f does not map actions across threads. Otherwise, $a \leq_{\mathrm{po}} l \leq_{\mathrm{sw}} q \leq_{\mathrm{hb}} b$ for some

[6] For brevity, we mostly ignore thread management actions in this section, but all results can easily be extended to cover them as well.

release action l and an acquire action q. But by Definition 4 this gives $f(a) \leq'_{\text{po}} f(l)$. As above we see $f(l) \leq'_{\text{sw}} f(q)$, and $f(q) \leq'_{\text{hb}} f(b)$ follows since q is an acquire action. In summary, we get $f(a) \leq'_{\text{hb}} f(b)$, establishing the claim. □

Now our first result follows effortlessly:

Theorem 2. *If there is a data race between two actions $f(a)$ and $f(b)$ in execution E', then there is already a data race between a and b in E.*

Proof. Follows directly from the previous lemma and the definition of a data race. □

This result ensures that a synchronisation respecting restructuring can never introduce a new data race between two actions carried over from the original program, although there may well be a data race involving actions introduced by the transformation.

We immediately gain an important corollary:

Corollary 1. *A restructuring transformation that does not introduce any new actions will map correctly synchronised (i.e., data race free) programs to correctly synchronised programs.*

A similar result can be established for deadlocks.

Lemma 3. *If there is a deadlock in an execution E' of $\rho(P)$ caused by locking actions in* $\text{rng}(f)$*, then the same deadlock occurs in E.*

Proof. We sketch the proof for the case of two deadlocking threads. In that case, there are threads ϑ and ϑ' and monitor enter actions l_1, l'_1, l_2, l'_2. Writing $L(l_1)$ for the lock acquired by action l_1 and similarly for the others, we require $L(l_1) = L(l'_1)$ and $L(l_2) = L(l'_2)$; actions l_1 and l_2 belong to ϑ, whereas l'_1 and l'_2 belong to ϑ'.

By the definition of f, the same must be true of $f(l_1), f(l'_1), f(l_2), f(l'_2)$. In order for ϑ and ϑ' to deadlock, we must have $f(l_1) \leq_{\text{po}} f(l_2)$, $f(l'_2) \leq_{\text{po}} f(l'_1)$, $f(l_1) \leq_{\text{so}} f(l'_1)$, and $f(l'_2) \leq_{\text{so}} f(l_2)$. By definition of f, the same is true of l_1, l'_1, l_2, l'_2 due to the totality of $\leq_{\text{so}}$ over all synchronisation actions, the fact that $\leq_{\text{po}}$ and $\leq_{\text{so}}$ are consistent, and the first point of Definition 4. □

Again, this proves that the transformation cannot introduce a deadlock only involving actions from the original program, but does not preclude the existence of deadlocks involving newly introduced actions.

The above two results establish a certain baseline. They apply to a wide range of refactorings (and indeed non-behaviour-preserving transformations), but only guarantee very basic properties.

We conclude this subsection by establishing the correctness of our two example refactorings on correctly synchronised programs. Programs with races will be discussed below.

Theorem 3. *The refactorings* EXTRACT TEMP *and* INLINE TEMP *preserve the behaviour of correctly synchronised programs if they preserve the behaviour of sequential programs and respect synchronisation dependencies.*

Proof Outline. For EXTRACT TEMP, note that if we extract copies $e_1, \ldots, e_n$ of an expression e into a local variable, there cannot be any acquire or thread management actions between the individual copies, since the refactoring needs to preserve synchronisation dependencies. In a well-behaved execution this means that the read actions in all copies see the same values, hence correctness for sequential programs ensures behaviour preservation.

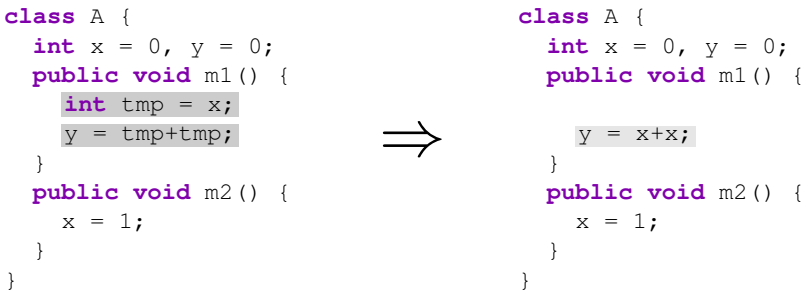


Figure 3.12: INLINE TEMP in the presence of a possible data race.

For INLINE TEMP, the refactoring may move field reads over acquire actions, which would seem to make it possible for them to see values written by different field writes than before the refactoring. For correctly synchronised programs, however, this is not possible, since the read must already have been preceded by corresponding acquire actions in the original program to prevent data races. So again read actions will see the same values as in the original program, and sequential correctness ensures behaviour preservation. □

To make the above argument precise, we would need a formalisation of the correspondence between source level Java programs and programs as they are viewed by the JMM. Such an undertaking is beyond the scope of this work, so we content ourselves with this intuitive argument.

3.2.5 Handling programs with races

Theorem 3 only states that EXTRACT TEMP and INLINE TEMP are correct for programs without data races. For programs *with* data races, it is possible for these refactorings to remove or introduce concurrent behaviours. For example, consider Figure 3.12, where INLINE TEMP is applied to `tmp` in `m1`. If `m1` and `m2` can run concurrently, there is a possible data race on the `x` field. In the original program, `m1` can assign either `0` or `2` to field `y`, depending on when `m2` executes. In the modified program, `x` is read twice, enabling the new behaviour of `m1` assigning `1`.

The transformation in the example is, in fact, behaviour preserving if the methods are never invoked concurrently. Hence the correctness of the refactoring depends on whether `m1` and `m2` can execute in parallel. Unfortunately, for Java programs, determining what code can execute in parallel requires expensive whole-program analysis [96]. Hence it seems impractical to require refactorings to preserve the concurrent behaviour of programs with races while still enabling standard transformations.

Apart from these pragmatic considerations, there are some technical details of the JMM specification that make it unlikely that the formal correctness result can be extended to hold on programs with data races. Concretely, our definition of dependency edges allows INLINE TEMP to move accesses into a **`synchronized`** block, a transformation that is known under the somewhat whimsical name “roach motel reordering” in the literature [87]. Similarly, we allow EXTRACT TEMP to perform a redundant read elimination. While the authors of the JMM originally claimed that these two transformations are formally behaviour preserving [87], later research has shown that this is not the case [22, 58]. Arguably this indicates a fault with the specification of the JMM, but it remains an open problem whether the model can be changed to accommodate these and similar transformations.

Our definition of synchronisation dependencies is overly cautious with respect to final fields: the JMM allows reads of final fields to be moved across synchronisation constructs with much greater freedom than normal field reads, essentially requiring that the value of a final field is immediately visible to all threads once an object's constructor has finished executing. Treating them like normal fields as we do is, of course, safe, but leaves some room for improvement.

A rather peculiar feature of the JMM is that it allows non-atomic reads and writes of (64-bit) `double` and `long` values. More precisely, a read or write of a non-`volatile` variable or array element declared to be of either type can be performed as two 32-bit non-atomic reads or writes by the virtual machine. This does not concern refactorings, however: even if the access is split in two, there is still only a single access at the source level, so there is no danger of independently reordering the non-atomic accesses.

3.2.6 Implementation considerations

Using the control flow framework introduced earlier, it is quite easy to compute synchronisation dependencies for control flow nodes, in particular because no alias analysis needs to be done.

Since our analysis is intra-procedural, it treats method calls as thread management actions, forestalling any reordering across such calls. To approximate calling contexts, the start node of every method also counts as an acquire action, and the end node as a release action. Of course, dependency computation is only required for refactorings like INLINE TEMP and EXTRACT TEMP that alter memory traces; the many refactorings that do not affect memory traces require no changes (see Theorem 1).

As with name binding, synchronisation dependencies are exposed through the by now familiar locking metaphor: a refactoring that needs to preserve the synchronisation behaviour of a piece of code can call into the framework to lock all synchronisation dependencies of the code in question, and later unlock them. Unlocking these dependencies simply means checking whether all dependencies have been preserved in the expected way, aborting and undoing the refactoring if this turns out not to be the case.

The conservative treatment of method calls means that we could potentially introduce many spurious synchronisation dependencies, which could prevent many harmless refactorings from going through. In particular, we can extract and inline expressions involving method calls only in very specific circumstances. The same kind of problem, however, arises with data flow analysis, where our current implementation also makes conservative assumptions about data dependencies or side effects of method calls. For refactoring engines adopting a more sophisticated data flow analysis, it would be easy to extend this analysis to also gather information about the synchronisation actions that could be triggered by a method call.

3.2.7 Related work

This section is based on the author's paper "Concurrent Refactoring of Concurrent Java Code" [115] published at ECOOP 2010, which is joint work with Julian Dolby, Manu Sridharan, Emina Torlak and Frank Tip of IBM Research. The key ideas presented here (synchronisation dependencies and their preservation conditions) are the author's, although discussions with the co-authors were invaluable for their development and clarification. The correctness proofs were done for the most part by the author, with some help from Emina Torlak. The implementation is the author's work; Manu Sridharan conducted a feasibility study for a more sophisticated analysis of the synchronisation behaviour of method calls, which is included in the publication, but has been omitted from this presentation.

To the best of the author's knowledge, this is the first work to offer a systematic treatment of the refactoring of concurrent Java code. In the literature, issues arising from concurrent execution are usually not considered. When they are (as in [65]), the authors usually strengthen their preconditions to prevent refactoring code that looks like it might be run in a concurrent setting. In particular, there does not seem to be any previous work on relating refactorings to the Java Memory Model.

There has, however, been a lot of interest in refactoring programs to enhance their concurrent behaviour. The REENTRANCER [140] tool transforms programs to be reentrant, enabling safe parallel execution. The CONCURRENCER tool [34] aims to adapt sequential Java code to use concurrent libraries, and the RELOOPER tool [35] reorganises loops to execute in parallel. In contrast to our work, these papers propose new refactorings specifically tailored to a concurrent setting.

Going beyond Java, some preliminary research has been done on refactorings for X10 [23], a Java-based language with sophisticated concurrency support, and has yielded promising first results [89].

Java compilers are generally very cautious about optimising concurrent code. While we could not find any published work on the optimisations performed by recent systems, it appears that previous versions of the Jikes virtual machine's just-in-time compiler utilised a notion of synchronisation dependency edges not unlike the one we use in this paper to prevent code motion of memory operations across synchronisation points [19]. Their dependencies would appear to be more restrictive than ours (forbidding, for instance, roach motel reordering), and they are based on the pre-Java 5 memory model. Also recall that for practicality, we allow some non-behaviour preserving transformations for programs with data races which clearly must be disallowed in a compiler.

There has been some work in the slicing community on slicing concurrent programs. For this purpose, several authors have proposed new dependencies to complement the classic control and data dependencies in analysing concurrent programs. Cheng [24] proposes three new kinds of dependencies: selection dependency to model control dependencies arising from non-deterministic selection, synchronisation dependency to model synchronisation between processes[7], and communication dependency to model inter-process communication.

Krinke [73] instead introduces interference dependencies that model the interaction between threads due to the use of shared variables; no synchronisation constructs are considered. In particular, it seems that such dependencies would have to take data races and all their possible outcomes into account, which would make them unsuitable for our purposes.

Both authors treat the problem at a fairly abstract level. Zhao [141] considers the problem of computing the dependencies proposed by Cheng for Java programs. His approach does not seem to be directly based on the Java Memory Model, though, and in particular does not handle volatile accesses.

[7] Despite the name, this is a very different concept from the synchronisation dependencies introduced in this work.

Chapter 4

Language Restrictions and Extensions

In the previous two chapters we have seen how static semantic properties of programs such as their name binding structure or their data flow play a central role in making refactorings reliable and correct. By formulating refactorings in terms of the properties they ought to preserve we obtain high-level, effectively checkable criteria for ensuring behaviour preservation. While the precise way in which static semantic information is computed and checked may differ between languages, the concepts involved transcend individual languages.

This opens up an appealing prospect for making refactorings generic: if refactorings can simply be formulated in terms of static semantic dependencies, then their implementation could abstract away from the syntax of the object language. All the refactoring needs to know is how to compute the appropriate dependencies of every language construct, the rest would then be language independent.

Alas, this level of genericity seems unattainable in practice. Syntactic peculiarities of the object language become eminently important when we want to describe how the refactoring transforms its input program, since the output program must, of course, be well-formed. When a refactoring rearranges expressions or statements, it may be enough from a semantic perspective to check that certain control and data flow dependencies are preserved, but real-world languages often have fairly idiosyncratic rules about where certain kinds of expressions or statements are allowed to occur. In addition, they often provide syntactic sugar that abbreviates commonly occurring idioms, but in turn may introduce static semantic dependencies that are not immediately obvious from the program text.

This chapter proposes a way to deal with such problems. While it seems unrealistic to hope for a completely generic formulation of even very simple refactorings, we will at least demonstrate how to encapsulate the treatment of troublesome language features in Java in a modular and reusable way, so they can be dealt with once and for all.

The basic idea is very simple: if it is the Java language that troubles us, we simply change it. While our refactorings will still accept arbitrary Java programs as input and produce well-formed Java programs as output, they internally work on a language that is more suitable to their purpose: for example, it is often easier to restrict the input language for a refactoring by desugaring complex constructs to make them easier to treat; conversely, it is sometimes useful to introduce language extensions in intermediate steps so as to make the refactoring easier to formulate, or easier to structure.

However, this process must be transparent to the user: all extensions must ultimately be eliminated or translated away in the output program, and syntactic sugar should be reintroduced where possible. As we shall see, the same language restriction or extension can be useful for more than one refactoring. These

```
class A {
  synchronized int f() {

    return 23;

  }

  static synchronized int g() {

    return 42;

  }
}
```

≡

```
class A {
  int f() {
    synchronized(this) {
      return 23;
    }
  }

  static int g() {
    synchronized(A.class) {
      return 42;
    }
  }
}
```

Figure 4.1: Desugaring synchronised methods

refactorings can then share common code for enforcing restrictions or translating away extensions, which makes them more modular and more reusable.

We have already encountered a very useful language extension: locked names. While they should never occur in the output program, many refactorings benefit from being able to lock an existing name to its declaration or to introduce a new locked name that is guaranteed to bind to a given declaration. The details of how to replace a locked name by a normal name are handled by the general naming framework and need not concern the implementer of any particular refactoring. In the same way, locked overriding and flow and synchronisation dependencies can be understood as language extensions, albeit with a very simple elimination procedure: if the dependencies are intact, the code stays unchanged; if they are violated, the refactoring is aborted.

In this chapter, we will first discuss some examples of language restrictions and show how they can be enforced on input programs. The second half of the chapter discusses language extensions, in particular the language extension of *anonymous methods*, and shows how it helps to structure the implementation of the EXTRACT METHOD refactoring, making it possible to understand this fairly complex refactoring as the composition of several small-scale refactorings. The next chapter will then present the results of a major case study, showing that our approach of formulating refactorings in terms of the dependencies they need to preserve and of simplifying and modularising their description with the help of language restrictions and extensions is powerful enough to give succinct specifications and robust implementations of just about any refactoring.

4.1 Language restrictions

A typical example of the usefulness of language restrictions is the `synchronized` method modifier. Recall that an instance method that is declared `synchronized` acquires the monitor lock of its receiver object whenever it is called, and releases the lock on return; similarly, a `synchronized static` method acquires the lock of the class object of the class in which it is declared. Thus we can see the `synchronized` modifier simply as syntactic sugar for a `synchronized` block around its body, as shown in Figure 4.1.

This simple transformation makes many refactorings much easier to express. Consider, for example, the PULL UP METHOD refactoring, which moves a method from a child class to its superclass [45]. If the method to be moved is `static synchronized`, naïvely moving it to the superclass would change the lock that is

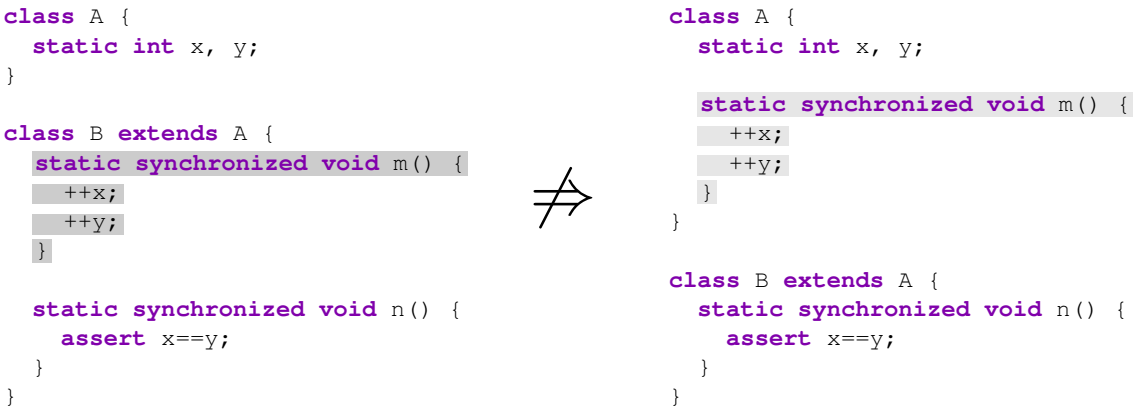


Figure 4.2: Problematic example of PULL UP METHOD

being acquired, which could change the concurrent behaviour of the program. Consider, for example, the program in Figure 4.2 on the left. Methods `m` and `n` both synchronise on `B.class`, so they can never execute concurrently, and `n` can never observe `x` and `y` to have different values. If `m` is pulled up to class `A` without adjusting synchronisation, it now synchronises on `A.class`, so there is no longer mutual exclusion between the two methods, and the assertion in method `n` may fail.

Instead of extending the implementation of PULL UP METHOD with code that caters to this special case, we prefer to specify this refactoring on the restricted language of Java without `synchronized` modifiers: as the very first step of the refactoring, `synchronized` modifiers are desugared as shown in Figure 4.1, marking the created `synchronized` block as stemming from a modifier. Then the refactoring can perform its transformations without having to provide special treatment for `synchronized` methods. Finally, any `synchronized` block that was marked in the first step is examined to see whether it can be "re-sugared" into a modifier. For example, this is possible when pulling up instance methods, since the acquired lock is the receiver object, which does not change during the refactoring.

Our refactoring engine makes this desugaring step available for refactoring implementations, but for performance reasons we generally refrain from desugaring *all* `synchronized` modifiers: for PULL UP METHOD, it is certainly enough to desugar the modifier of the method on which the refactoring is applied, if any.

4.2 Language extensions

It is easy to imagine that language restrictions can ease the implementation of refactorings by eliminating the need to handle special constructs. What is maybe more unexpected is that *language extensions* can provide similar benefits. In addition, language extensions often make it possible to break down more complicated refactorings into smaller *microrefactorings* that each perform one particular step of the overall transformation. Every microrefactoring is itself a behaviour preserving operation that can be understood, specified, implemented and tested in isolation. Often, microrefactorings can even be reused between different (macro-)refactorings.

We demonstrate the advantages of language extensions and microrefactorings on the EXTRACT METHOD refactoring as our running example. While an implementation of EXTRACT METHOD is part of most modern

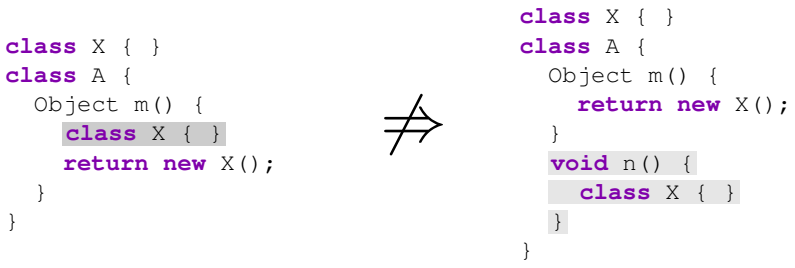


Figure 4.3: Changed name binding during method extraction

industrial-strength refactoring engines, these implementations are sometimes not very reliable and quite regularly fail to ensure that the refactored program is behaviourally equivalent to the original program. We show that our approach leads to a clean and modular implementation that correctly handles many cases where other implementations fail.

4.2.1 Challenges

Let us begin by discussing some of the challenges that an implementation of EXTRACT METHOD has to overcome.

Name binding issues

Since the refactoring introduces new names and declarations into the program, care has to be taken not to accidentally change existing name bindings. Take, for example, the program in Figure 4.3 on the left; the method `m` in class `A` constructs an instance of the locally declared class `X` and returns it.

In NetBeans, extracting the declaration of `X` to a method `n` yields the program on the right, where the instance constructed is no longer of the local class `X`, but of the global class of the same name. This particular program does not change its behaviour, but slightly extended examples either make NetBeans produce an output program that does not compile (which is annoying to the user, if comparatively harmless), or that still does compile but behaves differently.

Other IDEs employ simple heuristics to guard against this kind of situation, but no systematic binding preservation seems to be attempted, and similar bugs can be discovered in all cases [118].

Control flow issues

While Java mostly uses structured control flow, it also provides the unstructured branching statements **`break`** and **`continue`**. The former exits from an enclosing loop, which can be further specified by a label, while the latter only exits from the current iteration and starts the next one. These "**`goto`**s in disguise" cannot be moved into the newly created method blindly: if their target loop is not also moved, the resulting program will not compile.

Even more subtle is the **`return`** statement: If types match, an extracted method with an embedded return will still compile, but of course the statement now returns from the *extracted* method, not the original method as it did before. As an example, consider the program in Figure 4.4 on the left, and assume we want to extract the **`if`** statement into a method `n`. A naïve implementation might produce the program on the right, which

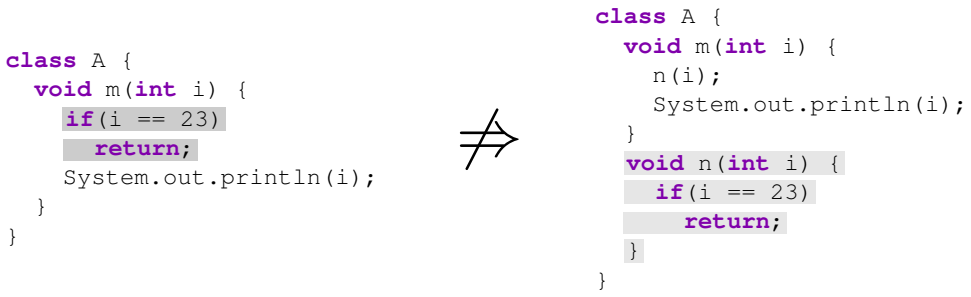
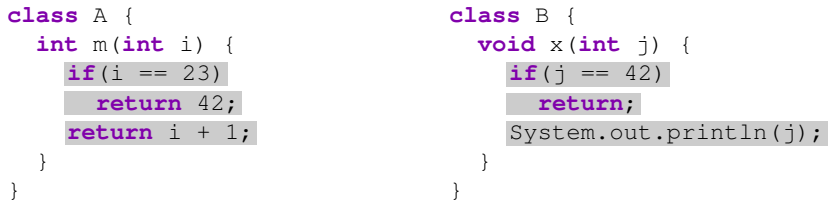


Figure 4.4: Changed control flow during method extraction



(a) (b)

Figure 4.5: Preservation of control flow during method extraction

has different behaviour from the original program: while originally calling `m(23)` would return immediately, it now prints `23`.

A precondition-based approach might categorically forbid extraction of any code that contains these branching statements, but that would reject many potentially useful refactorings: As a simple example, consider the program in Figure 4.5 on the left, and assume we want to extract the whole body of `m` into a new method `n`. This can easily be done if, instead of replacing the extracted code by the method invocation `n(i)`, we instead replace it by `return n(i)`.

Thus a more advanced precondition might be to allow extraction if all control paths end in a `return` statement. But this is again too stringent a requirement, as the example program in the same figure on the right shows. Our preconditions would not allow us to extract the whole body of `x` into a new method `y`, although this would actually be unproblematic, since the code to be extracted is right at the end of the enclosing method anyway.

Relaxing the preconditions further to allow this refactoring as well is certainly possible, but there is always a danger of accidentally introducing unsoundness this way, especially if new language versions introduce constructs that influence control flow, such as the `assert` statement in Java 1.4.

A natural alternative is to directly check whether control flow is preserved: Before extracting statements into a method, compute their control flow successors; recompute afterwards and make sure that every CFG node still has the same successors. This immediately rules out extracting `break` or `continue` statements without their target loop, since their control flow successors will no longer be defined at all.

As it stands, however, the requirement for precise preservation of successor nodes is too strong. Recall that in the examples in Figure 4.5 we would insert a `return` statement around the call to the extracted method. This new `return` node will show up in the successor set of final control flow nodes of the extracted method, and hence violate the requirement. To allow this kind of "fix up", we identify some nodes as being *flow-through* nodes whose only purpose it is to transfer control to another node without any other semantic

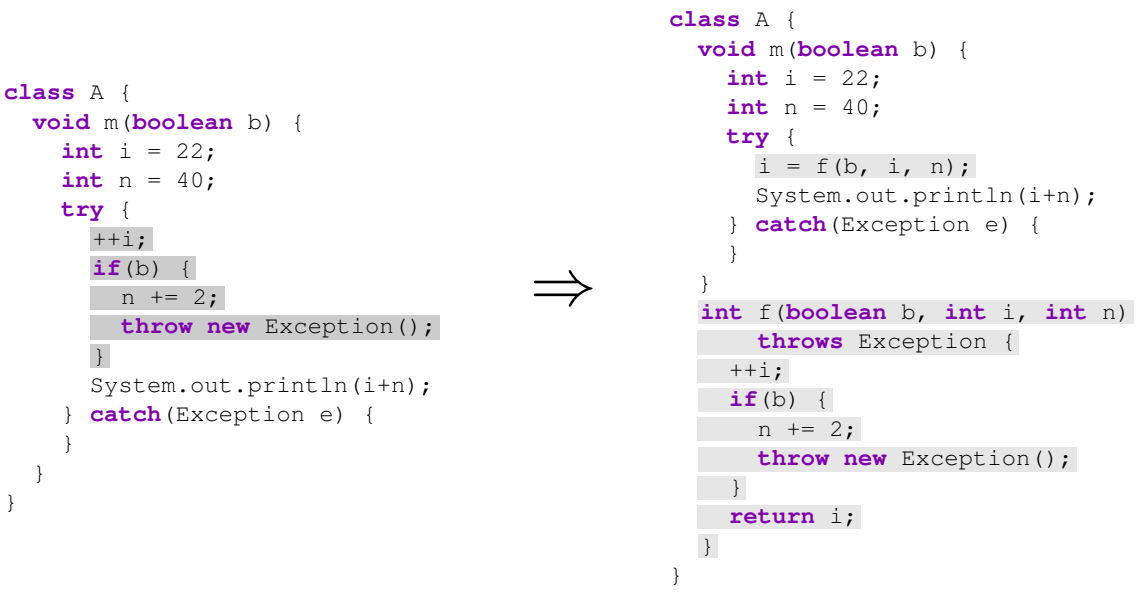


Figure 4.6: A refactoring rejected by Eclipse

effect: in Java, the flow-through nodes are precisely the nodes corresponding to **break**, **continue**, **return** and **assert** statements (whereas the expression that is being returned or the truth of which is asserted is, of course, not flow-through). We can then relax our requirement of preserving the set of successor nodes to allow preservation modulo flow-through nodes.

Data flow issues

One of the most intricate aspects of the EXTRACT METHOD refactoring is to determine the parameters of the extracted method. Intuitively, it is clear that we need to pass the new method the values of any local variables it might need, and that the new method in turn should return the values of any local variables it has changed, if they are needed for further computation.

