

FACULTY OF ENGINEERING OF THE UNIVERSITY OF PORTO

# Interactive Decompilation

José Manuel Rios Fonseca

Graduated in Mechanical Engineering  
by the Faculty of Engineering of the University of Porto

Ph.D. in Mechanical Engineering  
by the University of Wales Swansea

Dissertation submitted in partial fulfillment of  
the requirements for the degree of  
Master of Informatics Engineering

Dissertation prepared under the supervision of  
Dr. Ademar Manuel Teixeira de Aguiar  
from the Department of Electrical and Computer Engineering  
of the Faculty of Engineering of the University of Porto  
and of  
Dr. João Alexandre Baptista Vieira Saraiva  
from the Department of Informatics  
of the University of Minho

Porto, August 2006



## Resumo

As técnicas de engenharia reversa em geral, e de descompilação de código máquina em particular, podem ser úteis para o desenvolvimento e manutenção de software. Este trabalho debruça-se na incorporação da interactividade no processo de descompilação, assim alargando a sua utilidade ao permitir a intervenção do utilizador aquando da descompilação para eliminar ambiguidades semânticas do código, organizar-lo, e melhorar a sua legibilidade. É definido um catálogo de *refactorings* de engenharia reversa para código C de baixo nível (quase Assembly), onde cada *refactoring* ajuda a tornar o código de baixo-nível incrementalmente mais inteligível; deste modo a aplicação combinada e sucessiva destes *refactorings* permite efectivamente transformar código de baixo nível em alto nível, preservando a sua semântica. Para validar e testar a aplicabilidade da abordagem da descompilação interactiva através dos *refactorings* definidos, foi concebida uma ferramenta interactiva que automatiza a aplicação destes *refactorings* – a ferramenta IDC.



## **Abstract**

Reverse engineering techniques in general, and machine code decompilation in particular, can be useful for software development and maintenance. This work focuses on the incorporation of human interactivity in the decompilation process, increasing its usefulness by allowing an user to provide the necessary input during decompilation to disambiguate code semantics, organize code, and improve its readability. A catalog of reverse engineering refactorings for low-level (near-Assembly) C code is defined, where each refactoring helps making the low-level code incrementally more intelligible; so the combined and successive application of these refactorings can effectively transform a low-level machine code to a higher-level code, while preserving its semantics. To validate and test the applicability of the interactive decompilation approach through the defined refactorings, an interactive tool to automate the application of these was developed – the IDC tool.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem . . . . .	1
1.2	Objective . . . . .	1
1.3	Proposed strategy . . . . .	1
1.4	Outline . . . . .	2
<b>2</b>	<b>Background and review</b>	<b>3</b>
2.1	Decompilation . . . . .	3
2.1.1	Definition . . . . .	3
2.1.2	Motivation . . . . .	3
2.1.3	Legal implications . . . . .	5
2.1.4	Feasibility . . . . .	5
2.1.5	State of existing reverse-engineering tools . . . . .	7
2.2	Program Transformation . . . . .	10
2.2.1	Program Representation . . . . .	10
2.2.2	Transformation Paradigms . . . . .	12
2.3	Refactoring . . . . .	13
2.3.1	Definition . . . . .	13
2.3.2	Refactoring and decompiling . . . . .	13
2.3.3	State of existing interactive refactoring tools . . . . .	13
2.4	Review of related work . . . . .	14
<b>3</b>	<b>Catalog of low-level refactorings</b>	<b>15</b>
3.1	Function prototyping . . . . .	18
3.1.1	Extract Function . . . . .	18
3.1.2	Set Function Return . . . . .	19
3.1.3	Add Function Argument . . . . .	20
3.2	Organizing data . . . . .	21
3.2.1	Extract Local Variable . . . . .	21
3.2.2	Inline Temp . . . . .	21
3.2.3	Split Temporary Variable . . . . .	22
3.2.4	Replace Magic Number with Symbolic Constant . . . . .	23
3.2.5	Replace Data Values with Record . . . . .	23
3.2.6	Replace Type . . . . .	24
3.2.7	Dead Code Elimination . . . . .	24
3.2.8	Rename Symbol . . . . .	25

