Thesis Overview

A Toolkit for Constructing Refactoring Engines

Jeffrey L. Overbey
Department of Computer Science
University of Illinois at Urbana-Champaign
Advisor: Ralph E. Johnson
jeffrey.l@over.bz

Refactoring is a disciplined technique for restructuring a software system in which a programmer uses a sequence of small-scale, behavior preserving changes to effect a larger-scale, behavior-preserving change to the system [1, 2]. Each of these small-scale changes, or *refactorings,* makes an incremental change to the system's internal design or code quality while leaving the externally observable behavior of the system unchanged. By performing many such changes in sequence, ensuring after each step that all tests pass and the system's behavior has not changed, the programmer can make substantial design changes while minimizing the likelihood of introducing new bugs. Often, systems are refactored because they require a new feature or bug fix that cannot be accommodated by the system's existing design. Changing the design first, ensuring that all tests still pass, and postponing behavioral changes until afterward tends to be much less error-prone than changing everything at once.

Many refactorings are simple but tedious, which makes them good candidates for automation. Common refactorings include renaming identifiers, moving code between classes or functions, and encapsulating variables. Most integrated development environments (IDEs) – including Eclipse, IntelliJ IDEA, Microsoft Visual Studio, and Apple Xcode – provide support for automated refactoring. These features allow the programmer to select a portion of the source code and select a particular refactoring to apply. The IDE then performs a static analysis of the source code, determining whether the desired change will change its behavior. If the behavior will not change, the IDE modifies the source code, showing the user a side-by-side, before-and-after view of the source code so that he can visually inspect the changes.

The Problem

While automated refactorings are available in many IDEs (and in fact have been commercially available since the late 1990s), building an automated refactoring tool for a new programming language is still a very expensive and difficult task. The majority of the code in a refactoring tool is devoted to supporting components, including a parser and source code manipulation infrastructure; only a small fraction of the code is devoted to the refactorings themselves. Many languages do not yet have automated refactoring tools available, and these massive infrastructural requirements can make developing new tools prohibitively expensive.

The Solution

This dissertation shows that it is possible to build libraries and code generators that provide many of the supporting components for a refactoring engine. This can significantly reduce the amount of code needed to build a refactoring tool for a new language.

The dissertation contains two major contributions (based on two previously published papers [3, 4]). The first focuses on generating the syntactic components of a refactoring engine: the lexer/parser, abstract syntax tree (AST), and source code manipulation infrastructure. The second focuses on precondition checking, the procedure that determines whether or not a refactoring will change a program's behavior.

The techniques were used to build the refactoring engine in Photran, an Eclipse-based IDE for Fortran, as well as prototype refactoring tools for PHP and BC. In all three tools, about 90% of the code was either generated or contained in a language-independent library; the majority of the hand-written code was devoted to the implementation of refactoring transformations.

Generating Rewritable Abstract Syntax Trees

The primary program representation in a refactoring tool is a data structure known as an *abstract syntax tree (AST).* While many compilers also use ASTs, they are often not suitable for use in a refactoring tool. In a refactoring tool, an AST must (1) accurately model the original source code, (2) maintain a precise mapping

between AST nodes and locations in the source code, and (3) capture enough information about preprocessing (in the case of languages like C and Fortran) to facilitate manipulation of the original, un-preprocessed code.

Since ASTs designed for refactoring tools have such specialized requirements, it may be desirable to construct an AST specifically for refactoring, rather than attempting to retrofit these requirements onto a compiler's AST. One way to do this is to take the grammar supplied to a parser generator and *annotate* it with information about how to construct ASTs. Then, the parser generator can be extended so that it generates AST node classes as well as parsers that construct ASTs comprised of these nodes.

Such a system has been implemented in a tool called Ludwig. Ludwig's generated ASTs implement a very rich API. The API provides methods to traverse the tree, as well as to search for particular nodes. More importantly, it makes it easy to manipulate source code. There are API methods for adding, moving, copying, modifying, and deleting AST nodes, as well as emitting revised source code from a modified AST. When the AST is modified and revised source code is emitted, it preserves the source code's original formatting, including spacing and comments.

As one notable example, Ludwig was used to construct Photran's Fortran 2008 parser and AST. The Ludwig-generated AST and source manipulation API were used to build all 31 of the refactorings available in Photran 7.0. Its Fortran 2008 grammar was about 5,000 lines long, but that generated about 90,000 lines of Java code.

Differential Precondition Checking

When a programmer wants to implement a new automated refactoring, often, the most difficult part is not designing the source code transformation but rather ensuring that the transformation will not change the behavior of the program being transformed. Traditionally, refactoring tools have guaranteed behavior preservation by exhaustively checking for everything that could cause an error, and then performing the source code transformation only after all such checks pass. The process of checking for potential errors prior to transformation is called *precondition checking.*

An appealing alternative to traditional precondition checking is to transform the program first, and then analyze the program *after* it has been transformed, attempting to detect any unintended semantic changes. Differential precondition checking is a specific technique for performing precondition checking in this way. A differential precondition checker builds a "semantic model" of the original source code, performs the refactoring transformation to modify the source code, and then builds a second semantic model from the modified code. The two models are compared, and the refactoring indicates what differences are expected. If all of the actual differences between the two semantic models are expected, then the refactoring is deemed to be behavior preserving and proceeds as usual; if unexpected differences are found, they are displayed to the programmer with an informative error message, and the programmer is given the option to abort the transformation.

Perhaps the most interesting aspect of differential precondition checking is that it can be implemented in a way that is language-independent; a differential precondition checker can be built once, optimized, placed in a library, and reused in refactoring tools for many different languages. Such a differential precondition checker was used to implement 18 refactorings for three different languages: 7 for Fortran, 2 for PHP, and 9 for BC.

The full dissertation [5] is available online at http://jeff.over.bz/dissertation.

Acknowledgment

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign, its National Center for Supercomputing Applications, Cray, and the Great Lakes Consortium for Petascale Computation.

Jeffrey L. Overbey
jeffrey.l@over.bz

References

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

[2] M. Fowler. *MF Bliki: RefactoringMalapropism.* Refactoring Malapropism. http://martinfowler.com/bliki/RefactoringMalapropism.html, January, 2004.

[3] J. Overbey and R. Johnson, "Generating Rewritable Abstract Syntax Trees." In *Software Language Engineering: First International Conference,* volume 5452 of *Lecture Notes in Computer Science,* pages 114-133, Springer-Verlag, Berlin, Heidelberg, 2009.

[4] J. Overbey and R. Johnson, "Differential Precondition Checking: A Lightweight, Reusable Analysis for Refactoring Tools." In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011).*

[5] J. Overbey. *A Toolkit for Constructing Refactoring Engines.* PhD thesis. University of Illinois at Urbana-Champaign, Champaign, IL, 2011.