Refactoring is not (yet) about transformation *


torbjoern.ekman, max.schaefer, mathieu.verbaere}@comlab.ox.ac.uk

Abstract

In order to ensure correctness, refactorings have to check extensive preconditions before performing the transformation. These preconditions usually involve subtle analyses of the program to be refactored, and as long as there is no good support for implementing them, refactoring is not about transformation, but about analysis. In most cases, these refactoring analyses are very similar to analyses implemented in a compiler and require the same level of detail to ensure behaviour preservation. We therefore propose to implement a refactoring engine on top of a compiler to leverage existing infrastructure, and complement it with refactoring-specific functionality.

Many simple refactorings appear as building blocks in more complex refactorings. We have implemented two such building blocks that are widely useful: The first one allows to move symbolic names from one place in the program to another while preserving binding structure; it frees the developer from having to worry about issues like name clashes and accidental overriding. The second building block encapsulates data flow and control flow analyses, enabling the developer to specify precise conditions for validity of a transformation in terms of concepts like dominance and liveness.

Based on these approaches, we have implemented a refactoring engine as part of a larger effort to generate IDEs from declarative language specifications using the JastAdd meta-compiler tools. The described building blocks were successfully used as a foundation for other refactorings such as Rename, Extract Method, and Encapsulate Field.

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

General Terms Languages

Keywords Refactoring, IDEs, extensible compilers

1. Introduction

Most useful refactoring transformations can be composed from more low-level, “simpler” transformations. Opdyke [Opd92] was the first to make this observation and gave a set of useful primitives for refactoring object-oriented programs, such as create an empty class or change a member function name. He also defined several functions for describing preconditions that the program should meet to ensure that its behaviour will be preserved by the transformation. A few years later, Roberts introduced additional postconditions for the composition of low-level refactorings into high-level refactorings [Rob99]. Since then, most refactoring works have built on these pioneering concepts, revealing the importance of primitive transformations and analysis functions. Looking at existing implementations of mainstream refactoring tools, however, it seems that these primitive transformations and analyses are still a common source of flaws in the automation of refactorings.

Motivating examples Take for instance Extract Class, a refactoring that has been recently introduced in Eclipse 3.4. Although all of its transformation steps have been carefully thought out, we could easily find common cases where the current implementation fails to refactor the program correctly. Figure 1 shows an example where the return type of a method is accidentally changed due to name shadowing. The issue does not lie in the composition of the transformation steps, but rather in the analysis of one of the low-level transformations for inserting a nested class.

Similar problems affect Extract Interface in Eclipse: The core constraint generation and solving have been well implemented, but the refactoring may still break compilation because of insufficient checks regarding name bindings. In particular, it is wrongly assumed that the new top-level interface is directly visible from any point in the program. This kind of flaw is in fact not related to the core ingredients of the implemented refactoring, and inherent to the complexity of name binding rules in mainstream languages. It could therefore be avoided if a set of carefully crafted building blocks were available to refactoring developers.

Challenges Such building blocks are, however, very difficult to implement correctly for mainstream languages. The example of Figure 1 shows that even a primitive transformation like create a member class, which seems trivial at first, must recurse to a compiler-like analysis in order to preserve name bindings. Besides, this analysis has to be fully consistent with the compiler used within the IDE. Analysis
functions are therefore more important and much harder to get right than the transformations themselves.

Correctly implementing the analyses is not enough, though. We ought to make them easily extensible and reusable: extensible by compiler developers when adding new language constructs, and reusable by refactorers when defining refactorings for the implementation of high-level refactorings. By hiding the complexity of analysis functions into user-friendly building blocks, we hope refactoring authors will be able to focus, at last, on transformation steps only.

The paper is organised as follows. In Section 2, we discuss how to reuse and complement compiler analyses from JastAddJ, an extensible Java compiler. In Section 3, we show how to package these analyses into convenient building blocks to be used when implementing refactorings. In Section 4, we present our current implementation of these ideas. In Section 5, we conclude and discuss future work.

2. Reusing compiler analyses

Mechanising refactoring transformations often requires the same kind of analysis as in compiler passes. In contrast to some of our previous work where analyses were implemented from scratch [VEdM06], we propose here to implement them as direct extensions to JastAddJ, a full-fledged extensible Java compiler [EH07]. The compiler is built using the meta-compiler system JastAdd, which enables us to create modular extensions that can easily be composed with the existing compiler.