3.2.9	Simplify Expression . . . . .	25
3.3	Structuring control flow . . . . .	27
3.3.1	Structure <i>If</i> Statement . . . . .	27
3.3.2	Structure <i>If-Else</i> Statement . . . . .	27
3.3.3	Structure <i>Do-While</i> Statement . . . . .	28
3.3.4	Structure Infinite Loop . . . . .	28
3.3.5	Structure Continue Statement . . . . .	29
3.3.6	Structure Break Statement . . . . .	29
3.3.7	Structure While Statement – Form I . . . . .	30
3.3.8	Structure While Statement – Form II . . . . .	30
3.3.9	Structure While Statement – Form III . . . . .	31
3.3.10	Inline Return Statement . . . . .	32
3.3.11	Consolidate Boolean <i>And</i> Expression . . . . .	32
3.4	Example . . . . .	33
<b>4</b>	<b>Design of the IDC tool</b>	<b>35</b>
4.1	Requirements . . . . .	35
4.2	Design decisions . . . . .	36
4.2.1	Programming language . . . . .	36
4.2.2	Program representation and transformation . . . . .	36
4.2.3	GUI toolkit . . . . .	39
4.3	Architecture . . . . .	39
4.3.1	Overall architecture . . . . .	39
4.3.2	Program representation . . . . .	40
4.3.3	Program transformation . . . . .	45
4.3.4	Machine instruction semantics . . . . .	52
4.3.5	Code pretty-printing . . . . .	53
4.3.6	Transforming ATerms into non-ATerms . . . . .	55
4.3.7	Refactoring . . . . .	56
4.3.8	User interface . . . . .	58
<b>5</b>	<b>The IDC tool</b>	<b>63</b>
5.1	About . . . . .	63
5.2	Features . . . . .	63
5.3	Availability . . . . .	63
5.4	Tutorial . . . . .	63
5.4.1	Main window . . . . .	64
5.4.2	Extract function prototype . . . . .	64
5.4.3	Dead code elimination . . . . .	66
5.4.4	Control flow simplification . . . . .	70
5.4.5	Data flow simplification . . . . .	70
5.4.6	Variable renaming . . . . .	71
5.4.7	Variable renaming . . . . .	71
5.5	Current limitations . . . . .	71
5.6	Extending the tool with new refactorings . . . . .	75



<b>6</b>	<b>Conclusions</b>	<b>79</b>
6.1	Contributions . . . . .	79
6.2	Directions for future work . . . . .	79
	<b>Bibliography</b>	<b>81</b>



# List of Figures

2.1	Compilation process . . . . .	3
2.2	Compilation example . . . . .	4
2.3	Decompilation process . . . . .	4
2.4	Binary translation . . . . .	6
2.5	dcc decompilation stages . . . . .	8
3.1	Decompilation as a sequence of refactorings . . . . .	16
4.1	Main modules of the IDC tool . . . . .	36
4.2	IR decision tree . . . . .	37
4.3	Package dependency diagram of the interactive decompilation tool . . . . .	41
4.4	Class diagram of the aterm package . . . . .	43
4.5	Transformation class diagram . . . . .	46
4.6	Transformation context class diagram . . . . .	47
4.7	Transformation combinators class diagram . . . . .	48
4.8	Example of transformation language . . . . .	51
4.9	Example of the Assembly loading and translation process . . . . .	54
4.10	Example of code pretty-printing via Box representation . . . . .	55
4.11	Refactoring class diagram . . . . .	56
4.12	Main activity diagram . . . . .	59
4.13	Model-View class diagram . . . . .	60
4.14	Path annotation for the pointing problem . . . . .	61
5.1	Tool main window . . . . .	64
5.2	Pretty-printed view of the Intermediate Representation . . . . .	65
5.3	Context sensitive refactoring menu . . . . .	66
5.4	Control Flow Graph view . . . . .	67
5.5	Term Inspector view . . . . .	68
5.6	Code after applying the Extract Function refactoring . . . . .	68
5.7	Specifying the return symbol for the Set Function Return refactoring . . . . .	69
5.8	Code after applying the Set Function Return refactoring . . . . .	69
5.9	Code after applying the Add Function Argument refactoring . . . . .	70
5.10	Code after applying the Dead Code Elimination refactoring . . . . .	71
5.11	CFG after applying the Dead Code Elimination refactoring . . . . .	72
5.12	Code after applying the control structuring refactorings . . . . .	73
5.13	Code after applying the Inline Temp refactoring . . . . .	73
5.14	Code after applying the Simplify Expression refactoring . . . . .	74

5.15 Final code after applying the Rename Symbol refactoring . . . . .	74
5.16 Side-by-side comparison of the example source code . . . . .	75

