NEW TECHNIQUES FOR THE RAPID CONSTRUCTION OF REFACTORING ENGINES

BY

JEFFREY L. OVERBEY

B.S., Southeast Missouri State University, 2004
M.S., University of Illinois at Urbana-Champaign, 2006

DISSERTATION PROPOSAL

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Abstract

Production refactoring tools are complex and time-consuming to build, often requiring tens to hundreds of thousands of lines of hand-written code. The proposed thesis will suggest several techniques which can expedite the process of building a refactoring tool for a new language. Many of a tool's infrastructural components can be encapsulated in frameworks, libraries, and code generators, allowing the developer to focus on building analyses and refactorings rather than infrastructure. But it is also possible to reduce the implementation cost and improve the correctness of refactorings themselves through the use of a language-agnostic precondition checking engine.

One part of the proposed thesis will address syntactic aspects of refactoring, suggesting that a grammar can be used to generate not only a parser but also a preprocessor-aware, syntactic rewriting infrastructure. A concise set of grammar annotations is proposed which allows the structure of an abstract syntax tree to be specified in a grammar. A tool can then generate both a parser and a fully rewritable abstract syntax tree which can be used to manipulate source code while preserving every aspect of the original source code, including spacing and comments. Moreover, the generated infrastructure can support C preprocessor directives in the language, even if the language is not C or C++.

The second part of the thesis will address the semantic aspects of refactoring, proposing a new architecture for refactoring engines in which the refactoring engine maintains a model of certain aspects of the program's semantics, and the sequence of changes applied to the abstract syntax tree is used to define an equivalence class among semantic models. Rather than attempting to check an exhaustive set of preconditions, the program transformations can be performed, and the semantic models of the original and transformed programs can be tested for equivalence to determine whether or not the transformation is a valid refactoring. This semantic model can be easily extended to include preprocessor directives as well. One instantiation of this architecture will be investigated in detail, along with its advantages, disadvantages, and applicability.

As proofs of concept, these techniques are being used to develop the refactoring engines in Photran, an Eclipse-based IDE and refactoring tool for Fortran, and Ludwig, a lexer/parser/AST generator, as well as a prototype refactoring tool for Lua, a dynamically-typed scripting language.

Acknowledgments

Portions of this work were supported by the United States Department of Energy under Contract No. DE-FG02-06ER25752 and are 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, IBM, and the Great Lakes Consortium for Petascale Computation.

Special thanks to Ralph Johnson for his exceptional advisement; Darko Marinov for several discussions about testing, translation validation, and credible compilation; Robert Bowdidge for discussing the refactoring engine in Apple Xcode; John Brant for discussing the Smalltalk Refactoring Browser; Nick Chen for introducing me to OmniGraffle and BibDesk; and Brett Daniel for his help with TestOrrery.

Introduction

1. The Problem

Reuse is common

Building a production-quality refactoring tool generally requires a substantial amount of infrastructure—often tens to hundreds of thousands of lines of code—and so attempts to reuse components are common. Refactoring support in Apple Xcode 3.0 was built using a stock C/C++/Objective-C parser and used Xcode's *snapshot* feature to support multi-file undo [18]. Refactoring support in NetBeans is based on the front end from Sun's javac compiler [17]. The JastAddJ compiler was similarly modified to support refactoring [101]. Peter Sommerlad's group at the Institute for Software [60] has successfully added refactoring support onto Eclipse IDEs for C/C++ [52], Ruby [30], Python, and Groovy [68] by extending their existing language infrastructures and using the Eclipse Language Toolkit (LTK) to provide the GUI. Xrefactory adds C/C++ refactoring support to Emacs based on a front end from the Edison Design Group [113].

Reuse impacts architecture

Of course, creating a refactoring tool by gluing together stock components and custom infrastructure has ramifications. It can clearly impact the internals of the tool (say, when code quality is compromised as refactoring-specific concerns are hacked onto components that were never intended to support them), but reuse can also have more visible effects. For example, the refactorings in the Eclipse C/C++ Development Tools (CDT) support only a single configuration of the C preprocessor, so if one block of code is guarded by `#ifdef WIN32` and another by `#ifdef LINUX`, it will only refactor one of them. This is directly attributable to the fact that the CDT's language infrastructure was designed to support only a single preprocessor configuration (indeed, that is acceptable for most IDE functionality); modifying it to support multiple configurations was estimated to require at least eight person-months of effort [95].

Reuse is not always possible

Unfortunately, it is not always possible to reuse stock components in a refactoring tool. Sometimes this is due to licensing issues. Sometimes this is due to technical issues. For example, if a parser outputs a lowered program representation like three-address code, it can be difficult or impossible to map elements of the lowered representation back to positions in source code, limiting its usefulness in a refactoring tool. Similarly, if code must be preprocessed before it is parsed, it may be difficult or impossible to correlate the parser's output with locations in the unpreprocessed source code.

Reuse is ad hoc

Perhaps the bigger problem, though, is that no serious attempt has been made to reuse components *among refactoring tools*. In each of the tools described above, components were reused from compilers or IDEs; the decision about what to reuse was based on what happened to be available for their particular language. A substantial effort was still required to add refactoring-specific functionality onto the existing infrastructural components.

This problem is exacerbated when few or no components can be reused from existing tools. This can often mean that the programmer must implement tens of thousands of lines of infrastructure before he can even begin writing actual refactorings.

Infrastructure demands are high

The proposed dissertation will identify several ways in which commonality among refactoring tools can be exploited. The end result will be not only a suite of "tools for building refactoring tools," but also several architectural and algorithmic advances that can make refactoring tools easier to build in general.

Better reuse is possible

2. Background

Architectural Forces: Refactoring Engines vs. Compilers

In the next subsection, we will describe a typical architecture for refactoring tools. Many of the forces driving this architecture derive from the (obvious) fact that a refactoring tool is *not* a compiler. There is some commonality between refactoring tools and compilers, but there are also significant differences.

Refactoring engine ≠ compiler

Ultimately, a refactoring tool must modify the user's source code, and *the user will maintain the refactored code.* So a refactoring cannot just prettyprint an AST, ruining the user's formatting. It cannot even remove comments. And it certainly cannot lower the representation and output a semantically equivalent but visually dissimilar program.

The user owns the refactored code

Thus, the primary program representation in a refactoring tool is always one that can be mapped directly to the user's source code. This is usually an abstract syntax tree. And, generally speaking, all analyses and transformations must be done directly on the AST. (Contrast this with a compiler, where analysis and transformation are generally done on a much simpler intermediate form.)

No lowered representation

Furthermore, in languages like C, C++, C#, and Fortran, the user's code is generally run through a preprocessor before a compiler ever sees it. However, since a refactoring tool must modify the *user's* source code, it must be able to parse, analyze, and transform code with embedded preprocessor directives.

No preprocessing

Another major difference between refactoring tools and compilers are the types of transformations that are performed. Most compiler optimizations are highly nontrivial, intraprocedural transformations. Refactoring transformations tend to be much simpler, but they are oriented at design-level constructs—methods, fields, classes, etc.—and tend to require potentially program-wide analysis and transformation.

Easy but program-wide transformations

Finally, refactoring tools are interactive. First, this means that analyses do not have to be completely conservative, and transformations do not have to be completely accurate; often it is acceptable for the tool to make a "guess" then ask the user to visually inspect the result. Second, the user decides what refactoring to invoke and can provide input, so there is generally no need for the tool to estimate the profitability of its transformations; that responsibility has been transferred to the user. Finally, there are speed considerations. Users generally refactor much less frequently than they compile, so refactorings can be slower than typical compiler transformations, as long as they are reasonably so. However, more expensive compiler optimizations (intended to be used only during automated, overnight builds) can arguably be slower than refactorings, since they are not intended to be run in "interactive time."

Interactivity relaxes requirements

The Architecture of Refactoring Engines

Figure 1 illustrates the architecture of a typical refactoring tool, biased somewhat toward the design of the Eclipse Java and C/C++ Development Tools (JDT and CDT, respectively). The architecture follows the relaxed layered model [20, p. 45], where each layer generally depends on several of the layers below it.

Relaxed layered architecture

AST is critical

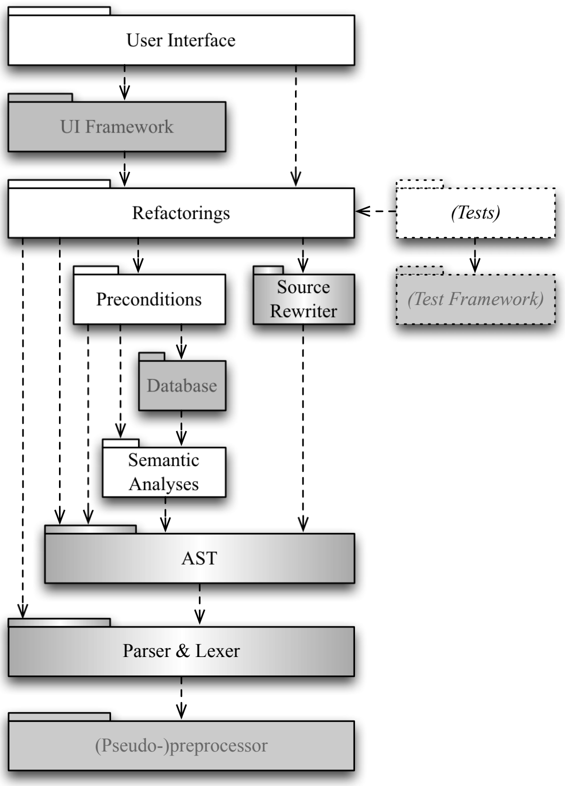


Figure 1: Architecture of a typical refactoring tool. Components that can be implemented generically, accoording to the proposed dissertation, are shown in *solid gray* if they can be implemented in a library or framework and in a *gray-white gradient* if they can be implemented in a code generator. Testing packages are also shown to emphasize that, although they are not technically part of the architecture, they are an essential part of such a tool.

The bottommost tiers consists of the abstract syntax tree (AST) and the machinery needed to construct it: the lexer, parser, and (if applicable) preprocessor. The AST plays a critical role in a refactoring tool, as virtually every other component depends on it and makes assumptions about its structure.[1] Clearly, the AST must be used in analysis and precondition checking. But many refactoring tools also use the AST for source code transformation, effectively adding, deleting, moving, or modifying nodes in order to make changes to source code.

When a language must be preprocessed before it can be parsed, some information about the effects of preprocessing must be included in the AST in order to refactor successfully. For example, it is often necessary to know which AST nodes originated from `#include` directives in the original code. This requires additional information to be passed from the preprocessor to the lexer and parser; otherwise the preprocessor behaves exactly as it would for a compiler. However, correctly handling conditional compilation (`#if`) directives requires substantial changes to the preprocessor's behavior (discussed later); Garrido [48] calls the resulting tool a *pseudo-preprocessor*.

Pseudo-preprocessing

Choice of semantic analyses varies

[1]This is analogous to a three-tier Web application, where the data access tier is at the bottom, and changes to the database schema tend to ripple through the data access and business logic tiers as assumptions about the data model change.

Immediately above the AST is a suite of semantic analyses. Exactly what analyses are implemented depend on what refactorings are implemented. Rename and Move, for example, require only name binding analysis; Extract Method requires some level of flow analysis; and loop transformations require array dependence analysis. The choice of analyses can also depend on the language being refactored. For example, Extract Local Variable requires type checking in a statically-typed language but not in one that is dynamically typed; and Extract Method can be implemented without flow analysis if the language supports pass-by-reference.

Large projects require a cross-reference database

When the refactoring tool will operate on large projects, often some semantic analysis information must be saved to disk in order to achieve reasonable performance. For example, renaming a method requires knowing the all of the call sites of that method and of any methods that override it; most tools choose to save this information (or some approximation thereof), based on the observation that (1) the brute-force approach of parsing and analyzing every file in the project while the user waits on the refactoring to complete would be prohibitively expensive, and (2) in a million-line project, most of the files will likely *not* call that method anyway. In an IDE, the cross-reference database can be updated incrementally as the user edits individual files, so maintaining this database need not affect the tool's responsiveness.

Preconditions, source rewriting, transformations, UI

The remaining components support refactoring more directly. There is a suite of precondition checks which are common among several refactorings. A source rewriter maps AST transformations to textual (offset/length) transformations, accounting for comments and reindentation. And the topmost layer is, of course, the user interface, which for convenience is generally integrated into an IDE or text editor.

3. The Solution

Libraries, Frameworks, & Code Generators

Most of the components in a refactoring tool *need not be implemented by hand.* Specifically, in Figure 1, the components shown in solid gray can be implemented in a library or framework, and those shown in a gray-white gradient can be implemented in a code generator.