We can make this intuition precise using the data flow concepts introduced in Chapter 3: a variable should become a parameter to the extracted method if it has a use within the code to be extracted whose reaching definition lies before the extracted selection, and it should be returned if there is a definition which is the reaching definition for some use that comes after the selection. This is a fairly natural criterion, since the flow edges between these nodes would be "broken" by method extraction; they have to be rerouted through parameters in order to be preserved.

As an example, assume that in the program in Figure 4.6 on the left we want to extract the highlighted code. We note that all of `b`, `n`, and `i` have to become parameters to the new method, and `i` should be returned so that its new value is available for the `println` statement, as shown in the program on the right.

The value of `n`, however, does *not* need to be returned: if `n` is changed by the extracted code, an exception is immediately thrown and control transfers to the **catch** clause; in other words, the assignment to `n` has no reached use, and hence its value does not need to be returned. Eclipse, for example, does not detect this, and determines that both values need to be returned. This is not easily accomplished in Java, and hence the refactoring is (unnecessarily) rejected.

4.2.2 EXTRACT METHOD in five steps

As shown in the above examples, the main challenge in implementing the EXTRACT METHOD refactoring is the intertwining of name binding, control flow, and data flow, which are all delicate by themselves, and even more so in conjunction.

By using our existing naming framework, we can easily achieve binding preservation where necessary, and we also have the required machinery to compute the required control and data flow information. It is then a natural idea to simplify our task by splitting the refactoring into two parts that deal with control flow and data flow separately. Unfortunately, this is difficult to achieve in plain Java: At some point during the refactoring, the statements to be extracted have to be moved into a new method, and at that point both their control and data flow will change at the same time.

We hence introduce *anonymous methods* into the language as a lightweight extension that helps to break up the transformation. Just as an anonymous class is a class that is defined and instantiated at the same time and may access local variables from the surrounding method, an anonymous method is a method that is defined and invoked at the same time, and (besides its own parameters and local variables) can also access variables from the surrounding method.

In control flow terms it behaves like an ordinary method, in particular `break` and `continue` statements cannot escape the anonymous method. In data flow terms, however, it behaves like a block in that it can access and modify variables from the enclosing scope in a lexically scoped fashion.

Thus an anonymous method provides a convenient half-way point for the EXTRACT METHOD refactoring: If we can package up the statements to be extracted into an anonymous method, it means that we have successfully preserved the control flow. We can then tackle the task of preserving data flow by successively introducing parameters and return values as needed.

To facilitate this process, it will be useful to allow anonymous methods to have *reference parameters*, so that changes to parameters can be propagated back to the surrounding method from which code is being extracted.

Once the anonymous method does not reference any local variables from the surrounding method anymore, reference parameters are rewritten into normal parameters whose value is returned as a result, and the anonymous method is promoted to a normal method.

We thus propose to split the refactoring into the following five steps:

1. EXTRACT BLOCK pulls the statements to be extracted together into a block.

2. INTRODUCE ANONYMOUS METHOD turns that block into an anonymous method without parameters.

3. CLOSE OVER VARIABLES eliminates any references to local variables of the surrounding method from within the anonymous method by introducing parameters. If a variable is not only only read but also written, it is made a reference parameter.

4. ELIMINATE REFERENCE PARAMETERS gets rid of parameters that need to be passed by reference, since this is not supported by Java. Their final value is instead returned and assigned to the corresponding local variable of the surrounding method.

5. LIFT ANONYMOUS METHOD turns the anonymous method into a named method within the same type as the method we are extracting from.

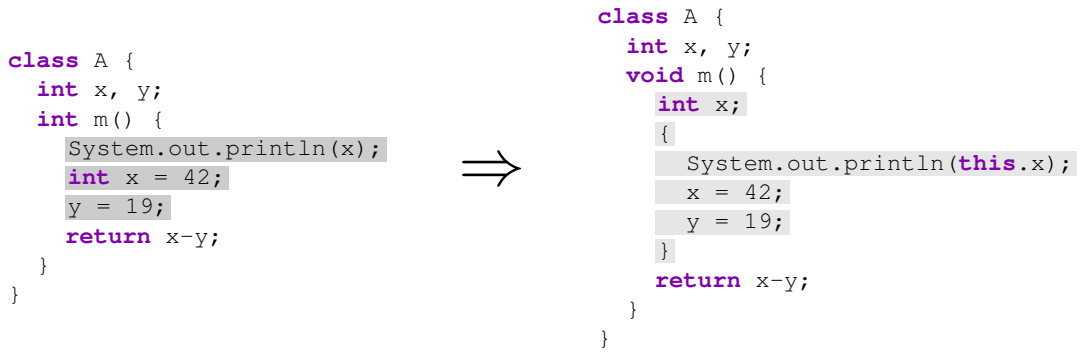


Figure 4.7: Applying the EXTRACT BLOCK microrefactoring

The whole EXTRACT METHOD refactoring is simply a sequential composition of these five microrefactorings, which we now are going to discuss in greater detail.

4.2.3 EXTRACT BLOCK

The EXTRACT BLOCK refactoring takes as its input a block b composed of statements b_0 to b_{n-1} and two indices i and j such that $0 \leq i \leq j < n$. The goal of the refactoring is to put statements b_i to b_j into a new block b' and insert it into b to replace the original statements.

Since blocks do not affect control or data flow, this step is usually very easy to perform. The only complication arises with name binding preservation as shown in Figure 4.7: in the program on the left, we want to wrap the highlighted statements into a block. Note that among them is a declaration of a local variable `x`, which is referenced in the **`return`** statement. This **`return`** statement, however, is not one of the statements to be extracted. For this reason, we have to leave the declaration of `x` outside the block, only moving its initialising assignment. To avoid name capture (as for the reference to field `x` in the example), we lock all names in the statements to be wrapped into the block. The naming framework then will automatically take care of adding qualifiers where necessary. In order for the output program to be syntactically valid, we also cannot allow any of the statements to be extracted to carry a **`case`** or **`default`** label.

4.2.4 INTRODUCE ANONYMOUS METHOD

Next we want to convert the block created in the previous step into an anonymous method without parameters. JastAddJ, of course, has no built-in support for anonymous methods, since they are not part of the Java language. Its extensible implementation, however, makes it very easy to add language extensions.

Support for anonymous methods can be added by providing an additional production for an abstract syntax tree node in the object language:

```
AnonymousMethod : Expr ::= Parameter:ParameterDeclaration*
                           ReturnType:Access Exception:Access* Block Arg:Expr*;
```

In words, this production says that anonymous methods are a kind of expression, i.e., they can occur anywhere the Java language grammar allows an expression to occur. They contain a list of parameters, represented by the same node type as parameters for plain Java methods and constructors; a return type which is an `Access`, i.e. a possibly qualified name; a list of thrown exceptions, likewise given as accesses; a body given as a `Block`; and finally a list of arguments, which may be arbitrary expressions.

We do not specify any parsing rules for anonymous methods, since we do not want to make them available for programmers, but for presentation purposes we use the concrete syntax $(\overline{p} : r \textbf{ throws } \overline{x} \Rightarrow b)(\overline{e})$ to represent an anonymous method with body b that takes parameters $\overline{p}$, is invoked with arguments $\overline{e}$, throws exceptions $\overline{x}$ and has return type r.

Conceptually, the invocation of an anonymous method works like for a normal method: parameters are bound to arguments, the body is executed, and may return a value by executing a `return` statement. Exception handling likewise works as for methods: exceptions thrown but not caught within the anonymous method propagate to the enclosing scope and onwards until a corresponding `catch` clause is found. Like a normal method, an anonymous method may declare and use local variables in addition to its parameters, and it may also access any variable or field visible in the surrounding method, subject to lexical scoping.

No code generation needs to be implemented for anonymous methods, but name lookup rules and control and data flow analysis have to be extended to take them into account. This is simply a matter of adapting the corresponding rules for blocks and methods.

To turn a block into an anonymous method without parameters, we do not need to adjust any data flow or name bindings: all these work the same way for blocks as they do for anonymous methods. We do, however, need to lock the control flow successors of every statement in the block, and unlock it again after the block has been wrapped into an anonymous method in order to make sure successors stay intact.

More precisely, given a block b in a context with return type T[1], we perform the following steps:

1. For every statement in b, lock its set of control flow successors, skipping flow-through nodes.
2. Compute all uncaught checked exceptions thrown in b, and use the naming framework to compute locked accesses $e_1, \ldots, e_n$ for them.
3. Construct an anonymous method C of the form

   `(() : `R` throws `$e_1,\ldots,e_n$` ⇒ `b`)()`

   where R is T if b cannot complete normally[2], or `void` otherwise.
4. If b can complete normally, replace it by C`;`, otherwise by `return` C`;`.
5. Unlock control flow, aborting the refactoring if the flow has changed.

In the second example program of Figure 4.5, for example, the block can complete normally, so we perform the following transformation:

```
class A {                          class A {
  void m(int i) {                    void m(int i) {
    {                                  (() : void ⇒ {
      if(i == 23)                        if(i == 23)
        return;                            return;
      System.out.println(i);             System.out.println(i);
    }                                  })();
  }                                  }
}                                  }
```

[1] That is, T is either the return type of the enclosing method, or it is `void` if b is not inside a method.

[2] That is, if every control flow path through b ends in a control transfer statement like `return`; for the precise definition see the Java Language Specification [48, §14].

In the program on the left, the only control flow successor of both the `return` statement and the `println` statement is the exit node of the enclosing method. The same is true in the refactored program on the right, and since all the other successor statements are likewise preserved, the refactoring can continue.

In the first example program of Figure 4.5, the block *cannot* complete normally (both control paths end in a `return` statement), hence we transform as follows:

```
class A {
  int m(int i) {
    {
      if(i == 23)
        return 42;
      return i + 1;
    }
  }
}
```

```
class A {
  int m(int i) {
    return (() : int ⇒ {
      if(i == 23)
        return 42;
      return i + 1;
    })();
  }
}
```

Again, we can verify that control flow successors have not changed: the newly inserted `return` is a flow-through node, and hence does not disrupt control flow.

4.2.5 CLOSE OVER VARIABLES

Our next task is to reroute data flow edges that go across the boundaries of the anonymous method through parameters. This is akin to the operation of lambda lifting [60] for functional languages.

We want to defer the handling of return values to the next step, so we introduce another language extension in the form of *output* and *reference* parameters, marked with the modifiers `out` and `ref`, respectively. The argument given for such parameters must be a variable of the enclosing scope, and any changes the anonymous method makes to the parameter are reflected in that argument, thus these arguments are (conceptually) passed by reference.

Parameters marked `ref` may be read before they are assigned, which is not possible for `out` parameters. These two kinds of parameters behave similar to their counterparts in C# [57], but they only occur as ephemeral language constructs during refactoring, not as genuine language features. A parameter that is marked neither `out` nor `ref` is called a *value parameter*.

In data flow terms, we need to handle two situations:

- Inside the anonymous method, there could be a read of a local variable `x` such that the witnessing path from one of its reaching definitions crosses the boundaries of the anonymous method. We then say that this read has an *incoming data flow edge*.
- Inside the anonymous method, there could be a write of a local variable `x` such that the witnessing path to one of its reached uses crosses the boundaries of the anonymous method. We then say that this write has an *outgoing data flow edge*.

In the first case, `x` needs to be made a value parameter of the method, in the second case `y` should become an output parameter. Of course, both situations may apply to the same variable, which should then be classified as a reference parameter.

After all variables have been treated in this manner, we are guaranteed that all data flow edges have been safely rerouted. There might, however, still be references to local variables from the enclosing scope in the

```
class A {
  int k;
  void m(boolean b) {
    int n = 23, m;
    (() : void ⇒ {
      if(b)
        n += 19;
      m = k = 56;
    })();
    System.out.println(n);
  }
}
```

⇒

```
class A {
  int k;
  void m(boolean b) {
    int n = 23, m;
    ((boolean b, ref int n) : void ⇒ {
      int m;
      if(b)
        n += 19;
      m = k = 56;
    })(b, n);
    System.out.println(n);
  }
}
```

Figure 4.8: Closing over local variables

```
void m() {
  int x = 23;
  for(int i=0;i<20;++i)
    (() : void ⇒ {
      out.println(x+=i);
    })();
}
```

⇒

```
void m() {
  int x = 23;
  for(int i=0;i<20;++i)
    ((ref int x, int i) : void ⇒ {
      out.println(x+=i);
    })();
}
```

Figure 4.9: Closing over variables updated in a loop

body; this happens if the use of a local variable inside the anonymous method is independent of its use in the enclosing method, and we can then safely make it a local variable of the anonymous method instead.

As an example, consider the program in Figure 4.8 on the left. The CLOSE OVER VARIABLES refactoring considers the three references to `b`, `n`, and `m` inside the anonymous method (but not the reference to `k`, since it is a field).

- `b` has an incoming data flow edge; the parameter declaration counts as a definition, there are no intervening definitions of `b`, and the definition is outside the anonymous method. It has no outgoing data flow edge, since it is never used afterwards.
- `n` has both an incoming and an outgoing data flow edge; its reaching definition is its declaration, which is outside the anonymous method; its reached use is the `println` statement, which is likewise outside the anonymous method.
- `m` has neither incoming nor outgoing data flow edges, since it has no reaching definitions or reached uses.

Thus, `b` should be made a value parameter, `n` a reference parameter, and `m` a local variable, yielding the program on the right.

Note that control flow cannot be influenced by the transformations done in this step, so no control flow locking and unlocking is needed. Also, we forgo binding preservation in this step: uses of local variables from the surrounding scope will now bind to the corresponding parameters instead. In all the preceding (and most of the following) steps, however, we do indeed want to preserve binding. Our decomposition of the refactoring helps to clarify this situation and makes it possible to precisely pinpoint at which stages binding preservation is required, and where we have deliberately chosen to relax it.

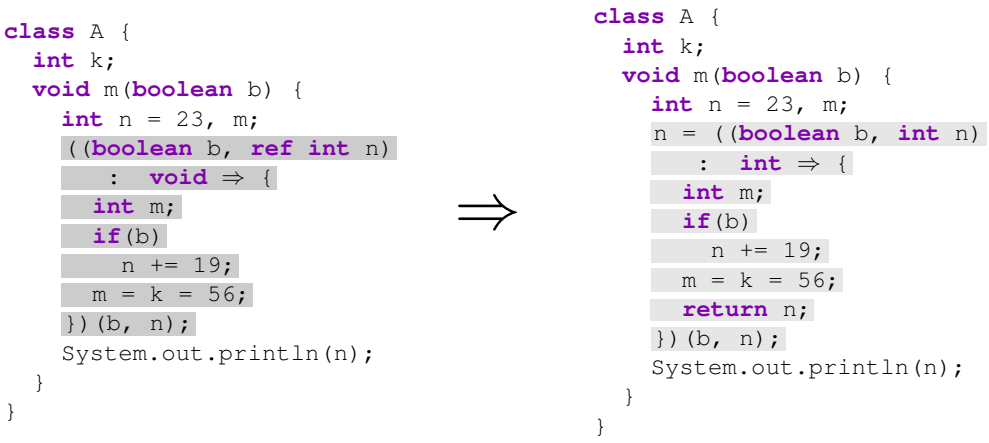


Figure 4.10: Eliminating reference parameters

The reader may find the formulation of the conditions for converting local variables into parameters slightly awkward. Looking at the example, it seems that one could abbreviate the conditions to saying that a variable should become a value parameter if it has a use inside the anonymous method whose reaching definition is outside, and similar for output parameters. That this is not sufficient in the presence of loops is demonstrated by the example program in Figure 4.9. Here, the single use of `x` inside the anonymous method has two reaching definitions (itself and the initialisation of `x`), so it should clearly be made a parameter. In fact, it should be made a reference parameter to make sure that its value keeps being incremented. To see this, consider again the single use of `x`: it is also a write of `x`, with itself as its only reached use. This reached use is lexically inside the anonymous method, but the witnessing path that shows their relationship has to cross outside the method into the loop, which shows that the value of `x` must be propagated back to the enclosing method to preserve data flow behaviour.

4.2.6 Eliminate Reference Parameters

Our refactored program has now reached its apogee from the Java language specification: not only are we working with an anonymous method, a construct unknown to the JLS, but it even might feature output and reference parameters, which are likewise not supported by the language. It is now our task to safely remove these extra features to bring our program back into the fold of standard compliant Java programs.

In this step, we eliminate **`out`** and **`ref`** parameters. The basic idea is that a **`ref`** parameter can be simulated by a value parameter whose value is returned to the surrounding method, and there assigned to the corresponding variable; an **`out`** parameter is a local variable whose value is passed back in the same fashion.

More precisely, we perform the following steps:

1. If there is no output or reference parameter, the refactoring is a no-op. If there is more than one such parameter, abort (since Java methods can only return a single value).

2. If the anonymous method already has a non-**`void`** return type, abort likewise, for the same reason.

3. Let `x` be the only non-value parameter; its corresponding argument `a` must be a local variable access. If `x` is a reference parameter, change it into a value parameter. Otherwise make it into a local variable of the anonymous method and remove `a` from the list of arguments.

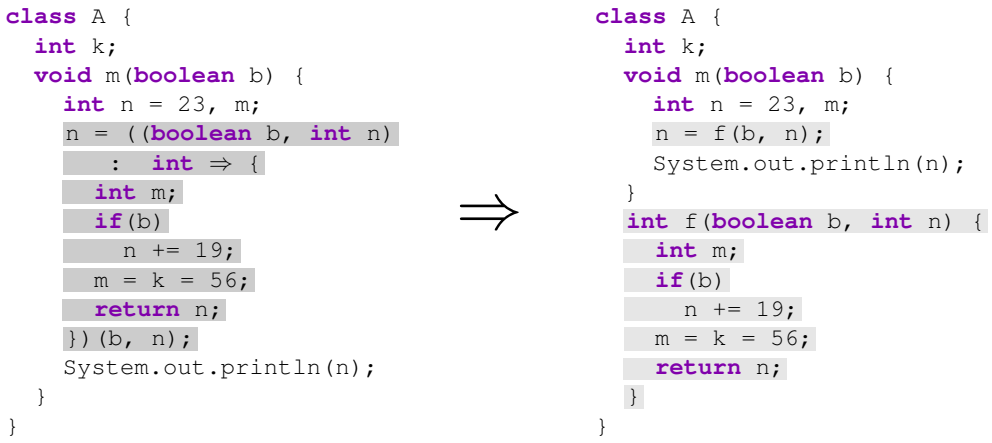


Figure 4.11: Lifting an anonymous method to a named method

4. Change the return type of the anonymous method to the type of `x`.

5. Turn any return statement inside the anonymous method into `return x`, and insert an additional return statement if the body can complete normally. Wrap the whole anonymous method into an assignment to `a` in the surrounding method.

All these steps are quite straightforward to implement; for example, the program from Figure 4.8 is transformed as shown in Figure 4.10

As explained, this refactoring step rejects any anonymous method that needs to return the value of two or more variables, which results in a behaviour similar to Eclipse. An alternative would be to package up the necessary return values into a wrapper object, return that object, and unwrap it again in the calling method.

4.2.7 LIFT ANONYMOUS METHOD

To finally turn our refactored program back into a normal Java program, we need to eliminate the anonymous method. We know that it has only value parameters and its body does not reference any local variables from the enclosing scope; hence it is semantically equivalent to a call to a named method with the same body and parameters and the same arguments.

Assuming that we want to extract the anonymous method to a method named `f`, we simply convert $(\overline{p} : r \textbf{ throws } \overline{x} \Rightarrow \overline{b})(\overline{e})$ into the method call $\texttt{f}(\overline{e})$, and insert the definition

```
r f(p̄) throws x̄ {
  b̄
}
```

into the surrounding class declaration. Our naming framework is put to use to ensure that the call to `f` really binds to the newly inserted method `f` and that all type bindings are preserved, and we lock overriding to ensure dynamic dispatch behaviour does not change.

The example program from above, of course, poses no such problems, and we can successfully complete the refactoring as shown in Figure 4.11.

4.2.8 Putting it all together

The microrefactorings introduced above are all implemented as methods on AST node types, which are just Java classes. For example, the EXTRACT BLOCK refactoring is a method in class `Block` with the signature

```
Block extractBlock(int i, int j)
```

When invoked as `b.extractBlock(i, j)`, the method extracts statements `i` to `j` of block `b` into a new block, and returns it as a result. If, at any point, the refactoring cannot proceed (for example due to an invariant violation), an exception is thrown.

The other refactorings are implemented in a similar way, so that the complete EXTRACT METHOD refactoring can be implemented (almost) as a one-liner:

```
MethodDecl Block.extractMethod(int i, int j, String n) {
  return extractBlock(i, j).introduceAnonymous().
         closeVariables().eliminateRef().lift(n);
}
```

This method again belongs to class `Block`, and is invoked on the block from which we want to extract statements `i` to `j`; the parameter `n` determines the name of the new method. Since all the microrefactorings return the result they produce, an invocation chain can be used to implement their sequential composition.

Most pleasingly, the inverse refactoring INLINE METHOD can be decomposed in the same way as EXTRACT METHOD: first turn the method invocation into an anonymous method; introduce a reference parameter if the result of the call is assigned to a local variable; replace argument passing by assignment; inline the anonymous method into a block; and finally inline the block into a sequence of statements. Of course, each of these five microrefactorings performs the inverse task of the corresponding microrefactoring for method extraction.

To keep the presentation simple, we have so far ignored complications arising from generic methods. If the method from which we are extracting is generic and the extracted code uses any of its type parameters, the extracted method has to be provided with its own copies of these type parameters, for the original ones are not going to be visible anymore. LIFT ANONYMOUS METHOD as described above handles this situation correctly, if not very gracefully: when trying to unlock type accesses in the newly created named method, the name binding framework will be unable to construct accesses to the type parameters, and hence abort the refactoring.

A more satisfactory solution would be to allow for anonymous methods to have type parameters just like normal methods. We refrain from defining type inference rules for these type parameters, instead insisting that in addition to its normal arguments every anonymous method also have a list of type arguments, one per type parameter.

We then only need to extend the CLOSE OVER VARIABLES microrefactoring to close over type variables as well. This is much easier than for normal variables, since type variables are not subject to any kind of data flow; we simply need to identify all type parameters V of the surrounding method or constructor that are accessed by the anonymous method, and give the anonymous method an extra type parameter V' with the same bounds as V; finally, we add V as a type argument to initialise V'.

When converting the anonymous method into a named method in LIFT ANONYMOUS METHOD, we check whether it has any type parameters. If not, the microrefactoring proceeds as outlined above. Otherwise, we turn the anonymous method into a generic method with the same list of type parameters, and instead of a

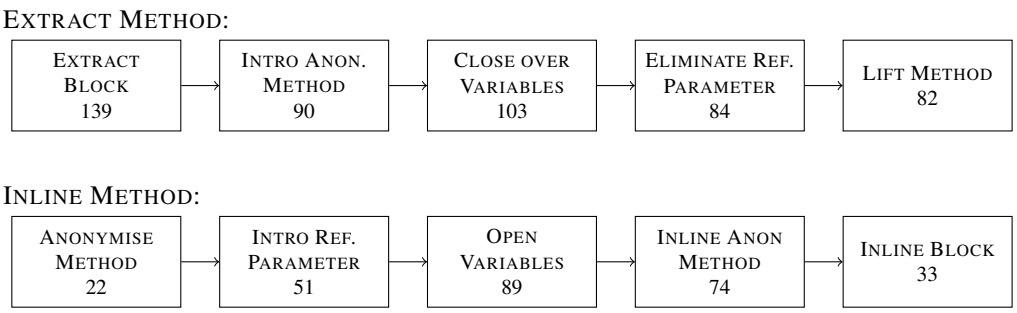


Figure 4.12: Structure and code size for EXTRACT METHOD and INLINE METHOD

normal method access we insert a parameterised method access that initialises all the type parameters to their correct types.

The implementation of INLINE METHOD, on the other hand, does not need to deal with generic methods: in Java, a method call can never directly invoke a generic method; it always invokes a particular parameterisation of that method, in which all type parameters have been assigned concrete types.

4.2.9 Evaluation

To evaluate our approach, we first present some statistics on the amount of code needed to implement the discussed refactorings in our refactoring engine.

The implementation of anonymous methods requires about 150 lines of code for the definition of corresponding AST node types, pretty printing, name analysis, and control and data flow analysis.

The size of the individual refactorings we have implemented is summarised in Figure 4.12: for each refactoring we show its decomposition in micro-refactorings, and the size of each micro-refactoring.

EXTRACT METHOD takes around 500 lines of code, quite evenly distributed between the individual micro-refactorings: EXTRACT BLOCK needs about 140 lines, INTRODUCE ANONYMOUS METHOD and CLOSE OVER VARIABLES around 100 lines each, whereas ELIMINATE REFERENCE PARAMETERS and LIFT ANONYMOUS METHOD each are implemented in less than 90 lines. The numbers for INLINE METHOD are similar, as shown in the same figure; overall it is a bit smaller than EXTRACT METHOD.

These numbers compare very favourably both to Eclipse's implementation of EXTRACT METHOD, which comprises more than 1500 lines of code, and other implementations like the one presented by Juillerat [62], which seems to offer less functionality for only a subset of Java at around 1000 lines of code.

To test the correctness of our refactorings, we ran our implementation on the test suite for Eclipse 3.5, which is publicly available, and some tests from the test suite for IntelliJ IDEA 8.0, which JetBrains kindly provided for us to use.

The former includes 395 test cases for EXTRACT METHOD, of which 23 test functionality that we do not support yet: Two test cases concern the extraction of a method into a surrounding class other than the immediate host class. Another 21 cases test duplicate elimination, where a number of occurrences of the same expression are jointly extracted into a method. This feature is largely orthogonal to the extraction process proper and relies on clone detection [12], which we have not implemented in our framework yet. All of the remaining 372 test cases are handled correctly by our implementation, among them three on which Eclipse's own refactoring engine fails with a null pointer exception, and one on which it produces a wrong result.

The test cases that JetBrains provided to us comprise 74 tests for EXTRACT METHOD, of which 19 again concern duplicate elimination. We pass all the remaining 55 tests.

Where our engine did not produce identical code to Eclipse or IntelliJ, we manually checked to make sure that our output is equally valid. Such discrepancies are mostly due to a different ordering of the parameters of the extracted method or similar syntactic variations.

4.3 Related work

This chapter is based on the author's paper "Stepping Stones over the Refactoring Rubicon" [119] published at ECOOP 2009, which is joint work with Mathieu Verbaere, Torbjörn Ekman and Oege de Moor. The key idea of using language extensions to decompose refactorings is the author's, as is its application to the specification and implementation of EXTRACT METHOD. Mathieu Verbaere provided assistance with setting up the Eclipse test suite and surveying related work. All co-authors provided valuable feedback on the development and the presentation of the material.

The refactoring literature usually presents statement-level refactoring like EXTRACT METHOD as primitive transformations useful in the composition of larger-scale refactorings [71, 100]. We are, however, by no means the first to advocate the decomposition of that kind of refactoring into yet smaller components. Perera [104] paints a compelling picture of how the use of microrefactorings changes the way refactorings are used by the programmer, with method extraction in Java as one of his examples. His proposed building blocks, e.g., PUSH STATEMENT INTO METHOD for moving statements one by one into a freshly created method, are still quite complex, however, as they need to yield a valid Java program at each intermediate step. It is hence not obvious that his decomposition actually makes it easier to implement a behaviour preserving refactoring engine. By comparison, our use of lightweight language extensions makes the individual microrefactorings much simpler and easier to think about and implement.

Reichenbach *et al.* [110] likewise propose to split refactorings into smaller steps. To achieve a more fine-grained decomposition, they suggest dropping the requirement of behaviour preservation for intermediate steps; instead, behaviour preservation is checked at the end of a composite transformation by comparing program models of the original program and the transformed program. They have implemented several refactorings for Java and report promising results in a comparative evaluation with Eclipse.