# List of Tables

3.1	Refactoring catalog . . . . .	17
4.1	IR schema . . . . .	44
4.2	Transformation combinators . . . . .	47
4.3	Schema of the Box representation . . . . .	54



# Listings

4.1	Making ATerms with the Python ATerm library . . . . .	40
4.2	Matching ATerms with the Python ATerm library . . . . .	42
4.3	Implementation (in Python) of the <code>Not</code> transformation combinator . . . . .	48
4.4	Implementation (in Python) of the <code>BottomUp</code> transformation traverser . . . . .	50
4.5	Python implementation of the <code>FoldR</code> transformation factory . . . . .	51
4.6	Mixing the transformation language in Python code . . . . .	52
4.7	SSL specification of the Intel IA-32 <code>ADDL</code> instruction . . . . .	53
4.8	Excerpt of the compiled instruction lookup table for the Intel Pentium . . . . .	53
4.9	ATerm walker example – Box language writer . . . . .	57
5.1	Example input Assembly code ( <code>factorial.s</code> ) . . . . .	65
5.2	Complete Rename Symbol refactoring example . . . . .	77





# Acronyms

<b>ASDL</b>	Abstract Syntax Description Language
<b>AST</b>	Abstract Syntax Tree
<b>ATerm</b>	Annotated Term
<b>CFG</b>	Control Flow Graph
<b>DAG</b>	Directed Acyclic Graph
<b>DOM</b>	Document Object Model
<b>GUI</b>	Graphical User Interface
<b>IDC</b>	Interactive Decompiler
<b>IDE</b>	Integrated Development Environment
<b>IR</b>	Intermediate Representation
<b>JIT</b>	Just In Time
<b>OOP</b>	Object Oriented Programming
<b>PDG</b>	Program Dependency Graph
<b>SSA</b>	Static Single Assignment
<b>SSL</b>	Semantics Specification Language
<b>UI</b>	User Interface
<b>XML</b>	Extensible Markup Language
<b>XOR</b>	Exclusive Or
<b>XSLT</b>	XSLT Transformations



# Chapter 1

## Introduction

Software development is a fast paced technology field where new computer hardware, programming languages, and a myriad of associated technologies are developed every year. Competition is fierce. Software development and hardware supplying companies can disappear as quickly as they appeared. Once developed, software can easily be duplicated, leading companies to protect their investment in development with patents and their know-how with non-disclosure agreements. Software can also be a carrier of malicious code such as viruses and Trojans.

Reverse engineering techniques can be of crucial importance in this aggressive software development world. Reverse engineering can be employed to port to new programming languages or hardware architectures, to maintain software from a disappeared vendor, to attest the violation of patents or business secrets, or to detect malicious code.

Decompilation is a particular reverse engineering technique that aims to produce code in a high-level language from machine code, i.e., the reverse process of compilation.

### 1.1 Problem

Despite advances in decompilation techniques and tools, intrinsic characteristics of the compilation process – ambiguity and information discard – limit the abstraction level and maintainability of code generated by automatic decompilation tools.

### 1.2 Objective

Human action can step in where automatic decompilation techniques falter, providing the necessary input to disambiguate code semantics, organize code, and improve readability.

The objective of this thesis is to integrate human interaction in the decompilation process in order to help improving the quality of the generated code.

### 1.3 Proposed strategy

The approach investigated in this work was:

- First, to define a set of transformations of low-level (near Assembly) code that aims at improving its structure, readability, semantics without changing its behavior. Such

transformations are also called refactorings.

- Second, to develop an interactive decompilation tool that assists the user in the task of reverse engineering Assembly code, by automating the application of the above mentioned transformations.

## 1.4 Outline

Chapter 2 introduces the main concepts behind decompilation, program transformation, and refactoring and reviews the existing tools and technologies for each of these topics.

Chapter 3 presents a catalog of reverse engineering refactorings. For each refactoring is described the motivation for usage, the application mechanics, and an illustrative example.

Chapter 4 details the design of the developed interactive decompilation tool – the IDC tool. It lists the main requirements, justifies the main design decisions, and describes the tool’s architecture.

Chapter 5 describes the usage of the IDC tool. It gives a brief tutorial on how to decompile a simple example program, summarizes the current features and limitations, and explains how to extend the tool with new refactorings.

Chapter 6 presents the main conclusions of this work together with some directions for future work.

## Chapter 2

# Background and review

This chapter introduces concepts that are relevant to this work and reviews other related work. The main notions behind decompilation, program transformation, and refactoring are given; and the existing tools and technologies for each of these topics are reviewed.