Reduced infrastructure cost

Libraries, frameworks, and code generators can provide several advantages. The most obvious is that they can reduce the amount of code that must be written and thus reduce development time. However, by reducing the amount of new code that must be written, they can also increase reliability. In the end, they should allow the developer to focus on the "interesting" parts of the implementation rather than tedious, mundane, or routine support code.

Focus on analysis and transformation

And that is what we want to achieve with regard to refactoring engines. Arguably, the "interesting" parts of a refactoring tool—the parts where a developer would expect to expend the most effort—are the analyses, refactorings, and the corresponding tests and UI components. Notice that, in Figure 1, these are exactly the components that are left entirely to the developer; all of the infrastructural components can be implemented in libraries, frameworks, or code generators.

Lex/Yacc, LTK, TestOrrery

For several of these components, this is possible with existing technology. Lexer and parser generators existed as early as the 1960s [63]. AST generators also exist, although they are less commonly used. One example of the "UI Framework" component is Eclipse, a framework for building integrated development environments, extended with the Language Toolkit (LTK), which provides (among other things) a wizard-based interface for refactoring, including a *diff*-like GUI for previewing the changes that will result. The best example of the "Test Framework" component is the JUnit unit testing framework [4] extended with TestOrrery [6], a library for bounded-exhaustive test data generation. By using TestOrrery to produce Java ASTs, this framework was used to build ASTGen [2], a system used to test and expose several bugs in the Eclipse JDT and NetBeans' refactorings for Java [32, 47].

Database, rewriter, preprocessor

On the other hand, it is less clear that the program database, source rewriter, and (pseudo-)preprocessor can be implemented in a truly language-independent fashion, as these three components all depend heavily on the language being refactored. Clearly, there could be some commonality among refactoring tools for similar languages (say, C and Fortran, or Java and C#). This is less obvious if the languages are, say, Fortran, Lua (a dynamically-typed scripting language), and EBNF grammars. Finding commonality in these components among such diverse languages is exactly what the proposed dissertation will address.

Proposal

1. Overview

The proposed thesis will focus on three major topics. (A detailed list of contributions appears in Figure 3.)

1. Generating rewritable ASTs

First, it will show that it is possible to generate an AST-based, syntactic rewriting infrastructure from a BNF or EBNF grammar. The grammar is annotated using a concise set of six annotations which describe the structure of an abstract syntax tree. A tool can then generate both a parser and a rewritable AST which can be used to manipulate source code while preserving every aspect of the code's formatting, including spacing and comments.

2. Universal precondition checking

The second part of the proposed research will propose that refactorings need not check an exhaustive set of preconditions *a priori*. Instead, the programmer can simply transform an AST, indicating what analyses should be preserved after the transformation has been applied, and the refactoring engine can determine whether that actually happens. The thesis will describe a specific algorithm ("universal precondition checking") and a language-independent library, called the *virtual program graph* (VPG), which implements this algorithm at scale while also providing a cross-reference database and indexing infrastructure.

3. C preprocessor support

The third and final part of the dissertation will discuss what is needed to support C preprocessor directives in the language being refactored, even if the language is not C or C++. The most challenging aspect is illustrated in Figure 2, where the conditional (`#ifdef`) directive splits a statement; any naïve approach (like treating preprocesor directives as comments) will fail to parse this, and "normal" preprocessing would ignore all but one arm of the conditional, which is generally insufficient for refactoring. The proposed research will describe a mechanism whereby any LR(1) parser can be modified to parse code with embedded conditional directives; it will also describe how to subsequently incorporate preprocessing directives into ASTs and the VPG.

```
printf("You are running "
#ifdef WIN32
  "Windows"
#else
  "%s", run_command("uname -s")
#endif
);
```

Figure 2: Example of an incomplete conditional.

Contributions

Generating Rewritable ASTs

1. A new set of annotations for specifying AST structure in a grammar, and a method for augmenting these ASTs with concrete syntax so that they can be used for source code rewriting
2. A formalization of the AST generation algorithm, including constraints that an annotated grammar must satisfy for AST generation to succeed and proofs of several safety properties

Universal Precondition Checking & VPG

3. The notion that many preconditions can be specified (or verified) by expressing analysis preservation requirements during the transformation, and then checking whether the preservation was satisfied, and an algorithm for doing so ("universal precondition checking") using a program representation based on the *program graph* semantic model
4. A scalable implementation of a universal precondition checking library
5. Specifications of several common refactorings using this model

Language-agnostic Pseudo-preprocessing

6. An algorithm for language-agnostic conditional completion in LR(1) parsers
7. Mechanisms for integrating pseudo-preprocessing with generated ASTs
8. A strategy for representing preprocessor directives in a program graph and including them in universal precondition checking

Figure 3: Contributions of the proposed thesis.

2. Generating Rewritable Abstract Syntax Trees

Two contributions

The first part of the thesis will expound the idea that a BNF or EBNF grammar can be annotated and used to generate an AST-based syntactic rewriting infrastructure. It will make two contributions:

1. A new set of annotations for specifying AST structure in a grammar, suitable for generating a syntactic rewriting infrastructure; and
2. A formalization of the AST generation algorithm, including constraints that an annotated grammar must satisfy for AST generation to succeed and proofs of several safety properties.

Annotations in [91]

This work is, for the most part, complete. A paper presenting the grammar annotations and an overview of how they are used to generate a rewriting infrastructure was presented in the First International Conference on Software Language Engineering [91]. It describes the grammar annotations in detail, provides a link to our reference implementation (Ludwig), and briefly describes our experience implementing the refactoring engine in Photran, a refactoring tool for Fortran. An example of an annotated grammar appears below in Figure 4.

Purpose of formalization

The formalization is mostly complete and serves as the basis of the implementation in Ludwig, but it has not yet been published. An outline appears in Figure 6. The purpose of formalizing the AST construction is to address the fact that there are some combinations of annotations that do not "make sense" or that lead to ambiguities in the (static or dynamic) structure of the AST. A simple example is shown in Figure 5: The generated `IfStmtNode` contains only one member for a statement, and it is not clear whether this should represent the then-statement or the else-statement. Formalizing the AST construction will allow us to establish sufficient criteria to guarantee that cases like this will not occur, i.e., to guarantee that every program that can be parsed will also have a unique AST representation.

Annotated grammar:	Generated AST node class:

$\langle \textit{if-stmt} \rangle$ ::=
~~IF~~ $\langle \textit{expr} \rangle$ ~~THEN~~ $\overbrace{\langle \textit{stmt} \rangle}^{\text{thenStmt}}$ ~~ENDIF~~
| ~~IF~~ $\langle \textit{expr} \rangle$ ~~THEN~~ $\overbrace{\langle \textit{stmt} \rangle}^{\text{thenStmt}}$ ~~ELSE~~ $\overbrace{\langle \textit{stmt} \rangle}^{\text{elseStmt}}$ ~~ENDIF~~
	`class IfStmtNode {`
	`  public ExprNode expr;`
	`  public StmtNode thenStmt;`
	`  public StmtNode elseStmt;`
	`}`

Figure 4: Example of generating an AST node from an annotated grammar. A single `IfStmtNode` class is generated, corresponding to the $\langle \textit{if-stmt} \rangle$ nonterminal on the left-hand side of the ::= symbol. The symbols on the right-hand side correspond to fields in this class. The labels "thenStmt" and "elseStmt" assign the AST nodes for the two $\langle \textit{stmt} \rangle$ nonterminals to fields with those names. Terminal symbols that are ~~struck out~~ indicate that the corresponding tokens can be omitted from the AST.

Annotated grammar:	Generated AST node class:

$\langle \textit{if-stmt} \rangle$ ::=
~~IF~~ $\langle \textit{expr} \rangle$ ~~THEN~~ $\langle \textit{stmt} \rangle$ ~~ELSE~~ $\langle \textit{stmt} \rangle$ ~~ENDIF~~
	`class IfStmtNode {`
	`  public ExprNode expr;`
	`  public StmtNode stmt;`
	`}`

Figure 5: Example of an ill-defined annotated grammar. Cf. Fig. 4.

1. Formally define an annotated grammar (similar to the definition of a context-free grammar).
2. Given an annotated grammar, define what AST node classes will be generated and the inheritance relationship among these classes.
3. Establish criteria sufficient to guarantee (1) that every symbol in the grammar corresponds to exactly one concrete AST node class, and (2) that the inheritance relationship among these classes is valid.
4. Define the members of each AST node class.
5. Establish criteria sufficient to guarantee that every class member has a unique type and visibility and that its name is unique within the class.
6. Define an attribute grammar which describes how ASTs are built from the generated classes at parse time.
7. Prove that (1) this grammar is S-attributed, (2) it only instantiates concrete node classes, (3) assignments to node members are valid (well-typed), (4) each node member is assigned at most once, (5) all tokens are present in the AST, and (6) the original ordering of the tokens is preserved in the AST.

Figure 6: Outline of the formalization of the AST construction

3. Universal Precondition Checking & VPG

The second part of the thesis will make three contributions:

Three contributions

1. The notion that many preconditions can be specified (or verified) by expressing analysis preservation requirements during the transformation, and then checking whether the preservation was satisfied, and an algorithm for doing so ("universal precondition checking") using a program representation based on the *program graph* semantic model
2. A scalable implementation of a universal precondition checking library
3. Specifications of several common refactorings using this model

3.1 Preconditions and Analysis Preservation

Generally, refactorings proceed in four steps.

Refactoring = preconditions + transformation

1. *Check initial preconditions.* Initial preconditions are usually very inexpensive checks to ensure that the most basic criteria for applying the refactoring have been met. For example, the file to refactor must be writable, and the user's cursor may need to be placed on a particular construct in the program.
2. *Request user input.*
3. *Check final preconditions.* Final precondition checks validate the user input and then perform any program analyses necessary to (attempt to) guarantee that transforming the program will preserve the behavior of the original program.
4. *Modify the source code.*

Compilability & preservation

When specifying or implementing a refactoring, the most difficult part is final precondition checking: attempting to guarantee that the transformation will preserve the behavior of the original program. Generally, this means ensuring (1) that the refactored program will still compile after it has been transformed *(compilability)*, and (2) that a particular program analysis will be preserved *(preservation)*.

Example

One nontrivial example is the Extract Method refactoring, where the user selects a sequence of statements, and they are moved into a new method and replaced with a call to that method. Two critical parts of precondition checking are (1) ensuring that the statements do not include a `return` statement or a `goto` referencing a label outside the selection, and (2) ensuring that the effects of any variable assignments are propagated back from the extracted method. Stated in terms of analysis preservation, these are attempting to guarantee that the control flow and du-chains, both into and out of the original statement block, are preserved when it is replaced with the method call.

Transform, then check

In practice, refactoring tools generally try to determine whether analyses will be preserved *before* they transform the AST. However, it has been observed in the literature that it is possible to simply transform the AST and *then* determine whether or not analyses were preserved. This after-the-fact checking is used in some dependence-based program transformations for parallelization [112, 65] and was also used in Griswold's restructurer [53]. We will refer to this alternative as *a posteriori checking,* and we will use *a priori precondition checking* to refer to the more common alternative where the checking is completely done *before* transforming the AST.

Advantages of *a posteriori* checking

The main advantage of *a posteriori* checking is that it makes it easier to ensure that every case has been covered. Consider preserving control flow in the Extract Method example above. *A posteriori* checking is easy: One must simply verify that the two flow graphs are isomorphic. On the other hand, checking for control flow preservation *a priori* means explicitly checking for `return` statements, certain `goto` statements, and any other construct that could cause the control flow graph to deviate.

Goal: generic *a posteriori* checking

The proposed research will capitalize on the idea of *a posteriori* checking by proposing a *generic* way to check whether an AST preserves certain program analyses, regardless of what language the AST represents and regardless of what the program analyses are. To achieve this, we will use a variation on a semantic model known as a *program graph.*

3.2 Program Graphs

Program graph

A *program graph* "may be viewed, in broad lines, as an abstract syntax tree augmented by extra edges" [81, p. 253]. These "extra edges" represent semantic information, such as variable scopes and bindings, control flow, inheritance relationships, and so forth.

AST-based semantic analyses

Unlike a compiler, where the AST, symbol table, and control flow graph are usually separate data structures, a program graph combines all such information into a single data structure. In a program graph, there are no symbol tables; rather, some nodes in the AST correspond to declarations, and references contain an edge pointing to their declaration. Scopes can similarly be represented by AST nodes; symbols can point to the scope in which they are defined. The control flow successors of a node are other AST nodes. A variable's du-chains would likely consist of AST nodes representing assignment statements and variable-use expressions. And so forth. In other words, in a program graph, all program analyses are stated in terms of AST nodes.