The concept of control and data flow preservation features prominently in the influential paper of Komondoor and Horwitz [72] on procedure extraction. However, their work mostly focusses on moving statements together to arrange them into a contiguous block suitable for extraction; the actual process of extracting into a procedure they dismiss as "straightforward".

In our specification of EXTRACT METHOD, we understand the step of closing over variables from the surrounding method as rerouting "broken" data flow dependencies. Intriguingly, the same idea can be found in recent work by Murphy-Hill and Black [95], who introduce GUI support for displaying this kind of dependency in an IDE in order to make it easier for the user to understand which variables will become parameters, and why this step may fail.

The composition of smaller refactorings into larger ones has been the focus of work by Kniesel and Koch [71]. For the refactorings we have considered in this chapter, a very simple form of sequential composition is sufficient, which they call AND-sequence. Their work is also concerned with composing the pre- and post-conditions of constituent refactorings, which does not directly apply to our invariant-based presentation.

An invariant-based approach to structural refactoring has been suggested some time ago by Griswold [49]. He provides a small catalogue of simple "program restructurings" that preserve data and control dependencies. While these restructurings are quite similar in scope and intent to our microrefactorings, they are presented for a very small and well-behaved object language (a first-order subset of Scheme), where they are much easier to implement and reason about than with Java.

Chapter 5

Specifying and Implementing Refactorings

The preceding chapters have provided us with a toolbox for specifying and implementing refactorings. We have shown that static semantic dependencies provide a convenient way to think about correctness issues, we have explored the use of language restrictions and extensions to work around complexities and idiosyncrasies of the object language, and we have demonstrated that complicated macrorefactorings can often be broken down into simple microrefactorings.

We have motivated each of these concepts with one or more specific refactorings for whose implementation they prove beneficial. By now the reader may have become a bit suspicious: maybe we just cleverly selected refactorings that fit our purpose. Is there any evidence that the techniques we have introduced are more widely applicable? Can they be used to implement all those other refactorings offered in modern IDEs?

The present chapter answers this question in the affirmative. We start out by defining an informal pseudocode notation for specifying refactorings that is precise enough to serve as the basis of an implementation. We show how the refactorings introduced in the preceding chapters can be specified in this way. We then present the results of a case study in which we have specified and implemented all the refactorings available in recent versions of Eclipse, except for some refactorings that have already been thoroughly discussed in the literature. We give an overview of the language restrictions and extensions used and report on the degree of reusability achieved by microrefactorings. Our implementation is then put to the test by running it on Eclipse's internal refactoring test suite. The results show that our implementation, while only a prototype, compares favourably to that of Eclipse in terms of features, yet is much more compact and can correctly handle many programs where Eclipse produces wrong output.

5.1 Specifying refactorings

While the high-level specification of refactorings has attracted considerable interest (see Section 5.3 on related work below), past attempts at providing such specifications have usually concentrated either on very simple object languages or on very simple refactorings to keep the specification readable.

The concepts and techniques introduced in the previous chapters allow us to overcome this limitation. A dependency-based approach to refactorings allows us to simplify the description of refactorings, with

complex name binding and flow analysis transparently taken care of by a general framework; language extensions and restrictions encapsulate the handling of problematic special cases; and the decomposition into microrefactorings makes the description of even complicated refactorings manageable.

So far, however, all we have given are prose descriptions. Such descriptions tend to become unwieldy and unclear, and it is not always obvious how far the informal description is removed from an actual implementation. We now rectify this omission by providing more disciplined pseudocode descriptions of the refactorings we have discussed so far. The next section will discuss how these informal descriptions serve as the basis of a concrete implementation of the refactorings.

5.1.1 Specifying RENAME

Let us start with the RENAME refactorings. Recall that to rename a type or variable, it is enough to lock any names that are "endangered" by the renaming, i.e., that might become captured, change the name, and unlock the names again. In addition, some rules regarding name clashes have to be respected.

The algorithm for renaming a field is shown in more detail in Algorithm 1. Refactorings are presented as imperative procedures that directly mutate the syntax tree. They can take arguments, here f and n, and return results, although RENAME FIELD does not. Arguments and return values are informally typed: in the example, argument f is meant to be an AST node corresponding to a declaration of a field, whereas n is an identifier, i.e., a string that fulfils the lexical conditions of a valid Java identifier. A list of all types used in the specifications is given in Table B.1 in Appendix A.

Algorithm 1 RENAME FIELD(f: *Field*, n: *ident*)

Input Language Restrictions:
Output Language Extensions: locked names

1: **assert** host type of f contains no other field of name n
2: $n_o \leftarrow$ name of f
3: **for all** accesses a that could refer to a variable named n or n_o **do**
4: lock a
5: set name of f to n

Every specification of a refactoring describes its input language and output language, indicating any language restrictions that the input program has to obey, as well as any language extensions that may occur in the output. The specification does not contain code to enforce these restrictions, or to eliminate the extensions, since this is done in the same way for every refactoring. By convention, we assume that restrictions are enforced before the refactoring starts to execute; language extensions are eliminated from the output program if the refactoring has completed successfully, but only if it is being performed as a stand-alone refactoring. When a refactoring is invoked as part of a larger refactoring, the decision of when to eliminate language extensions is left to the caller. This is crucial for refactorings like EXTRACT METHOD, specified in detail below, where one subrefactoring creates a language extension that is later eliminated by another.

For RENAME FIELD, the refactoring does not require any language restrictions on the input program, but may produce an output program containing locked names.

The main body of the refactoring is specified as informal imperative pseudocode. It first checks a precondition: the host type of f cannot already contain a field of name n. This precondition needs to hold in order for the output program to be well-formed. If it fails to hold, the whole refactoring is aborted and any changes

made to the syntax tree are rolled back. An actual implementation would additionally need to provide an explanatory error message, but this is elided from the specification.

If the precondition holds, the refactoring determines the current name n_o of the field, and then locks all endangered variable accesses. To do this, it searches the whole program for accesses a that could refer to a variable named n or n_o; such an access could be a variable access, but also a type or package access that appears in a syntactically ambiguous position. An implementation may, of course, wish to restrict the scope of locking for performance reasons.

All that remains to be done now is to actually change the name of the field. At this point, the work of the refactoring is done. As mentioned above, we rely on a general naming framework such as the one whose implementation was described in Chapter 2 to eliminate locked names in favour of regular names. This unlocking process may still fail, causing the whole refactoring to be aborted.

Renaming local variables is very similar, as shown in Algorithm 2: instead of a *Field* f, we rename a *LocalVar* v, i.e., either a local variable or a parameter. We need to check that the scope of v is disjoint from the scope of any other local variable or parameter n, since in Java local variables of the same name cannot have nested scopes. We then proceed to lock endangered accesses, where we can restrict our attention to the scope of the renamed variable. Finally, we change the name of v and rely on the naming framework to reconstruct any bindings to declarations that might have become hidden.

Algorithm 2 RENAME LOCAL(v: *LocalVar*, n: *ident*)

Input Language Restrictions:
Output Language Extensions: locked names

1: **assert** scope of v does not intersect scope of any other *LocalVar* named n
2: $n_o \leftarrow$ name of v
3: **for all** accesses a in the scope of v that could refer to a variable named n or n_o **do**
4: lock a
5: set name of v to n

Since the specifications of RENAME TYPE and RENAME PACKAGE are very similar, we omit them. RENAME METHOD, on the other hand, merits a brief discussion, since it needs an additional language extension: locked overriding.

Algorithm 3 RENAME METHOD(m: *Method*, n: *ident*)

Input Language Restrictions:
Output Language Extensions: locked names, locked overriding

1: $n_o \leftarrow$ name of m
2: **for all** methods m' of name n or n_o **do**
3: lock overriding of m'
4: lock accesses to m'
5: **for all** m' such that $\exists m''.m <:^* m'' \wedge m' <:^* m''$ **do**
6: **assert** m' is not native
7: $s \leftarrow$ signature of m' after renaming
8: **assert** host type of m' contains no local method of signature s
9: **assert** m' can override or hide any ancestor method of signature s
10: **assert** m' can be overridden or hidden by any descendant method of signature s
11: set name of m' to n
12: remove any static import of m' if it would become vacuous

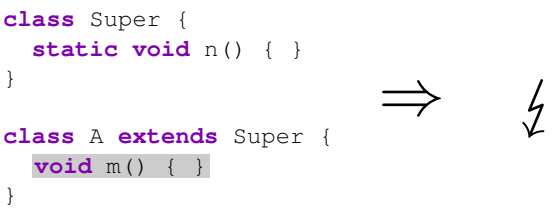


Figure 5.1: Example where method `m` cannot be renamed to `n`

As seen in Algorithm 3, we declare this language extension as part of the output language, relying on the framework to eliminate it if necessary, which in this case simply means to check that the method still overrides precisely the same methods as before, and aborting the refactoring otherwise.

The overall structure of the refactoring is slightly more complicated than for the other RENAME refactorings, since we need to rename not only method m, but also all its relatives, i.e., all methods m' such that both m and m' (reflexively, transitively) override the same method m''. We write the overriding relationship between methods as $<:$, and its reflexive transitive closure as $<:^*$.

For every such method m', we ensure that it is not native (otherwise the renaming would introduce a mismatch between the method's Java declaration and its native-code implementation). Then we ensure that there will not be any clash with an already existing method of the same name in the host class. For any ancestor methods of the same signature (i.e., any methods that m' would override or hide after the refactoring) m' must be able to, in fact, override or hide them, and the same for descendant methods that would override or hide m'. This prevents the refactoring from going ahead in a situation like the one shown in Figure 5.1.

Besides changing the name of method m' to the new name, we also need to remove any static import of m' that would become vacuous, i.e., that would no longer import anything at all, since such imports are forbidden in well-formed Java programs.

5.1.2 Specifying INLINE TEMP

Next, let us consider the INLINE TEMP refactoring, which was discussed in some detail in Chapter 3. Its specification, given in Algorithm 7, makes use of three microrefactorings SPLIT DECLARATION, INLINE ASSIGNMENT and REMOVE DECL, whose specifications we consider first.

The SPLIT DECLARATION refactoring, specified in Algorithm 4, operates on a *Temp* d, i.e., a node that represents a declaration of a local variable, but not a parameter. If that declaration has an initialiser, it turns it into a stand-alone assignment a, otherwise it does not do anything. To accommodate both cases, the refactoring has an ML-style `option` return type: in the former case, it returns `Some` a, in the latter `None`.

While the output of this microrefactoring will not contain any language extensions beyond those that may have been present in the input program, it requires that its input program should not contain compound declarations, i.e., statements that declare more than one variable at the same time. Obviously, such compound declarations can always be rewritten into a series of individual declarations that each form a statement by themselves.

Furthermore, it requires that there are no array initialisers, which are a special kind of expression for constructing array literals that is only allowed to occur in variable initialisers, but nowhere else. This little syntactic unevenness is easily eliminated in favour of an array creation expression, as shown in Figure 5.2, which also shows how to desugar compound declarations.

```
class A {
  void m() {
    int x = 23, y,
        z[] = { 23, 42 };
  }
}
```

≡

```
class A {
  void m() {
    int x = 23;
    int y;
    int z[] = new int[] { 23, 42 };
  }
}
```

Figure 5.2: Desugaring compound declarations and array initialisers

Algorithm 4 SPLIT DECLARATION(*d*: *Temp*): `option` *Assignment*

Input Language Restrictions: no compound declarations, no array initialisers

Output Language Extensions:

1: **if** *d* has initialiser *e* **then**
2: $x \leftarrow$ variable declared in *d*
3: $a \leftarrow$ new assignment $x = e$
4: insert *a* as statement after *d*
5: remove initialiser of *d*
6: **return** `Some` *a*
7: **else**
8: **return** `None`

With these simplifications, SPLIT DECLARATION is now quite easy to express. Note that we may not be able to insert *a* as a statement after *d* if *d* is not immediately inside a block: it may, for example, be the initialising statement of a `for` statement. In that case, the refactoring should abort.

The next step is the microrefactoring INLINE ASSIGNMENT, which takes an assignment *d* and inlines it into all its uses. This refactoring has to perform some data flow analysis to ensure behaviour is preserved, as discussed in Chapter 3; the specification (Algorithm 5) closely follows the steps outlined there. To ensure purity of *e*, we check that it does not contain any method or constructor calls in Line 5, and a similar check is performed in Line 14. We use this fairly conservative check in anticipation of the correctness argument to be presented in Chapter 6, which is phrased in terms of such calls.

Note that in Line 13 we ensure that any use into which we inline is located in the same body declaration as the assignment, and not a local or anonymous class nested inside it: otherwise the inlining would cross method boundaries, transcending the capabilities of a purely intraprocedural data flow analysis. We also lock synchronisation dependencies as described in Section 3.2 to ensure correctness for concurrent programs.

The microrefactoring requires the absence of implicit assignment conversion from its input program, or at least from assignment *d*. To see why this is necessary, consider the program in Figure 5.3 on the left. It assigns the integer literal 42 to the local variable `l`, which is declared to be of type `long`. Hence the invocation of `n` in the next line will be resolved to method `n(long)`, printing `long` to the console.

If we directly inline the expression `42` into all uses of this assignment as shown in the program on the right, however, the invocation will now resolve to `n(int)`, printing `int` instead. We should, instead, inline the expression `(long)42`: the cast from `int` to `long` is implicit in the input program, since the right hand side of an assignment is subjected to a process of assignment conversion [48, §5.2] to fit the type of the left hand side. For the purposes of INLINE ASSIGNMENT, it is much easier if that implicit conversion is made explicit first, so we require it as a language restriction on the input program.

The final step of INLINE TEMP, the REMOVE DECL microrefactoring shown in Algorithm 6, is quite

Algorithm 5 INLINE ASSIGNMENT(d: *Assignment*)

Input Language Restrictions: no implicit assignment conversion

Output Language Extensions: locked names, control flow, reaching definitions, synchronisation dependencies

1: $x \leftarrow$ left hand side of d
2: $e \leftarrow$ right hand side of d
3: **assert** d forms a statement by itself
4: **assert** x refers to local variable
5: **assert** e does not contain method or constructor invocations, and no variable writes
6: $U \leftarrow$ all reached uses of d
7: $C \leftarrow$ targets of exceptional control flow edges originating in e
8: **if** $C \neq \emptyset$ **then**
9: **assert** there is at least one $u \in U$ on every path from d to an exit node
10: **for all** $u \in U$ **do**
11: **assert** u only has one reaching definition and is not an lvalue
12: **assert** immediately enclosing body declaration of u is same as of e
13: **for all** $c \in C$ and n such that $d \rightarrow_s^* n \rightarrow_s^* u$ **do**
14: **assert** n is not a method or constructor invocation
15: **assert** n does not write a location that is live at c
16: $e_u \leftarrow$ copy of e
17: lock names, reaching definitions, exceptional control flow, synchronisation dependencies in e_u
18: replace u with e_u
19: replace d with an empty statement

```
class A {
  void m() {
    long l;
    l = 42;
    n(l);
  }

  void n(int i) {
    System.out.println("int");
  }

  void n(long l) {
    System.out.println("long");
  }
}
```

$\not\Rightarrow$

```
class A {
  void m() {
    long l;
    n(42);
  }

  void n(int i) {
    System.out.println("int");
  }

  void n(long l) {
    System.out.println("long");
  }
}
```

Figure 5.3: Problematic example of INLINE ASSIGNMENT (V)

easy to specify: for a declaration d of a local variable, check that it is not referenced anywhere and has no initialiser, and then remove it from the syntax tree. If it is still used or does, in fact, have an initialiser, the refactoring does nothing. Note that for convenience we again assume that d is rewritten into a stand-alone declaration if it is part of a compound declaration.

Algorithm 6 REMOVE DECL(d: *Temp*)

Input Language Restrictions: no compound declarations

Output Language Extensions:

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

Putting these three microrefactorings together, we obtain the specification of INLINE TEMP in Algorithm 7. The first line performs an implicit pattern matching on the result of SPLIT DECLARATION: if that microrefactoring returns `None`, the matching fails and INLINE TEMP is aborted. The invocation of INLINE ASSIGNMENT in the second line appears between floor brackets $\lfloor\cdot\rfloor$: we use this convention to indicate that any language extensions produced by this microrefactoring should be eliminated from the program before proceeding. This is not needed for the other two microrefactorings, which produce pure Java.

Algorithm 7 INLINE TEMP(d: *Temp*)

Input Language Restrictions:

Output Language Extensions:

1: `Some` $a \leftarrow$ SPLIT DECLARATION(d)
2: $\lfloor$INLINE ASSIGNMENT$\rfloor$(a)
3: REMOVE DECL(d)

5.1.3 Specifying EXTRACT METHOD

As our final example of pseudocode specifications, let us consider the EXTRACT METHOD refactoring discussed in Chapter 4. This refactoring is composed of five microrefactorings, whose specification we will present in turn before giving a specification of EXTRACT METHOD itself.

Algorithm 8 EXTRACT BLOCK(b: *Block*, i: *nat*, j: *nat*): *Block*

Input Language Restrictions: no compound declarations
Output Language Extensions: locked names

1: $[s_0;\ldots;s_{n-1}] \leftarrow$ statements in b
2: **assert** $0 \le i \le j < n$
3: lock all variable and type names in b
4: **for all** $i \le k \le j$ **do**
5: **assert** s_k is not a `case` or `default`
6: **if** s_k declares a variable referenced after s_j **then**
7: SPLIT DECLARATION(s_k)
8: move s_k before s_i
9: $b' \leftarrow$ new block with statements $s_i,\ldots,s_j$
10: set statements of b to $s_0,\ldots,s_{i-1},b',s_{j+1},\ldots,s_{n-1}$
11: **return** b'

The first constituent microrefactoring is EXTRACT BLOCK, specified in Algorithm 8. It takes as its argument a block b and two natural numbers i, j indicating the indices of the first and the last statement from b to extract into a new block; that new block replaces the extracted statements and is returned as the result. The input is required not to contain compound declarations, and the output program may contain locked names.

After checking that the given indices make sense, the refactoring locks all names in the block. It uses SPLIT DECLARATION to split off the initialiser of any declaration among the statements to be extracted which is used after the last extracted statement: for such declarations, we can only extract the initialiser, not the declaration itself, which has to stay in the enclosing block. None of the statements to be moved into the block can be a `case` or `default` of a `switch`, as the result would not be syntactically correct. Finally, the new block is constructed and inserted into the original block.

Algorithm 9 INTRODUCE ANONYMOUS METHOD(b: *Block*): *AnonymousMethod*

Input Language Restrictions:
Output Language Extensions: locked control flow, locked names, `return void`, anonymous methods

1: lock control flow successors in b
2: $[e_1; \ldots; e_n] \leftarrow$ locked accesses to all uncaught checked exceptions thrown in b
3: **if** b can complete normally **then**
4: $\quad c \leftarrow (() :$ `void throws` $e_1, \ldots, e_n \Rightarrow b)()$
5: $\quad$ replace b with c`;`
6: **else**
7: $\quad$ **if** b is in a method m **then**
8: $\quad\quad T \leftarrow$ locked access to return type of m
9: $\quad$ **else**
10: $\quad\quad T \leftarrow$ `void`
11: $\quad c \leftarrow (() :$ T `throws` $e_1, \ldots, e_n \Rightarrow b)()$
12: $\quad$ replace b with `return` c`;`
13: **return** c

The next step towards extracting a method is refactoring INTRODUCE ANONYMOUS METHOD, specified in Algorithm 9. It takes a block b and returns the anonymous method it has been turned into; thus obviously its output will contain anonymous methods, as well as locked names and locked control flow successors. The language extension `return void` will be discussed below.

The refactoring locks control flow dependencies that need to be preserved, determines the checked exceptions that need to be declared, and then constructs the anonymous method. If the block b can complete normally, i.e., if control can "fall off" its last statement, it is replaced by an expression statement containing the anonymous method.

Otherwise, the anonymous method is wrapped into a `return` statement. Note, however, that the anonymous method may have type `void` if the enclosing body declaration is a method with return type `void` or a constructor. Instead of introducing extra code to deal with this case, we simply allow the refactoring to occasionally produce return statements that "return" an expression of type `void`: this is the `return void` language extension alluded to above. Of course, the extension is easy to eliminate: simply replace `return` e`;` with e`; return;`. Expressions of type `void` are always promotable, and hence can form an expression statement by themselves.

The next step isolates the anonymous method from its surrounding method in terms of data flow: the CLOSE OVER VARIABLES refactoring, specified in Algorithm 10, finds incoming and outgoing data flow

Algorithm 10 CLOSE OVER VARIABLES(*a*: *AnonymousMethod*)

Input Language Restrictions:
Output Language Extensions: anonymous methods, `out` and `ref` parameters, locked names

```
1:  m ← body declaration enclosing a
2:  V ← ∅; Val ← ∅; Out ← ∅; Ref ← ∅
3:  for all variable accesses va in a do
4:      v ← variable va binds to
5:      assert if va is a write, then v is not final
6:      if v is a local variable or parameter of m then
7:          V ← V ∪ {v}
8:          if va has an incoming data flow edge then
9:              if v ∈ Out then
10:                 Out ← Out \ {v}
11:                 Ref ← Ref ∪ {v}
12:             else if v ∉ Ref then
13:                 Val ← Val ∪ {v}
14:         if va has an outgoing data flow edge then
15:             if v ∈ Val then
16:                 Val ← Val \ {v}
17:                 Ref ← Ref ∪ {v}
18:             else if v ∉ Ref then
19:                 Out ← Out ∪ {v}
20: for all v ∈ V do
21:     if v ∈ Val ∪ Out ∪ Ref then
22:         p ← new parameter with same name and type as v
23:         make p ref if v ∈ Ref, out if v ∈ Out
24:         add p as parameter to a
25:         add access to v as argument to a
26:     else
27:         v′ ← new local variable with same name and type as v
28:         add v′ as local variable to a
29: for all type parameters V of m used in a do
30:     add type parameter V′ with same name and bounds as V to a
31:     add type argument V to a
```

edges, i.e., witnessing paths between definitions and uses that go across the boundary of the anonymous method, and reroutes them through parameters.

The algorithm computes four sets of variables: V is the set of all local variables and parameters of the enclosing body declaration m which are accessed within a; $Val \subseteq V$ are those variables that should be made value parameters, $Out \subseteq V$ those that should become output parameters, and $Ref \subseteq V$ those that should become reference parameters. These sets are determined by iterating over all variable accesses in a. At the same time, it is checked that a does not write any final field: if m is a constructor, then a, being lexically within m, may write final fields of the enclosing class; but this is no longer possible once a is turned into a stand-alone method.

Once the sets have been computed, the microrefactoring creates parameters for the variables in $Val \cup Out \cup Ref$, and local variables for all the other variables. Finally, it closes over type variables by introducing corresponding type parameters and arguments.

As the penultimate step, the refactoring ELIMINATE REFERENCE PARAMETERS, specified in Algorithm 11, removes any `ref` or `out` parameters the anonymous method may have. This can only be done

Algorithm 11 ELIMINATE REFERENCE PARAMETERS(*a*: *AnonymousMethod*): *AnonymousMethod*

Input Language Restrictions: no implicit `return`
Output Language Extensions: anonymous methods

1: **if** *a* has `ref` or `out` parameters **then**
2: **assert** *a* has a single `ref` or `out` parameter
3: **assert** return type of *a* is `void`
4: $x \leftarrow$ the `ref` or `out` parameter of *a*
5: $v \leftarrow$ the variable access passed as argument into *x*
6: replace *a* by $v = a$
7: set return type of *a* to type of *x*
8: replace every `return;` statement with `return` *x*`;`
9: **if** *x* is a `ref` parameter or it is live at the entry of *a* **then**
10: make *x* a value parameter
11: **else**
12: make *x* a local variable
13: remove argument *v*
14: **return** *a*

if there is at most one such parameter and if the anonymous method does not return a result, since the final value of the parameter should be returned and assigned to the corresponding variable in the surrounding method.

To simplify the transformation, we require that implicit `return` statements of the anonymous method be made explicit, so that every control path on which execution of the method terminates normally ends with a return statement. We then simply adjust all the return statements to return the value of `x`. If `x` is a reference parameter, it should now be made a value parameter; the same should happen if it is an output parameter that is live at entry to the method. Of course, this cannot happen in a valid anonymous method, since output parameters may not be read before they have been written, but variable reads violating this condition may have been inserted in the previous step when we updated the `return` statements.

Otherwise, `x` can be demoted to a local variable of the closure, and its corresponding argument (which must simply be a local variable access) removed.

Algorithm 12 LIFT ANONYMOUS METHOD(*n*: *ident*, *a*: *AnonymousMethod*): *Method*

Input Language Restrictions:
Output Language Extensions: locked names

1: **assert** *a* does not reference any local variables from surrounding body declaration
2: **assert** *a* has no `ref` or `out` parameters
3: lock all names in *a*
4: $\overline{e} \leftarrow$ argument list of *a*
5: $m \leftarrow$ turn *a* into method named *n*
6: make *m* `static` if *a* occurs in static context
7: $T \leftarrow$ innermost type surrounding *a*
8: **assert** *T* has no member method with same signature as *m*
9: **assert** no subtype of *T* declares a method that would override or hide *m*
10: lock all calls to methods named *n*
11: insert *m* into *T*
12: $c \leftarrow$ locked call of *m* on arguments $\overline{e}$
13: replace *a* with *c*
14: **return** *m*

The final step is to turn the anonymous method into a named method, which is done by the microrefactoring LIFT ANONYMOUS METHOD, specified in Algorithm 12. We check that the anonymous method a does not reference local variables of the method from which it is extracted, and has no `ref` or `out` parameters: both conditions are obviously already satisfied when this microrefactoring is invoked as part of EXTRACT METHOD, but they have to be checked in order for LIFT ANONYMOUS METHOD to be behaviour preserving as a stand-alone refactoring.

We need to lock all names in a to prevent accidental name capture, then we construct a named method m corresponding to a and insert it into the surrounding type T. We require that T does not declare or inherit a method with the same signature as m: the former would lead to an invalid output program, the latter might change dynamic method call dispatch. For the same reason, no subtype of T may declare a method that would override or hide m. Finally, we replace the anonymous method with a locked call to the extracted method.

Algorithm 13 EXTRACT METHOD(b: *Block*, i: *nat*, j: *nat*, n: *ident*): *Method*

Input Language Restrictions:

Output Language Extensions:

1: $b' \leftarrow \lfloor$EXTRACT BLOCK$\rfloor(b,i,j)$
2: $a \leftarrow$ INTRODUCE ANONYMOUS METHOD(b')
3: CLOSE OVER VARIABLES(a)
4: ELIMINATE REFERENCE PARAMETERS(a)
5: **return** $\lfloor$LIFT ANONYMOUS METHOD$\rfloor(n,a)$

Specifying EXTRACT METHOD is now simply achieved by plugging together the above five microrefactorings, as shown in Algorithm 13. Note again that we use floor brackets to indicate the points at which language extensions should be eliminated: in this case only at the beginning and at the very end, since the anonymous method created by INTRODUCE ANONYMOUS METHOD should survive until it is removed by LIFT ANONYMOUS METHOD.

5.2 Implementing refactorings

The specification examples in the preceding section will hopefully have given the reader a flavour of how the techniques introduced in this thesis can be used to both handle deep problems concerning the preservation of static semantic properties and shallow problems engendered by the specific syntactic rules of the object language.

In order to show that these techniques carry further than a couple of well-chosen example refactorings, we set ourselves the goal of implementing all the refactorings offered in 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 [129].