## 2.1 Decompilation

### 2.1.1 Definition

Compilation is the process whereby source code written in a high-level language (such as C or C++) is translated into a low-level language, typically the Assembly language, which is near to the machine code (fig. 2.1 and 2.2).

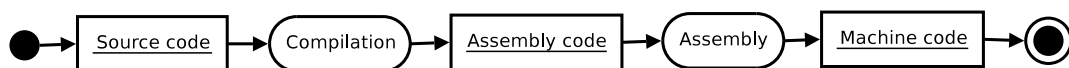


Figure 2.1: Compilation process

Decompilation is the reverse process of compilation (fig. 2.3). It aims to translate machine or Assembly code into a higher-level language [1].

Decompilation creates a representation of the input program at a higher level of abstraction. It is, therefore, a reverse engineering technique [2].

### 2.1.2 Motivation

The motivations for using reverse engineering techniques, such as decompilation, typically fall into two categories: software maintenance and security [1, sec. 1.6].

As a comprehension aid, the decompilation of a program can be used to:

- revise the binary code, such as when:
  - detecting vulnerabilities,
  - detecting malicious code,
  - or verifying that a machine code matches the supplied source code;
- learn an algorithm;

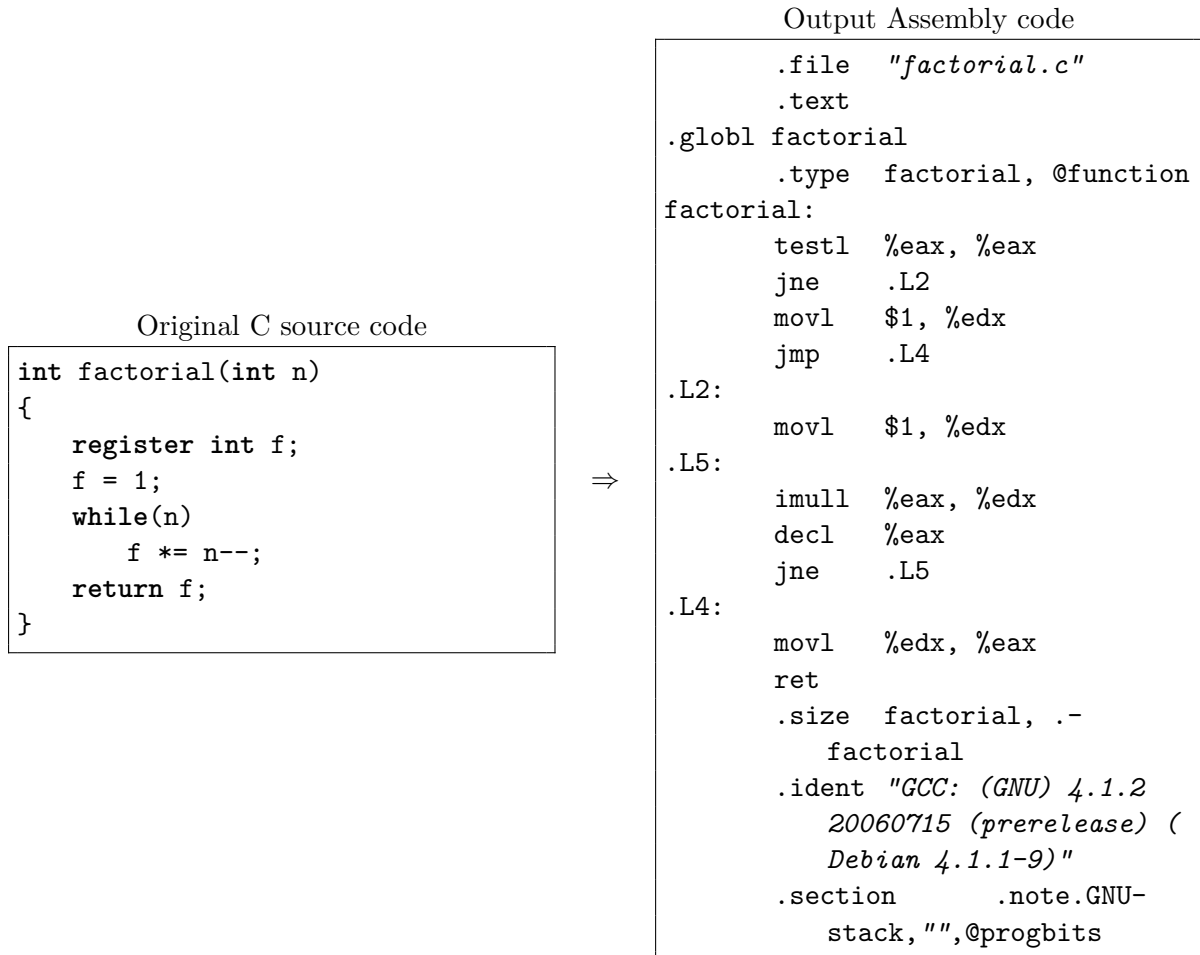


Figure 2.2: Compilation example

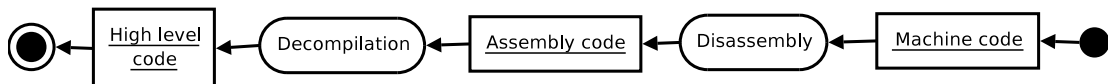


Figure 2.3: Decompilation process

- improve interoperability.

With little or no human intervention, decompilation can be also used to:

- optimize to a new platform, e.g., recompile old code for the Intel 80386 processor in order to optimize it for the Pentium 4 processor;
- port to another platform where the same set of system libraries is available, e.g., recompile a binary for the Intel version of Windows to the Alpha version.

With some human intervention and effort (to produce maintainable code), the decompilation of a program can be used to:

- recover lost source code, either by accident or malevolence;
- correct errors;
- add new features.

Further reasons to employ decompilation are given in [3].

### 2.1.3 Legal implications

Like other technologies, not all uses of the decompilation techniques are legally accepted [4]. Indeed, computer programs are covered by *Author Copyrights*, which protect the *expression* of an idea in the form of a program, forbidding duplication or creation of derivative products, and thereby protecting the investment made by the programmer or company. Software is also frequently bundled with user agreements that restrict its use in order to prevent decompilation or disassembly of the program by the user.

Depending on the country, there are exceptions in the law that allow decompilation/disassembly of programs for:

- interoperability with another software or hardware when the interface is not specified;
- correcting errors when the copyright holder is not available;
- and for determining whether parts of the program violate rights other than the authors' (e.g., such as patents or business secrets).

### 2.1.4 Feasibility

Fully automated decompilation of executable machine code into the original source code is not always possible:

- **Ambiguity** – there is an ambiguous correspondence between high-level language statements and the respective machine code instructions. The same source code can be compiled into different machine codes. That is the case, for example, when different compiler optimization strategies are chosen. And vice-versa: different high-level constructs can be compiled into the same machine code. High-level languages, such as the C language, offer to the programmer a rich choice of syntactical constructs for similar ends: *for* vs. *while* loops, *select* vs. *if-then-else* conditionals, *enum* vs. *int* data types, and data structures vs. pointer arithmetic.

- **Information loss** – much of the original information is discarded during the compilation process. For example, variable names, function names, data structure definitions, and code comments are not included in the final executable, unless debugging information is explicitly required.
- **Distinction between code and data** – even the static disassembly of machine code – a step that precedes the actual decompilation – is not a trivial task. The distinction between data and code in an executable is often blurred. In the extreme it is possible to encounter self-modifying code. That is the case of self-unpacking executables, executable obfuscators, and viruses.

Nevertheless, some degree of success is possible if one or more of the above limitations are somehow relaxed:

- **Binary translation**, used in *Just In Time* (JIT) compilers and emulators, is one of those cases where decompilation techniques have been successfully applied [5]. During binary translation the machine code of the source platform is decompiled to an intermediate representation and then recompiled to machine code of the target platform (fig. 2.4). The intermediate representation is at a lower level than the original source code in order to allow fully automated decompilation, but still higher than the original machine code. The higher the intermediate representation level is, more optimizations are possible and more efficient the generated code will be.