Superimposition

Alternatively, one might think of a program graph as an AST with the graph structures of a control flow graph, program dependence graph, du-chains, etc. "superimposed" [108]. The nodes of the AST also serve as nodes of the various graph structures; the edges connecting them are different.

Example in Fig. 7

An example of a Java program and a plausible program graph representation are shown in Figure 7. The underlying abstract syntax tree is shown in outline form; the dotted lines are the extra edges that make the AST a program graph. We have shown four types of edges. *Scope* edges link a declaration to the class or method in which it is defined. *Binding* edges link the use of an identifier to its corresponding declaration. Within the method body, *control flow* edges form the (intraprocedural) control flow graph; the method declaration node is used as the entry block and null as the exit block. Similarly, there are two du-chains, given by *def-use* edges.

Language-neutral, combined representation

The advantages of the program graph representation are twofold. First, it is generic enough that it a program graph can be built to represent virtually any conventional programming language. Every program has a syntax which can be represented as a syntax tree, and we hypothesize that the semantics needed to perform common refactorings can be represented as extra edges (and node annotations) on this tree. The second advantage of the program graph is that it summarizes the "interesting" aspects of both the syntax and semantics of a program in a single representation, obviating the need to maintain a mapping between several distinct representations.

3.3 Universal Precondition Checking

In addition to the advantages above, we claim that using a program graph-based program representation for refactoring has another advantage: It can serve as the basis for a language-agnostic, *a posteriori* precondition checking algorithm we call *universal precondition checking.*[2]

Universal checking is predicated on the idea that refactoring transformations can be described in terms of five *primitive transformations*, specifying what program analyses (i.e., what types of extra edges in a program graph) should be preserved across these transformations. We will first describe these primitive transformations; then we will describe what it means to preserve a program transformation across a transformation. Finally, we will use the Extract Method refactoring on a Java program to illustrate the definitions concretely.

3.3.1 Primitive AST Transformations

5 primitives

The AST transformations in most refactorings tend to be relatively straightforward. We hypothesize that most refactoring transformations can be described as a composition of the following five *primitive operations:*

- **Add (α).** A new subtree is inserted into the AST.
- **Eliminate (ϵ).** A subtree is deleted from the AST.
- **Replace (ρ).** A subtree of the AST is replaced with a different subtree.
- **Move (μ).** A subtree of the AST is deleted, and re-inserted at a different location.
- **Copy (κ).** A subtree of the AST is copied to a different location.

3.3.2 Analysis Preservation Across Primitive Transformations

Analysis preservation = edge preservation

When we transform an AST—i.e., when we add a node, replace a node, etc.—we want to be able to specify which program analyses should be preserved. In the context of a program graph, this means specifying which types of extra edges should (or should not) be present in a program graph before and after the transformation.

Def. boundary-crossing edge

Our definition of edge preservation will be based on the notion of a boundary-crossing edge. Given a subtree T in the AST underlying a program graph, an extra

[2]Such a name is admittedly over-optimistic but captures the essential idea that it can serve as the fundamental precondition checking mechanism for many refactorings in many languages—even refactorings and languages that have not yet been invented.

edge in the program graph is *boundary-crossing relative to T* iff either (1) it has a source in T and a sink outside T, or (2) it has a sink in T and a source outside T.

Def. e-preservation

Now suppose we want to preserve edges of type e. We will let T and T' denote the abstract syntax tree before and after transformation, respectively. We will say that each primitive transformation is *e-preserving* iff the following hold:

- **Add (α).** There are no boundary-crossing edges of type e relative to the added subtree in T'.
- **Eliminate (ϵ).** There are no boundary-crossing edges of type e relative to the deleted subtree in T.
- **Replace (ρ), Move (μ), Copy (κ).** The boundary-crossing edges of type e relative to the replaced subtree in T are in 1–1 correspondence with the boundary-crossing edges of type e in the replacement subtree in T'.

Intuition

Intuitively, we are trying to capture the idea that, if we want to substitute one subtree with a different subtree that supposedly does the same thing, then all of the edges that extended into the old subtree should also extend into the new subtree, and all of the edges that emanated from the old subtree should also emanate from the new subtree. In a sense, the old subtree and the new subtree should interface with their surroundings in the same way.

Approximation

Note that this is an *approximation* of semantic preservation. Our definition places no constraints on edges that are entirely contained inside the affected subtree. Moreover, for ρ, μ, and κ, the boundary-crossing edges must be in 1–1 correspondence, but the definition places no constraints on *what* nodes in the subtree those edges point to.

3.3.3 Example: Extract Method

Example: Figs. 7–10

To provide a concrete illustration, we will use an example Java program based on the second Extract Method benchmark from the C2 Wiki [96]. The Java program and a corresponding program graph are shown in Figure 7; the underlying abstract syntax tree is shown in outline (tabular) form to conserve space. Suppose we want to apply the Extract Method refactoring to the two statements "`i++; field++;`." The code that should result after the refactoring has been applied is shown in Figure 8 along with its corresponding program graph. Note that the local variable i must be passed and returned in order to preserve the meaning of the original program.

Flow-preserving ρ-operation

As discussed earlier, the most important preconditions in Extract Method ensure the preservation of control flow and du-chains when the original statements are replaced with the method call. In our example, this is a Replace (ρ) operation that should preserve control flow and def-use edges in the program graph.

Boundary-crossing edges adjusted

Figure 9 shows the original program from Figure 7 again, but the statements/subtree to be extracted have been circled and shaded. The boundary-crossing edges have been adjusted: The circled subtree is treated as an endpoint, rather than particular nodes within that tree, and all of the edges entirely contained within the subtree have been removed. (This will make it easier to check for a 1–1 correspondence between boundary-crossing edges.) Furthermore, we have only marked control-flow and def-use edges since those are the only edges we are interested in preserving. Thus, the only edges shown in the figure are those that must be preserved across the ρ-operation. Likewise, Figure 10 shows the refactored program from Figure 8 again, but the replacement statement/subtree has been circled and shaded, the boundary-crossing edges have been adjusted, and only control-flow and def-use edges have been marked.

1–1 = ρ-preservation

Observe that the extra edges in the program graphs in Figures 9 and 10 are in 1–1 correspondence, meeting our definition of edge preservation across a ρ-operation.

Thus, this operation “preserved meaning” as intended, and so the refactoring can be allowed to commit.

Erroneous conditions

One can also verify that this test would have caught several erroneous conditions. For example, suppose the extracted statements contained a `return` statement; then the control flow edge from that statement would have been omitted from the refactored program’s program graph. Or suppose the refactoring was buggy and did not assign the variable *i*; then the def-use edges would not have been in 1–1 correspondence.

3.3.4 Operation of a Universal Precondition Checker

Compilability & preservation

Recall that most preconditions are intended to ensure *compilability* and *analysis preservation.* Testing for edge preservation in a program graph covers the latter, but by incorporating this into an appropriate architectural design, a test for the former comes almost “for free.”

Front-end checks verify compilability

A refactoring tool generally performs the same types of front-end checks that a compiler would. For example, if a program contained two subroutines with the same name, this would be detected in the process of constructing name-binding edges in a program graph. Or if a `goto` statement referenced a nonexistent label, this would be detected while constructing control flow edges. This can be used to our advantage: By re-running these front-end analyses on the refactored program, we can use these checks to verify that the refactored program will still compile.

Operation of a universal engine

By combining this idea with the idea of edge preservation in a program graph, we can propose the following (unorthodox) operation for a refactoring engine. (If the source code cannot be parsed in Step 1 or 4, or if there are errors detected during the static analysis in Step 2 or 5, then an error must be reported to the user, and the refactoring cannot proceed.)

1. A parser parses the source code, producing an *initial AST.*
2. Static analyses are performed on the initial AST, producing a model of the program’s semantics (the *initial model).* Our algorithm will assume that this model is a program graph.
3. A sequence of primitive operations are performed, transforming the initial AST into a *modified AST.*
4. Source code is generated from the modified AST, and this source code is re-parsed to produce the *derivative AST.*
5. The derivative AST is statically analyzed, producing a *derivative model* (program graph).
6. The initial and derivative model are tested for equivalence modulo the sequence of primitive operations performed on the initial AST. In our case, this means testing for the preservation of certain types of edges in the program graphs. If the models (program graphs) are found to be equivalent, the transformation preserved meaning, and the refactoring can commit.

Fig. 11

This process is illustrated in Figure 11. The cornerstone of an efficient implementation is a fast algorithm for performing Step 6: testing the semantic models for equivalence. When the semantic model is a program graph, we will propose that this can be done by rewriting the initial and derivative program graphs to a normal form which can be quickly tested for equality.

3.3.5 Normalization

Illustrated in §3.3.3

The process of normalizing a program graph was already illustrated in our Extract Method example: All of the edges that are entirely contained in the affected subtree are removed, and boundary-crossing edges are adjusted so that they do not have endpoints inside the subtree but rather the subtree is treated as an endpoint in itself.

Leaves common AST nodes and boundary-crossing edges

This procedure can be applied to both the initial and derivative program graphs, and the two normalized program graphs should be isomorphic in a very predictable way. By collapsing the affected subtrees into a single node and eliminating edges within that node, the remaining nodes and edges of the resulting graphs are exactly the common AST nodes and boundary-crossing edges, respectively.

3.3.6 Equality Testing

Textual span

To turn the idea of normalization into an algorithm, we can capitalize on the observation that every node in an AST can be described as an interval denoting the textual span of its tokens in the program text. In other words, every AST node can be described by an interval describing a starting and ending textual offset. The root of an AST would have a textual span encompassing the entire program text; a node representing a function would encompass everything from its first token (or perhaps a leading comment) through the end of the function; and an identifier token in the AST would only have the textual span of that token's text. Note that, with this representation, the textual span of a subtree will always be a subinterval of the textual span of its parent node in the AST.

Primitive operations change intervals predictably

Now, consider how each of the primitive operations affects AST nodes' intervals.

- **Add (α).** If n characters are added at offset i, then all AST nodes with a textual offset beyond i will be moved n characters later.
- **Eliminate (ϵ).** If n characters are removed beginning at offset i, then all AST nodes with a textual offset beyond $n + i$ will be moved n characters earlier.
- **Replace (ρ), Move (μ), Copy (κ).** Similar.

In other words, by tracking what primitive operations are performed, we can use the textual span of a node in the initial AST to predict the textual span of the corresponding node in the derivative AST.

Endpoints as textual intervals

Since AST nodes serve as the endpoints of the "extra edges" in a program graph, and AST nodes can be represented as (textual) intervals, the edges in a program graph can similarly be represented as an ordered triple containing

- the source node's interval,
- the sink node's interval, and
- the edge's type.

Then we can use this ability to predict textual offsets in the derivative AST to predict, for each edge in the initial program graph, what the corresponding edge in the derivative program graph will be.

Equivalence checking algorithm

In the end, this gives rise to the following procedure for checking two program graphs for equivalence modulo a sequence of primitive operations.

1. Represent the initial and derivative program graphs as a list of ordered triples, one triple per edge, using textual spans to denote the endpoints.
2. Normalize the initial program graph, forming a new list of triples, and use the sequence of primitive operations to store the *expected* endpoints in the derivative AST rather than the actual endpoints in the initial AST.