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

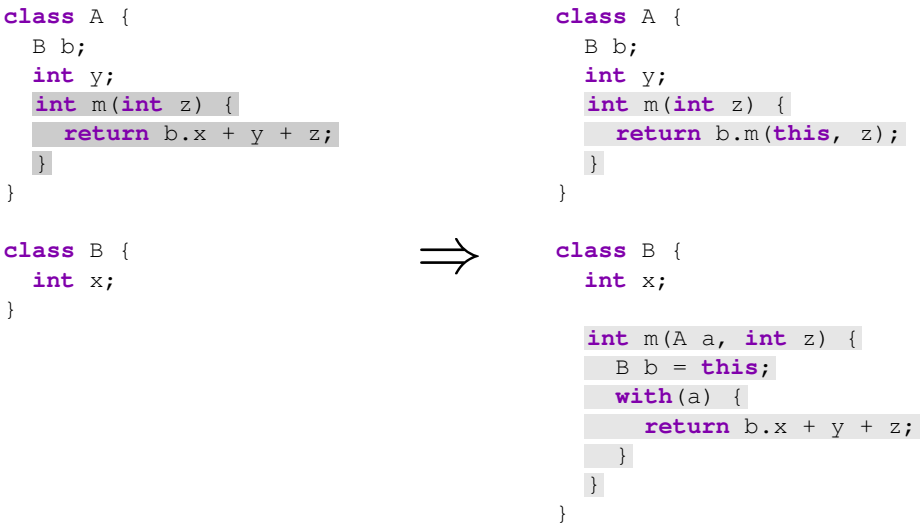


Figure 5.4: Moving a method with the help of a **with** block

All the other refactorings can be given quite compact specifications that our implementation follows closely. Besides anonymous methods, which are used in EXTRACT METHOD and INLINE METHOD, we have found it useful for several refactorings to use JavaScript-like **with** blocks.

Take, for example, the program in Figure 5.4 on the left. Here, we have a method `m` that accesses the field `x` of the field `b` of its receiver object, the field `y` of its receiver object, and the parameter `z`. Assume we want to move this method into class `B` such that `b` becomes its receiver object; normally, one would perform this refactoring if the method relies more heavily on the state of `b` than on that of its receiver object (which, admittedly, is hardly the case in this example).

The program on the right shows the first step in performing this movement: the original method is turned into a delegating method that invokes the moved method, passing along **this** as the first parameter so that the moved method can still access fields of its former receiver object. In the moved method, we initialise a local variable `b` to **this**: all accesses to the former field `b` are now bound to the local variable `b`, so at runtime they will resolve to the same reference as before.

The **with** block that surrounds the old method body makes the code contained within it execute as if its receiver object were the argument of the **with** block, namely `a` in this case: any field that would normally be looked up on **this** will be looked up on `a` instead, so for instance the unqualified access to field `y` will resolve to `a.y`. Local variables, such as `b` and the parameter `z`, from outside the **with** block are still visible inside, but no instance fields of the surrounding instance can be accessed.

Of course, the refactoring does not end here. It is not very hard to eliminate **with** blocks; essentially, all we have to do is to make the implicit qualifications effected by the **with** block explicit. As a final step, we can then use INLINE TEMP to get rid of the local variable `b`.

Elevating **with** blocks to the rank of a language extension would be of doubtful benefit if MOVE METHOD were the only refactoring that uses it. Pleasingly, this is not so. It is not hard to see that MAKE METHOD STATIC, a refactoring that exposes the receiver object of a method as a parameter, is easy to implement using **with** blocks; perhaps more unexpectedly, the MOVE INNER TO TOPLEVEL refactoring that promotes an inner class to a toplevel class also becomes much easier to formulate with the help of **with** blocks.

Refactoring	Total	Inapplicable	Missing Feature	We Reject	Eclipse Rejects	Same Result	Lines of Code	
							Eclipse	Us
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 5.1: Evaluation of the correctness of our refactoring engine on Eclipse's test suite

Although we have had occasion to use some other language extensions apart from anonymous methods and `with` blocks, they are without exception very minor features simply designed to smooth over some wrinkle of the Java syntax. A full list of all the language restrictions and extensions used in the specification and implementation of our refactoring engine is given in Appendix B.

Table 5.1 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 RENAME refactorings, which were shown in Chapter 2, and for EXTRACT METHOD and INLINE METHOD, which were discussed in Chapter 3.

For the remaining 17 refactorings the table shows the total number of test cases in column "Total", and the number of test cases where our engine produced the same results as Eclipse, modulo trivial differences, in column "Same Result". The other columns categorise sources of disagreement.

Column "Inapplicable" 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 next 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 presumable refactor most or all of these test cases, we leave the implementation of a principled clone detector and its integration with the refactoring engine to future work.

The following two columns 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 data flow analysis of our implementation: both these refactorings require the movement of expressions across method borders; our implementation only allows this for expressions composed entirely of compile-time constants and accesses to final fields, as well as calls to methods that return such an expression.

This is perhaps a reasonable choice for INLINE CONSTANT, but may be somewhat limiting for INTRODUCE PARAMETER; a more sophisticated implementation of that refactoring would need some interprocedural analysis capabilities to be able to prove that the refactoring can go ahead. Another possible solution, as outlined in Chapter 3, would be to push this kind of issue into the user interface, 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 [123] these heuristics are rather crude and can easily lead to subtle changes in behaviour.

Our refactoring engine does not attempt to solve this problem at the moment. 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. 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, but most of the reduction in size comes from the fact that complicated issues of static semantic preservation are handled by the underlying framework, not by every individual refactoring. Generally, the implementation of a refactoring has about twice as many lines of code as its pseudocode description, which is a decent average, considering that JastAdd is based on Java, which is famous for its verbose syntax.

Adding up the line counts in the last column, one obtains a total 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: while about a third of the 28 microrefactorings used in the specification of these refactorings are only used in one macrorefactoring, more than half are used in two or three macrorefactorings, and a handful of very versatile microrefactorings are used up to five times.

The source code size measurements do 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 been discussed above. A further 800 lines provide functionality for introducing and removing the language restrictions and extensions used by the refactorings.

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 5.1 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 it is somewhat lacking in performance. The former is by choice, since our goal was to develop a working prototype, not an end-user tool. We have not implemented any undo functionality yet, which is particularly crucial in our approach where a refactoring may fail halfway through the transformation. Providing such functionality is not very hard, since our refactoring implementations all access and mutate the syntax tree through the API provided by JastAdd. One could then use the aspect-oriented features of JastAdd to attach a piece of advice to these mutation methods that records the performed actions on a global undo stack, so that they can be undone later.

The performance problem is more serious: 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.

5.3 Related work

The material in this chapter is based on the author's paper "Specifying and Implementing Refactorings" [114] published at SPLASH 2010, which is joint work with Oege de Moor. All specifications and implementations discussed in this chapter were developed by the author, who also undertook the evaluation. The co-author played an important role in improving and clarifying the presentation.

As far as we know, this is the first large-scale effort to give high-level specifications of refactorings on a realistic programming language that are precise enough to serve as a basis for implementation.

The classic approach to the specification of refactorings, as pioneered by Opdyke and Roberts [100, 111], mostly concentrates on giving a formal account of the preconditions needed to ensure behaviour preservation,

whereas the transformation itself is usually only described informally. Roberts describes his refactorings for Smalltalk, whereas Opdyke bases his work on a restricted subset of C++.

There still exists a rather wide gap between such descriptions and actual implementations of refactorings. For this reason, there has been quite some interest in providing executable specifications of refactorings. For instance, Lämmel [75] 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 [47], who give executable specifications of several Java refactorings in the rewrite logic-based system Maude. This approach is appealing in that rewrite logic is flexible enough to also describe the operational semantics of Java programs, even including concurrent behaviour. The authors prove the correctness of their specifications, but these proofs are pen-and-paper and not immediately based on the formalised semantics. Furthermore, the object language under consideration is a very early version of Java that is missing some of its more complicated features.

The JunGL language proposed by Verbaere [134] 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 [135], and even complex type-based refactorings are within reach of the language [102].

In their work on type-based refactorings such as EXTRACT INTERFACE and INFER GENERIC TYPE ARGUMENTS, Tip *et al.* [7, 68, 129, 130] formulate constraint rules to ensure behaviour preservation in great detail, covering most relevant features of Java. The actual refactorings, however, are only described informally; the implementation of type-based refactorings in Eclipse is based on this work.

Steimann and Thies [123] 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.

Kegel and Steimann [65] 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 [140], 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 [121] 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.

Chapter 6

Verifying Refactorings

One of the defining features of our exposition so far has been an abundance of counterexamples. We have used them to expose bugs in well-known refactoring engines, or to motivate dependencies to be preserved, language restrictions to enforce, or extensions to allow. In every case we could successfully handle the issues brought to the fore by these examples, and our evaluation in the last chapter lends some credibility to the claim that the techniques and concepts introduced so far are powerful enough to serve as a basis for specifying and implementing just about any refactoring.

But can we go beyond mere empirical evidence? Can we reason about the correctness of our specifications, and gain some assurance that we have not simply overlooked some very subtle counterexamples? These are the questions that we tackle in this chapter.

We will start by discussing the question of what it is exactly we would like to prove. Refactorings should be behaviour preserving, but many classic definitions of program behaviour take an overly monolithic, batch-oriented view that hardly seems appropriate in the context of Java. It seems rather difficult to adapt these definitions to yield a definition of behaviour and its preservation that is both precise and general, so we suggest to treat the overall concept of behaviour preservation as an intuitive guideline and to concentrate on finding more precise correctness criteria for individual refactorings, that are specific to a given refactoring.

As an example of this approach, we recapitulate the criterion of binding preservation that is the basis of our specification of the RENAME refactorings, and discuss to what extent this specific criterion implies the intuitive requirement of behaviour preservation. A more involved example is the INLINE TEMP refactoring developed in Chapter 3 and specified in Chapter 5. We present a detailed informal argument for the correctness of this specification, using a method-level notion of correctness that we then relate to intuitive behaviour preservation. The decomposition of the refactoring into three steps aids in identifying central parts of the refactoring that can be reasoned about in a modular way.

Our correctness argument for RENAME depends on the correctness of the naming framework, a fully formal account of which is the subject of the penultimate section of this chapter. We discuss how reference attribute grammars can be embedded into type theory and show how the name lookup and access construction for NameJava from Chapter 2 can be expressed in the theorem prover Coq; we then report on a mechanised proof that access construction is a partial right inverse to name lookup.

We end this chapter by surveying some related work on the correctness of refactorings.

6.1 Program behaviour and its preservation

An ideal definition of program behaviour and behaviour preservation from the viewpoint of refactorings should satisfy three criteria:

1. It should be *realistic* in that it captures the essence of a program's behaviour, not just an idealistic approximation.

2. It should offer a *precise* description of behaviour and of what it means to preserve it.

3. It should be *general* in that as many well-known refactorings as possible can be seen to be behaviour preserving under this definition.

We will consider several definitions from the literature and discuss how well they fit these criteria, starting with Opyke's definition [100]. He specifies seven program properties that refactorings should preserve. The first six of these concern type correctness and scoping rules, and their violation would usually lead to an uncompilable output program; the final property to preserve is *semantic equivalence*, which Opdyke defines as follows: "let the external interface to the program be via the function *main*. If the function *main* is called twice (once before and once after the refactoring) with the same set of inputs, the result set of output values must be the same". He further clarifies that the "set of output values" includes any side effects that might happen during the execution of the main function.

This definition seems appealing since it is general: any transformation is allowed as long as it does not affect the behaviour of the main function. It also is fairly generic and easily applied to other languages than C++, for which Opdyke originally formulated it; after all, most languages have a concept similar to the `main` function of a C++ program as the point where program execution starts. Java programs, for instance, often have a static `main` method in one of their classes that serves as the entry point.

At least for Java, however, Opdyke's definition is not realistic. Although the `main` method is the entry point when the program is run as a stand-alone application, this method has no special status for programs that are run as applets or web applications, let alone for libraries; hence it hardly seems justified to make this method the basis of a definition of program behaviour.

As for precision, while it is clear what the "inputs" to the main function are (both in C++ and Java that function takes an array of string literals as its arguments), the concept of "output values" is much less well-defined, since it is far from clear what it means for the program to cause the same side effects. Presumably we could understand this as a requirement that the program performs the same I/O operations in the same order, but this may again be problematic in the context of Java: for instance, Opdyke explicitly affirms that a refactoring that adds an unused field to a class should count as being behaviour preserving. But in a distributed Java application that uses some form of serialisation to communicate objects between different components, such a change in memory layout may incur a change in I/O behaviour, arguably without affecting the essence of program behaviour.

In the subsequent refactoring literature, not a lot of progress has been made on extending Opdyke's definition to other real-world programming languages or making it more precise. Roberts, for instance, completely abandons the concept of behaviour preservation: he instead specifies, for each individual refactoring, a set of preconditions and postconditions such that the postconditions are satisfied on the output program if the preconditions are on the input program [111]. Going yet a step further, many influential practitioners of refactoring do not even consider the correctness of individual refactorings, but only the correctness of a

particular refactoring step on some particular program: behaviour preservation in this case amounts to the program still passing its test suite after the refactoring has been performed [45].

At the other end of the spectrum there has been work on applying concepts from the area of formal programming language semantics to obtain a precise definition of program behaviour, and then to prove that the semantics of the output program of a refactoring is the same as the semantics of its input program [9, 41]. This line of work is appealing for its precision and the reuse of well-understood concepts, but it invariably concentrates on very simple languages.

One might hope to gain some leverage from the substantial body of work on formalising the semantics of Java [13, 36, 43, 55, 56, 69, 99], but unfortunately yet again the focus here is on describing miniature subsets of the language that are missing many of those features of the language that make refactoring challenging: none of them tackle the full complexity of name lookup or method overloading; control flow is generally much simplified, leaving out unstructured control flow; class initialisation is usually ignored, as are native methods and dynamic class loading; finally, we are not aware of any formalisation that incorporates reflection or concurrency. In all cases program semantics is given with respect to a single entry point.

One might hope that at least for functional programming languages it should be easier to come up with a workable concept of behaviour preservation. Although only Standard ML has a fully formalised semantics [93], there exists a rich body of work on source level transformations of functional programs for use in program derivation and optimisation [14, 20, 105]. The Haskell Refactorer project [82] aims at exploiting the strong theoretical basis of functional languages to implement and verify refactorings for Haskell [83]. However, as Brown notes [18], the program transformations proposed in the literature usually are very localised, while many refactorings are not. Hence, correctness proofs for Haskell refactorings still tend to either concentrate on small subsets of the language with a well-defined semantics [83, 127] or to rely on an informal input-output preservation argument [18].

We consider the problem of finding a refactoring-friendly notion of program behaviour and its preservation for modern programming languages unsolved. It is certainly too much to hope for a precise, formal account of the semantics of all of a language like Java, and even if we had such a semantics proving that even a very simple refactoring preserves it would likely be an arduous task. A useful definition of program behaviour would perhaps be more along the lines of Opdyke's original definition, forgoing precision for generality, though updated to more accurately reflect the realities of modern programming languages.

In this work, we do not aim to provide such a definition. We rather take our cue from Roberts, in that we consider the correctness of *individual* refactorings. Our specifications of refactorings are given in terms of static semantic dependencies, and we can show in detail how these are preserved. We will show in the next section how these properties can be used to argue for the behaviour preservation of specific refactorings. While informal, such arguments can be carried out in some detail, and by focussing on particular refactorings we are often able to establish fairly strong results.

6.2 Correctness of individual refactorings

In this section we will present correctness arguments for the RENAME and INLINE TEMP refactorings. We will argue that our specifications of these two refactorings preserve behaviour in a suitable sense. These arguments are informal in that they are not based on a precise formulation of the semantics of Java, but still fairly detailed. Throughout this section, we will restrict our attention to sequential programs and not consider concurrent features, which can be dealt with separately as shown in Section 3.2.

6.2.1 Correctness of RENAME

The RENAME refactorings as described in Chapter 2 and specified in Chapter 5 rename declarations in the program while preserving its binding structure.

More precisely, when successfully applied to a valid Java program P, a RENAME refactoring yields a valid Java program P', for which we can find a function r mapping AST nodes of P to AST nodes of P' such that

1. if d is a declaration in P, then $r(d)$ is a declaration of the same kind in P'; the name of the entity declared by $r(d)$ might be different than the name of the entity declared by d, but all subtrees of d and $r(d)$ correspond under r;
2. if n is a name in P that binds to a declaration d, then $r(n)$ is a (possibly different) name in P' that binds to $r(d)$;
3. for any other node e in P, $r(e)$ is a node of the same kind in P', and subtrees of e and $r(e)$ correspond under r;
4. if method m_1 overrides method m_2 in P, then $r(m_1)$ overrides $r(m_2)$ in P' and vice versa.

Inspecting the different kinds of expressions in Java, we see that the result an expression evaluates to depends on the value of the receiver object, and the values returned by variable accesses and by method and constructor invocations.

Since name bindings are preserved, corresponding variable accesses in P and P' will return the same value, assuming they are evaluated on the same heap. Method invocations will resolve to the same method in P and P': Condition 2 above ensures that overloading resolution is performed in the same way, and dynamic dispatch will select the same method due to Condition 4. If the invoked method or constructor is a source method, it has been refactored as well, so it will return the same value in P' as in P; this need, however, not be the case for native methods.

For example, the reflective method `Class.forName` that looks up a class by its name may very well return a different result in P' than in P; if we rename a class `A` in P to `B`, then the method invocation `Class.forName("A")` will return the class object of `A` in P, but it will throw a `ClassNotFoundException` in P'. Similar issues arise from the use of class literals, which can be understood as syntactic sugar for invocations of `Class.forName`.

We can thus give the following correctness guarantee for our specification of RENAME: Let us say that method m and m' behave identically if, when invoked on the same heap with the same arguments, they will either both return the same value or both throw the same exception or both not terminate; furthermore, they will call the same methods and constructors in the same order with the same argument values; finally, if they terminate, then they will leave the heap in the same state.

It is then clear that any method or constructor m of P behaves identically to its counterpart $r(m)$ in P' if it does not use class literals, and every method or constructor m' that m invokes behaves identically to $r(m')$. If m neither uses class literals nor (directly or indirectly) invokes native methods, this implies that m and $r(m)$ do, in fact, behave identically. Otherwise, behaviour preservation is not guaranteed, although it is reasonable to assume that behaviour is still preserved if none of the invoked native methods make use of the reflection API.

In summary, for programs that do not use reflection, our specifications of the RENAME refactorings preserve program behaviour method by method with the mapping r providing a sort of API migration path

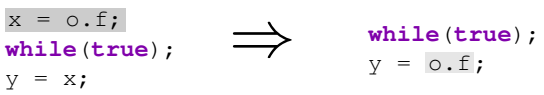


Figure 6.1: Example of INLINE ASSIGNMENT in the presence of non-termination

for external users of the program. Of course, this behaviour preservation guarantee is conditional on the correct implementation of the name binding framework, which we will discuss in Section 6.3.

6.2.2 Correctness of INLINE TEMP

For the verification of INLINE TEMP, we will concentrate on INLINE ASSIGNMENT. It is fairly clear that SPLIT DECLARATION and REMOVE DECLARATION are behaviour preserving in a very strong sense: the former will not affect the generated bytecode at all, the latter simply discards an unused local variable.

For INLINE ASSIGNMENT, we will again prove that it preserves behaviour method by method. Since the refactoring is specified in an intra-procedural manner, it makes sense to phrase its correctness similarly intra-procedurally, regarding method and constructor invocations as opaque events. Hence we characterise the execution of a method by its *execution trace*, which is the sequence of method and constructor invocations it performs (including their argument values, if any), and by its *outcome*, i.e., the value returned or exception thrown. The state the method leaves the heap in upon completion also forms part of the outcome.

We call two methods or constructors having the same signature *trace equivalent* if they produce the same execution trace and outcome when they are executed in the same state, i.e., on the same heap and with the same parameter values.

The correctness criterion we aim to establish for INLINE TEMP is the following: If program P' results from the valid Java program P by a successful application of INLINE TEMP, then P' is also a valid Java program with the same types, methods and constructors, and every method or constructor in P' is trace equivalent to the corresponding method or constructor in P.

We will need to slightly restrict this result to avoid issues with termination, as exemplified in the program in Figure 6.1 on the left. We are asked to inline an assignment to variable `x` into its single use, which happens to lie beyond an infinite loop. If `o` is a null pointer, the evaluation of `o.f` will throw an exception; in the refactored program, however, this exception will no longer happen, because the program gets stuck in the loop. The example suggests that we need to require that executions do not get stuck between the assignment and any of its uses.

While this criterion for behaviour preservation is appealing in its relative simplicity, it neglects some aspects of program behaviour. For example, inlining an expression may increase the number of push operations in the generated bytecode, which could, under very specific circumstances, lead to the refactored program throwing a `StackOverflowError` where the original program would not. It seems likely, however, that trying to preserve behaviour so precisely that even the stack layout of the virtual machine never changes would make most refactorings not behaviour preserving, so it seems justifiable to ignore this subtlety.

In order to rigorously define the notion of execution traces and their equivalence, we would need a fully developed operational semantics of Java. For the purposes of this informal argument, we instead simply assume that there is *some* operational semantics of Java that allows us to determine for a given statement

and a system state the set of possible executions starting at that statement and proceeding to the end of the surrounding method.[1] More formally, we assume the existence of a function

$$\mathit{Traces}\colon \Sigma \times \mathit{Stmt} \to \mathcal{P}(T \times O)$$

where Σ is the type of states, *Stmt* the type of statements, given as AST nodes. An execution is simply a pair of an execution trace $\tau \in T$ and an outcome $o \in O$.

An execution trace is a finite or infinite sequence, where each element of the sequence represents a method or constructor invocation. An invocation is specified by the invoked method or constructor and a list of argument values, including the receiver object in the case of instance methods.

The outcome o is of one of the following three forms:

- $o = \langle \mathit{Ret}\, v, \sigma \rangle$ where v is a value and $\sigma \in \Sigma$ a state; this indicates that execution of the method completes normally in state s, returning v. For constructors, initialisers and `void` methods, we allow v to be the special symbol $\bullet$.
- $o = \langle \mathit{Exc}\, e, \sigma \rangle$; this indicates abrupt completion in state s due to some exception e that is not caught within the method.
- $o = \bot$; this indicates non-termination.

If the trace is infinite, the outcome must be $\bot$. Note that in this case we do not care about the final heap state, since in a sequential setting it is not observable anyway.

We connect the control flow graph to the dynamic execution behaviour as follows:

Definition 5. *A* formal trace *is a (finite or infinite) path through the CFG of a method where every vertex on the path is decorated with a value that matches the node's type. Expressions of type* `void` *and statements are decorated with* $\bullet$*. Expressions that do not complete normally are decorated with the exception they throw, non-terminating method or constructor invocations are decorated with* $\bot$*. Let* $s(t)$ *denote the first node of the formal trace* t*, and* $\mathit{nodes}(t)$ *the set of all nodes appearing as vertices on* t*.*

For a formal trace t*, we define* $|t|$ *to be the execution trace resulting from it by removing all inessential vertices, just keeping invocations and their argument values. The outcome* $o(t, \sigma)$ *of* t *with respect to a start state* σ *is defined to be the returned value or thrown exception together with the updated state, or* $\bot$ *if the trace is infinite or ends with a node decorated with* $\bot$*.*

We call a formal trace t feasible *if there is a state* σ *such that* $\langle |t|, o(t, \sigma) \rangle \in \mathit{Traces}(\sigma, s(t))$*. Feasibility is only defined for formal traces that start at statements.*

We assume that execution traces are conservatively predicted by the CFG in the sense that for every execution $\langle \tau, o \rangle \in \mathit{Traces}(\sigma, s)$ there is a formal trace t such that $\tau = |t|$ and $o = o(t, \sigma)$.

To show trace equivalence, we show that executions traces can be mapped between the original method and its refactored counterpart in both directions.

Definition 6. *A method* m' *is said to* result from method m by inlining *if there is an assignment statement* d *of the form* `x = e;` *in* m*, where* `x` *is a local variable, and a set* U *of accesses to* `x` *such that* m' *is the same as* m*, except that* d *is replaced with an empty statement* d'*, and every* $u \in U$ *with a copy* e_u *of the expression* e*.*

We say that m' *results from* m *by* circumspect inlining *if additionally the following conditions hold:*

[1] For brevity, in the rest of this section whenever we say "method" we mean "method, constructor or initialiser".

1. *U is the set of all reached uses of the assignment; every $u \in U$ has d as its only reaching definition, and none of them is an lvalue.*
2. *Every name n_u in a copy e_u of e binds to the same declaration as the corresponding name n in e.*
3. *The reaching definitions of a variable read r_u in a copy e_u are the same and go through the same loops as those of the corresponding read r in e.*
4. *If there is an exceptional edge from a node x_u in a copy e_u to some node c, then there is an exceptional edge from x to c, and vice versa. Let C be the set of all such nodes c.*
5. *If e may throw exceptions, then on every path from the assignment to the method exit there occurs some $u \in U$.*
6. *If e may throw exceptions, then there is no node n with $d \to_s^* n \to_s^* u$ for some $u \in U$ such that n is a method or constructor invocation, or a write of some location that is live at a node $c \in C$.*

Our specification of INLINE ASSIGNMENT performs a circumspect inlining according to this definition. We now aim to show that the set of execution traces is not changed by the refactoring.

Lemma 4. *If m' results from m by circumspect inlining and neither m nor m' have non-terminating executions, then*

$$\mathit{Traces}(\sigma, s(m)) \subseteq \mathit{Traces}(\sigma, s(m')),$$

for any state $\sigma \in \Sigma$, where $s(m)$ is the start node of m and $s(m')$ is the start node of m'.

Proof. Consider $E = \langle \tau, o \rangle \in \mathit{Traces}(\sigma, s(m))$. We find a corresponding formal trace t by our assumption. To fix terminology, let $s(e)$ be the first subexpression of e in evaluation order, i.e., the root node of the subgraph of the CFG of m induced by the nodes in e. Note that this node is unique, since there is no way in Java to "jump" into the middle of an expression from outside it. There are two main cases to consider.

1. If $s(e) \notin \mathit{nodes}(t)$, then E never evaluates e, so by Condition 1 it cannot evaluate any of its uses either, i.e., $\mathit{nodes}(t) \cap U = \emptyset$. That means that none of the vertices of t are touched by the refactoring, hence it is a feasible formal trace of m'.
2. Otherwise, let us first restrict our attention to the case where t only contains a single occurrence of $s(e)$. There are two subcases.

   (a) If $d \notin \mathit{nodes}(t)$, then t represents an execution where the evaluation of e is started, but does not complete normally. We then know that t is of the form $\ldots, s(e), \ldots, f, c, \ldots$ where f is the subexpression of e whose evaluation throws the exception, and c the node where the exception is caught, or an exit node of m.

   We consider some execution $E' = \langle \tau', o' \rangle$ of the refactored method that arrives at $s(e)$ in the same state as E. Let t' again be the corresponding formal trace.

   Since we assume termination and by Condition 5 we know that t' must contain some $s(e_u)$ as a vertex. Take the prefix of t' up to the earliest such vertex, and create a formal trace t'' by splicing together this prefix with the suffix $s(e_u), \ldots, f_u, c, \ldots$, where f_u is the node corresponding to f in copy e_u. This is a formal trace of the refactored program: no $u \in U$ can occur on the path after c, and exceptional control flow edges are preserved by Condition 4. By Condition 6 no locations

that may be read after c are written between d' and $s(e_u)$, so this trace is also feasible. Of course $|t''| = \tau$ and $o(t'', \sigma) = o$.