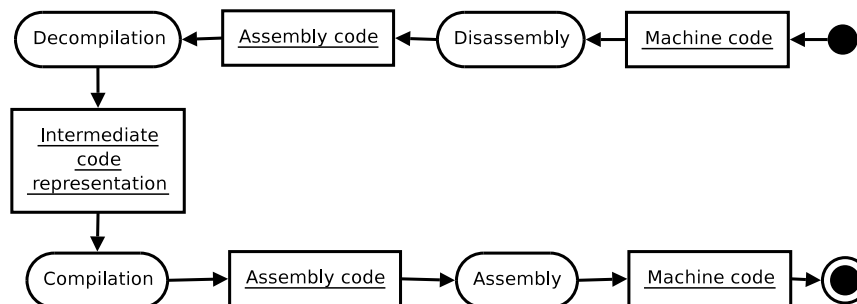


Figure 2.4: Binary translation

- **Byte-code compilation** – another successful case is the decompilation of compiled *byte code* of interpreted languages, such as Java, Python, and .NET languages. Because the compiled byte code is meant to be interpreted by a virtual machine or compiled to native machine code by a JIT compiler, it preserves much of the original syntactic information. This allows to obtain high-level code similar to the original source code in a fully automated fashion.
- **Human intervention** can step in where automatic decompilation techniques falter, providing the necessary input to disambiguate code semantics, organize code, and improve readability.

If the purpose is to obtain maintainable source code on a high level language, such as the C language, starting from machine code, then the only possible relaxation to the requirements is to allow human intervention. This is precisely the focus of this work.



### 2.1.5 State of existing reverse-engineering tools

This section reviews the existing machine code reverse engineering tools, including, but not limited to, code decompilation tools.

#### The `dcc` decompiler

The `dcc` decompiler [6] was a pioneer program developed by Cristina Cifuentes [1] that decompiles EXE files from DOS into C programs.

The analysis performed by the `dcc` decompiler relies on traditional compilation optimization techniques and graph theory. The former is used to eliminate register and intermediate instructions in order to reconstruct higher level instructions; the latter is used to determine the high-level control structures. The final program retains the Assembly code that cannot be decompiled.

**Decompilation stages** The `dcc` decompilation analysis is divided in several stages (fig. 2.5), similar to the stages in a compiler [1, p. 9].

During the loading stage the input executable is read, in the DOS EXE format.

Unlike compilers, there is no lexical analysis stage, as there is no mean to know whether a byte belongs to the beginning, middle, or end of an instruction.

During the parsing stage the code is syntactically analyzed, and transformed in grammatically equivalent expressions. For example, the Assembly instruction `sub cx, 50` is transformed in the expression `cx = cx - 50`. The major difficulty of this stage is separating code from data, as both are mixed in the executable, and virtually any byte combination constitutes a valid machine code instruction. To reduce the size and complexity of this task, available auxiliary information (such as compiler signatures, library signatures, and library prototypes) is taken in consideration, allowing to focus on the code unique to the executable. After parsing it is possible to generate a disassembly as a decompilation side-product. To facilitate subsequent analysis, the parse tree is transformed into an intermediate language.

Next, *Control Flow Graphs* (CFGs) are generated for each subroutine. These are necessary for the recognition of high-level control structures.

During the semantic analysis, the semantic meaning of groups of instructions is verified, type information is annotated and propagated through the subroutines. Idioms are identified and transformed into semantically equivalent expressions. For example, if it is known that the first argument of a subroutine is a long integer, then the instructions

```
asgn [bp+2], 0
asgn [bp+4], 14h
```

would be transformed into a single instruction

```
asgn [bp+2]:[bp+4], 14h
```

The data flow analysis simplifies the intermediate code by eliminating the use of temporary registers and conditional flags. For example, the instruction sequence

```
asgn ax, [bp-0Eh]
asgn bx, [bp-0Ch]
asgn bx, bx * 2
asgn ax, ax + bx
asgn [bp-0Eh], ax
```

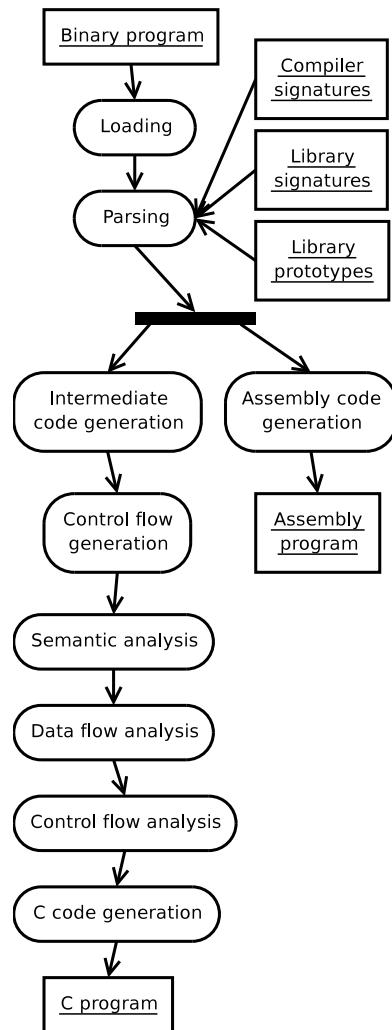


Figure 2.5: dcc decompilation stages

would be converted into a single instruction

```
asgn [bp-0Eh], [bp-0Eh] + [bp-0Ch] * 2
```

During the control flow analysis, high-level control structure patterns are recognized, thereby transforming the spaghetti of *goto* statements into nested conditionals and loops.