3. Normalize the derivative program graph, forming a new list of triples.
4. If the two lists of triples are equal (that is, they contain exactly the same ordered triples), then the program graphs are equivalent.

```
class Test2 {
  int field = 0;
  void fun() {
    int i = 0;
    i++;
    field++;
    System.out.println(i);
  }
}
```

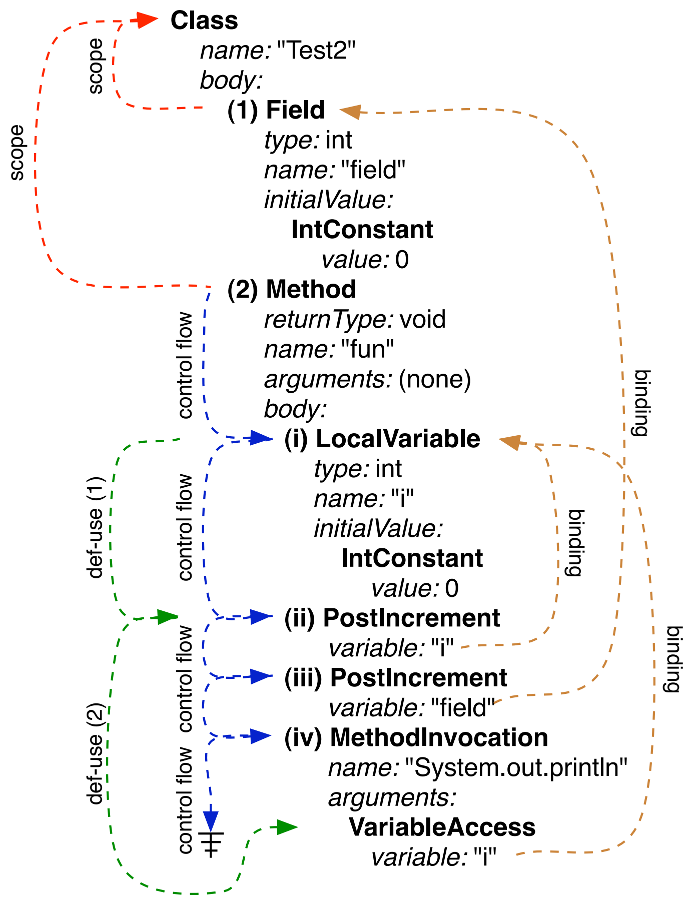


Figure 7: Example Java program and corresponding program graph

```
class Test2 {
  int field = 0;
  void fun() {
    int i = 0;
    i = newMethod(i);
    System.out.println(i);
  }
  i newMethod(int i) {
    i++;
    field++;
    return i;
  }
}
```

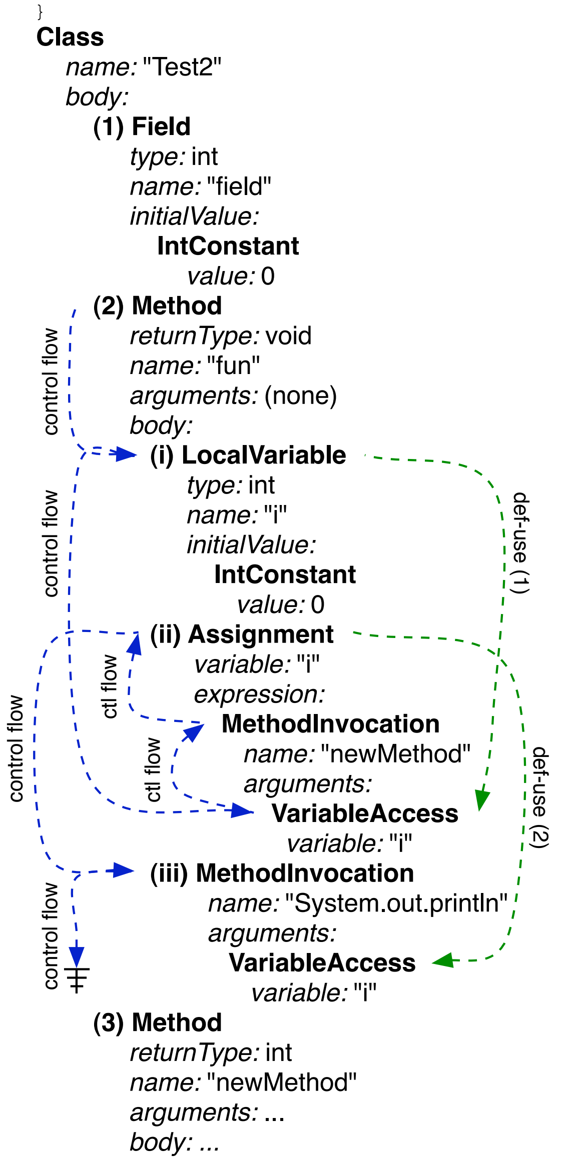
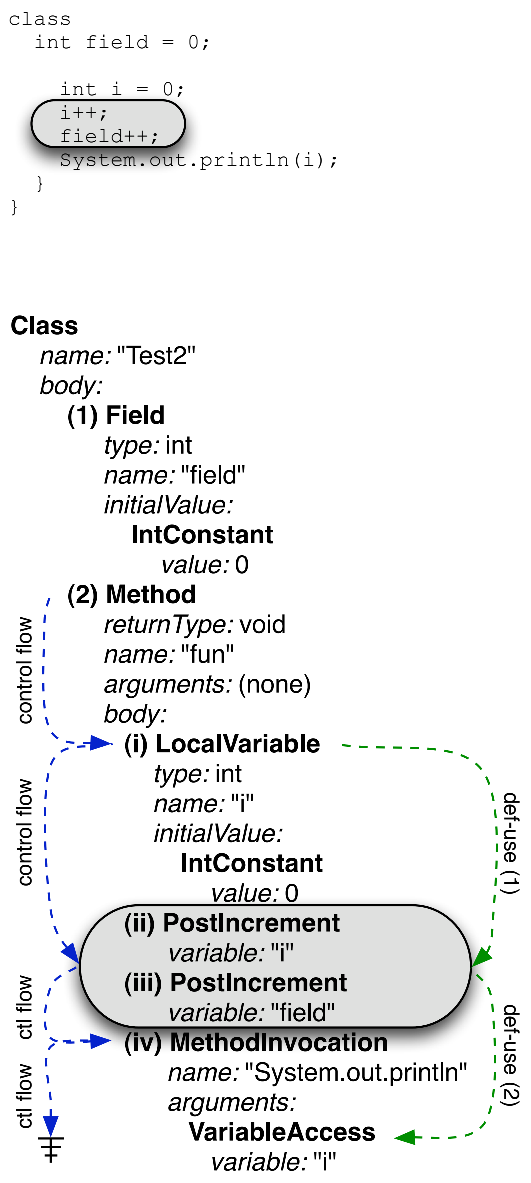


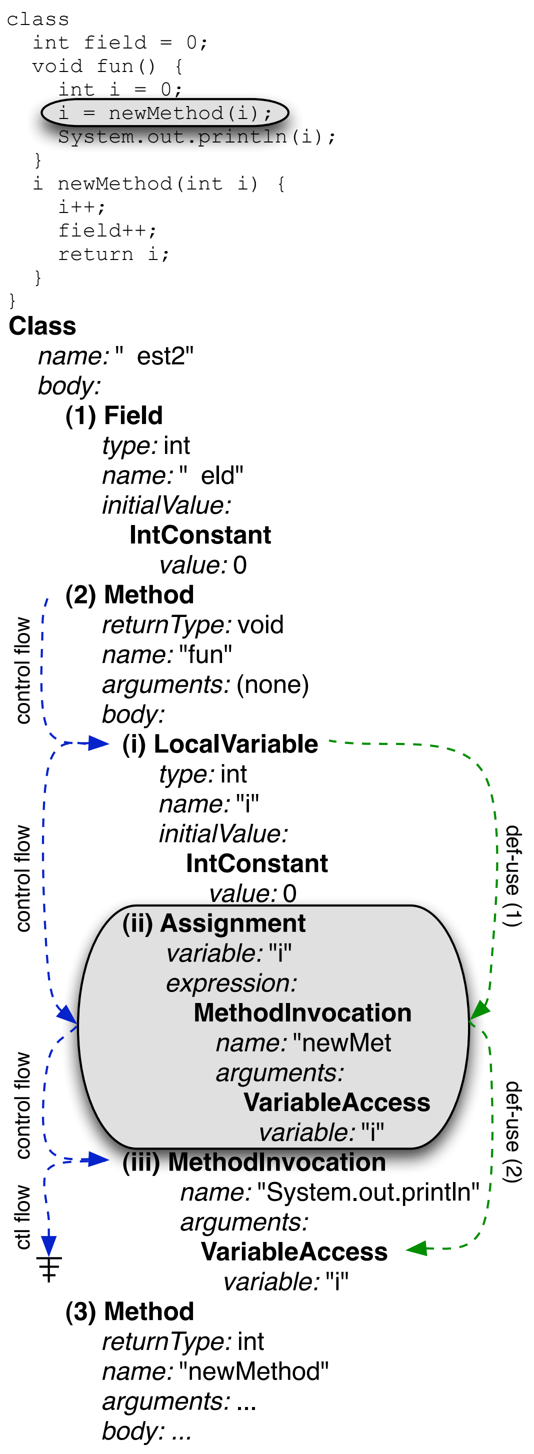
Figure 8: Program graph from Figure 7 after applying the Extract Method refactoring to the two post-increment statements. Scope and binding edges have been omitted.



```
class
  int field = 0;

    int i = 0;
    i++;
    field++;
    System.out.println(i);
  }
}
```

Figure 9: Program graph from Figure 7, normalized prior to applying the Extract Method refactoring to the two post-increment statements



```
class
  int field = 0;
  void fun() {
    int i = 0;
    i = newMethod(i);
    System.out.println(i);
  }
  i newMethod(int i) {
    i++;
    field++;
    return i;
  }
}
```

Figure 10: Program graph from Figure 8, normalized after applying the Extract Method refactoring to the two post-increment statements

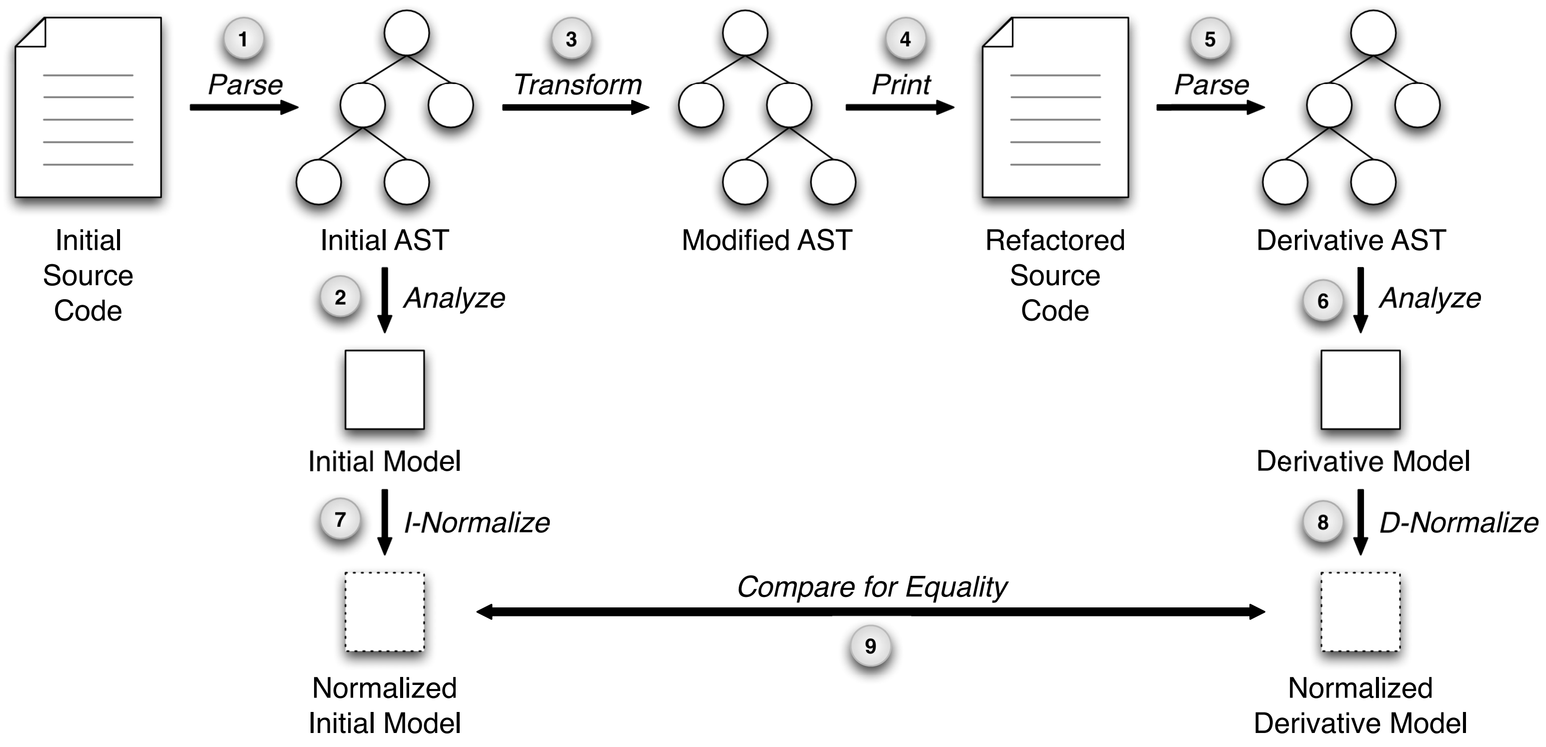


Figure 11: Operation of a normalization-based universal refactoring engine

4. Language-Agnostic C Pseudo-Preprocessing

The final part of the thesis will make three contributions:

Three contributions

1. An algorithm for language-agnostic conditional completion in LR(1) parsers
2. Mechanisms for integrating pseudo-preprocessing with generated ASTs
3. A strategy for representing preprocessor directives in a program graph and including them in universal precondition checking

Previous work

Our initial paper on generating rewritable ASTs [91] also lays the foundations for representing C preprocessor directives in the AST, starting from earlier work by Garrido [48]. A subsequent paper [92] defines the notions of *token-whitetext affixes* and *substitution tokens* as a means of incorporating syntactic information about single-configuration preprocessing directives into ASTs for arbitrary languages.