Name analysis Name analysis in JastAddJ is implemented as a family of attributes on the nodes of a program’s abstract syntax tree. The definitions of these attributes are given by defining equations that specify the result of looking up a name at a certain node in terms of the lookup attributes of neighbouring nodes. For example, the equation for looking up a variable name \( n \) on a statement in a block specifies that \( n \) should first be looked up as a local variable in the block, and if no such variable is found, it should be recursively looked up on the parent node (i.e., the block itself).

JastAdd supports a form of aspect oriented programming, where the defining equations for attributes and supporting pure Java code can freely be organised into different aspects according to their concern, and are then woven together into a compiler for the whole language. This makes the compiler very modular and extensible: support for new language features is added as a set of aspects that can be modularly combined with the basic implementation.

Access computation Refactoring specific analyses can be implemented as an extension of the JastAddJ compiler, extending and reusing existing analyses of the compiler. One very useful extension is access computation. Whereas, for example, variable lookup determines which declaration \( d \) a name \( n \) refers to at a certain node \( p \), variable access computation does the reverse: It determines a (possibly qualified) name \( n \) that can be used to access a declaration \( d \) from a program position \( p \).

The implementation of access analysis as an attribute makes it easy to implement access computation: For every defining equation for name lookup, we can give a corresponding equation for access computation. We show in [SEdM08] that this correspondence is systematic, and that the resulting access computation is both correct and efficient, even on fairly large input programs. And just like the underlying name analysis can easily be expanded with equations for new language features, we can also maintain the access computation for the extended language with little effort.

Renaming Given this implementation of access computation, a Rename refactoring is easily implemented: For every variable name in the program to be refactored, we temporarily store the declaration they refer to (we call this step locking down all names) before performing the renaming. Then we go over the program again, and check for every name whether it still refers to the same declaration as before (using the name analysis of the compiler). If it does not, access computation is used to obtain a name that does, and the old name is replaced by the new one (this is the unlocking step).

From a correct implementation of access computation, we thus obtain a Rename refactoring that does not change the binding structure of a program. This is arguably a more practical correctness criterion for renaming than the classical requirement of semantics preservation, which is impossible to achieve in the presence of features such as reflection or dynamic class loading.

This general framework of access computation greatly simplifies the maintenance of the refactoring. When extending the underlying language with new constructs, it suffices to extend the equations for lookup and access computation to get an up-to-date implementation of Rename. But the frame-
work is much more widely applicable than this: Many refactorings (such as Extract Method or Inline Variable) need to move or duplicate pieces of code. One necessary (though obviously not sufficient) requirement for such operations to be performed correctly is that names in the code still refer to the same declarations as they did before moving.

3. Building blocks

The reuse of compiler analyses greatly simplifies the task of implementing correct refactorings, yet having to implement every new refactoring from scratch would put an undue burden on the programmer. It is therefore desirable to have a library of primitive transformations and analysis functions that refactoring developers can leverage [Opd92, Rob99, KK04]. We describe several such “building blocks” here, how they are implemented in our framework, and how they are used in other refactorings.

Two well-known examples of building blocks are inserting a method into a class (Insert Method) and inserting a member class into a class (Insert Nested Class). When inserting a method into a class, one would usually want to check if the newly inserted method overrides any other method, or is overridden by another method in turn. This is usually not desired, as it may affect dynamic method resolution at runtime. In a language like Java, however, the rules for method overriding are quite subtle, and corresponding checks are often omitted by the refactoring implementations of popular IDEs.

On the other hand, every Java compiler must implement overriding checks for purposes of error checking. In the JastAdd compiler, these checks are implemented as a predicate overrides, which our implementation of Insert Method can directly reuse. Similar checks are needed for Insert Nested Class in order to prevent the kind of flaw shown in Figure 1.

Besides these special building blocks, there are two very general components that we have found to be eminently useful in implementing many of our refactorings: One is the name analysis toolkit introduced in the previous section, which allows the locking and unlocking of names to ensure that they refer to the desired declarations, another is a toolkit for control and data flow analysis. The former handles the more static aspects of a program, whereas the latter is concerned with its dynamic runtime behaviour.

Preserving bindings As we have explained, preserving bindings is at the heart of any refactoring that moves, creates or duplicates code. Before editing code, all names are locked down. The code is then edited, and afterwards names are unlocked again, potentially qualifying or otherwise adapting them to ensure they still refer to the correct declarations. If, in some rare unavoidable circumstances, the name analysis component cannot preserve name bindings, the refactoring is aborted, and we can provide the user with an error message pinpointing the precise location of the problem.