The last stage is the generation of the high-level C code. Names are given to all subroutines, arguments, local variables, and registers.

**Limitations** The `dcc` source code is available under an open-source license, but it has not been updated ever since it was written in 1994, having several limitations. It is a *proof of concept* program, which can only cope with simple academic examples, and it supports a single processor/platform combination.

## UQBT

UQBT [7] is a framework for the construction of generic recompilers aimed at the binary translation between different processors. Its current implementation generates low level C intermediate code that is then compiled and optimized to the target platform by an ordinary C compiler [5].

Although UQBT's main purpose is not the construction of decompilers, this framework defines a new language for specifying the semantics of machine instructions – the *Semantics Specification Language* (SSL) [8]. Such specifications can be reused for decompilation purposes, and the UQBT source distribution includes the SSL specifications for several processors.

## IDA Pro

IDA Pro [9] is a popular commercial disassembler with support for multiple processors. Albeit not a decompiler, IDA Pro employs several of `dcc`'s decompilation techniques to generate richer Assembly code, namely the mapping and tracking of local variables in the stack and the recognition of standard library function signatures.

Nevertheless, most of IDA Pro's popularity stems from its user interactivity. It provides a graphical interface to manually adjust several aspects of the Assembly code, such as to:

- mark code and data regions;
- delimit functions;
- define complex data structures;
- attribute semantic meaning to operand constants, making them as global pointers, stack variable offsets, structure members offsets, etc.

The graphical interface also provides effective visualization of derived program information, such as control flow graphs and data cross-reference tables, which assist the user in the program understanding and navigation.

## Boomerang

The Boomerang project [10] aims to develop a generic decompiler through the open-source community. It is developed in the C++ language. It reuses the decompilation techniques developed for the dcc decompiler and the machine semantics specifications from UQBT framework, and has been subject of research and experimentation of new decompilation techniques [11].

Still, the Boomerang decompiler currently has some limitations, mostly derived from its design. There are currently plans for a rewrite of the tool, and one of the main drives for rewriting is to allow inclusion of interactive decompilation, with the same spirit of IDA Pro [12].

## CodeSurfer/x86

CodeSurfer [13] is a powerful tool for code analysis, understanding, and inspection. It is able to:

- visualize the *Program Dependency Graph* (PDG) – a graph that, unlike the CFG, takes into account both the *data* and control dependency [14, 15];
- perform program *slicing* – the discovery of all statements and predicates that might affect the value of a given variable at a given point of a program [16, 17];
- perform program *chopping* – the discovery of the statements that transmit effects between two program points (an extension of the program slicing concept) [18].

CodeSurfer/x86 [19] is a prototype system that integrates CodeSurfer with IDA Pro for the analysis of Intel x86 executables [20]. It employs *Value Set Analysis* (VSA) – an algorithm to determine an over-approximation of the set of integer values (or memory addresses) that each data object can hold at each program point [21] – to recover the data flow in the executable without using a symbol-table or debugging information.

## Other tools

An exhaustive list of known machine code decompilers is given in [22].

## 2.2 Program Transformation

The topic of program decompilation is part of a much larger field, which is the field of program transformation. Examples of other program transformations are program compilation, program optimization, program refactoring, and program obfuscation.

This section reviews the commonly used approaches for program transformation systems.

### 2.2.1 Program Representation

The choice for program representation directly impacts the kind of transformation systems used. Visser [23] identified the most common program representations, which are described below.

### Textual representation

Programmers normally write programs as text. It is not the most convenient representation for complex program transformations. Usually programs are first parsed into abstract structured representations by a parser and afterwards converted back into textual representation by an unparser. Nevertheless, some systems work directly with text, most noticeably tools included in interactive program editing and developing environments.

### Parse Tree

A parse tree (or concrete syntax tree) represents the syntactic structure of the program according to the rules of a grammar. A parse tree contains nodes and edges which do not affect the semantics of the program, such as white-space, comments, and grouping parenthesis. Nevertheless, it is used in some applications where it is necessary to restore that information as much as possible, such as for program refactoring.