(b) Otherwise, t must be of the form $\ldots, s(e), \ldots, d, \ldots$. We construct a formal trace t' from t by omitting the sub-path $s(e), \ldots, d$ and replacing every $u \in U$ that occurs on the trace with a copy of that sub-path. Clearly, t' is a formal trace of m'. It is also feasible: since declarations and reaching definitions have not changed (Condition 2 and Condition 3), variable reads in every inlined copy see the same values as in the original expression (recall Lemma 1 from Subsection 3.1.2). Hence by Condition 1 every inlined copy e_u in the refactored method evaluates to the same value as the corresponding variable read u in the original method. The effect of e_u on the heap is the same as that of u, namely none at all. Since there are no method calls in the changed segments we have $|t'| = \tau$, and also $o(t', \sigma) = o$.

If there is more than one occurrence of $s(e)$ on the trace, we can repeat the above construction for every use of $s(e)$ to arrive at a feasible trace t' corresponding to the same execution.

□

The other direction is proved similarly:

Lemma 5. *For σ, m and m' as above we have $Traces(\sigma, s(m')) \subseteq Traces(\sigma, s(m))$.*

Proof. Consider $E' = \langle \tau', o' \rangle \in Traces(\sigma, s(m'))$. We find a corresponding formal trace t' by the assumption. We distinguish similar cases as above.

1. If $d' \notin nodes(t')$, then $s(e_u) \notin nodes(t')$ for every e_u, so t' is a feasible trace of the original program.

2. As above, we first assume that t' only contains a single occurrence of d'. There are two subcases.

   (a) If there is no $s(e_u) \in nodes(t')$, then e cannot throw exceptions. We choose an execution of the original program that arrives at $s(e)$ in the same state as E' arrives at d', and find its corresponding formal trace t; note that this trace must pass through d since the expression cannot throw an exception. We construct a formal trace t'' of the original program by splicing t and t' at d. Since e cannot change the state and no further vertices of the trace are affected by the refactoring, t'' is a feasible trace of the original program with $|t''| = \tau'$ and $o(t'', \sigma) = o'$.

   (b) Otherwise, consider the first $s(e_u)$ to occur on t'. We can identify a subpath $s(e_u), \ldots, f_u, c$ where all vertices up to f_u are from the inlined copy of e, but c is not. We consider two cases.

      i. If c is an exception handler or exit node, then there cannot be any other $s(e_v)$ on the trace after c, since c must belong to a `try` block that encloses d' or it must be the method's exit node. We choose a trace t as in the previous subcase, take its prefix up to $s(e)$, append the subpath up to f, i.e., the node corresponding to f_u in e, and then the rest of t', yielding a trace t'', which is clearly a formal trace of the original program, and is feasible since no relevant write or method call can have occurred between d' and $s(e_u)$. As before, we have $|t''| = \tau'$ and $o(t'', \sigma) = o'$.

      ii. Otherwise, we claim that the subpath of t' starting at any other $s(e_v)$ for $v \in U \setminus \{u\}$ is the same as that at $s(e_u)$: since name binding and reaching definitions are preserved, every read returns the same result in every copy, so the evaluation of every copy proceeds in the same

way. We construct a trace t of the original program as follows: up to $s(e)$, it is the same as t'; then follows the common subpath from all the $s(e_u)$; then we append the rest of t', starting at d', but replacing every subpath resulting from an inlined expression with its corresponding reference to `x`.
This gives us a trace of the original program. By preservation of name binding and reaching definitions it is also a feasible trace, and $|t| = \tau'$ and $o(t,\sigma) = o'$.

As before, we generalise to the case where there are multiple occurrences of d' by considering them one by one.

□

Theorem 4. INLINE TEMP *preserves behaviour for terminating methods.*

Proof. By the previous two lemmas we can map executions from the original program to the refactored program and back again such that they have the same trace and outcome, hence the refactored method has the same behaviour as the original method. No other parts of the program are affected. □

We can slightly relax the termination requirement: inspecting the proofs of the two lemmas above, we can see that the only place where termination is required is in the case that e can throw exceptions. Even in this case, it is enough to require that any execution that reaches the assignment d eventually reaches one of the uses U, i.e., that execution cannot get stuck between definition and use of `x`. In this case, the formal traces can be extended as required in the proof, and the rest of the proof goes through. If e cannot throw exceptions, we do not need to stipulate any requirements on termination.

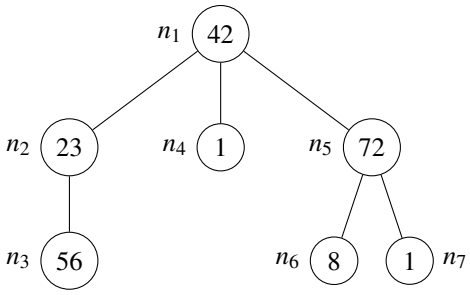
6.3 Correctness of the naming framework

The previous subsection has shown how to reason in detail, yet informally, about the correctness of individual refactorings in terms of static semantic properties such as name binding or data flow. This seems like an appropriate tradeoff, since we can gain some confidence in the correctness of our specifications without having to spend too much time on proving the correctness of simple refactorings.

For the underlying dependency framework that captures these static semantic properties in the form of dependencies, on the other hand, one might be willing to invest some more time and energy into proving their correctness. After all, this framework is used by many different refactorings, whose individual correctness crucially depends on the correctness of the framework.

Formally verifying those parts of the framework dealing with data flow and concurrency seems like quite a formidable task, since it would have to be based on a formal operational semantics of Java. As explained in Section 6.1 there does not yet appear to be a formalisation that would suit our purposes. The name binding framework, on the other hand, looks more manageable: all we need to verify its correctness is a specification of name lookup, we do not need to dive into the intricacies of formalising the operational semantics at all. Furthermore, the naming framework has a very clear-cut correctness criterion, provided by Equation 2.1, namely that access construction, the process of finding a potentially qualified name binding to a certain declaration, is a partial right inverse of access resolution, the process of looking up the declaration a potentially qualified name binds to.

In this section we present such a formalisation. At the heart of our formalisation is a shallow embedding of reference attribute grammars as they are used in JastAdd into the type theory of the Coq theorem prover.




```
MkTree 42
  [MkTree 23
     [MkTree 56 []];
   MkTree 1 [];
   MkTree 72
     [MkTree 8 [];
      MkTree 1 []]]
```

Figure 6.2: A rose tree

Abstract syntax trees are encoded as inductive data types with the help of a zipper data structure; attributes become recursively defined functions on these data types. We show how the name lookup attributes are straightforwardly translated, as are their inverse attributes for access construction. Finally, we discuss a mechanised proof of the correctness of access construction for NameJava. We also show that the formalisation is easily extended with other language features that are orthogonal to renaming.

6.3.1 Abstract syntax trees and nodes in Coq

Let us start by considering how to represent abstract syntax trees, and in particular nodes in the abstract syntax tree in Coq. Our goal is to show how to encode the grammar of NameJava, but for simplicity we will start by considering the much simpler grammar of *rose trees* over the natural numbers. Rose trees are unranked trees, where every node carries a value, in this case a natural number, so we can describe them by the following JastAdd grammar, where we assume that `nat` is the type of natural numbers.

```
Tree ::= ⟨Value:nat⟩ Tree*;
```

Grammars like this one are easily encoded as inductive data types in Coq, with one type per nonterminal of the grammar:

```
Inductive tree :=
  MkTree : nat → list tree → tree.
```

This data type only has a single constructor `MkTree`, corresponding to the single production appearing in the grammar. We make use of the type `nat` of natural numbers and the polymorphic type `list` of lists, both defined in Coq's standard library.

An example rose tree, represented both graphically as a tree and as a Coq term, appears in Figure 6.2. Every node in the tree corresponds to an occurrence of `MkTree` in the term, and is labelled with its value. Furthermore, we have given the nodes in the tree representation names $n_1, \ldots, n_7$.

Every node in the tree is itself the root of a subtree, for example node n_7 is the root of the very simple tree `MkTree 1 []` that consists of only one node. However, a node is not uniquely characterised by this subtree: for instance, the subtree of node n_4 is the same as that of node n_7. What distinguishes n_4 and n_7 is their position within the tree: whereas n_4 is the second child of the root, n_7 is the second child of the third child of the root.

A node's position can very naturally be expressed as a list of natural numbers that describe how to get from the root to that node: for instance, the path $[2]$ says to start at the root, and then take the second child; thus it describes the position of n_4. The position of n_7, on the other hand, is $[3;2]$, and the position of the root itself is the empty list $[]$. Thus we can think of a node as being given by a tree and a position within it.

Describing positions in this way has one drawback, however: there are many lists of natural numbers that do not identify any node; for instance, in our example $[3;3]$ is not the position of any node.

A much more elegant representation of positions is known as the *zipper* [53]. The basic idea of the zipper is that the position of a node can be described as a tree with a single "hole" that identifies the place where the node would go to form a complete tree. For instance, the position of node n_4 is described by the following pseudo-term:

```
MkTree 42
  [MkTree 23
     [MkTree 56 []];
   •;
   MkTree 72
     [MkTree 8 [];
      MkTree 1 []]]
```

The position of the root node n_1 is simply •. Of course, these position descriptions are no longer valid terms of type `tree`, since they contain the subterm •. It will not do to declare • as an additional constructor of type `tree`, for then we could construct terms with many bullets in them, which would no longer uniquely describe the position of a node.

One way out would be to instead define a new type `tree_with_hole` like this:

```
Inductive tree_with_hole :=
| • : tree_with_hole
| Step : nat → list tree → tree_with_hole → list tree → tree_with_hole.
```

Note that the second constructor makes sure that there is only one hole by singling out among its children exactly one `tree_with_hole`, whereas the other children are regular `tree`s. Alternatively, we could turn things around and, instead of describing the path from the root to the hole, describe the path from the hole to the root:

```
Inductive pos :=
| Top : pos
| Step : nat → list tree → pos → list tree → pos.
```

Using this representation, the position of the root node n_1 is `Top`, the position of node n_4 is

```
Step 42
     [MkTree 23 [MkTree 56 []]]
     Top
     [MkTree 72 [MkTree 8 []; MkTree 1 []]]
```

and the position of node n_7 is

```
Step 72
     [MkTree 8 []]
     (Step 42
           [MkTree 1 [];
            MkTree 23 [MkTree 56 []]]
           Top
           [])
     []
```

We follow the usual convention of recording in reverse order the elements in the first list of every `Step` constructor, i.e., those subtrees to the left of the hole. This makes it easier to find the node immediately to the left of a given node, as we shall see.

Note that our type of positions is in fact nothing more than a list of steps, so we will instead use the following definitions:

```
Inductive step :=
  Subtree : nat → list tree → list tree → step.
Definition pos := list step.
```

A syntax tree node is now quite simply described as a pair of a position and a subtree:[2]

```
Inductive node := ⟨_,_⟩ : pos → tree → node.
```

Using these definitions, node n_4 in the example is represented as

```
⟨[Subtree 42
          [MkTree 23 [MkTree 56 []]]
          [MkTree 72 [MkTree 8 []; MkTree 1 [][]]],
 MkTree 1⟩
```

and node n_7 becomes

```
⟨[Subtree 72
          [MkTree 8 []]
          [];
  Subtree 42
          [MkTree 1 [];
           MkTree 23 [MkTree 56 []]]
          []],
  MkTree 1⟩
```

While cumbersome to read, this representation gives us a concept of *node identity*: two nodes are only equal if they both have the same subtree *and* are located at the same position in the tree. Note also that every node contains all the data in the whole tree, so there is no need to carry around a representation of the entire AST.

One advantage of representing positions the way we do is that it is easy to determine the parent and the siblings of a node. If the position of a node is the empty list `[]`, then the node is in fact the root node, so there is no parent or sibling. If, on the other hand, the position is of the form $s :: p$ with s being a step `Subtree n ls rs`, then the position of the parent node is p, and its subtree is obtained by "plugging" the subtree t of the child into the hole described by step s. This is implemented in Coq as follows (where `rev` is the standard library function for reversing lists):

```
Definition plug s t := match s with
                         Subtree n ls rs ⇒ MkTree n (rev ls ++ t::rs)
                       end.
```

As another example, the left sibling of a node is determined by this function:

[2] We use a Haskell-like syntax to declare a constructor with mixfix syntax, where each underscore represents an argument position; i.e., for `p` of type `pos` and `t` of type `tree`, the constructor application would be written as `⟨p,t⟩`. In actual Coq syntax, we need an additional `Notation` declaration to achieve this, which we elide for brevity.

```
Inductive bodydecl :=
| Field : vardecl → option expr → bodydecl
| Initialiser : block → bodydecl
| MemberClass : classdecl → bodydecl
with classdecl :=
| Class : string → option access → list bodydecl → classdecl.

Inductive program :=
| Prog : list classdecl → program.
```

Figure 6.3: Coq encoding of parts of the NameJava abstract grammar

```
Definition left (nd:node) : option node :=
match nd with
| ⟨Subtree n (l::ls) rs, t⟩ ⇒ Some ⟨Subtree n ls (t::rs), l⟩
| _ ⇒ None
end.
```

The real power of the zipper, however, lies in the ease with which it generalises to more complex data structures. The basic approach stays the same: inductive data types represent trees, there is a `step` data type to represent steps from children to their parents within the tree, a `pos` type which contains a list of steps to the root, and nodes are simply pairs of a position and a tree.

The `step` data type can be derived mechanically from the grammar: a well-formed syntax tree must be derivable using the grammar, so if we have a node n in the tree with a child node n', the subtree rooted at n must have been derived using some production p, and n' must be derived using the rules of some non-terminal appearing on the right hand side of that production.

Let us consider the abstract grammar of a part of NameJava, whose Coq encoding is given in Figure 6.3. As before, inductive data types correspond to non-terminals; note that types `bodydecl` and `classdecl` are mutually defined, and hence connected by the keyword **`with`**. We use the standard library types `option` and `list` to encode optional and list children, respectively.

The `step` data type for this excerpt is given in Figure 6.4. Since the grammar has multiple non-terminals, `step` is now a doubly indexed data type, with the first index describing the non-terminal of the child, and the second the non-terminal of the parent. These indices are taken from a type `NT` that contains a code for every non-terminal of the grammar. Apart from this complication, the definition of `step` follows the same strategy as for rose trees; for instance, the type of the most complicated constructor `Class_Bodydecl` should be read as follows: the position of a `bodydecl` relative to the `classdecl` that contains it is given by a `string` (the name of the class), an optional `access` (the optional **`extends`** clause), the list of body declarations that come before it, and the list of body declarations that come after it.

Observe that optional children are treated like normal nonterminal children as far as the `step` data type is concerned: if we can give the position of an optional child, then it must obviously be present in the tree.

The indexing of steps is important when we want to construct well-typed positions: for instance, a position starting with a `step Vardecl Bodydecl` must continue with a `step Bodydecl n` for some other nonterminal `n`. Indeed, in this grammar `n` must be `Class`; the next step could then be a `step Class Program`, which would bring us to the root. This typing discipline reflects the fact, easily verified in the grammar, that a body declaration must appear as child of a class declaration, and that a class declaration can appear

```
Inductive NT := Program | Class | Bodydecl | ...

Inductive step : NT → NT → Type :=
| Field_Decl : option expr → step Vardecl Bodydecl
| Field_Init : vardecl → step Expr Bodydecl
| Initialiser_Block : step Block Bodydecl
| MemberClass_Decl : step Class Bodydecl
| Class_Super : string → list bodydecl → step Access Class
| Class_Bodydecl : string → option access → list bodydecl → list bodydecl →
                   step Bodydecl Class
| Prog_Class : list classdecl → list classdecl → step Class Program.
```

Figure 6.4: Partial step data type for NameJava

as child of a program. On the other hand, a `step Vardecl Bodydecl` should never be composable with a `step Class Program`, since the target of the first step and the source of the second step do not match.

For this reason, we define the data type `pos` of positions and the type `node` of nodes as follows:

```
Inductive pos : NT → Type :=
| Root : pos Program
| Step : ∀ n n', pos n' → step n n' → pos n.

Inductive node : NT → Type :=
  ⟨_,_⟩ : ∀ n, pos n → tree n → node n.
```

Both types are indexed over `NT`: for instance, `pos Bodydecl` is the type of positions where a body declaration could occur. In the definition of `node`, indexing ensures that the position and the subtree of the node fit together: they must both be indexed over the same code `n`. The function `tree` that maps non-terminal codes to the representing types is elided here.

Recall that in the case of rose trees and binary trees as presented above, we only have a single nonterminal, so the type `NT` has precisely one element, and the indexing becomes trivial.

Deriving the inductive data types for encoding trees, the type `NT` of codes, and the type `step` from the grammar is mechanical, but tedious. We have extended the tool Ott [120], which already offers functionality for translating abstract grammars into inductive type definitions for Coq or other theorem provers, to automatically generate definitions of `NT` and `step`, as well as several other auxiliary functions such as `plug`.

Our dependently typed representation of positions and nodes is essential for guaranteeing that only well-formed AST nodes can be represented; for instance, the definition of `plug` depends vitally on the type indices to guarantee that the subtree to be plugged in fits into the given position. However, the indices make it rather cumbersome to describe sideways movement in the tree; given a node of type `node n`, it is in general hard to tell what type, say, its right sibling might have. For instance, in NameJava a `node Access` could be the node representing the **`extends`** clause of a class, in which case its right sibling would be a `node Bodydecl`, or it could be on the left side of a dot, in which case its right sibling would be a `node Name`.

Fortunately, for the purposes of name lookup we only need a very special kind of sideways movement, namely moving to the right or left sibling within a list child. For instance, when looking up a local field within a class, we would like to iterate over all body declarations to find it. It is not difficult to implement a function `right` (and a corresponding function `left`) that returns the right sibling of a node that is part of a list child, and `None` for all other nodes; its implementation for NameJava, which again can be systematically

```
Definition right n (nd:node n) : option (node n) :=
match nd with
| ⟨Step (Prog_Classdecl lcds (cd'::rcds)) p,  cd⟩ ⇒
  Some ⟨Step (Prog_Classdecl (cd::lcds) rcds) p), cd'⟩
| ⟨Step (Class_Bodydecl C s lbds (bd'::rbds)) p,  bd⟩ ⇒
  Some ⟨Step (Class_Bodydecl C s (bd::lbds) rbds) p), bd'⟩
| ⟨Step (Block_Stmt lstmts (stmt'::rstmts)) p,  stmt⟩ ⇒
  Some ⟨Step (Block_Stmt (stmt::lstmts) rstmts) p), stmt'⟩
| _ ⇒ None
end.
```

Figure 6.5: Function to find right sibling of a node

```
(* computes number of right siblings of a node *)
Definition rdist n (nd:node n) : nat :=
match nd with
| ⟨Step (Prog_Classdecl lcds rcds) p,  _⟩ ⇒ length rcds
| ⟨Step (Class_Bodydecl C s lbds rbds) p,  _⟩ ⇒ length rbds
| ⟨Step (Block_Stmt lstmts rstmts) p,  _⟩ ⇒ length rstmts
| _ ⇒ 0
end.

(* the number of right siblings decreases as we move to the right *)
Lemma right_wf : ∀ n (nd nd':node n),
  right nd = Some nd' → rdist nd' < rdist nd.
(* proof omitted *)
Hint Resolve right_wf : measures.

(* class of types with decidable equality *)
Class Comparable (A:Type) := {
  _=?=_ : ∀ (a a':A), {a=a'}+{a≠a'}
}.
Instance Comparable string. (* proof omitted *)
Instance Comparable node.   (* proof omitted *)

(* monadic bind with swapped arguments for option type *)
Notation "f ␣=≪␣ o" := (match o with Some x ⇒ f x | None ⇒ None end).
```

Figure 6.6: Some generic utility functions used by name lookup

derived from the grammar, is given in Figure 6.5. These functions are generalisations of the attributes `prev` and `next` from Chapter 2.

6.3.2 Encoding reference attribute grammars

With the above encoding of nodes as pairs of positions and subtrees, we can now implement attributes as functions on nodes. For instance, the attribute `local_field` to look up a local field starting at a body declaration and iterating over all body declarations to the right of it (corresponding to attribute `localField` from Figure 2.12) can be implemented as shown in Figure 6.7, using some utility functions defined in Figure 6.6.

Let us briefly go over the latter first: function `rdist` (for "right distance") returns the number of right siblings of a node, which is 0 for most nodes, except those that occur in list children. The lemma `right_wf`

```
(* looks up f as a local field on nd and its right siblings *)
Function local_field (f:string) (nd:node Bodydecl) {measure rdist nd}
  : option (node Vardecl) :=
  match nd with
  | ⟨p, Field (Var _ f' as d) i⟩ ⇒
      if f =?= f' then
        Some ⟨Step (Field_Decl i) p, d⟩
      else
        lookup_local_field f =≪ right nd
  | _ ⇒ lookup_local_field f =≪ right nd
  end.
(* termination proofs *)
Proof.
  auto using measures.
  auto using measures.
Defined.
```

Figure 6.7: Local field lookup as a Coq function

proves the rather obvious fact that this distance decreases with every invocation of `right`; we add this lemma to the hint database `measures`, which we will use for termination proofs later.

Coq data types do not come with a built-in equality check; for most inductive data types, however, such a check is easy to implement. We define a type class `Comparable` that provides a comparison operator `_=?=_`, and show that `string` and `node` are instances of this class, i.e., that their equality is decidable.

Finally, we define a notation for the monadic bind operator for the `option` type.

Given these utility functions, `local_field` is easy to write: for a given node, check whether it is a declaration of a field, and compare the field name to the name we are looking for. If they match, we return the node corresponding to the declaration, otherwise we recursively invoke `local_field` on the next body declaration to the right. For the rightmost body declaration, `right` will return `None`, so the whole bind construction will evaluate to `None` as well.

Like many systems based on dependent type theory, Coq requires all recursive functions to be terminating. By default, all functions must be structurally recursive, i.e., recursive invocations have to happen on a subterm of the argument on which the function recurses. This, of course, is rather restrictive; in this example, for instance, we want to perform a recursive invocation on the right sibling, which is not structurally smaller than the node itself.

Fortunately, Coq's type system is powerful enough to encode arbitrary well-founded relations as structurally recursive ones. The **`Function`** package, which we use in this example, provides some syntactic sugar to make functions that are defined by well-founded recursion almost as easy to express as those that are defined by structural recursion. It needs an additional **`measure`** annotation that takes a measure function, here `rdist`, and one of the function arguments, here `nd`. When applied to that argument, the measure function should return a natural number.

In order to prove termination, we now need to show that on every recursive invocation the measure function returns a smaller number for the argument of the recursive call than it did for the original argument. In this case, **`Function`** generates two proof obligations to establish this for both recursive calls, which are easily discharged using the built-in tactic `auto`, if we tell it to use the `measures` database, which contains the lemma `right_wf`.

```
Fixpoint lookup_class (A:string) n (p:pos n) (t:tree n) : option (node Class) :=
  match p, t with
  | Root, _ ⇒
      None
  | Step p (Prog_Classdecl _ _ as s), cd ⇒
      toplevel_class A ⟨p, plug s cd⟩
  | Step p (Class_Bodydecl _ _ _ _ as s), bd ⇒
      match member_class A p (plug s bd) with
      | Some d ⇒ Some d
      | None ⇒ lookup_class A p (plug s bd)
      end
  | Step p s, t ⇒
      lookup_class A p (plug s t)
  end.
```

Figure 6.8: Implementing class lookup in Coq

Attributes `local_variable`, `local_class` and `toplevel_class`, corresponding to the JastAdd attributes `localVariable`, `localClass` and `toplevelClass`, are implemented very similarly, again using well-founded recursion on the right-sibling relation.

As an example of an inherited attribute, let us consider `lookup_class`, the attribute that looks up an identifier as a class name by traversing nested scopes, using `member_class` to look up member classes of classes and `toplevel_class` to look up toplevel classes within the program; the code is shown in Figure 6.8.

As an inherited attribute, `lookup_class` matches on the position of the node it is invoked on. If the node is the root node, the search terminates and returns `None`. Otherwise the position is of the form `Step p s`, where `p` is the position of the parent node and `s` describes the location of the matched node relative to its parent. The attribute then matches on `s`: if `s` is `Prog_classdecl`, then the node is a toplevel `Classdecl` node, so we invoke `toplevel_class`, which is implemented similar to `local_class` above.

Similarly, if `s` is `Class_bodydecl`, we first try to find class `A` using `member_class`; if that function returns `None`, however, we recursively invoke `lookup_class` on the parent node. That is also what we do if `s` is neither of these two. We pass the position and the subtree as separate arguments simply to make termination obvious enough for Coq's **`Fixpoint`** construct to accept the definition without further proof.

Note the close correspondence between the cases of the **`match`** construct of `lookup_type` and the equations we give for inherited attributes in JastAdd: the `Root` provides a default value for when attribute evaluation reaches the root; the second and third cases match on specific parent-child patterns, which we would write `Program.getClassdecl()` and `Classdecl.getBodydecl()` in JastAdd; and the final case implements the default copy rule that simply returns the value the attribute evaluates to on the parent node. Any inherited attribute can be encoded by a match construct like this.

It remains to implement the attribute `member_class` and an auxiliary attribute `lookup_classname` to look up potentially qualified class names. This attribute allows us to decouple type lookup from name disambiguation, which somewhat simplifies proofs: remember that in the JastAdd version `memberClass` invokes `type`, which invokes `decl`, which does disambiguation; but of course we already know that the superclass access is a class access. We might try to implement these two attributes as two mutually recursive functions as shown in Figure 6.9.

The intuition behind this code is fairly simple: to look up a member class `A` on a class declaration `nd`, first check whether there is a locally declared class of name `A` in `nd`; if there is, return that class. Otherwise,

```
Fixpoint member_class (A:string) (nd:node Classdecl) :=
  match local_class A nd with
  | Some nd ⇒ Some nd
  | None ⇒
    match nd with
    | ⟨p, Classdecl _ None _⟩ ⇒
      None
    | ⟨p, Classdecl _ (Some S) _⟩ ⇒
      member_class A =≪ lookup_classname S nd
    end
  end
with lookup_classname (tn:access) n (nd:node n) :=
  match tn with
  | SimpleAccess A ⇒
    lookup_class A nd
  | QualAccess (AccessAsExpr q) B ⇒
    member_class B =≪ lookup_classname q nd
  | _ ⇒ None
  end.
```

Figure 6.9: A failed attempt at implementing member class lookup in Coq

check whether `nd` declares a superclass. If it does not, return `None` to indicate that no member class of name `A` exists. Otherwise, look up the superclass, and then recursively look up `A` as a member of the declaration of the superclass. The definition of `lookup_classname` simply recurses over the structure of the access, looking up qualified classes as members of their qualifying classes.

There are two problems with this definition: firstly, there is a mutual dependency between `lookup_class`, `lookup_classname` and `member_class`, so they should be declared by mutual recursion and assembled into a single **`Fixpoint`** definition. Secondly, and far more seriously, the recursion in `member_class` is not actually well-founded.

To see that this is not an artifact of the formalisation, consider the following Java program:

```
class A extends B { }
class B extends A { }
```

If we were to look up a member class, say, `C` on class `A`, we would not find it as a locally declared class, so we would recursively look it up on class `B`, where we again would not find it, recurse back to `A`, and so get caught in an infinite loop.

Of course, the above program is not a valid Java program: the language specification forbids cycles in the inheritance hierarchy, so in a valid Java program recursion up the inheritance chain is always well-founded. The catch is that in order to determine whether a class hierarchy is acyclic (or indeed to formulate what it means for the class hierarchy to be acyclic!) we already need an implementation of class lookup.