Let us illustrate the use of this building block on a simple example. Say we want to create a setter method for a field $x$. We first create an empty method with a single parameter named $x$, too. For this step we use Insert Method, which we do not detail here. Then, we simply add, into the body of the new method, a statement that assigns the field with the parameter. The code reads as follows:

```java
AdjustmentTable table = new AdjustmentTable();
VarAccess left = field.createLockedVarAccess(table);
VarAccess right = param.createLockedVarAccess(table);
body.addStmt(new ExprStmt(new AssignExpr(left,right)));
table.adjust();
```

The variable field is the field declaration of $x$, while param is the parameter declaration of the setter method. The AdjustmentTable is an auxiliary data structure that keeps track of names to be adjusted. We create variable accesses for both of them to be used in the assignment. Although both declarations have the same name, the last call to adjust() unlocks previously locked names and ensures they point to their original declarations. In this precise example, the framework therefore automatically constructs the assignment this.x = x.

Of course, the approach scales to much more complex cases. It not only simplifies precondition checking (which can ignore any name-related issues), but also avoids a host of common pitfalls [EESV08].

Preserving flow Many analyses require precise control flow and data flow analysis. When extracting a block of code into a method, it may for instance be necessary to compute the set of live and updated variables in that block to compute method parameters and the return value. In general we need such analyses when moving around code to preserve the control-flow in the refactored program. We have implemented a building block providing extensible control-flow analysis at the source level, rather than at an intermediate representation [NNEHM08]. This is particularly convenient for refactorings that require a direct mapping to the source code.

The control flow graph is extremely useful when dealing with non-local transfer of control in Java programs such as exceptions, break, continue, and return. In particular, intricacies due to interactions between non-local transfer and finally blocks in Java give raise to many subtle errors when moving code. The flow building block enables the developer to reason about the validity of a transformation in terms of successors, domination, and liveness. Additional abstractions can easily be built on top of these analyses using the equation based fixed-point computation framework in JastAdd. A typical example is to compute liveness within a set of selected statements rather than for a complete method.

While the flow analysis framework itself is quite mature its interface to refactorings is still work in progress, and we take inspiration from important research on restructuring programs at statement level, e.g. [GN93, Ett07].
4. Implementation

We have implemented the two high-level building blocks described in Section 3 in a refactoring engine extending the JastAdd extensible Java compiler. That engine is an integral part of a larger effort to generate IDEs from high-level language specifications based on attribute grammars using the JastAdd meta-compiler tools. From these specifications, an IDE is generated supporting various refactorings and other services such as name completion, content outline, cross referencing, and various semantic search facilities.

Renaming is supported for all declarations using the name analysis toolkit described in Section 2. This toolkit has also been successfully extended to support inter-type declarations from AspectJ which heavily modifies the name binding rules in Java. We have also used it as a building block with the described locking and unlocking steps as a foundation for other well-known refactorings.

Consider for instance the Encapsulate Field refactoring, which makes a public member field private, adds getter and setter methods to its host class, and replaces every access to the field by invocations of the appropriate methods. The locking and unlocking facilities makes this refactoring almost trivial to implement: For every access to the field being encapsulated we can determine how to access the appropriate getter or setter method from this particular position in the syntax tree, and insert the computed access. The framework ensures that tricky corner cases are handled correctly, e.g., a getter method being overridden by a local method and hence rendered inaccessible.

Another example we implemented is Extract Method, which extracts a piece of code into a new method. The naming toolkit proved useful in this setting as well, for instance to detect accidental overriding and to construct type accesses for the throws clauses in the extracted method. This refactoring crucially relies on the control and data flow analysis toolkit. In particular, a selected piece of code can not be extracted if there is a break statement in that code which targets a loop that is not included in the selection. Another example would be the computation of which exceptions may be thrown by the extracted code in order to construct the throws clause of the extracted method.

5. Conclusion and future work

By reusing analyses from the JastAdd extensible Java compiler we gain an extensible refactoring engine that is easy to maintain as the language evolves. We have presented two extensible building blocks that encapsulates name binding and flow analysis intended to be used as a base when building refactorings.

The combined system provides an excellent platform for implementing extensible refactorings. As an example we extended the refactoring engine to support inter-type declarations which heavily affects name analysis. Renaming can easily be extended to cater for the new language feature and other refactorings that depend on the binding building block, e.g., Extract Method, are automatically updated. An added bonus of building on the compiler is that we get consistent analyses throughout the IDE, from interactive wizards to compilation.

As future work we plan to build a library of useful basic operations and do more large scale experiments with composition. Our current approach is manual, but it would be interesting to provide a composition framework or even to verify composition statically in the spirit of [KK04]. Finally, the extensible analysis framework and refactoring engine opens up possibilities to implement framework specific refactorings with moderate effort.

References