Conditional compilation problem

As alluded to in Section 1, the biggest challenge in refactoring preprocessed code involves conditional compilation. In an example such as the following, a Rename refactoring would need to modify the *variable* references in both arms of the conditional directive, but it is not immediately obvious how to parse this—much less represent or refactor it.

```
if (
 #if defined(A) || defined(B)
 variable
 #else
 function() < 1 && variable
 #endif
 < 2) x = 3;
```

Garrido's technique

Garrido's [48] technique for handling conditional compilation involves modifying the grammar to specify where conditionals may appear and then inserting a pass between lexical analysis and parsing which ensures that conditional directives appear only at these locations: If a conditional directive appears at an unexpected location, it is moved forward or backward as necessary, and tokens are copied into each branch of the conditional. This process is called *completing the conditional.* Although reasonable, it is language-specific, heuristic, and becomes complicated in the presence of nested conditionals. It also requires modifying the grammar and adding AST nodes for conditional compilation directives.

Solution based on GLR, Celentano

Our solution for representing multiple-configuration sources, which is intended to be language-agnostic and grammar-independent, is based on concepts underlying Generalized LR (GLR) parsing [107, p. 5], an extension of the usual LR parsing algorithm [69, 33] which can handle ambiguities in a language, as well as some algorithmic ideas from Celentano's method for incremental LR parsing [25, 9, 90].

4.1 Representing Conditionals

AST should capture all configurations

Conceptually, conditional compilation allows for variation in an AST depending on the preprocessor's *configuration,* that is, the set of macro definitions under which it is operating: The structure of the AST under one configuration is not necessarily the same as for another. Our solution is to construct a single AST which captures *all* of the AST variations that may occur under *all* feasible configurations—just as GLR parsers can use a single parse tree to represent several ambiguous interpretations of a phrase.

Fig. 12

Figures 12(a) and (b) show the individual ASTs that would be constructed for the preceding example under each preprocessor configuration. Notice that the smallest subtree that differs between the two is the expression under the if-statement node.

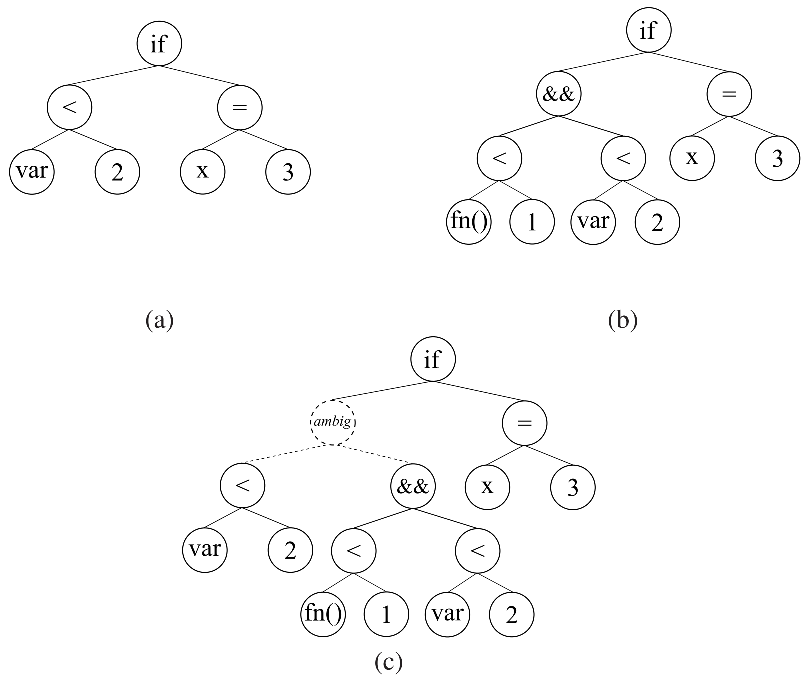


Figure 12: (a) The AST for the configuration `defined(A) || defined(B)`. (b) The AST for the configuration `!(defined(A) || defined(B))`. (c) The multiple-configuration AST. The dotted node represents an "ambiguous" expression whose children are each guarded by a different preprocessor configuration.

Figure 12(c) shows a "multiple-configuration" AST which combines the previous ASTs by inserting an "ambiguous" expression node whose children are the individual expression nodes guarded by the various preprocessor configurations.

4.2 Parsing Conditionals

Construction in LR(1) parsers

This representation can be constructed by modifying a deterministic LR(1) parser. The following method is oversimplified—its positioning of ambiguous nodes in the AST is often less than ideal—but it nevertheless conveys the gist of our technique.[3] The parser proceeds as usual until it reaches an `#ifdef` directive in the token stream. The parser clones itself so that a different copy of the parser can process each branch of the `#ifdef`. Each clone independently processes the tokens under its branch of the conditional, stopping when it is about to shift the token following the `#endif`. The clones are then compared for *equivalence:* If two or more clones are in the same state and have equivalent stack contents, those clones are merged into a single parser.[4] When parsers are merged, the topmost elements on their stacks are made to be children of an "ambiguous" node (as in Figure 12(c)), and this "ambiguous" node becomes the topmost element on the stack of the merged parser. After testing for equivalence and possibly merging parsers, each subsequent token is fed to all remaining clones, which shift the token and perform any reductions, stopping when they are prepared to shift the following token. The clones are again tested for equivalence, merged as necessary,

[3]It appears that the application of the same essential ideas to preprocessing was independently discovered and implemented in [94].

[4]Why is it safe to merge parsers under these conditions? In mathematical models of deterministic shift-reduce parsers, the transition relation between states is a *function* of the stack contents, the current state, and the remaining input: If all of these are identical among parsers, the parsers' subsequent behaviors will be identical.

and the cycle continues until only a single parser remains. In the worst case, this will happen at the end of the input.

4.3 Future Work

Although the multiple-configuration program representation and the simplified parsing algorithm described above have been published, more work remains to be done.

4.3.1 Enhancements to the Parsing Algorithm

Specify ambiguous nodes

First, the algorithm above assumes that *any* AST node can be made into an ambiguous node. This makes the structure of the resulting AST unpredictable, which makes program analyses and transformations difficult to write. One enhancement will be to refine this algorithm so that the user can *define* which AST nodes are allowed to be ambiguous (or, rather, which nonterminals in the grammar have ambiguous AST nodes associated with them).

Finer-grained parser merging

Second, testing for parser equivalence only prior to *shift* operations can place ambiguous nodes significantly higher in the AST than necessary. For example, if the last statement in a program is guarded by an `#ifdef`, the root of the AST will be made into an ambiguous node rather than just the last statement node: The statement will be reduced, but the entire program will also be reduced (creating the root AST node), before the end-of-input token is "shifted" (i.e., an *accept* action is taken). Allowing the user to specify "ambiguous" nonterminals can help here as well. If these can be ordered according to the containment relationship of the corresponding AST nodes—for example, expression < statement < function < program—then the parser can order the *reduce* operations among the parser clones to ensure that ambiguous nodes are placed as low as possible in the AST.

4.3.2 Universal Precondition Checking for Preprocessed Code

Finally, the dissertation will address how to integrate preprocessor directives into a program graph and how they can be integrated with a universal precondition checker. The basic strategy for including preprocessor directives in ASTs (generated or hand-written) is discussed in our SLE paper [91]; these can act as nodes in a program graph as well.

Guarded edges

Garrido [48] observed that handling multiple configurations means that symbol table entries must be "guarded" by a preprocessor configuration, since some declarations may exist only under certain configurations, or an entity with a particular name may have several possible types depending on the preprocessor configuration. The proposed dissertation will address how to handle such situations in a program graph representation. The obvious analog would be to associate such a "guard" with the edges in a program graph. A name could bind to several possible declarations according to the preprocessor configuration; there could be several possible control flows; du-chains could vary; etc.

Universal prepreprocessor preconditions

The dissertation will also address how to integrate such a representation with a universal precondition checker. Generally, refactoring preprocessed code means checking for two preconditions.

1. Nodes/tokens resulting from file inclusion or a macro expansion cannot be modified unless the included file or macro definition is modified. The latter can only be done if *every* occurrence of the correspoding node/token can be changed similarly.

2. Preprocessor directives cannot be reordered if it will change the meaning/interpretation of those directives. (For example, one can swap the order of two

method definitions in a C++ class, but not if that would move a macro use above its previous `#define`, or if it would move it out of an `#ifdef` it was in before.)

Uniform replacement

The simplest way to handle condition (1) is to simply not allow nodes resulting from an inclusion or expansion to be modified. It is not yet clear whether a program graph representation can handle the latter case—detecting if the changes propagated from modifying an included file or macro definition will be valid.

Preprocessor directive dependence graph (PDDG)

However, it appears that a program graph can certainly handle precondition (2): Detecting whether a refactoring that would otherwise be valid will reorder preprocessor directives in such a way that it would no longer be invalid. The strategy would be to construct a representation similar to a program dependence graph among preprocessor directives in the AST. The relationship between macro definitions and uses is analogous to a data dependence, and and there is a relationship analogous to a control dependence between conditional compilation directives and the macro uses and control lines inside each arm of the conditional. The usual theory from compiler optimization extends naturally: A reordering transformation is valid if it preserves dependences [65, §2.2.3].

5. Scope, Evaluation, and Plan of Action

As indicated earlier, the work on generating rewritable ASTs from annotated grammars is mostly complete. In contrast, universal precondition checking and our technique for handling conditional compilation in LR(1) parsers are largely untested ideas.

5.1 Three Refactoring Tools

As a platform for testing the proposed ideas, three Eclipse-based refactoring tools are being developed.

- **Photran, an IDE and refactoring tool for Fortran 2008.** Although it originated from UIUC, Photran is now an open-source project hosted by the Eclipse Foundation and a component of the Eclipse Parallel Tools Platform. It has an active user community; new releases generally receive about 20,000 downloads. The parser, AST, and syntactic rewriting infrastructure are all generated from an annotated grammar using the technique described in Section 2, and Photran currently uses a program graph-based representation that follows the model described in Section 3. The parser and AST have been extended by colleagues at Fujitsu Japan to support XPFortran [78, 86], and refactorings are currently being developed by several students at UIUC as well as by colleagues at Unijuí Universidade Regional (Brazil) and Universidad Nacional de la Plata (Argentina). Photran 5.0, planned for release during the 4th quarter of 2009, will contain about 15 refactorings.

- **A prototype refactoring tool for Lua.** Lua is a popular scripting language that is used in Adobe's Photoshop Lightroom and World of Warcraft and is "the leading scripting language in games" [77]. Using the technique in Section 2, an Eclipse-based refactoring tool for Lua with two small refactorings (Rename Local Variable and Interchange Loops) was developed in about 7.5 hours.

- **Ludwig, a lexer/parser/rewritable AST generator and refactoring tool for EBNF grammars.** Ludwig is the tool used to generate the parser, AST, and rewriting infrastructure in Photran, as well as for the Lua refactoring tool and for the refactorings in Ludwig itself.

The decision to refactor Fortran, Lua, and EBNF was based on the disparity of these languages. Fortran is a very large, statically-typed, compiled language. Lua is a very small, dynamically-typed scripting language. EBNF is a specification language that is almost nothing like either Fortran or Lua.

5.2 Universal Precondition Checking

Photran currently uses a program graph-based semantic representation, which has been isolated into a library we call the *Virtual Program Graph,* or VPG. It has been performance-tuned and tested on projects up to one million lines of code.

A universal precondition checker will be built into the VPG and used to build several refactorings for Fortran, Lua, and EBNF. Since many refactorings have already been built and tested (particularly in Photran) using traditional precondition checking, for the purpose of comparison, several of these refactorings will be re-implemented using the universal precondition checker. This will allow the size (in lines of code) and performance of the refactorings to be compared.

5.3 Language-Agnostic C Pseudo-Preprocessing

Photran 5.1 is scheduled to include single-configuration C prepreprocessor support in refactorings. This is being supported by modifying the C preprocessor in the Eclipse C/C++ Development Tools (CDT) to interface with Photran's lexer, allowing information about preprocessor directives to be added to its abstract syntax tree. This will provide a platform for testing many of our ideas for representing single-configuration preprocessed code.

Our technique for language-agnostic conditional completion and for representing multiple-configuration preprocessed code will be prototyped in the Lua refactoring tool.[5]

[5]Multiple-configuration preprocessing may or may not be integrated into Photran, for technical and logistical reasons.

Related Work

1. General

Terminology

Refactoring ≠ restructuring

"Refactoring" and "restructuring" often seem to be interchangable terms, although there is actually a subtle difference between them. Refactoring is a specific technique for performing restructuring which uses small-scale, behavior-preserving changes to achieve larger, behavior-preserving changes in software systems [45]. In contrast, restructuring could also be achieved through one massive change that does not preserve behavior in the interim.