6.3.3 Encoding non-terminating attributes

There are several ways to approach this problem. One could, for example, observe that `member_class` certainly never needs to search the same class declaration twice, so we can bound the maximal number of recursion steps by the total number of classes in the program. That number is, of course, easily determined by a (well-founded) recursion over the abstract syntax tree of the whole program.

```
CoInductive #_ (A:Type) :=
| Ret : A → #A
| App : ∀ B, (B → #A) → #B → #A.

Notation "'call'" := (App Ret).
Notation "x ← c ; r" := (App (fun x ⇒ r) c).
```

Figure 6.10: The coinductive computation monad

We can then furnish our implementation of `member_class` with an extra parameter `ttl` (for "time to live"), a natural number that indicates the number of recursive calls we are allowed to make. On every recursive invocation, `ttl` is decreased by one; if it drops to zero, we return `None` instead of performing a recursive invocation. The definition of `member_class` then becomes structurally recursive on the parameter `ttl`, and is accepted by Coq.

The drawback of this solution is that the addition of the `ttl` parameter obscures both implementation and proofs. Arguably, the naming framework should not have to worry about whether or not lookup terminates; after all, its correctness can be phrased relative to termination, stating that *if* access construction terminates and returns a name, and *if* looking up that name terminates, it will return the declaration we were constructing an access to.

Of course, we can then no longer hope to encode lookup, or access construction, for that matter, as a recursive function, since those always have to terminate. Fortunately, in recent years there has been some work on encoding potentially non-terminating programs in type theory [15, 21]. The latter, in particular, makes use of coinductive types and corecursive functions to achieve the encoding.

A coinductive type in Coq looks syntactically just like an inductive type, but whereas (intuitively) the elements of an inductive type are all finite trees that can be built up using its constructors, a coinductive type also includes all the infinite trees that can be constructed in this way. A corecursive function is a function that produces elements of a coinductive type. Unlike a recursive function, a corecursive function does not have to terminate (otherwise it would be impossible to construct infinite trees), but it has to satisfy a so-called *productivity requirement*, meaning roughly that on every recursive invocation it has to contribute at least one constructor to the tree it is building up.

To represent potentially non-terminating computations, we use the coinductive computation monad introduced by Megacz [90], the definition of which is shown in Figure 6.10. Just like an inductive type, the type of computations has constructors, in this case two: an element of type `#A`, representing computations that either diverge or return an element of type `A`, is either of the form `Ret a`, representing a computation that terminates and returns `a`, or of the form `App f c`, representing a computation that first evaluates `c`, and then, if it terminates and returns some value `b`, proceeds to evaluate `f b`. We also introduce some syntactic sugar for a Haskell-like "do notation".

A terminating computation is represented by a term built up by finitely many applications of these constructors, a non-terminating computation by an infinite term. For instance, the infinite term

```
App Ret (App Ret (App Ret ...
```

can be built by the following corecursive function:

```
CoFixpoint diverge := App Ret diverge.
```

```
CoFixpoint lookup_class (A:string) n (nd:node n) : #option (node Class) :=
  match nd with
  | ⟨Root, _⟩ ⇒
      Ret None
  | ⟨Step (Prog_Classdecl _ _ as s) p, cd⟩ ⇒
      Ret (toplevel_class A ⟨p, plug s cd⟩)
  | ⟨Step (Class_Bodydecl _ _ _ _ as s) p, bd⟩ ⇒
      r ← member_class A ⟨p, plug s bd⟩;
      match r with
      | Some cd ⇒ Ret (Some cd)
      | None ⇒ lookup_class A ⟨p, plug s bd⟩
      end
  | ⟨Step s p, t⟩ ⇒
      call (lookup_class A ⟨p, plug s t⟩)
  end
with member_class (A:string) (nd:node Classdecl) : #option (node Class) :=
  match local_class A nd with
  | Some nd ⇒ Some nd
  | None ⇒
    match nd with
    | ⟨p, Classdecl _ None _⟩ ⇒
      Ret None
    | ⟨p, Classdecl _ (Some S) _⟩ ⇒
      App (member_class A) (lookup_classname S nd)
    end
  end
with lookup_classname (tn:access) n (nd:node n) : #option (node Class) :=
  match tn with
  | SimpleAccess A ⇒
    call (lookup_class A nd)
  | QualAccess (AccessAsExpr q) B ⇒
    App (member_class B) (lookup_classname q nd)
  | _ ⇒ Ret None
  end.
```

Figure 6.11: Corecursive encoding of class lookup in Coq

Notice that `diverge`, being corecursive, can call itself on any argument of the right type, no matter whether it is structurally greater or smaller, or even without any argument at all, as in this case, as long as the invocation is part of a constructor application, here of constructor `App`. In our intuitive operational reading of the computation monad, the computation `App Ret c` behaves the same as `c`, but it always obeys the productivity condition; hence we introduce the handy abbreviation `call c` for it.

Given these definitions, we can now rescue our failed attempt at implementing class lookup by converting it into a corecursive definition, as shown in Figure 6.11.

To preserve termination of the type theory, corecursive definitions in Coq are evaluated lazily: the next recursive step is only evaluated when required by another function. Hence, while it is easy to write functions that produce computations of type `#A`, there is no direct way to "run" these computations within Coq.

To get around this problem, Megacz defines a judgement `_↓_`, read "evaluates to", where `c ↓ x` means that computation `c` terminates and returns value `x`. This judgement is easily written up as an inductive data type in Coq, as shown in Figure 6.12 on the left; the two constructors of this type, corresponding to derivation rules for the judgement, codify the informal operational meaning of the computation monad given above.

Since this is an inductive type, its elements (which are proofs of the judgement) are of finite size, and

```
Inductive _↓_ : ∀ A, #A → A → Prop :=
| RetEval : ∀ A (a:A), Ret a ↓ a
| AppEval : ∀ A B (f:A↛#B) (c:#A) a b,
    c ↓ a → f a ↓ b → App f c ↓ b.
```

```
CoInductive _⇝_ : ∀ A, #A → A → Prop :=
| RetYields : ∀ A (a:A), Ret a ⇝ a
| AppYields : ∀ A B (f:A↛#B) (c:#A) a b,
    c ⇝ a → f a ⇝ b → App f c ⇝ b.
```

Figure 6.12: Judgements denoting results of computations

```
Inductive symbolic_class_access (cd:node Class) :=
| TLAcc : symbolic_class_access cd
| MemClassAcc : node Class → bool → symbolic_class_access cd.

Definition move_into cd (sta:symbolic_class_access cd) (cd':node Class) :=
  r ← member_class (class_name cd) cd';
  match r with
  | Some _ ⇒
    match sta with
    | TLAcc ⇒ Ret None
    | MemClassAcc host _ ⇒ Ret (Some (MemClassAcc host true))
    end
  | None ⇒ Ret (Some sta)
  end.
```

Figure 6.13: Symbolic class accesses in Coq

can be recursed over. One can then write a function `eval` which, given a computation `c` and a proof that it evaluates to a value, returns that value.

We are, however, not particularly interested in evaluating our lookup attributes, but rather in proving properties about them. It turns out that formulating such proofs in terms of the _↓_ judgement is rather cumbersome, since it again forces us to reason about the termination of lookup and access construction, which we want to avoid.

Hence we use a different judgement _⇝_, read "yields" and shown in the same figure on the right, that looks almost the same as _↓_, except that it is coinductive. For this reason, we cannot understand `c ⇝ x` to mean "`c` evaluates to `x`": after all, the proof term could be infinite. This judgement rather means "if `c` terminates, then it returns `x`". It is not difficult to show that `c ↓ x` and `c ⇝ x'` together imply `x=x'`, i.e., the two judgements agree on terminating computations.

6.3.4 Access construction

In the previous subsection we have shown how to encode class lookup for NameJava as a corecursive function. Variable lookup can, of course, be implemented in the same manner. Quite analogous to the development in Chapter 2, we can now manually invert these functions to obtain access construction functions. Just like there we introduce symbolic accesses as an abstract representation for class accesses; for instance, symbolic class accesses are defined as shown in Figure 6.13.

The type `symbolic_class_access` is indexed over the class it is providing a symbolic access for. A `TLAcc` is a symbolic access to a toplevel class, whereas a symbolic access of the form `MemClassAcc host b` records that `cd` is locally defined in class `host`, with the boolean flag `b` indicating whether or not the access to be built must be qualified.

The function `move_into` moves an access `sta` to a class `cd` into a class `cd'`. To do this, it checks whether `cd'` has a member class of the same name as `cd`; if that is the case, the symbolic access needs to be updated to reflect that qualification is now necessary. Toplevel classes cannot be qualified in NameJava, so `move_into` returns `None` to indicate that the class becomes inaccessible. For member classes, it simply turns on the qualification flag. A similar function `move_downto` moves an access that is valid in a superclass to a subclass, again adding qualification where necessary.

Using these two functions, it is easy to define an access construction function for classes that parallels the definition of `lookup_class`. Its type is

```
access_class : ∀ (cd:node Class) n, node n → #option (symbolic_class_access cd)
```

indicating that, given a class `cd` to build an access to and a node from where to build the access, it will either return a symbolic class access to `cd`, or `None` if this is impossible; as indicated by the hash mark, this function might not terminate.

To prove the correctness of `access_class`, we define a function `resolve_scacc` that simulates the process of splicing the symbolic class access into the AST and determining its declaration. We can then state and formally prove the following correctness criterion for `access_class`:

```
Lemma access_correct : ∀ cd n (nd:node Classdecl)
                                (scacc:symbolic_class_access cd),
  access_class cd nd ⇝ Some scacc →
    resolve_scacc scacc nd ⇝ Some cd.
```

Encoding name lookup and access construction for NameJava, and proving this correctness criterion for class access construction and the corresponding criterion for variable access construction requires around 800 lines of specification and 350 lines of proof script, as determined by the `coqwc` utility.

Of course, NameJava is only a tiny subset of Java, so one might object that this is yet again only a fruitless exercise in miniaturisation, where a desired property can be proved on a well-behaved mini-language without any realistic hope of extending it to the full language.

Fortunately, this is less true for naming than for many other language features: after all, most language constructs in Java do not affect naming at all. For instance, introducing an **`if`** statement into the language only requires nine additional lines of specification, all of which can be auto-generated from the abstract grammar. Neither lookup nor access computation have to be changed, and all the proofs still go through unchanged. For language features such as the **`for`** loop, which do influence the lookup without introducing any new naming related concepts, the effort is still moderate: about 60 lines of specification, and 80 lines of proof. Adding, e.g., interfaces would certainly require more effort, but the experience of JastAddJ shows that reference attribute grammars are powerful enough to handle the full Java language.

6.4 Related work

The results in Section 6.3 have previously been published in the paper "Formalising and Verifying Reference Attribute Grammars in Coq" at ESOP 2009 [117], which is joint work with Torbjörn Ekman and Oege de Moor. The formalisation and the proofs are the author's work, the co-authors made important contributions to presentational aspects. The material in the other sections of this chapter has not been published before.

Some related work on the correctness of refactorings has been alluded to above, in particular Opdyke's original definition as preservation of input-output behaviour [100], and Roberts' approach of substituting

refactoring-specific postconditions for a general criterion of behaviour preservation [111]. A third approach is found in Griswold's thesis [49]: his object language is a first-order subset of Scheme with a straightforward procedural semantics; this semantics can be captured by a program dependency graph (PDG), and Griswold formulates his refactorings in terms of meaning-preserving graph transformations. Work by Horwitz and others on procedure extraction uses a similar approach [72]. Program dependency graphs are most suitable for reasoning about the semantics of procedural languages without heap-allocated variables: in this case it can be shown that PDGs are an adequate representation in the sense that programs with isomorphic PDGs are strongly equivalent, i.e., on any given input state they either both diverge, or both terminate in the same state [52]. Suitability for programs with heaps is less clear; although a similar adequacy theorem can be proved in this case [106], the PDGs need to be refined with dynamic information only available at runtime.

The correctness of refactorings is usually only established by an informal argument [27, 100, 111]. In recent years, several researchers have taken a more formal approach: Ettinger [41] establishes the correctness of a slice-based refactoring using predicate transformers. The previously mentioned formalisation of several Java refactorings in Maude [47] is accompanied by (informal) correctness proofs.

Li and Thompson [83] prove the correctness of two refactorings for an extended lambda calculus using equational reasoning. Cornélio [31] formalises refactorings as refinement rules on the refinement-oriented language ROOL, the correctness of which can be established by algebraic reasoning; these rules and their correctness proofs have more recently been mechanised in the algebraic specification language CafeOBJ by Júnior *et al.* [63]. Bannwart and Müller [9, 10] combine formal reasoning about behaviour preservation based on an operational semantics of a small subset of Java with the use of assertions that are inserted into the refactored program to dynamically check postconditions. Their formalisation of program behaviour is very similar to the execution traces we use in our correctness argument for INLINE TEMP.

The most rigorous correctness proofs that we are aware of have been given by Sultana and Thompson [127], who formalise several refactorings in the Isabelle theorem prover [101], and mechanically check their correctness. Their work, again, is restricted to extended lambda calculi, and they do not provide any substantial evidence for the feasibility of their techniques beyond this simple setting.

A common feature of these formal correctness proofs is the inordinate amount of work required even for proving the correctness of simple refactorings on simple languages. It is likely, moreover, that these proofs are very sensitive to details of the implementation: if the implementation changes, the proofs have to be redone. The same problems, of course, arise in the formal verification of optimising compilers. One very promising approach to dealing with such issues is the concept of *translation validation*, first introduced by Pnueli *et al.* [109]. Instead of proving that the output program generated by the compiler or one of its optimisation passes is always semantically equivalent to the input program, they employ a validator which checks semantics preservation for a single run of the compiler.

The crucial question, of course, is how to implement the validator, and how to ensure *its* correctness. One approach is for the validator to generate verification conditions to be proved by a theorem prover [142]. A less powerful but potentially more efficient approach is to use abstract interpretation to obtain a symbolic representation of program behaviour, and then compare these representations to prove semantics preservation. This approach was used, for instance, by Necula [97] to implement a translation validator for optimisations in the GCC compiler. His approach uses symbolic execution of basic blocks of intermediate code to summarise their effects on memory registers. Often, the correctness of the validator is established by informal proof only; more recently, however, Tristan and Leroy [131] have reported on the implementation of a validator whose correctness has been mechanically proven using Coq.

Our approach of checking dependency preservation to ensure correctness of refactorings has a certain similarity to translation validation, which it would be interesting to explore further. Following Necula's approach, one could even imagine using a form of symbolic execution to check semantics preservation of INLINE TEMP and EXTRACT TEMP. Symbolically executing source code, however, is likely to be much more complicated than for intermediate code, so the correctness of the validator could be difficult to establish.

A much more lightweight approach is automated testing of refactoring engines, as pioneered by Daniel and others [32]. They present the ASTGen tool, a combinator-based syntax tree generator library, which can be used to randomly generate input programs for refactoring engines. While they were able to discover some bugs in widely used refactoring engines, their work showed that it is challenging to generate test programs that are both syntactically correct and interesting from a refactoring perspective.

We appear to be the first to discuss an embedding of reference attribute grammars into type theory, but there has been some work on the verification of attribute grammars before. Katayama and Hoshino [64] discuss the verification of properties of attributes in classical attribute grammars without references. They derive some general principles and a high-level procedure for generating sufficient verification conditions to establish such properties. Their method relies on a building a dependency graph for attributes, which is undecidable in the presence of reference attributes. Bauer [11] presents an induction principle for inherited attributes, which is similar to the induction principle we obtain from representing nodes using zippers.

Traditionally, attribute grammars have been encoded in lazy functional languages as families of functions, one per nonterminal, that take inherited attributes as parameters and return synthesised attributes as results [61]. Our method represents different attributes by different functions, thus staying closer to the original attribute grammar. It is not clear if the traditional encoding could accommodate reference attributes. The zipper has been used before to implement inherited attributes [6, 132], but without making the connection to reference attributes.

One major shortcoming of our formal verification effort is that the verified code is an encoding of the naming framework, not its actual JastAdd implementation. Currently, there is no tool support for formalised reasoning about JastAdd grammars. However, since the attribute definitions are translated into plain Java one could use a verification tool for Java, such as Krakatoa [42, 88], to reason about the generated code. This would likely be an arduous task, since the generated code exposes details of the evaluation mechanism that are implicit in the JastAdd definition. Furthermore, in order to formalise the inversion property of access construction, one would have to axiomatise name lookup inside the logic of the verification tool.

While the access construction machinery is an important component of the common framework underlying the individual refactorings, it is not the only one. Control and data flow analysis, for instance, also play a very important role, but we have not yet attempted to verify our implementation of these analyses. Since the analyses we use are based on similar concepts as the data flow analyses used in optimising compilers, it seems likely that approaches developed for the verification of such analyses could be reused. Techniques for automatically deriving soundness proofs of data flow analyses, as in the Cobalt [78] and Rhodium [79] systems, seem particularly appealing. The usual caveat applies, however, that it is not clear how easily techniques designed for working on intermediate representations generalise to the source level.

Chapter 7

Discussion

The previous chapters have presented the concepts and techniques underlying our approach to the specification, implementation and verification of refactorings. In this chapter we take a step back and view our work in a broader context. First, we will discuss possible applications and extensions beyond Java. While we have made this language the focus of our investigation as an example of a modern real-world programming language, much of our approach carries over to other object-oriented languages, and even beyond.

Next, we will revisit our choice of JastAdd as the implementation language for our refactoring engine. We will analyse to what extent this choice has shaped our implementation, and relate our experience with its benefits and shortcomings. We briefly consider other languages that have been used to implement refactorings, highlighting some of their salient features from the viewpoint of refactoring.

The next section is devoted to common features of popular refactoring engines that we have not implemented, and more generally to open problems that our approach does not directly solve. Some of these are rather practical problems that our implementation could be extended to support with some engineering effort, while others are deep issues that would require further research to solve satisfactorily.

We conclude this chapter and the present dissertation with a general assessment of what has been attempted, what has been achieved, and what remains to be done.

7.1 Applicability beyond Java

Software refactoring is widely accepted as a standard technique in the world of object-oriented programming, with support for automated refactoring available in most modern IDEs for Java and C#. But refactorings have also been proposed and implemented for other languages, for instance Haskell [82] and Erlang [81]. The work described in this dissertation has exclusively focussed on specifying, implementing and verifying refactorings for Java, so it is only fair to ask whether any of our development carries over or can be extended to other programming languages.

We think that the dependency-based approach to handling the preservation of static semantic properties that we have advocated here for Java is probably applicable to many other programming languages as well, although the precise definition of the dependencies, or even what dependencies to preserve, may differ between languages. The language restrictions and extensions we have discussed in Chapter 4 are much more specific to Java, as handling its peculiarities is their *raison d'être*. However, one could certainly hope that the concepts of language restrictions and extensions may be helpful in other languages as well.

	Normal Access	Volatile Read	Monitor Enter	Volatile Write	Monitor Exit
Normal Access				×	×
Volatile Read	×	×	×	×	×
Monitor Enter	×	×	×	×	×
Volatile Write				×	×
Monitor Exit			×	×	×

Table 7.1: C# Reordering Matrix

We will now consider in slightly more detail some of the static semantic dependencies we have introduced in Chapter 2 and Chapter 3 and consider their applicability to other languages.

Naming The concept of name binding is universal, and the requirement that a RENAME refactoring should not change the name binding structure of a program makes sense in every language that we are familiar with. While lookup rules differ between languages, the interpretation of access construction as a right inverse of lookup gives us an implementation strategy for implementing and preserving name binding dependencies for any language.

Java's name lookup rules are exceptionally complicated; for languages with simpler rules it may not be worth the effort to invert the name lookup rules one by one if a simpler access construction process suggests itself. This is especially true for functional languages, which generally tend to have fewer ways of qualifying names to bind to declarations that are not directly visible.

The case of overriding dependencies in Java shows that special semantic features such as dynamic method dispatch may make it necessary to extend the basic picture of renaming beyond locking names.

Control and data flow Control and data flow are likewise concepts that transcend language boundaries, but their applications in refactoring are much less straightforward than that of name binding. In the case of INLINE TEMP, for instance, we had to carefully formulate and prove several lemmas about data flow and its preservation in order to reassure ourselves that our dependencies and preconditions were sufficient to preserve behaviour. This makes it seem unlikely that these conditions are portable across languages, although it may be possible to adapt our specification of INLINE TEMP to a language like C# that is fairly closely related to Java.

Synchronisation Our definition of synchronisation dependencies and their preservation conditions is entirely based on the Java Memory Model, and hence *prima facie* not portable to other languages. That being said, the concept of synchronisation dependencies seems to be directly applicable to C#: its memory model is defined in terms of a reordering matrix similar to that given by Lea, which we have seen in Table 3.1. The analogous matrix for C#, taken from [54], is given in Table 7.1. We can see that actions are categorised in the same way as for Java, and the reordering constraints are almost identical, except that there is greater freedom for reordering with respect to volatile reads and writes. This matrix could be used as the basis of a definition of synchronisation dependencies for C#, which could be used in the same way as the corresponding dependencies for Java to prevent potentially behaviour-changing reorderings.

7.2 Using attribute grammars to implement refactorings

Initially, our decision to use JastAdd for implementing refactorings was influenced by purely practical considerations: we had a modular, extensible Java compiler frontend at hand, namely JastAddJ, which seemed like a good basis for a refactoring engine. It was then an obvious choice to implement the engine using JastAdd as well.

This choice has certainly turned out to be very fortunate. First and foremost, we could reuse many existing analyses provided in the compiler frontend, such as its name binding and type analysis, but we could also immediately benefit from other developments in the JastAdd community, for instance the source level control flow framework implemented by Nilsson-Nyman and others, which was later refined by Mao.

In a wider sense, the reference attribute grammar paradigm, which views attributes as overlaying a more general graph structure onto the tree-structured abstract syntax, has had a marked influence on our dependency-based approach to the specification and implementation of refactorings.

JastAdd's modularity mechanisms, which allow the specification of both abstract syntax and attributes to be freely distributed over different modules, make it easy to extend the object language, which has greatly simplified the implementation of those refactorings that need language extensions. Adding language extensions would presumably have necessitated many more global changes in a more traditional, purely object-oriented frontend without JastAdd's aspect-oriented modularity features.

One very basic design choice with JastAdd is that although attribute definitions encourage a declarative programming style, this is never enforced, and attributes can be defined using arbitrary Java code, including side effects and mutation. This has at times been helpful, and even necessary, since JastAdd provides no specific language features to express syntax tree transformations, which have to be expressed as imperative tree modifications.

However, this is also the Achilles' heel of JastAdd: for performance reasons, many attribute values have to be cached to avoid spurious recomputation, but with arbitrary imperative code accessing the tree it is very difficult to determine when such cached values have to be discarded and recomputed. JastAdd does not attempt to do this at all, instead exposing the attribute cache mechanism to the programmer, so that they can flush stale attribute values by hand.

For many refactorings it is very hard to tell which attributes have been invalidated in which part of the tree: inheritance between classes means that changes to one class may have repercussions in all its child classes, which can be located anywhere in the program; the flexible way in which class definitions can be nested in Java only compounds the problem.

Our implementation hence errs on the side of caution, generously flushing attributes after every transformation. While for some refactorings, such as RENAME, this does not seem to impact performance notably, it incurs a heavy performance penalty for others, in particular for highly modular refactorings that are composed of many microrefactorings.

The problem of how to efficiently recompute 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 to handle this issue transparently. The classic work of Demers, Reps and Teitelbaum [33] 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 [103, 113] concentrates on caching attribute values, but again for traditional attribute grammars without reference attributes.

The doctoral thesis of Maddox [85] presents an incremental attribute grammar system with some more advanced features, although the eclectic nature of the system makes it hard to judge to what extent the results would apply to JastAdd. More recently, Boyland has proposed an incremental evaluation mechanism for remote attribute grammars [16], 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 it seems unrealistic to expected 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 [1–3], 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.

7.3 Alternative implementation languages

While JastAdd seems to provide a good balance of declarativity and power, it is, of course, not the only possible language for implementing refactorings. In this section, we will briefly discuss some alternatives.

Verbaere's JunGL language [134, 135] is a domain-specific language for implementing refactorings. It combines an imperative core with ML-like algebraic data types for representing syntax trees and functions defined by pattern matching with features for defining attributes on and edges between AST nodes. The central mechanism for defining edges are path queries, which are a kind of regular expression for describing paths in the syntax tree.

Path queries can be used to give elegant descriptions of many of the static semantic analyses required by refactorings; in particular control and data flow properties become very easy to describe. As with JastAdd, however, JunGL provides no high-level support for expressing transformations, instead relying on imperative modification of the abstract syntax tree.

An alternative approach is to use term rewriting, implemented in systems like Maude [28], Stratego [17], Tom [8], or Strafunski [76], for implementing refactorings. These languages make it easy to describe syntax tree transformations and provide a clean semantic model for reasoning about them.

Generally, they tend to view syntax trees as terms without a concept of node identity. Usually, terms are maximally shared, i.e., structurally identical subtrees are only stored in memory once. Contrast this with the situation in JastAdd or JunGL, where an AST node is characterised not only by its subtree, but also by its position within the whole tree. Features such as inherited attributes or JunGL edges crucially rely on node identity, and hence have no direct equivalent in a term-based representation of syntax trees.

A recent attempt at bridging the gap between analysis-oriented languages and transformation-oriented ones is the RASCAL language [70], which combines term rewriting for transformation with relational algebra and logic programming for analysis. Syntax trees are again represented as maximally shared terms. It will be interesting to see how well RASCAL succeeds in expressing, for instance, object-oriented lookup or control and data flow analyses, whose formulation in JastAdd and JunGL is greatly simplified by exploiting node identity.

JastAdd comes with some limited support for rewriting, which is purely used for purposes of normalisation: the programmer can specify conditional rewrite rules for nodes that are applied whenever a node is accessed through the `getChild` method of its parent node. For instance, JastAddJ contains rewrite rules that insert a default constructor into classes that do not explicitly declare one, and rules to break up compound

variable declaration statements. These rewrite rules are implicitly applied as part of the attribute evaluation process, and hence cannot directly be used to implement the transformations needed for refactoring.

7.4 Unsupported features

While the evaluation in Chapter 5 shows that our implementation compares favourably with industrial-strength engines, it is only a prototype and not meant as a tool for developers. As such, it is missing many pragmatic features that programmers have come to expect from refactoring engines. We will discuss some of these features here and consider what would need to be done to support them in our implementation.

IDE integration Currently, our implementation is not integrated into a development environment. It has a command-line interface that we use for testing purposes, but in its current status it is obviously unusable for developers. We have experimented with an integration into Eclipse's LTK toolkit in the context of an effort to achieve better integration of JastAdd-based solutions into Eclipse, which, however, has not led to either usable tools or formal publications.

Our experiments have shown that there are no fundamental obstacles to such an integration: the LTK toolkit is not tied to the internal AST representation used by Eclipse, so we do not need to fundamentally change the implementation of our refactoring engine. There is, however, some work to be done in smoothly integrating our implementation with the UI; in particular, LTK assumes that refactoring implementations are based on the classic precondition paradigm, which our implementation is not. Instead of checking preconditions, then, our implementation would have to go ahead and immediately perform the refactoring, catching any exceptions and reporting them as precondition failures. This does not fit very well with the Eclipse model in which precondition checks are supposed to be quite fast, especially given the performance problems our implementation currently suffers from anyway.

Layout and comment preservation A crucial feature of refactoring support in IDEs is the preservation of program layout and comments: a refactoring should preserve original program layout as much as possible, and transform comments along with the code to which they belong. The latter is especially important for Javadoc comments associated with program elements such as classes or methods; for instance, when a method is moved to a different class, its Javadoc comment should be moved along with it. Most state-of-the-art refactoring engines even have some support for refactoring arbitrary comments; for instance, when renaming a program element, they may try and update references to the renamed elements inside comments.

By its very nature, this is a deeply heuristic feature. While the structure of Javadoc comments is standardised to a certain degree, and hence amenable to refactoring, general comments have no such fixed structure. Our current prototype implementation does not support preservation or refactoring of either layout or comments: the refactored source code is pretty-printed in a standardised layout, discarding all comments.

While it has not been a major focus of refactoring research, there has been some work on layout preservation in the literature. Li [80] discusses the approach taken by the Haskell refactoring engine HaRe. In Haskell, layout preservation is an essential feature, since the parser takes indentation into account when parsing nested constructs. The approach taken by HaRe is to concurrently maintain two views of the program, one as an abstract syntax tree which does not contain layout information, and one as a token stream that contains precise information about whitespace and comments. Care has to be taken to keep these two representations consistent, especially when creating new pieces of code.

An alternative approach has been proposed by van den Brand and Vinju, and implemented in the ASF+SDF system [133]: they extend the abstract syntax tree with *layout nodes*, which are ignored when matching rewrite rules. Thus the layout of those parts of the program that are not rewritten is preserved, but any layout information for rewritten parts is discarded. Vinju has later extended this line of work towards a first-class representation of layout, which can be selectively ignored or utilised, depending on the needs of a specific transformation [136, Chapter 8].

Handling reflection One particularly troublesome feature of Java from the viewpoint of refactoring is reflection. The Java reflection API allows a program to dynamically load classes, create instances, and access their fields and methods without any static checks. In particular, the names of classes to load or members to access are simply given as string objects that can be constructed in an arbitrary way.

Obviously, this makes it very hard to implement a correct RENAME refactoring: for instance, when renaming a class `A` to `B`, not only do we have to adjust all static references to class `A` and all references that might become captured by the renaming, we also have to fix up code that dynamically constructs instances of class `A`. Reflection also provides run-time access to the names of program entities, and hence makes it possible to write code that can observe a renaming even if it does not influence the name binding structure.

In order to guarantee behaviour preservation, an implementation of RENAME would strictly speaking have to reject any program that makes use of reflection. Given the popularity of reflection-based libraries and frameworks such as Spring [122] and Hibernate [50] in Java, this is not a practical choice.

Our current implementation does not attempt to preserve behaviour of programs that use reflection, which is in keeping with common practice: none of the major refactoring engines for Java tries to systematically refactor programs that use reflection.[1] There has been some work on the static analysis of reflection-based Java code, which might be applicable in this context: Christensen *et al.* [25] present a string analysis and propose to use this analysis for determining the actual classes loaded by invocations of the reflective method `Class.forName`. Livshits *et al.* [84] propose a more precise reflection analysis which, however, requires a fairly heavy points-to analysis, potentially making it unsuitable for use in a refactoring engine.

A pragmatic approach that is gaining traction among popular IDEs is to provide special support for the most popular reflection-based APIs. These libraries often make use of XML-based configuration files that specify the names of methods or classes to be accessed by reflection. Compared with a direct use of the reflection API, this approach is much more disciplined and amenable to refactoring; it is indeed a special case of the next feature we consider.

Cross-language refactoring It is becoming increasingly common for different parts of large applications to be written in different languages. Particularly performance-critical bits of code may be written in a low level language and compiled to native code to be invoked through `native` methods in Java. Frameworks such as Spring and Hibernate rely heavily on XML configuration files, and scripting languages like Groovy [124] compile to Java bytecode for seamless integration with Java code.

These scenarios can be quite challenging for refactoring tools, since the non-Java parts of the program may reference elements of the Java program in some way, and these references may need adjusting if the Java elements are renamed or moved.

Again, our current implementation provides no support for this. In principle, it is not hard to add cross-language support for, e.g., renaming: instead of just locking and unlocking accesses within the Java code, we

[1] Eclipse has an option for performing name replacement within string literals when performing a renaming, but this is a purely heuristic feature.

also need to track accesses within the XML configuration files, or other parts of the program. In the simplest case, we consider these accesses un-adjustable, so if they fail to bind to the correct declaration after renaming, the refactoring is aborted. Alternatively, we may be able to adjust these accesses, for instance by changing attribute values within the XML file. Supporting this kind of feature is mostly an engineering challenge.

Strein and others [125, 126] propose to use a common meta-model to capture in a language-independent way static semantic information that is relevant for program analysis or refactoring tools. While their papers are a bit short on details, it seems that they mostly aim to capture the same kind of dependencies that we have considered in Chapter 2 and Chapter 3. For instance, they present a description of RENAME METHOD where they use a possibly cross-language *calls* relationship to identify call sites and other methods that need to be renamed. While such an approach is appealing for its generality, it is not clear how a refactoring specified at such a level of abstraction can respect language-specific conventions and rules. Their specification of RENAME METHOD, for instance, does not appear to handle method overloading.

For the specific case of Java programs with embedded data base query strings, Tatlock and others [128] show how to map references in the query strings back to Java, using an approach they term *deep typechecking*. Based on this, they develop *deep refactorings* that refactor both the Java program and the query strings at the same time, and show how to "deepen" RENAME CLASS and RENAME FIELD.

The Groovy-Eclipse plugin offers some support for cross-language refactoring between Groovy and Java [66]. It employs an *ad hoc* algorithm for finding references to Groovy program elements in Java code, and uses the Eclipse refactoring API to hook into refactorings that may require cross-language adjustments. Overall, the adopted solution seems very much tied to the Eclipse refactoring API and it is unclear whether it could be generalised to other situations.

Refactoring uncompilable code As it stands, our refactoring engine can only process valid Java programs that pass all the checks performed by the JastAddJ frontend. There are two main reasons for this:

For one, there are certain ordering assumptions in the name and type analysis implemented in JastAddJ, where later stages of the analysis assume that some validity properties have been checked at an earlier stage, and will fail if this is not so. Since our engine makes heavy use of these features, it will fail as well.

Secondly, most of the correctness arguments rely on the program being well-formed. For instance, when confronted with a program that declares two identically named local variables in the same scope, our access construction may construct an access to the wrong one.

The first of these problems is again mostly an engineering challenge. Already now, JastAddJ makes heavy use of the Null Object pattern [45]; for instance, if a type access is not bound, resolving this access will not return `null`, but a reference to the *unknown type*; looking up members on the unknown type will, in turn, return a reference to the unknown method or unknown field, as appropriate. This makes the code more robust and obviates the need for many null checks, though, of course, semantic checks need to be able to identify these null objects and distinguish them from valid types, methods and fields. Our refactoring engine currently aborts the refactoring whenever it comes across one of these null objects, but in many cases it could proceed, just issuing a warning instead.

For instance, in the case of renaming, it is not meaningful to lock an access that resolves to the unknown type or one of the other null objects. Hence our implementation of RENAME TYPE aborts the refactoring if such an access is found to be in danger of being captured. Instead, we could choose to issue a warning and not lock it. This will, of course, invalidate our guarantee of name binding preservation, but from a usability

perspective it may be preferable for a refactoring tool to at least offer some support for refactoring erroneous code, albeit without any guarantee for behaviour preservation, instead of rejecting it outright.

7.5 Conclusion

Refactorings are most useful if they encapsulate transformations that developers tend to do by hand anyway, and if they are easily explained on one or two simple examples. The refactorings commonly implemented in modern IDEs for Java and other object-oriented programming languages are of this kind.

Describing a refactoring *precisely*, however, is a perhaps unexpectedly difficult task. The complexities of real-world programming languages make it seemingly impossible to account for all corner cases, ensuring that a refactored program is always valid and behaves the same as the original program. For this reason, most refactorings have never been specified in detail and there exists a wide gap between the informal description of a refactoring and its concrete implementation.

We have proposed concepts and techniques that make it possible to give succinct, yet precise specifications of refactorings that are concrete enough to directly lead to high-quality implementations. We handle deep problems like name binding preservation or changes in data flow by giving our specifications in terms of the static semantics of the program, so that these problems are reduced to dependency preservation. To overcome shallow problems caused by idiosyncrasies or lacking features of the object language, we formulate our refactorings in terms of an enriched language with extensions that make the transformation easier to describe. This is often the key to decomposing a complex refactoring into multiple smaller, simpler refactorings, which in turn helps with verifying the correctness of critical parts of the specification.

We feel that having good specifications of refactorings is essential for both implementation and verification. By no means should one expect the specifications given in this thesis to be perfect. An implementation based on them may have bugs just like any other refactoring implementation. But whenever we discover a bug in an implementation based on a high-level specification, as indeed happened many times during the research that led to this thesis, we can propagate the fix back to that specification. Instead of just improving one particular implementation, then, we can deepen our understanding of the refactoring itself. In particular, if the fix concerns the underlying dependency preservation framework, any other refactoring that uses the same dependencies will automatically benefit. We hope that in this way the concepts and techniques introduced in this thesis will pave the way for a more principled approach to the specification, implementation and verification of refactorings.

Appendix A

NameJava

This appendix contains the full JastAdd implementation of NameJava with its abstract grammar, variable and class lookup, the name binding framework including access construction and locked names, and an implementation of the RENAME refactoring for variables and classes as discussed in Chapter 2.

A.1 Abstract grammar of NameJava

```
Program ::= ClassDecl*;

ClassDecl ::= ⟨Name:String⟩ [Super:Access] BodyDecl*;

abstract BodyDecl;
Field : BodyDecl ::= VarDecl [Init:Expr];
Initializer : BodyDecl ::= Block;
MemberClass : BodyDecl ::= ClassDecl;

VarDecl ::= Type:Access ⟨Name:String⟩;

abstract Stmt;
Block : Stmt ::= Stmt*;
LocalVar : Stmt ::= VarDecl [Init:Expr];

abstract Expr;

abstract Access : Expr;
SimpleAccess : Access ::= ⟨Name:String⟩;
QualAccess : Access ::= Left:Expr Right:SimpleAccess;

This : Expr;
QualThis : Expr ::= Type:Access;
CastExpr : Expr ::= Type:Access Expr;
```

A.2 Utility attributes for navigating the AST

```
/* auto-generated generic traversal and mutation methods on class ASTNode:
 *   int ASTNode.getNumChild()              -- get number of child nodes
 *   ASTNode ASTNode.getChild(int i)        -- retrieve the ith child
 *   ASTNode ASTNode.replaceWith(ASTNode n) -- replace by n, return n
 *   void ASTNode.flushCaches()        -- flush cached attributes in subtree */

// previous statement in same block
inh lazy Stmt Stmt.prev();
eq Block.getStmt(int i).prev() = i > 0 ? getStmt(i-1) : null;
eq Initializer.getBlock().prev() = null;

// next body declaration in same class
inh lazy BodyDecl BodyDecl.next();
eq ClassDecl.getBodyDecl(int i).next() {
  return i < getNumBodyDecl()-1 ? getBodyDecl(i+1) : null;
}

// innermost enclosing class
inh ClassDecl VarDecl.hostClass();
inh ClassDecl Expr.hostClass();
inh ClassDecl BodyDecl.hostClass();
eq ClassDecl.getChild().hostClass() = this;

// for a qualified access, return the QualAccess whose right child it is
inh QualAccess Access.getQualifyingDot();
eq ClassDecl.getChild().getQualifyingDot() = null;
eq QualAccess.getRight().getQualifyingDot() = this;

inh lazy Program Access.programRoot();
inh lazy Program VarDecl.programRoot();
inh lazy Program ClassDecl.programRoot();
eq Program.getChild().programRoot() = this;
```

A.3 Declarations and typing

```
interface Declaration { }
VarDecl implements Declaration;
ClassDecl implements Declaration;

syn ClassDecl Expr.type();
eq Access.type() = decl().type();
eq This.type() = hostClass();
eq QualThis.type() = getType().type();
eq Cast.type() = getType().type();
```

```
syn ClassDecl Declaration.type();
eq VarDecl.type() = getType().type();
eq ClassDecl.type() = this;

/* syntactic classification of names; any name occurring in a NameJava
 * program is syntactically either a variable name, a type name, or
 * an ambiguous name */
enum NameKind {
  VAR    { NameKind qualKind() { return EITHER; } },
  TYPE   { NameKind qualKind() { return TYPE; } },
  EITHER { NameKind qualKind() { return EITHER; } };

  // the kind of a name determines the kind of its qualifier
  abstract NameKind qualKind();
}

// determine the name kind according to syntactic position
inh NameKind Access.nameKind();
eq QualAccess.getLeft().nameKind() = getRight().nameKind().qualKind();
eq ClassDecl.getSuper().nameKind() = NameKind.TYPE;
eq Field.getInit().nameKind() = NameKind.VAR;
eq LocalVar.getInit().nameKind() = NameKind.VAR;
eq QualThis.getType().nameKind() = NameKind.TYPE;
eq Cast.getType().nameKind() = NameKind.TYPE;
eq Cast.getExpr().nameKind() = NameKind.VAR;

/* This attribute determines whether an access is a variable access or
 * a type access, and returns the variable declaration or class
 * declaration it binds to, respectively. If a name could refer either
 * to a variable or a class, we give preference to the variable.
 * This uses the attributes lookupVariable and lookupClass defined below. */
syn Declaration Access.decl();
eq SimpleAccess.decl() {
  if(nameKind() == NameKind.VAR || nameKind() == NameKind.EITHER) {
    VarDecl vd = lookupVariable(getName());
    if(vd != null)
      return vd;
  }
  if(nameKind() == NameKind.TYPE || nameKind() == NameKind.EITHER) {
    return lookupClass(getName());
  }
  return null;
}
eq QualAccess.decl() = getRight().decl();
```

A.4 Looking up variables

```
// local variable lookup within a block
syn VarDecl Stmt.localVariable(String name)
  = prev() == null ? null : prev().localVariable(name);
eq LocalVar.localVariable(String name)
  = name.equals(getVarDecl().getName()) ? getVarDecl()
                                        : super.localVariable(name);

// local field lookup within a class
syn VarDecl BodyDecl.localField(String name)
  = next() == null ? null : next().localField(name);
eq Field.localField(String name)
  = name.equals(getVarDecl().getName()) ? getVarDecl()
                                        : super.localField(name);

syn VarDecl ClassDecl.localField(String name) {
  return getNumBodyDecl() == 0 ? null : getBodyDecl(0).localField(name);
}

// member field lookup within a class and its ancestors
syn VarDecl ClassDecl.memberField(String name) {
  VarDecl f = localField(name);
  if(f != null)
    return f;
  if(hasSuper())
    return getSuper().type().memberField(name);
  return null;
}

// lexically scoped lookup
eq Block.getStmt(int i).lookupVariable(String name) {
  VarDecl v = getStmt(i).localVariable(name);
  if(v != null)
    return v;
  return lookupVariable(name);
}

eq ClassDecl.getBodyDecl(int i).lookupVariable(String name) {
  VarDecl f = memberField(name);
  if(f != null)
    return f;
  return lookupVariable(name);
}

eq QualAccess.getRight().lookupVariable(String name)
  = getLeft().qualifiedLookupVar(name);
```

```
eq Program.getChild().lookupVariable(String name) = null;

// qualified lookup
syn VarDecl Expr.qualifiedLookupVar(String name) = type().memberField(name);

// expose the attribute where we need to invoke it
inh VarDecl Access.lookupVariable(String name);
inh VarDecl Block.lookupVariable(String name);
inh VarDecl ClassDecl.lookupVariable(String name);
```

A.5 Looking up classes

```
// local member class lookup within a class
syn ClassDecl BodyDecl.localClass(String name) {
  return next() == null ? null : next().localClass(name);
}
eq MemberClass.localClass(String name) {
  return name.equals(getClassDecl().getName()) ? getClassDecl()
                                               : super.localClass(name);
}

syn ClassDecl ClassDecl.localClass(String name) {
  return getNumBodyDecl() == 0 ? null : getBodyDecl(0).localClass(name);
}

// member class lookup within a class and its ancestors
syn ClassDecl ClassDecl.memberClass(String name) {
  ClassDecl c = localClass(name);
  if(c != null)
    return c;
  return hasSuper() ? getSuper().type().memberClass(name) : null;
}

// toplevel class lookup
syn lazy ClassDecl Program.toplevelClass(String name) {
  for(ClassDecl cd : getClassDecls())
    if(name.equals(cd.getName()))
      return cd;
  return null;
}

// block-scoped lookup
eq ClassDecl.getBodyDecl(int i).lookupClass(String name) {
  ClassDecl c = memberClass(name);
  if(c != null)
    return c;
```

```
  return lookupClass(name);
}

eq Program.getClassDecl().lookupClass(String name) = toplevelClass(name);

eq QualAccess.getRight().lookupClass(String name)
  = getLeft().qualifiedLookupClass(name);

// qualified class lookup; note that classes can only be looked up as
// members of other classes, not on any other expression
syn ClassDecl Expr.qualifiedLookupClass(String name) = null;
eq Access.qualifiedLookupClass(String name)
  = decl().qualifiedLookupClass(name);

syn ClassDecl Declaration.qualifiedLookupClass(String name);
eq VarDecl.qualifiedLookupClass(String name) = null;
eq ClassDecl.qualifiedLookupClass(String name) = memberClass(name);

// expose the attribute where we need to invoke it
inh ClassDecl Access.lookupClass(String name);
inh ClassDecl ClassDecl.lookupClass(String name);
```

A.6 Symbolic accesses

```
interface Scope { }
ClassDecl implements Scope;
Block implements Scope;

syn boolean Scope.declaresVariable(String name);
eq ClassDecl.declaresVariable(String name)
  = memberField(name) != null;
eq Block.declaresVariable(String name)
  = getNumStmt() > 0 && getStmt(getNumStmt()-1).localVariable(name) != null;

interface SymbolicVarAccess {
  SymbolicVarAccess moveInto(Scope s);
  SymbolicVarAccess moveDownTo(ClassDecl c);
  Access toAccess(ClassDecl host);
  Access toAccess(Expr qual);
}

// symbolic access to a local variable
LocalVar implements SymbolicVarAccess;
public SymbolicVarAccess LocalVar.moveInto(Scope s) {
  return s.declaresVariable(getVarDecl().getName()) ? null : this;
}
public SymbolicVarAccess LocalVar.moveDownTo(ClassDecl c) {
```

```
  return null;
}

// symbolic access to a field
class SymbolicFieldAccess implements SymbolicVarAccess {
  private VarDecl target;
  // class declaring field target
  private ClassDecl source;
  // enclosing class whose ancestor is source
  private ClassDecl bend;
  // whether the field is shadowed or hidden
  private boolean needsQualifier;

  public SymbolicFieldAccess(VarDecl target) {
    this.target = target;
    this.source = target.hostClass();
    this.bend = this.source;
    this.needsQualifier = false;
  }

  public SymbolicVarAccess moveInto(Scope s) {
    if(s.declaresVariable(target.getName()))
      needsQualifier = true;
    return this;
  }

  public SymbolicVarAccess moveDownTo(ClassDecl c) {
    if(c.localField(target.getName()) != null)
      needsQualifier = true;
    bend = c;
    return this;
  }
}

public Access Access.splice(SymbolicVarAccess sacc) {
  QualAccess qacc = getQualifyingDot();
  Access acc;
  if(qacc == null) {
    acc = sacc.toAccess(hostClass());
    if(acc == null)
      return null;
    replaceWith(acc);
  } else {
    acc = sacc.toAccess(qacc.getLeft());
    if(acc == null)
      return null;
    qacc.replaceWith(acc);
```

```
  }
  return acc;
}

// convert symbolic access to actual access
public Access LocalVar.toAccess(ClassDecl host) {
  return new SimpleAccess(getVarDecl().getName());
}
public Access SymbolicFieldAccess.toAccess(ClassDecl host) {
  SimpleAccess acc = new SimpleAccess(target.getName());
  if(needsQualifier) {
    if(source == bend && source == host)
      return new QualAccess(new This(), acc);
    else if(bend == host)
      return new QualAccess(new Cast(new LockedClassAccess(source),
                            new This()), acc);
    else
      return new QualAccess(new Cast(new LockedClassAccess(source),
                            new QualThis(new LockedClassAccess(bend))), acc);
  } else {
    return acc;
  }
}

public Access LocalVar.toAccess(Expr qual) { return null; }
public Access SymbolicFieldAccess.toAccess(Expr qual) {
  SimpleAccess acc = new SimpleAccess(target.getName());
  if(needsQualifier) {
    if(bend != qual.type())
      return null;
    return new QualAccess(new Cast(new LockedClassAccess(source), qual), acc);
  } else {
    return new QualAccess(qual, acc);
  }
}

syn boolean Scope.declaresClass(String name);
eq ClassDecl.declaresClass(String name) = memberClass(name) != null;
eq Block.declaresClass(String name) = false;

// symbolic access to a class
interface SymbolicClassAccess {
  SymbolicClassAccess moveInto(Scope s);
  SymbolicClassAccess moveDownTo(ClassDecl c);
  Access toAccess(Access pos, ClassDecl host);
  Access toAccess(Access pos, Expr qual);
}
```

```
public Access Access.splice(SymbolicClassAccess sacc) {
  QualAccess qacc = getQualifyingDot();
  Access acc;
  if(qacc == null) {
    acc = sacc.toAccess(this, hostClass());
    if(acc == null)
      return null;
    replaceWith(acc);
  } else {
    acc = sacc.toAccess(this, qacc.getLeft());
    if(acc == null)
      return null;
    qacc.replaceWith(acc);
  }
  return acc;
}

// symbolic access to a toplevel class
ClassDecl implements SymbolicClassAccess;
public SymbolicClassAccess ClassDecl.moveInto(Scope s) {
  if(s.declaresClass(getName()))
    return null;
  return this;
}
public SymbolicClassAccess ClassDecl.moveDownTo(ClassDecl c) {
  if(c.localClass(getName()) != null)
    return null;
  return this;
}
// convert symbolic access to actual access, respecting name kinds
public Access ClassDecl.toAccess(Access pos, ClassDecl host) {
  if(pos.nameKind() == NameKind.TYPE ||
    pos.nameKind() == NameKind.EITHER && pos.lookupVariable(getName()) == null)
      return new SimpleAccess(getName());
  return null;
}
public Access ClassDecl.toAccess(Access pos, Expr qual) {
  return null;
}

// symbolic access to a nested class
class SymbolicMemberClassAccess implements SymbolicClassAccess {
  private ClassDecl target;
  private boolean needsQualifier;
  private ClassDecl host;
```

```
  public SymbolicMemberClassAccess(MemberClass mc) {
    this.target = mc.getClassDecl();
    this.needsQualifier = false;
    this.host = mc.hostClass();
  }

  public SymbolicClassAccess moveInto(Scope s) {
    if(s.declaresClass(target.getName()))
      needsQualifier = true;
    return null;
  }
  public SymbolicClassAccess moveDownTo(ClassDecl c) {
    if(c.localClass(target.getName()) != null)
      needsQualifier = true;
    return null;
  }
  public Access toAccess(Access pos, ClassDecl host) {
    SimpleAccess acc = new SimpleAccess(host.getName());
    switch(pos.nameKind()) {
    case VAR:
      return null;
    case TYPE:
      if(needsQualifier)
        return new QualAccess(new LockedClassAccess(host), acc);
      return acc;
    case EITHER:
      if(needsQualifier || pos.lookupVariable(target.getName()) != null)
        return new QualAccess(new LockedClassAccess(host), acc);
      return acc;
    }
    return null;
  }
  public Access toAccess(Access pos, Expr qual) {
      return null;
  }
}
```

A.7 Accessing variables

```
syn SymbolicVarAccess Stmt.accessLocal(VarDecl v) {
  return prev() == null ? null : prev().accessLocal(v);
}
eq LocalVar.accessLocal(VarDecl v) {
  return getVarDecl() == v ? this : super.accessLocal(v);
}

syn SymbolicVarAccess BodyDecl.accessLocalField(VarDecl v) {
```

```
  return next() == null ? null : next().accessLocalField(v);
}
eq Field.accessLocalField(VarDecl v) {
  return getVarDecl() == v ? new SymbolicFieldAccess(getVarDecl())
                           : super.accessLocalField(v);
}

syn SymbolicVarAccess ClassDecl.accessLocalField(VarDecl v) {
  return getNumBodyDecl() == 0 ? null : getBodyDecl(0).accessLocalField(v);
}

syn SymbolicVarAccess ClassDecl.accessMemberField(VarDecl v) {
  SymbolicVarAccess acc = accessLocalField(v);
  if(acc != null) return acc;
  if(hasSuper()) {
    acc = getSuper().type().accessMemberField(v);
    return acc == null ? null : acc.moveDownTo(this);
  }
  return null;
}

eq Block.getStmt(int i).accessVariable(VarDecl v) {
  SymbolicVarAccess acc = getStmt(i).accessLocal(v);
  if(acc != null) return acc;
  acc = accessVariable(v);
  if(acc != null) return acc.moveInto(this);
  return null;
}

eq ClassDecl.getBodyDecl(int i).accessVariable(VarDecl v) {
  SymbolicVarAccess acc = accessMemberField(v);
  if(acc != null) return acc;
  acc = accessVariable(v);
  return acc == null ? null : acc.moveInto(this);
}

eq QualAccess.getRight().accessVariable(VarDecl v)
  = getLeft().type().qualifiedAccessVar(v);

syn SymbolicVarAccess Declaration.qualifiedAccessVar(VarDecl v);
eq VarDecl.qualifiedAccessVar(VarDecl v) = getType().type().accessMemberField(v);
eq ClassDecl.qualifiedAccessVar(VarDecl v) = accessMemberField(v);

eq Program.getChild().accessVariable(VarDecl v) = null;

inh SymbolicVarAccess Access.accessVariable(VarDecl v);
inh SymbolicVarAccess Block.accessVariable(VarDecl v);
```

```
inh SymbolicVarAccess ClassDecl.accessVariable(VarDecl v);
```

A.8 Accessing classes

```
syn SymbolicClassAccess BodyDecl.accessLocalClass(ClassDecl cd) {
  return next() == null ? null : next().accessLocalClass(cd);
}

eq MemberClass.accessLocalClass(ClassDecl cd) {
  return cd == getClassDecl() ? new SymbolicMemberClassAccess(this)
                              : super.accessLocalClass(cd);
}

syn SymbolicClassAccess ClassDecl.accessLocalClass(ClassDecl cd) {
  return getNumBodyDecl() == 0 ? null : getBodyDecl(0).accessLocalClass(cd);
}

syn SymbolicClassAccess ClassDecl.accessMemberClass(ClassDecl cd) {
  SymbolicClassAccess acc = accessLocalClass(cd);
  if(acc != null)
    return acc;
  if(hasSuper()) {
    acc = getSuper().type().accessMemberClass(cd);
    if(acc != null)
      return acc.moveDownTo(this);
  }
  return null;
}

eq ClassDecl.getBodyDecl(int i).accessClass(ClassDecl cd) {
  SymbolicClassAccess acc = accessMemberClass(cd);
  if(acc != null)
    return acc;
  acc = accessClass(cd);
  if(acc != null)
    return acc.moveInto(this);
  return null;
}

eq Program.getClassDecl().accessClass(ClassDecl cd) {
  for(ClassDecl cd2 : getClassDecls())
    if(cd == cd2)
      return cd;
  return null;
}

eq QualAccess.getRight().accessClass(ClassDecl cd)
```

```
  = getLeft().type().qualifiedAccessClass(cd);

syn SymbolicClassAccess Declaration.qualifiedAccessClass(ClassDecl cd);
eq VarDecl.qualifiedAccessClass(ClassDecl cd) = null;
eq ClassDecl.qualifiedAccessClass(ClassDecl cd) = accessMemberClass(cd);

inh SymbolicClassAccess Access.accessClass(ClassDecl cd);
inh SymbolicClassAccess ClassDecl.accessClass(ClassDecl cd);
```

A.9 Locked accesses

```
ast LockedClassAccess : SimpleAccess ::= <Target:ClassDecl>;
ast LockedVarAccess : SimpleAccess ::= <Target:VarDecl>;

// convenience constructors for locked accesses
public LockedClassAccess.LockedClassAccess(ClassDecl target) {
  this(target.getName(), target);
}
public LockedVarAccess.LockedVarAccess(VarDecl target) {
  this(target.getName(), target);
}

// circumvent normal name lookup
eq LockedClassAccess.decl() = getTarget();
eq LockedVarAccess.decl() = getTarget();

// replace a normal access by a locked one
public void SimpleAccess.lock() {
  Declaration decl = decl();
  if(decl instanceof ClassDecl)
    replaceWith(new LockedClassAccess((ClassDecl)decl));
  else
    replaceWith(new LockedVarAccess((VarDecl)decl));
}

// lock all accesses to one of the given names in a subtree
public void ASTNode.lockNames(String[] names) {
  for(int i=0;i<getNumChild();++i)
    getChild(i).lockNames(names);
}

public void SimpleAccess.lockNames(String[] names) {
  for(int i=0;i<names.length;++i)
    if(getName().equals(names[i]))
      lock();
}
```

```
// unlock all locked names in a subtree; may fail with an exception
public void ASTNode.unlockNames() throws RefactoringException {
  for(int i=0;i<getNumChild();++i)
    getChild(i).unlockNames();
}

public void LockedClassAccess.unlockNames() throws RefactoringException {
  SymbolicClassAccess sacc = accessClass(getTarget());
  Access acc;
  if(sacc == null || (acc = splice(sacc)) == null)
    throw new RefactoringException("cannot␣unlock");
  acc.unlockNames();
}

public void LockedVarAccess.unlockNames() throws RefactoringException {
  SymbolicVarAccess sacc = accessVariable(getTarget());
  Access acc;
  if(sacc == null || (acc = splice(sacc)) == null)
    throw new RefactoringException("cannot␣unlock");
  acc.unlockNames();
}
```

A.10 Renaming

```
class RefactoringException extends Exception {
  public RefactoringException(String msg) {
    super(msg);
  }
}

// rename a variable
public void VarDecl.rename(String newname) throws RefactoringException {
  programRoot().lockNames(new String[]{getVarDecl().getName(), newname});
  setName(newname);
  programRoot().flushCaches();
  programRoot().unlockNames();
}

// rename a class
public void ClassDecl.rename(String newname) throws RefactoringException {
  programRoot().lockNames(new String[]{getVarDecl().getName(), newname});
  setName(newname);
  programRoot().flushCaches();
  programRoot().unlockNames();
}
```

Appendix B

Reading the Specifications

B.1 Pseudocode conventions

We give our specifications in generic imperative pseudocode. Parameters and return values are informally typed using one of the types from Table B.1. Additionally, we use an ML-like `option` type with constructors `None` and `Some` for refactorings 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. 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'.