Distinction is usage

That said, *individual* refactorings—the actual changes made—could also be called restructurings, because the two terms differ only in terms of the larger process. Thus, calling a restructuring tool a refactoring tool indicates a distinction only in the context in which it is expected to be used, not necessarily in what the tool does.

Distinction is not OO

A widely-cited survey of refactoring research [80] and a popular wiki [8] make an erroneous distinction between refactoring and restructuring, claiming that refactoring applies only to object-oriented systems. Although early work on refactoring happened to focus on object-oriented systems while work on restructuring at the same time did not (for example, contrast Opdyke's dissertation on "refactoring" with Griswold's on "restructuring"), that is not a defining distinction. This is bolstered by an abundance of more recent work on refactoring databases [13] and non-object-oriented languages [49, 79, 74].

History

Early work

The idea of program transformation has been around for almost as long as there have been programs to transform. Indeed, IBM's compiler for FORTRAN I (1959)—the first compiler for the first high-level language—performed common subexpression elimination, loop-independent code motion, and constant folding, among other transformations [11]. Behavior-preserving transformations at the source code level first gained interest during the 1970s, motivated initially by the desire to convert programs using `goto` statements into structured programs. This application led to the term *restructuring,* which took on a more generalized meaning[6] that followed it into the 1980s and 1990s.

Interactive GUIs

[6]A commonly-cited definition of *restructuring* appears in Chikofsky & Cross: "Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics). A restructuring transformation is often one of appearance, such as altering code to improve its structure in the traditional sense of structured design. [. . .] However, the term has a broader meaning that recognizes the application of similar transformations [. . .] in reshaping data models, design plans, and requirements structures." [26]

By that time, graphical user interfaces were becoming more widely available, and they proved to be a boon to restructuring tools. In contrast to the batch systems of the previous decades, the restructuring tools of this era were *interactive.*

Fortran parallelization

This proved particularly beneficial to researchers working on parallelizing compilers, who were discovering that fully-automatic, coarse-grained parallelization could not rival the work of a competent human. In response, they built tools like PTOOL [12], $\mathbb{R}^n$ [28, 29, 23], ParaScope [16, 66, 64, 27, 55], Faust [54], and D [58], which integrated their compilers' dependence analyses and loop transformations into an interactive tool. While the tool could perform the transformations and (attempt to) verify their correctness, the choice of which transformations to apply could be left to the programmer.[7]

Griswold & Opdyke

While those researchers were building interactive, behavior-preserving, source-level program transformation tools for performance tuning, two other researchers established the idea these tools could be used for an entirely different purpose: They could be used to help programmers make design changes during software maintenance. Bill Griswold's 1991 Ph.D. thesis [53] identified several common transformations—including moving, renaming, inlining, and extracting program entities—and detailed their implementation, prototyping them in a restructuring tool for Scheme. Around the same time, Opdyke and Johnson introduced the term "refactoring" into the literature [88][8]; Bill Opdyke's 1992 dissertation [87] catalogued transformations for building object-oriented frameworks and prototyped them in a refactoring tool for C++. While the big-picture ideas were similar, Griswold's dissertation focused largely on tool implementation and guaranteeing correctness, while Opdyke's focused more on the catalog of refactorings.

Refactoring Browser

Shortly thereafter, Brant and Roberts began developing the Smalltalk Refactoring Browser [99], which lead to Roberts' dissertation [100]. The Refactoring Browser was destined to be more than a research prototype; it was intended to be a useful, production-grade tool. Unlike previous restructuring tools, the Refactoring Browser was *integrated* into the Smalltalk development environment, allowing refactoring to be seamlessly intermixed with coding. It quickly gained popularity, most notably among the developers at Tektronix who later developed eXtreme Programming (XP). XP became the first software process to advocate refactoring as a critical step.

Proliferation

The popularity of XP, coupled with the subsequent publication of Fowler's *Refactoring* [44] in 1999, brought "refactoring" into the software development parlance. The 2000s saw a proliferation of refactoring tools. Automated refactoring became available to Java programmers on a large scale in 2001, when they were included in the (heavily Smalltalk-influenced) Eclipse JDT and IntelliJ IDEA. They were subsequently added to other IDEs including Microsoft Visual Studio, Sun NetBeans, and Apple Xcode, and other languages have been supported in the form of refactoring plug-ins for Eclipse, NetBeans, Visual Studio, and even emacs and vi.

2. Refactoring Tools

Architecture, Design, & Implementation

Refactoring Browser

Although Griswold [53] prototyped an interactive restructuring tool for Scheme and Opdyke [87] prototyped a refactoring tool for a subset of C++, Brant and Roberts' work on the Smalltalk Refactoring Browser [99, 100] resulted in the first production system and arguably had the greatest influence on subsequent tools. Perhaps its most important contribution was its intensely practical orientation: The Refactoring Browser

[7]Subsequent interactive parallelization tools include SUIF Explorer [76] and GPE [21].

[8]. . . although Johnson freely admits that Kent Beck and others at Tektronix were using the term conversationally before then.

established the ideas that refactorings must (1) be integrated into standard development tools, (2) be fast (or else a programmer would "just do [them] by hand and live with the consequences"), (3) avoid purely automatic reorganization, and (4) be "reasonably correct" [99].

Speed in RB

Achieving reasonable speed was critical. Griswold's prototype used a program dependence graph to make extensive guarantees about the correctness of its transformations, but it was very slow, sometimes taking several minutes to perform a transformation. In contrast, the Refactoring Browser was usable in interactive time but used only lightweight, syntactic analyses. However, *it did not sacrifice safety.* Most of the time, the lightweight analyses were sufficient to guarantee the safety of the transformation; indeed, many semantic analyses could be computed syntactically, thanks to Smalltalk's small grammar. When a safe transformation could not be guaranteed (e.g., inlining a dynamically-dispatched method), rather than attempt an expensive analysis, the tool would simply ask the user for input; then the correctness of the transformation depended on the accuracy of the user's input. While the Refactoring Browser did not sacrifice safety, avoiding expensive analyses *did* result in a sacrifice in precision. For example, renaming the selector (i.e., method) *open* would result in both *File#open* and *Socket#open* being renamed, even if the two were distinguishable (e.g., through type inference). [19]

Rubicon

Of course, many refactoring tools have been developed since then, varying widely in their capabilities and robustness. One popular distinction was originally promoted by Martin Fowler [43]: A refactoring tool that correctly implements the Extract Method refactoring has "crossed refactoring's rubicon." Since that refactoring is tremendously difficult to implement correctly without a fairly sophisticated program representation (an AST, often with flow analysis), the existence of a good Extract Method refactoring provided an easy way to distinguish "serious" refactoring tools from those attempting to use regular expression matching or other ill-informed techniques for program analysis; tools lacking an adequate analysis and transformation infrastructure would inevitably be doomed to failure, since the types of refactorings that could be implemented and the reliability of these refactorings would generally be limited.

Production tools

Even after the infrastructure to "cross the rubicon" is in place, building a robust and scalable refactoring tool is difficult, and for that reason, studying the design decisions and engineering trade-offs in widely-used commercial tools is particularly valuable. Unfortunately, there is little publicly-available documentation on their internals. Much of what is available has emerged from the annual ACM Workshop on Refactoring Tools [34, 37, 35]. Kieżun, Fuhrer, and Keller [46] trace the refactoring capabilities in the Eclipse Java Development Tools from Eclipse 1.0 (2001) through Eclipse 3.3 (2007), briefly describing its infrastructure and the *participant* model, which allows behavior to be "plugged into" refactorings like Rename and Move (this would allow an XML file to be updated when a Java class is renamed, for example). Bečička, Hřebejk, and Zajac [17] give a high-level overview of the refactoring infrastructure in NetBeans, while Jemerov [61] gives a much more detailed description of the facilities available in IntelliJ IDEA.

Research tools

In contrast, the research literature contains ample information on prototype refactoring tools. Many of these tools were a first attempt to refactor a particular language. These include tools to refactor Haskell and Erlang [74, 75] as well as Ada 95 [15], and Oberon [39]. Others were built to prototype a particular aspect of refactoring tool design. Morgenthaler's Cstructure [82] introduced *virtual control flow,* which was later implemented in Apple Xcode [18]. Garrido's CRefactory [48] served as a platform for prototyping C preprocessor-aware refactorings. The JastAddJ compiler [38] was extended to prototype a Rename refactoring that was more reliable and more easily extensible than the ones available in production refactoring tools for Java [101].

DSLs for Implementing Refactorings

Motivation for DSL

Most refactoring tools use AST manipulations or AST-guided text manipulations to transform source code. This model is straightforward, and it is sufficient for expressing the transformations in most common refactorings (e.g., Rename, Extract Method, Move Method). However, there are certainly more concise ways to specify program analyses and transformations. And in some cases, imperative tree manipulation can become outright confusing. Consider, for example, the complexity of the tree matching and manipulation in a refactoring which applies the distributive property $a(b - c) = ab - ac$ to an integer expression.[9] In cases like this, the disparity between specification and implementation becomes very apparent, and this could be lessened by implementing the analysis and/or transformation in a domain-specific language.

JunGL

One such language is JunGL [109, 108], "a hybrid of a functional language in the style of ML and a logic query language" [108] which was specifically designed for scripting refactorings. JunGL allows the programmer to define, query, manipulate, and pretty print a program graph-like representation. AST nodes can be defined as algebraic data types, and pattern-matching functions can be defined over these types. A variant of Datalog is used to define the "extra edges" representing semantic information, as well as to form queries over the graph during the precondition checking stage of refactoring. Finally, the AST can be transformed through mutators on the AST nodes and the result prettyprinted back to source code.

RB Rewrite Engine

A very different approach was taken several years earlier in in the Smalltalk Refactoring Browser. Smalltalk has a surprisingly small syntax [14]—small enough that the Refactoring Browser's preconditions were built using only name binding information and syntactic pattern matching, with no additional program analyses [19]. Because of the the heavy reliance on pattern matching, its developers soon integrated a rule-based rewrite engine. Originally, "the syntax for this engine was too complex for normal users," [98], but it was later simplified, making it possible for users to define their own transformations. In this simplified version, the expressions on the left- and right-hand sides of the rewrite rules looked like ordinary Smalltalk expressions, except that they could also contain *pattern variables* for matching arbitary expressions (subtrees); these pattern variables could be prefixed with special characters to exert more fine-grained control over the matching procedure. [98]

Empirical Data on Refactoring

Goal: Focus effort based on usage

When creating a new refactoring tool, it is helpful to know what refactorings are the most likely to be used, so that resources can be devoted to developing refactorings that will likely have the most impact. Unfortunately, none of the existing studies of refactoring usage are based on a statistical sample, and all are limited to Java, so one should not expect the results reported to extrapolate to other languages. Nevertheless, they are informative and useful as a point of reference.

API studies

Two such studies are the following. Using four frameworks and one library as case studies, Dig and Johnson [36] looked at the evolution of Java APIs, and, based on a manual inspection of API changes, concluded that 80% of the breaking API changes were refactorings. Counsell et al. [31] considered seven Java projects as case studies and used a tool to attempt to detect refactorings that occurred between versions.

Problems with these

Even disregarding the fact that their data is based on a very small number of case studies, these results are not particularly useful to someone building a refactoring tool. There are two reasons. First, their results indicate only what API changes *could have been* automated refactorings, according to the authors' discernment; they

[9]This is complicated by the fact that a, b, and c may be arbitrary expressions (as opposed to variables or constants), the computation of a is duplicated on the right-hand side, and the associativity and precedence of the expressions on the left- and right-hand sides of the equality are different.

Refactoring	Uses	Percentage
Rename	179,871	74.8%
Extract Local Variable	13,523	5.6%
Move	13,208	5.5%
Extract Method	10,581	4.4%
Change Method Signature	4,764	2.0%
Inline	4,102	1.7%
Extract Constant	3,363	1.4%
(16 Other Refactorings)	10,924	4.5%

Table 1: Usage of automated Java refactorings in Eclipse by approximately 13,000 developers, according to Murphy-Hill et al. [85, Table 1]. Refactoring commands were used a total of 240,336 times; the third column gives the percentage of uses relative to that number. The 16 refactorings aggregated in the last row each comprised less than 1% of total uses.

do not indicate which ones (if any) were actually performed by a tool. Perhaps more importantly, these results are limited to observable, API-level changes, so any refactorings that do not fall into that category (e.g., Extract Method, Extract Local Variable) are necessarily excluded. For these reasons, we will look to other sources to determine what the "most important" refactorings are for a refactoring tool.