We now list all the language restrictions and extensions used in the specifications and implementations comprising our refactoring engine. For each language extension and restriction we briefly explain its purpose, and discuss how to enforce or eliminate it, respectively.

Type	**Description**
AnonymousMethod	node representing an anonymous method
Assignment	node representing an assignment
Block	node representing a block
Field	node representing a field declaration
ident	string satisfying the lexical rules for an identifier [48, §3.8]
LocalVar	node representing a local variable declaration or parameter declaration
Method	node representing a method declaration
nat	natural number
Stmt	node representing a statement
Temp	node representing a local variable declaration
TypeDecl	node representing a type declaration

Table B.1: Node types

B.2 Language restrictions

Java provides a fair amount of "syntactic sugar" to abbreviate common idioms, which add some irregularity to the language. These abbreviations can be troublesome to handle for refactorings, so it is often easier to desugar them right at the beginning of the refactoring, replacing them with their more elaborate long form. After the refactoring is finished, the abbreviations should be reintroduced where possible, so our implementations marks syntax tree nodes that result from the expansion of syntactic sugar to enable this resugaring step.

For purposes of specification it is convenient to assume that some language restriction has been enforced on the whole program, but in practice nothing so drastic has to be done: it is usually enough to eliminate a certain construct from one given method or statement.

Array initialisers An array initialiser is a special expression that creates an array literal; it may only appear as the initialiser of a variable, or as part of an array creation expression. As shown in Figure 5.2, they can be desugared into array creation expressions, which are not restricted in their usability.

Compound assignment operators Like other C-based languages, Java provides a number of compound assignment operators such as +=, as well as unary increment and decrement operators. These operators offer convenient short-hands for frequently occurring code patterns, but for some refactorings they are not very convenient, since their operands are both reads and writes. It may then be easier to require compound assignments to be rewritten into normal assignments; this rewriting is not always easily done, and may cause the refactoring to fail.

Compound declarations A variable declaration statement in Java may declare more than one local variable or field. Sometimes it is more convenient to assume that every variable declaration is a statement by itself, which can be achieved by splitting up a compound declaration into a series of individual declarations as shown in Figure 5.2.

Implicit assignment conversion If the right hand side of an assignment does not have the same type as its left hand side, the result is implicitly cast to that type by a process known as *assignment conversion*. This is implicit in the syntax, but can lead to unpleasant surprises when implementing refactorings such as INLINE ASSIGNMENT; see Figure 5.3. It is usually easy enough to make this conversion explicit by inserting a type cast.

Implicit method modifiers Methods declared in an interface are implicitly `public` and `abstract`. For refactorings that move methods from interfaces to classes, it is more convenient if they can assume that these modifiers are always explicitly written out.

Implicit `return` Methods with return type `void` have an implicit return statement at the end of their body if it can complete normally. It is sometimes more convenient to make this explicit.

`synchronized` modifier The `synchronized` modifier indicates that a method implicitly acquires a lock upon its invocation: the monitor lock of its receiver object for instance methods, and that of its declaring class for static methods. This modifier can always be replaced by an explicit `synchronized` block around the method body, as shown in Figure 4.1, which makes the dependency on the lock more explicit.

Variable arity parameters The last parameter of a method or constructor may be declared to be *variable arity*, meaning that any number of actual arguments may be passed in the position corresponding to this parameter, with the variable arity parameter being assigned an array containing all these arguments. For refactorings that manipulate parameter lists of methods, it is usually more convenient to assume that there are no variable arity parameters, and corresponding arguments are passed by explicitly wrapping them into an array creation expression.

B.3 Language extensions

Language extensions often make the implementation of refactorings easier by providing convenient points at which to break up the refactoring into smaller parts. They also encapsulate commonly needed features that make the implementation of refactorings easier. All language extensions in our implementation are lightweight in the sense that they never occur in the output program, but are always eliminated or translated away into pure Java.

Anonymous methods Anonymous methods are nameless methods that are declared and immediately invoked on some arguments; syntactically, an anonymous method is an expression. Like named methods, anonymous methods may have parameters, return a value, and declare thrown exceptions. They can additionally access any variables declared in the surrounding scope. In terms of control flow, however, they behave like normal methods in that a `return` statement in an anonymous method returns from the anonymous method, and a `break` or `continue` statement cannot refer to a loop of the surrounding method.

Anonymous methods may declare `ref` and `out` parameters. The corresponding argument must then be a local variable access, which is considered to be passed by reference, so that any value the anonymous method assigns to it is visible to the surrounding method after the anonymous method has returned. While `ref` parameters are considered to be defined upon entry to the anonymous method, `out` parameters are not and have to be assigned a value before they can be used.

Anonymous methods are not easy to eliminate in general; our implementation makes no attempt to do so, but rather relies on the refactorings that introduce them to eliminate them later.

Fresh variable A fresh variable is a variable whose name does not clash with that of any other variable. It is invisible to normal name lookup, and can only be bound to using locked names (see below). To eliminate a fresh variable, an implementation can choose a fresh name that is not in use yet and change the declaration itself as well as any access to it to use this name.

Locked reaching definitions For every variable read, we can compute its set of reaching definitions, i.e., the set of all variable writes that may write the memory location that it reads, and that precede it in the control flow graph without any intervening writes of that location. For refactorings that move variable reads around, it may be useful to ensure that their reaching definitions do not change. To achieve this, a variable read's reaching definitions can be locked, i.e., cached and stored in the AST node, before the move; afterwards they can be unlocked, which simply means that the reaching definitions are recomputed, and the refactoring is aborted if they are not the same as before.

Locked names A locked name binds to a given declaration directly, bypassing normal lookup rules. To unlock a locked name, it may need to be qualified to escape name capture. For method names, it may

additionally be necessary to add type casts around arguments to ensure that the desired target method is indeed the most specific method under overloading resolution rules. It may not always be possible to construct a properly qualified name, in which case the refactoring is aborted.

Locked overriding Locking a method's overriding dependencies simply means that the set of methods it overrides is computed and stored in the node. To unlock, the set of overridden methods is recomputed and compared to the stored value; any change aborts the refactoring.

Locked successors When moving statements or expressions, it can be useful to ensure that their control flow successors do not change. This is achieved by caching control flow successors before the transformation, recomputing afterwards, and aborting the refactoring if they have changed.

Locked synchronisation dependencies As detailed in Subsection 3.2.2, there are three kinds of synchronisation dependencies:

- A CFG node *b* has an *acquire dependency* on a node *a* if *a* corresponds to an acquire action and there is a path from *a* to *b* in the CFG.
- A CFG node *a* has a *release dependency* on a node *b* if *b* corresponds to a release action and there is a path from *a* to *b* in the CFG.
- A CFG node *a* has a *thread management dependency* on any node *b* that corresponds to a thread management action and is reachable from it in the CFG

When moving statements and expressions, these dependencies should usually be preserved in the following sense: a normal access may never lose acquire dependencies; a normal access may never lose release dependencies; a node corresponding to a synchronisation action may never gain acquire or release dependencies; a node corresponding to any action may never gain or lose a thread management dependency.

The process of computing synchronisation dependencies and checking their preservation can again be abstracted into the locking-and-unlocking pattern.

On-demand `final` A local variable or parameter that is on-demand `final` can be accessed from a local class or anonymous class just like a `final` variable, but can be assigned to. When eliminating such a variable, it is made `final` only if it really is accessed in such a way; at that point, it will be ensured that the variable is only assigned once if it is a local variable, or not at all if it is a parameter. If the variable cannot be made `final`, yet is used in a local or anonymous class, the refactoring is aborted.

`return void` Sometimes a refactoring may construct a return statement `return e;` where expression `e` has type `void`. This is eliminated by rewriting it to `e; return;`, where it may additionally be possible to elide the return statement if it occurs at the end of a method.

`with` A `with` block allows to set the receiver object of the statement it qualifies. The argument to `with` then becomes the value of `this` within that block, and unqualified field accesses are resolved as member fields of that argument. Local variables from outside the `with` block are still visible within it, but instance fields are not. If the argument of the `with` block is a name, it is fairly easy to eliminate by making implicit qualifications explicit. For its use in implementing the MOVE METHOD refactoring, see Figure 5.4.

Bibliography

[1] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative Self-Adjusting Computation. In *Principles of Programming Languages (POPL)*, pages 309–322. ACM Press, 2008.

[2] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. An Experimental Analysis of Self-Adjusting Computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 32(1):96–107, 2009.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive Functional Programming. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):990–1034, 2006.

[4] *The AspectJ Programming Guide*, 2003. http://www.eclipse.org/aspectj.

[5] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. Modularity First: A Case for Mixing AOP and Attribute Grammars. In *Aspect-Oriented Software Development (AOSD)*, pages 25–35. ACM Press, 2008.

[6] Éric Badouel, Bernand Fotsing, and Rodrigue Tchougong. Yet Another Implementation of Attribute Evaluation. Technical Report 6315, INRIA Rennes, 2007.

[7] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring Support for Class Library Migration. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 265–279. ACM Press, 2005.

[8] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking Rewriting on Java. In *Rewriting Techniques and Applications (RTA)*, pages 36–47. Springer-Verlag, 2007.

[9] Fabian Bannwart. Changing Software Correctly. Master's thesis, Eidgenössische Technische Hochschule (ETH) Zürich, 2006.

[10] Fabian Bannwart and Peter Müller. Changing Programs Correctly: Refactoring with Specifications. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *Formal Methods (FM)*, pages 492–507. Springer-Verlag, 2006.

[11] Bernhard Bauer. Proving Properties over Attribute Grammars. Technical Report 9402, Justus-Liebig-Universität Giessen, AG Informatik, 1994.

[12] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone Detection Using Abstract Syntax Trees. In *International Conference on Software Maintenance (ICSM)*, pages 368–377. IEEE Computer Society, 1998.

[13] Gavin M. Bierman, Matthew J. Parkinson, and Andrew M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge, 2003.

[14] Richard Bird and Oege de Moor. *The Algebra of Programming*. Prentice Hall, 1996.

[15] Ana Bove. *General Recursion in Type Theory*. Ph.D. thesis, Chalmers University, 2002.

[16] John Boyland. Incremental Evaluators for Remote Attribute Grammars. *Electronic Notes in Theoretical Computer Science*, 65(3), 2002. Proceedings of Second Workshop on Language Descriptions, Tools and Applications (LDTA) 2002.

[17] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[18] Christopher Mark Brown. *Tool Support for Refactoring Haskell Programs*. Ph.D. thesis, University of Kent, 2008.

[19] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Java Grande Conference*, pages 129–141, 1999.

[20] Rod M. Burstall and John Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[21] Venanzio Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, 1(2), 2005.

[22] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *European Symposium on Programming (ESOP)*, pages 331–346. Springer-Verlag, 2007.

[23] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 519–538. ACM Press, 2005.

[24] Jingde Cheng. Slicing Concurrent Programs: A Graph-Theoretical Approach. In *Automated and Algorithmic Debugging (AADEBUG)*, pages 223–240. Springer-Verlag, 1993.

[25] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise Analysis of String Expressions. In *International Static Analysis Symposium (SAS)*, pages 1–18. Springer-Verlag, 2003.

[26] Alonzo Church. *Introduction to Mathematical Logic*. Princeton University Press, 1944.

[27] Mel Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. Ph.D. thesis, University of Dublin, Trinity College, 2000.

[28] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 System. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA)*, pages 76–87. Springer-Verlag, 2003.

[29] John Cocke. Global Common Subexpression Elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24. ACM Press, 1970.

[30] ADT Coq. The Coq Proof Assistant. http://logical.saclay.inria.fr/coq/, 2010.

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

[32] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated Testing of Refactoring Engines. In *Joint Meeting of the European Software Engineering Conference (ESEC) and the Symposium on the Foundations of Software Engineering (FSE)*, pages 185–194. ACM Press, 2007.

[33] Alan Demers, Tom Reps, and Tim Teitelbaum. Incremental Evaluation for Attribute Grammars with Applications to Syntax-directed Editors. In *Principles of Programming Languages (POPL)*, pages 105–116. ACM Press, 1981.

[34] Danny Dig, John Marrero, and Michael D. Ernst. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *International Conference on Software Engineering (ICSE)*, pages 397–407. ACM Press, 2009.

[35] Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph Johnson. RELOOPER: Refactoring for Loop Parallelism in Java. In *Companion to the Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 793–794. ACM Press, 2009.

[36] Sophia Drossopoulou and Susan Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In *Formal Syntax and Semantics of Java*, pages 41–82. Springer-Verlag, 1999.

[37] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 144–169. Springer-Verlag, 2004.

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

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

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

[41] Ran Ettinger. *Refactoring via Program Slicing and Sliding*. Ph.D. thesis, Oxford University Computing Laboratory, 2007.

[42] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification (CAV)*, pages 173–177. Springer-Verlag, 2007.

[43] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A Programmer's Reduction Semantics for Classes and Mixins. In *Formal Syntax and Semantics of Java*, pages 241–269. Springer-Verlag, 1999.

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

[45] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2000.

[46] Martin Fowler. Crossing Refactoring's Rubicon. http://martinfowler.com/articles/refactoringRubicon.html, 2001.

[47] Alejandra Garrido and José Meseguer. Formal Specification and Verification of Java Refactorings. In *Source Code Analysis and Manipulation (SCAM)*, pages 165–174. IEEE Computer Society, 2006.

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

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

[50] Red Hat. Hibernate. http://www.hibernate.org, 2010.

[51] David Hilbert and Wilhelm Ackermann. *Grundzüge der Theoretischen Logik*. Julius Springer, 1928.

[52] Susan Horwitz, Jan Prins, and Tom Reps. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Principles of Programming Languages (POPL)*, pages 146–157. ACM Press, 1988.

[53] Gérard P. Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[54] Thuan Quang Huynh and Abhik Roychoudhury. A Memory Model Sensitive Checker for C#. In *International Symposium on Formal Methods (FM)*, pages 476–491. Springer-Verlag, 2006.

[55] Atsushi Igarashi and Benjamin C. Pierce. On Inner Classes. *Information and Computation*, 177(1):56–89, 2002.

[56] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 132–146. ACM Press, 1999.

[57] ECMA International. Standard ECMA-334, C# Language Specification. http://www.ecma-international.org/publications/standards/Ecma-334.htm, 2006.

[58] Jaroslav Ševčík and David Aspinall. On Validity of Program Transformations in the Java Memory Model. In Jan Vitek, editor, *European Conference on Object-Oriented Programming (ECOOP)*, pages 27–51. Springer-Verlag, 2008.

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

[60] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Functional Programming Languages and Computer Architecture*, pages 190–230. Springer-Verlag, 1985.

[61] Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Functional Programming Languages and Computer Architecture*, pages 154–173. Springer-Verlag, 1987.

[62] Nicolas Juillerat and Béat Hirsbrunner. Improving Method Extraction: A Novel Approach to Data Flow Analysis Using Boolean Flags and Expressions. In *Workshop on Refactoring Tools (WRT)*. ACM Press, 2007.

[63] Antonio Carvalho Júnior, Leila Silva, and Márcio Lopes Cornélio. Using CafeOBJ to Mechanise Refactoring Proofs and Application. *Electronic Notes in Theoretical Computer Science*, 184:39–61, 2007. Proceedings of the Second Brazilian Symposium on Formal Methods (SBMF) 2005.

[64] Takuya Katayama and Yutaka Hoshino. Verification of Attribute Grammar. In *Principles of Programming Languages (POPL)*, pages 177–186. ACM Press, 1981.

[65] 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.

[66] Martin Kempf, Reto Kleeb, and Michael Klenk. Refactoring Support for the Groovy-Eclipse Plugin. Bachelor thesis, HSR – Hochschule für Technik Rapperswil, 2008.

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

[68] Adam Kieżun, Michael D. Ernst, Frank Tip, and Robert M. Fuhrer. Refactoring for Parameterizing Java Classes. In *International Conference on Software Engineering (ICSE)*, pages 437–446. ACM Press, 2007.

[69] Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(4):619–695, 2006.

[70] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Source Code Analysis and Manipulation (SCAM)*, pages 168–177. IEEE Computer Society, 2009.

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

[72] Raghavan Komondoor and Susan Horwitz. Semantics-preserving Procedure Extraction. In *Principles of Programming Languages (POPL)*, pages 155–169. ACM Press, 2000.

[73] Jens Krinke. Static Slicing of Threaded Programs. *SIGPLAN Notices*, 33(7):35–42, 1998. Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE) 1998.

[74] David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure, and Michael Wolfe. Dependence Graphs and Compiler Optimizations. In *Principles of Programming Languages (POPL)*, pages 207–218. ACM Press, 1981.

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

[76] Ralf Lämmel and Joost Visser. A Strafunski Application Letter. In Verónica Dahl and Philip Wadler, editors, *Practical Aspects of Declarative Languages (PADL)*, pages 357–375. Springer-Verlag, 2003.

[77] Doug Lea. The JSR-133 Cookbook for Compiler Writers, 2008. http://gee.cs.oswego.edu/dl/jmm/cookbook.html.

[78] Sorin Lerner, Todd D. Millstein, and Craig Chambers. Automatically Proving the Correctness of Compiler Optimizations. In *Programming Language Design and Implementation (PLDI)*, pages 220–231. ACM Press, 2003.

[79] Sorin Lerner, Todd D. Millstein, Erika Rice, and Craig Chambers. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *Principles of Programming Languages (POPL)*, pages 364–377. ACM Press, 2005.

[80] Huiqing Li. *Refactoring Haskell Programs*. Ph.D. thesis, University of Kent, 2006.

[81] Huiqing Li. Wrangler. http://www.cs.kent.ac.uk/projects/forse, 2009.

[82] Huiqing Li, Claus Reinke, and Simon Thompson. HaRe – The Haskell Refactorer. http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html, 2009.

[83] Huiqing Li and Simon Thompson. Formalisation of Haskell Refactorings. In Marko van Eekelen and Kevin Hammond, editors, *Trends in Functional Programming (TFP)*, pages 95–110. Intellect Books, 2005.

[84] V. Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection Analysis for Java. In *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 139–160. Springer-Verlag, 2005.

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

[86] Eva Magnusson and Görel Hedin. Circular Reference Attributed Grammars—their Evaluation and Applications. *Science of Computer Programming*, 68(1):21–37, 2007.

[87] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Principles of Programming Languages (POPL)*, pages 378–391. ACM Press, 2005.

[88] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

[89] Shane Markstrum, Robert M. Fuhrer, and Todd D. Millstein. Towards Concurrency Refactoring for X10. In *Principles and Practice of Parallel Programming (PPOPP)*, pages 303–304. ACM Press, 2009.

[90] Adam Megacz. A Coinductive Monad for Prop-Bounded Recursion. In *Programming Languages meets Program Verification (PLPV)*, pages 11–20. ACM Press, 2007.

[91] Tom Mens, Serge DeMeyer, and Dirk Janssens. Formalising Behaviour Preserving Program Transformations. In *Graph Transformation*, pages 286–301. Springer-Verlag, 2002.

[92] Sun Microsystems. NetBeans 6.8. http://www.netbeans.com, 2009.

[93] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[94] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[95] Emerson R. Murphy-Hill and Andrew P. Black. Breaking the Barriers to Successful Refactoring: Observations and Tools for Extract Method. In *International Conference on Software Engineering (ICSE)*, pages 421–430. ACM Press, 2008.

[96] Mayur Naik and Alex Aiken. Conditional Must Not Aliasing for Static Race Detection. In *Principles of Programming Languages (POPL)*, pages 327–338. ACM Press, 2007.

[97] George C. Necula. Translation Validation for an Optimizing Compiler. In *Programming Language Design and Implementation (PLDI)*, pages 83–94. ACM Press, 2000.

[98] Emma Nilsson-Nyman, Torbjörn Ekman, Görel Hedin, and Eva Magnusson. Declarative Intraprocedural Flow Analysis of Java Source Code. In *Language Descriptions, Tools and Applications (LDTA)*, 2008.

[99] David von Oheimb and Tobias Nipkow. Machine-checking the Java Specification: Proving Type-Safety. In *Formal Syntax and Semantics of Java*, pages 119–156. Springer-Verlag, 1999.

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

[101] Larry Paulson and Tobias Nipkow. Isabelle. `http://isabelle.in.tum.de`, 2010.

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

[103] Maarten Pennings, S. Doaitse Swierstra, and Harald Vogt. Using Cached Functions and Constructors for Incremental Attribute Evaluation. In *Programming Language Implementation and Logic Programming (PLILP)*, pages 130–144. Springer-Verlag, 1992.

[104] Roly Perera. Refactoring: To the Rubicon... and Beyond! In *Companion to the Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 2–3. ACM Press, 2004.

[105] Simon L. Peyton-Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *European Symposium on Programming (ESOP)*, pages 18–44. Springer-Verlag, 1996.

[106] Phil Pfeiffer and Rebecca Parsons Selke. On the Adequacy of Dependence-Based Representations for Programs with Heaps. In *Theoretical Aspects of Computer Science (TACS)*, pages 365–386. Springer-Verlag, 1991.

[107] Frank Pfenning and Conal Elliott. Higher-Order Abstract Syntax. In *Programming Language Design and Implementation (PLDI)*, pages 199–208. ACM Press, 1988.

[108] Andrew M. Pitts. Nominal Logic, a First Order Theory of Names and Binding. *Information and Computation*, 186(2):165–193, 2003.

[109] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 151–166. Springer-Verlag, 1998.

[110] Christoph Reichenbach, Devin Coughlin, and Amer Diwan. Program Metamorphosis. In Sophia Drossopoulou, editor, *European Conference on Object-Oriented Programming (ECOOP)*, pages 394–418. Springer-Verlag, 2009.

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

[112] Donald B. Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997. http://st-www.cs.uiuc.edu/users/brant/Refactory/.

[113] João Saraiva, S. Doaitse Swierstra, and Matthijs F. Kuiper. Functional Incremental Attribute Evaluation. In *Compiler Construction*, pages 279–294. Springer-Verlag, 2000.

[114] Max Schäfer and Oege de Moor. Specifying and Implementing Refactorings. In Martin Rinard, editor, *Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)*. ACM Press, 2010.

[115] Max Schäfer, Julian Dolby, Manu Sridharan, Frank Tip, and Emina Torlak. Correct Refactoring of Concurrent Java Code. In Theo D'Hondt, editor, *European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, 2010.

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

[117] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Formalising and Verifying Reference Attribute Grammars in Coq. In *European Symposium on Programming (ESOP)*, pages 143–159. Springer-Verlag, 2009.

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

[119] 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, *European Conference on Object-Oriented Programming (ECOOP)*, pages 369–393. Springer-Verlag, 2009.

[120] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective Tool Support for the Working Semanticist. In *International Conference on Functional Programming (ICFP)*, pages 1–12. ACM Press, 2007.

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

[122] SpringSource. Spring Framework. http://www.springsource.org, 2010.

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

[124] James Strachan and Guillaume Laforge. Groovy. http://groovy.codehaus.org, 2010.

[125] Dennis Strein, Hans Kratz, and Welf Löwe. Cross-Language Program Analysis and Refactoring. *Source Code Analysis and Manipulation (SCAM)*, 0:207–216, 2006.

[126] Dennis Strein, Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. An Extensible Meta-Model for Program Analysis. *IEEE Transactions on Software Engineering*, 33(9):592–607, 2007.

[127] Nik Sultana and Simon Thompson. Mechanical Verification of Refactorings. In *Partial Evaluation and Program Manipulation (PEPM)*, pages 51–60. ACM Press, January 2008.

[128] Zachary Tatlock, Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Deep Typechecking and Refactoring. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 37–52. ACM Press, 2008.

[129] Frank Tip. Refactoring Using Type Constraints. In *International Static Analysis Symposium (SAS)*, pages 1–17. Springer-Verlag, 2007.

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

[131] Jean-Baptiste Tristan and Xavier Leroy. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. In *Principles of Programming Languages (POPL)*, pages 17–27. ACM Press, 2008.

[132] Tarmo Uustalu and Varmo Vene. Comonadic Functional Attribute Evaluation. In *Trends in Functional Programming (TFP)*, pages 145–162. Intellect Books, 2005.

[133] Mark G. J. van den Brand and Jurgen J. Vinju. Rewriting with Layout. In Claude Kirchner and Nachum Dershowitz, editors, *Rule-based Programming (RULE)*. ACM Press, 2000.

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

[135] 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.

[136] Jurgen J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Ph.D. thesis, University of Amsterdam, 2005.

[137] Scott A. Vorthmann. Modelling and Specifying Name Visibility and Binding Semantics. Technical Report CMU-CS-93-158, School of Computer Science, Carnegie Mellon University, 1993.

[138] W3C. Jigsaw. http://www.w3.org/Jigsaw/, 2006.

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

[140] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for Reentrancy. In *Joint Meeting of the European Software Engineering Conference (ESEC) and the Symposium on the Foundations of Software Engineering (FSE)*, pages 173–182. ACM Press, 2009.

[141] Jianjun Zhao. Multithreaded Dependence Graphs for Concurrent Java Program. *International Symposium on Software Engineering for Parallel and Distributed Systems*, 1999.

[142] Lenore D. Zuck, Amir Pnueli, and Benjamin Goldberg. VOC: A Methodology for the Translation Validation of Optimizing Compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.