Studies on JDT usage

Somewhat more useful data are provided by Murphy et al. [84] and Murphy-Hill et al. [85]. Murphy et al. used the Mylar Monitor to collect data from 41 Java developers on the features used in the Eclipse IDE. Murphy-Hill et al. compare this data set with two others, including a publicly-available data set available from the Eclipse Foundation [7]. Eclipse 3.4 included the Usage Data Collector (UDC), a monitor that logged all of the commands executed by a user; this data set consists of these logs from more than 13,000 developers who consented to sending this information to the Eclipse Foundation.

Cf. Tbl. 1

Although it is a convenience sample, the UDC data set is by far the largest and probably gives the best indication of what automated refactorings are most used by Java developers. It is summarized in Table 1. Most striking is the fact that Rename is used more than all of the other refactorings combined, both in the UDC data set and in Murphy's data set [85]. This is bolstered by Murphy's observation that almost 100% of the developers in her sample used this refactoring, while the next-most-frequently used refactorings (Move and Extract) were each used by fewer than 60% of developers, followed by Pull Up and Inline closer to 30%.

3. Language-Independent and Language-Parametric Tools

In compiler research

The proposed research focuses on exploiting commonality among the infrastructural components of a refactoring tool. In the domain of compilers, nearly every component has been implemented in a library, framework, or DSL/code generator at some point in time. There have been lexical analyzer generators [63, 73], parser generators [62], AST generators [110], symbol table generators [97], data flow analyzer generators [67], code generator generators [24], and optimizing transformation generators [104], as well as entire compiler generators [106] and frameworks such as SUIF [105] and LLVM [72]. Lexer and parser generators are discussed in most compiler textbooks, including Aho et al. [10]; Muchnick discusses code generator generators [83, Ch. 6] and data-flow analyzer generators [83, §8.13].

Complementary work

Because the proposed research focuses on refactoring-specific issues, it generally does not compete with this work but rather complements it. For example, the AST generation algorithm could be implemented on top of virtually any parser generator. One could generate rewritable ASTs from Zephyr specifications, perhaps by storing offsets/lengths in AST nodes rather than concretizing the AST. These ASTs would be integrated with the VPG. A generated symbol table infrastructure could be helpful in building name-binding edges in a VPG. And so forth.

Generic refactoring

While there is been a plethora of work on making language-parametric tools for constructing compilers, there has been almost no such work targeted specifically at refactoring tools. However, there have been a few isolated attempts to find commonality in refactorings across languages. Lämmel [71] suggests that refactorings have both language-independent and language-dependent components, and that it may be possible to implement some refactorings "generically," parameterizing the language-dependent parts according to the semantics of the underlying language. Garrido's verification of refactorings using a formal semantics [51] acknowledges this as well, suggesting that the language specifics could be replaced and the refactoring re-verified with a different language. Finally, Mens [81] suggests that graph rewriting rules are an appropriate facility for specifying refactorings, assuming a program graph representation is adequate for capturing the relevant semantics of the underlying language.

4. Program Representation

Abstract Syntax

Although there is an abundance of existing work on abstract syntax, the proposed work is distinguished by (1) its annotation-based approach for the definition of abstract syntax (as opposed to a more traditional declarative approach), (2) its focus on incorporating C preprocessor directives in syntax trees for arbitrary languages, and (3) its focus on practical issues, including layout retention, customizability, and exposure as an API.

The Zephyr abstract syntax description language [110] is essentially a declarative language for "tree-like data structures" and resembles declarations of algebraic data types (à la ML). Wile [111] advocates a particular grammar notation (WBNF) which makes some aspects of abstract syntax (e.g., iteration, optionality, precedence, and nesting) more explicit and suggests a heuristic process by which Yacc grammars can be converted and abstract syntax derived automatically. Among existing tools, ANTLR [1, 93], LPG [5], and JavaCC/JJTree [3] all include AST generators; LPG's is derived directly from the concrete syntax, while ANTLR and JJTree rely on declarative specifications embedded in the grammar.

The proposed approach differs in many ways from all of these. For example, Zephyr is not integrated with a parser generator; ANTLR's ASTs allow a limited form of source rewriting [93, Ch. 9] but have a dramatically different structure than the ASTs we describe; and no existing AST generator can generate ASTs for C-preprocessed sources. However, the most prominent distinction of the present work is its annotation-based approach. Zephyr, ANTLR, and JJTree require the user to fully specify an abstract syntax. LPG constructs a primitive abstract syntax from the concrete syntax automatically. In contrast, our approach allows a *default* abstract syntax to be constructed from the concrete syntax and subsequently *refined* using annotations. This places less of an annotation burden on the programmer than requiring a fully explicit abstract syntax definition while simultaneously allowing more flexibility than a fully-inferred definition. One drawback of the annotation-based approach is that the structure of AST nodes is not immediately obvious, particularly when *(inline)* annotations are used; to remedy this, in Ludwig, we are developing a GUI which allows the user to view AST node structures as the grammar is being annotated.

Our heuristic for attaching whitetext to tokens in order to facilitate rewriting also appears to be new, although the fundamental idea of including whitetext in syntax trees appears elsewhere. Sellink and Verhoef [102] suggest placing whitespace and comments into a parse tree by (automatically) modifying the grammar to include a "layout" nonterminal prior to each occurrence of a token; Kort and Lämmel [70] include such information as annotations in the parse tree. Sommerlad [103], who built refactoring support into several IDEs not originally designed to support refactoring, provides an algorithm by which comments can be associating with AST nodes during refactoring, assuming they have been collected into a separate data structure by the lexer and assuming the comments and AST nodes have position information associated with them.

Semantic Representations

The program dependence graph, or PDG [41, 42] has been a classical program representation in research work on software development environments. Ottenstein and Ottenstein [89] first recognized in 1984 that a PDG could be used as a common program representation among the various components (compiler, debugger, etc.) in such a tool. They noted that a PDG can be updated incrementally and cited its advantages in program slicing (for debugging) and optimization/code generation, as well as its use in computing code complexity metrics.

The *program summary graph* [22] was used in Rice's PTOOL parallelization environment [12]. It is, in essence, a call graph extended with sufficient information to determine what global variables and passed-by-reference variables could be defined at a call site.

The Unified Interprocedural Graph (UIG) [56] was proposed in the context of a maintenance environment for C programs. It combined a program summary graph, call graph, interprocedural flow graph [57], and system dependence graph [59] (an interprocedural variant of a PDG) into a single representation which could provide data and control flow as well as data and control dependence information.

Griswold's prototype restructurer for Scheme [53] used PDGs as its underlying semantic representation. Because a PDG does not capture the scoping of name bindings, he augmented the PDG with *contours;* this allowed him to detect when a transformation would change name bindings. Because the AST and PDG were separate program representations, a great deal of effort was required to reconcile the two.

One obvious way to avoid mapping between separate program representations, as Griswold did, is to combine them into a single representation. For a source-level restructuring tool, this would mean combining the syntactic and semantic representations into a single data structure—i.e., annotating the AST, and/or superimposing one or more graph representations on its nodes. This idea does not appear to be attributable to any one individual, as variations on it appear throughout the literature. For example, Morgenthaler [82] describes a means by which the nodes in an AST can compute which other AST nodes serve as control flow successors and predecessors *(virtual control flow)*, avoiding the need to maintain a separate control flow graph. The idea of forming a graph structure from an AST is taken somewhat more literally by Mens [81], who uses such graphs to specify refactorings using graph rewriting rules. And JunGL [109, 108] uses an AST with a graph structure superimposed as its basic program representation.

5. C Preprocessor Support

Our work is also the first to treat language-independent handling of C-preprocessed code at length. An empirical confirmation of the importance of the C preprocessor is given by Ernst et al. [40]. The conditional completion problem is defined by Garrido [48]. The idea of modifying an LALR(1) parser appears to have been developed independently in [94], although the discussion is brief and is limited to the C language. Garrido [49, 50, 48] treats refactoring C-preprocessed code in detail; McCloskey and Brewer [79] take a different approach, requiring a semi-automated replacement of C preprocessor directives with a syntactically embedded macro language.

Appendices

A

Tentative Thesis Outline

0. Background

   History. Terminology. Notation.

1. Introduction

   Problems being addressed. State of the practice. Architecture of refactoring engines. Contributions.

Part I Generating Rewritable Abstract Syntax Trees

2. Annotating Grammars for the Generation of Rewritable ASTs

   Grammar annotations. AST concretization. Rewriting API. Empirical results. Related work.

3. An Algorithm for Constructing Rewritable ASTs

   Formalization of the construction. Soundness properties.

Part II Universal Refactoring

4. Universal Precondition Checking

   A posteriori checking. Program graphs. Primitive transformations. Program graph normalization. Textual span representation. Universal precondition checking algorithm. Related work.

5. The VPG Library

   VPG database schema. API. Empirical results.

Part III Language-Agnostic C Preprocessor Support

6. Single-Configuration C Preprocessor Support

   Prior work. Concretization of pseudo-preprocessed ASTs. Composition of preprocessors. Empirical results.

7. Multiple-Configuration C Preprocessor Support

   Language-agnostic LR(1) conditional completion. Proof of correctness. Source reproduction. Composition of preprocessors. Empirical results.

References

[1] ANTLR. `http://www.antlr.org/`.

[2] ASTGen: Automated testing of refactoring engines. `http://mir.cs.uiuc.edu/astgen/`.

[3] JJTree. `https://javacc.dev.java.net/doc/JJTree.html`.

[4] JUnit.org. `http://www.junit.org/`.

[5] LALR parser generator. `http://sourceforge.net/projects/lpg/`.

[6] TestOrrery. `http://testorrery.sourceforge.net/`.

[7] Usage data collector: Reports: Commands. `http://www.eclipse.org/org/usagedata/reports/data/commands.csv`.

[8] Program transformation wiki / Refactoring. `http://www.program-transformation.org/Transform/Refactoring`, 2004.

[9] Rekesh Agrawal and Keith D. Detro. An efficient incremental LR parser for grammars with epsilon productions. Acta Informatica, 19:369–376, 1983.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Prentice Hall, Upper Saddle River, NJ, 1986.

[11] F. E. Allen. A technological review of the FORTRAN I compiler. In AFIPS '82: Proceedings of the June 7-10, 1982, National Computer Conference, pages 805–809, New York, NY, USA, 1982. ACM.

[12] J. R. Allen, D. Baumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In Proceedings of the 1986 International Conference on Parallel Processing. IEEE Computer Society Press, August 1986.

[13] Scott W. Ambler and Pramodkumar J. Sadalage. Refactoring Databases: Evolutionary Database Design. Addison-Wesley Professional, 2006.

[14] American National Standards Institute. ANSI INCITS 319-1998 (R2007): Information Technology – Programming Languages – Smalltalk. 2007.

[15] Paul Anderson. A refactoring tool for Ada 95. In SIGAda '04: Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada, pages 23–28, New York, NY, USA, 2004. ACM.

[16] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The parascope editor: an interactive parallel programming tool. In Supercomputing '89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pages 540–550, New York, NY, USA, 1989. ACM.

[17] Jan Bečička, Petr Hřebejk, and Petr Zajac. Using Java 6 compiler as a refactoring and an analysis engine. In Proceedings of the 1st Workshop on Refactoring Tools (WRT'07), volume 57–58. Technical Report 2007-08, Technische Universität Berlin, 2007.

[18] Robert Bowdidge. Personal communication, 2007.

[19] John Brant. Personal communication, 2009.

[20] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. Pattern-Oriented Software Architecture: A System of Patterns, volume 1. John Wiley & Sons, 1996.

[21] C. R. Calidonna, M. Giordano, and M. Mango Furnari. A graphic parallelizing environment for user-compiler interaction. In ICS '99: Proceedings of the 13th International Conference on Supercomputing, pages 238–245, New York, NY, USA, 1999. ACM.

[22] D. Callahan. The program summary graph and flow-sensitive interprocedual data flow analysis. In PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, pages 47–56, New York, NY, USA, 1988. ACM.

[23] Alan Carle, Keith D. Cooper, Robert T. Hood, Ken Kennedy, and Linda Torczon. A practical environment for scientific programming. Computer, 20(11):75–89, 1987.

[24] Roderic G.G. Cattell. Code generation and machine descriptions. Technical Report CSL-79-8, Xerox Palo Alto Research Center, October 1979.

[25] Augusto Celentano. Incremental LR parsers. Acta Inf., 10:307–321, 1978.

[26] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. IEEE Software, 7(1):13–17, 1990.

[27] K.D. Cooper, M.W. Hall, R.T. Hood, K. Kennedy, K.S. McKinley, J.M. Mellor-Crummey, L. Torczon, and S.K. Warren. The parascope parallel programming environment. Proceedings of the IEEE, 81(2):244–263, Feb 1993.

[28] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization on the design of a software development environment. In Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, pages 107–116, New York, NY, USA, 1985. ACM.

[29] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. ACM Trans. Program. Lang. Syst., 8(4):491–523, 1986.

[30] Thomas Corbat, Lukas Felber, Mirko Stocker, and Peter Sommerlad. Ruby refactoring plug-in for eclipse. In OOPSLA '07: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 779–780, New York, NY, USA, 2007. ACM.

[31] S. Counsell, Y. Hassoun, G. Loizou, and R. Najjar. Common refactorings, a dependency graph and some code smells: an empirical study of java oss. In ISESE '06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, pages 288–296, New York, NY, USA, 2006. ACM.

[32] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 185–194, New York, NY, USA, 2007. ACM.

[33] Frank DeRemer. Practical Translators for LR(*k*) Languages. PhD thesis, M.I.T., Cambridge, MA, 1969.

[34] Danny Dig and Michael Cebulla, editors. Proceedings of the 1st Workshop on Refactoring Tools (WRT'07). Technical Report 2007-08, Technische Universität Berlin, 2007.

[35] Danny Dig, Robert M. Fuhrer, and Ralph Johnson. The 2nd workshop on refactoring tools (WRT'08). In OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 859–860, New York, NY, USA, 2008. ACM.

[36] Danny Dig and Ralph Johnson. How do apis evolve&quest; a story of refactoring: Research articles. J. Softw. Maint. Evol., 18(2):83–107, 2006.

[37] Danny Dig, Ralph Johnson, Frank Tip, Oege De Moor, Jan Becicka, William G. Griswold, and Markus Keller. Refactoring tools: Report on the 1st workshop WRT at ECOOP 2007. In M. Cebulla, editor, ECOOP 2007 Workshop Reader, volume 4906 of Lecture Notes in Computer Science, pages 193–202. Springer-Verlag Berlin Heidelberg, 2007.

[38] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 1–18, New York, NY, USA, 2007. ACM.

[39] Johannes J. Eloff. A software restructuring tool for Oberon. Master's thesis, University of Stellenbosch, 2001.

[40] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. IEEE Transactions on Software Engineering, 28(12):1146–1170, December 2002.

[41] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. Report RC-10208, IBM Research, August 1983.

[42] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9(3):319–349, 1987.

[43] Martin Fowler. Crossing refactoring's rubicon. `http://www.martinfowler.com/articles/refactoringRubicon.html`.

[44] Martin Fowler. Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[45] Martin Fowler. MF Bliki: RefactoringMalapropism. `http://martinfowler.com/bliki/RefactoringMalapropism.html`, January 2004.

[46] Robert M. Fuhrer, Markus Keller, and Adam Kieżun. Advanced refactoring in the Eclipse JDT: Past, present, and future. In Proceedings of the 1st Workshop on Refactoring Tools (WRT'07), pages 31–32. Technical Report 2007-08, Technische Universität Berlin, 2007.

[47] Kely Garcia. Testing the refactoring engine of the NetBeans IDE. Master's thesis, University of Illinois at Urbana-Champaign, 2007.

[48] Alejandra Garrido. Program refactoring in the presence of preprocessor directives. PhD thesis, Champaign, IL, USA, 2005.

[49] Alejandra Garrido and Ralph Johnson. Challenges of refactoring c programs. In IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution, pages 6–14, New York, NY, USA, 2002. ACM.

[50] Alejandra Garrido and Ralph Johnson. Refactoring C with conditional compilation. International Conference on Automated Software Engineering, 0:323, 2003.

[51] Alejandra Garrido and Jose Meseguer. Formal specification and verification of java refactorings. In SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, pages 165–174, Washington, DC, USA, 2006. IEEE Computer Society.

[52] Emanuel Graf, Guido Zgraggen, and Peter Sommerlad. Refactoring support for the C++ development tooling. In OOPSLA '07: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 781–782, New York, NY, USA, 2007. ACM.

[53] William G. Griswold. Program Restructuring as an Aid to Software Maintenance. PhD thesis, University of Washington, 1991.

[54] Jr. Guarna, V.A., D. Gannon, D. Jablonowski, A.D. Malony, and Y. Gaur. Faust: an integrated environment for parallel programming. Software, IEEE, 6(4):20–27, Jul 1989.

[55] Mary W. Hall, Timothy J. Harvey, Ken Kennedy, Nathaniel McIntosh, Kathryn S. McKinley, Jeffrey D. Oldham, Michael H. Paleczny, and Gerald Roth. Experiences using the ParaScope Editor: An interactive parallel programming tool. In PPOPP '93: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 33–43, New York, NY, USA, 1993. ACM.

[56] Mary Jean Harrold and Brian Malloy. A unified interprocedural program representation for a maintenance environment. IEEE Trans. Softw. Eng., 19(6):584–593, 1993.

[57] M.J. Harrold and M.L. Soffa. Computation of interprocedural definition and use dependencies. In International Conference on Computer Languages, 1990, pages 297–306, Mar 1990.

[58] Seema Hiranandani, Ken Kennedy, Chau-Wen Tseng, and Scott Warren. The d editor: a new interactive parallel programming tool. In Supercomputing '94: Proceedings of the 1994 Conference on Supercomputing, pages 733–ff., Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[59] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst., 12(1):26–60, 1990.

[60] Institut für software. http://ifs.hsr.ch/.

[61] Dmitry Jemerov. Implementing refactorings in IntelliJ IDEA. In Proceedings of the 2nd Workshop on Refactoring Tools. ACM, 2008.

[62] Stephen C. Johnson. Yacc – Yet another compiler-compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[63] Walter L. Johnson, James H. Porter, Stephanie I. Ackley, and Douglas T. Ross. Automatic generation of efficient lexical processors using finite state techniques. Commun. ACM, 11(12):805–813, 1968.

[64] K. Kennedy, K.S. McKinley, and C.W. Tseng. Interactive parallel programming using the ParaScope Editor. IEEE Transactions on Parallel and Distributed Systems, 2(3):329–341, 1991.

[65] Ken Kennedy and John R. Allen. Optimizing compilers for modern architectures: a dependence-based approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[66] Ken Kennedy, Kathryn S. McKinley, and Chau-Wen Tseng. Analysis and transformation in the parascope editor. In ICS '91: Proceedings of the 5th International Conference on Supercomputing, pages 433–447, New York, NY, USA, 1991. ACM.

[67] Gary A. Kildall. A unified approach to global program optimization. In POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 194–206, New York, NY, USA, 1973. ACM.

[68] Michael Klenk, Reto Kleeb, Martin Kempf, and Peter Sommerlad. Refactoring support for the groovy-eclipse plug-in. In OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 727–728, New York, NY, USA, 2008. ACM.

[69] Donald E. Knuth. On the translation of languages from left to right. Information and Control, 8:607–639, 1965.

[70] J. Kort and R. Lämmel. Parse-Tree Annotations Meet Re-Engineering Concerns. In Proc. Source Code Analysis and Manipulation (SCAM'03). IEEE Computer Society Press, September 2003. 10 pages.

[71] Ralf Lämmel. Towards generic refactoring. In RULE '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming, pages 15–28, New York, NY, USA, 2002. ACM.

[72] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar 2004.

[73] M. E. Lesk. Lex – a lexical analyzer generator. Computing Science 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[74] Huiqing Li and Simon Thompson. Tool support for refactoring functional programs. In PEPM '08: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pages 199–203, New York, NY, USA, 2008. ACM.

[75] Huiqing Li and Simon Thompson. Tool support for refactoring functional programs. In Proceedings of the 2nd Workshop on Refactoring Tools, 2008.

[76] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ghuloum, and Monica S. Lam. Suif explorer: an interactive and interprocedural parallelizer. In PPoPP '99: Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 37–48, New York, NY, USA, 1999. ACM.

[77] Lua: about. `http://www.lua.org/about.html`.

[78] Yuichi Matsuo. On emerging trends and future challenges in aerospace CFD using the CeNSS system of JAXA NS-III. International Conference on High Performance Computing and Grid in Asia Pacific Region, 0:388–395, 2004.

[79] Bill McCloskey and Eric Brewer. ASTEC: a new approach to refactoring C. In ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 21–30, New York, NY, USA, 2005. ACM.

[80] Tom Mens and Tom Tourwé. A survey of software refactoring. IEEE Trans. Softw. Eng., 30(2):126–139, 2004.

[81] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations: Research articles. J. Softw. Maint. Evol., 17(4):247–276, 2005.

[82] John David Morgenthaler. Static Analysis for a Software Transformation Tool. PhD thesis, University of California, San Diego, 1997.

[83] Steven S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[84] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? IEEE Softw., 23(4):76–83, 2006.

[85] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[86] 永井, 亨. XPFortran 入門. 情報連携基盤センターニュース, 5(2):129-168, 2006. (Nagai, Toru. An introduction to XPFortran.).

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

[88] William F. Opdyke and Ralph E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA). ACM, September 1990.

[89] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In SDE 1: Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pages 177–184, New York, NY, USA, 1984. ACM.

[90] Jeffrey L. Overbey. Practical, incremental, noncanonical parsing: Celentano's method and the generalized piecewise LR parsing algorithm. Master's thesis, University of Illinois at Urbana-Champaign, 2006.

[91] Jeffrey L. Overbey and Ralph E. Johnson. Generating rewritable abstract syntax trees. In Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers, volume 5452 of Lecture Notes in Computer Science, pages 114–133, Berlin, Heidelberg, 2009. Springer-Verlag.

[92] Jeffrey L. Overbey, Matthew D. Michelotti, and Ralph E. Johnson. Toward a language-agnostic, syntactic representation for preprocessed code. Submitted for publication.

[93] Terence Parr. The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[94] M. Platoff, M. Wagner, and J. Camaratta. An integrated program representation and toolkit for the maintenance of C an integrated program representation and toolkit for the maintenance of C programs. In Proc. Conf. Software Maint., pages 129–137, 1991.

[95] Chris Recoskie. Handling conditional compilation in CDT's core. `http://wiki.eclipse.org/images/b/b3/Handling_Conditional_Compilation_In_CDT%E2%80%99s_Core.pdf`, 2007.

[96] Refactoring benchmarks for extract method. `http://c2.com/cgi/wiki/wiki?RefactoringBenchmarksForExtractMethod`.

[97] Stephen P. Reiss. Generation of compiler symbol processing mechanisms from specifications. ACM Trans. Program. Lang. Syst., 5(2):127–163, 1983.

[98] Don Roberts and John Brant. Tools for making impossible changes – experiences with a tool for transforming large Smalltalk programs. IEE Proc.-Sw., 151(2):49–56, April 2004.

[99] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. Theor. Pract. Object Syst., 3(4):253–263, 1997.

[100] Donald Bradley Roberts. Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[101] Max Schäfer, Torbjörn Ekman, and Oege de Moor. Sound and extensible renaming for Java. In OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 277–294, New York, NY, USA, 2008. ACM.

[102] Alex Sellink and Chris Verhoef. Scaffolding for software renovation. In CSMR '00: Proceedings of the Conference on Software Maintenance and Reengineering, page 161, Washington, DC, USA, 2000. IEEE Computer Society.

[103] Peter Sommerlad, Guido Zgraggen, Thomas Corbat, and Lukas Felber. Retaining comments when refactoring code. In OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 653–662, New York, NY, USA, 2008. ACM.

[104] Steven W. K. Tjiang and John L. Hennessy. Sharlit—a tool for building optimizers. In PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, pages 82–93, New York, NY, USA, 1992. ACM.

[105] Steven W. K. Tjiang, Michael E. Wolf, Monica S. Lam, K. Pieper, and John L. Hennessy. Integrating scalar optimization and parallelization. In Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing, pages 137–151, London, UK, 1992. Springer-Verlag.

[106] Mads Tofte. Compiler generators: what they can do, what they might do, and what they will probably never do. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[107] Masaru Tomita, editor. Generalized LR Parsing. Springer, 1991.

[108] Mathieu Verbaere. A Language to Script Refactoring Transformations. PhD thesis, University of Oxford, 2008.

[109] Mathieu Verbaere, Arnaud Payement, and Oege de Moor. Scripting refactorings with jungl. In OOPSLA '06: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming, Systems, Languages, and Applications, pages 651–652, New York, NY, USA, 2006. ACM.

[110] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The zephyr abstract syntax description language. In DSL'97: Proceedings of the Conference on Domain-Specific Languages (DSL), 1997, pages 17–17, Berkeley, CA, USA, 1997. USENIX Association.

[111] David S. Wile. Abstract syntax from concrete syntax. In ICSE '97: Proceedings of the 19th International Conference on Software Engineering, pages 472–480, New York, NY, USA, 1997. ACM.

[112] Michael Joseph Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[113] Xrefactory. http://www.xref.sk/xrefactory/main.html.