### Abstract Syntax Tree

An *Abstract Syntax Tree* (AST) differs from a parse tree by omitting those nodes and edges which do not affect the semantics of the program.

### Direct Acyclic Graph

*Direct Acyclic Graphs* (DAGs) are often used instead of regular trees. A DAG allows subtree *sharing* – multiple references to the same subtree in different nodes –, making tree copying a constant time operation.

With *maximal sharing* only one copy of every subtree is kept in memory, achieving minimal memory usage and making the equality comparison of two subtrees a constant time operation.

The downside of sharing is that updating a subtree cannot be allowed, because a subtree can be shared in multiple points of the tree with a completely different context. Instead, updating a subtree must be achieved by rebuilding all its ancestor nodes. Also, the tree annotation of layout information, such as the originating file name and column, frequently used for error reporting, is not practical, as it reduces the possibility of subtree sharing.

### Term

In logical formalism, a term  $t$  is a constant (e.g., 1 or 0.1), a variable, or the result of acting on other terms by function symbols (e.g.  $C(t_1, t_2, \dots, t_n)$ ). Terms can be used to describe program ASTs, by corresponding the tree nodes labels to term constructors and the tree edges to the subterms.

### Full Fledged Graph

Programs can also be represented as fledged graphs, having for example back-links in the tree to represent loops (such as control flow graphs) and variable declarations [23].

## 2.2.2 Transformation Paradigms

### Term rewriting

A term rewrite rule has the form  $x \rightarrow y$ , where both  $x$  and  $y$  are terms and every variable in  $y$  also occurs in  $x$ . Term rewrite rules can be used to express basic program transformation rules.

A set of rules defines no particular order of application. A rewriting strategy is an algorithm for applying rules to achieve a certain goal, typically a term normalization [23]. Common normalizations are the *innermost* and *outermost* normalizations, which successively apply the set of rules starting from the subterms or root term, respectively.

Normalization of terms with respect to a set of rewrite rules is applicable in areas such as algebraic simplification of expressions and automated theorem proving [24, 25].

Nevertheless, the basic approach of normalizing a program tree with respect to a set of transformation rules is not adequate for program transformation due to a number of reasons given in [23]. This inadequacy motivated the development of many extensions and variations of the basic rewriting paradigm for program transformation systems. The most noticeable of these being the Stratego language [26] – an important reference for this work.

### Tree parsing

In tree parsing rules are written as tree grammar rules. These are used to traverse the tree with an applicable set of rules and execute corresponding actions.

Tree parsing is used in parser generators such as ANTLR, which can generate a tree walker from a tree grammar [27, 28].

### Attribute Grammars

Attribute grammars extend the context free grammar formalism with attributes and attribute equations [29]. Attributes are associated to non-terminal symbols and attribute equation to the productions of the grammar. The process of attribute evaluation consists of assigning values to the attributes of tree nodes (that are instances of non-terminal symbols). Typically, the attribute evaluation order is defined based on the dependencies induced by the attribute equations.

Attribute grammars can be used to create language translation tools such as compilers.

Lrc [30] is a system for generating efficient incremental attribute grammar evaluators, and can be used to generate language based editors and other advanced interactive environments [31].

### Other approaches

Most program transformation systems use one of the approaches above, or a variation. Other program transformation systems can be found on the Program Transformation Wiki [32].

## 2.3 Refactoring

### 2.3.1 Definition

Opdyke [33] first defined refactoring as a behavior-preserving program transformation, usually to make the program easier to understand and maintain. Fowler [34] subsequently presented a refactoring catalog that is aimed at improving software design, and Kerievsky [35] later proposed an high-level kind of refactorings to improve an existing design with design patterns.

### 2.3.2 Refactoring and decompiling

The decompilation of a program has both the same understanding and maintenance simplification aims and the same behavior-preserving property as does a refactoring. Thus decompilation of a program could be carried out as the composition of basic refactorings.

Refactoring has been used for low-level code optimization [36] and for reverse engineering design patterns [37], but no previous use of refactoring for decompilation was found.

### 2.3.3 State of existing interactive refactoring tools

A refactoring browser is an interactive tool that automatically applies refactorings to source code. Refactoring browsers are particularly relevant to this work, as they involve both program transformation and user interactivity.

This section reviews some of the existing tools for automated program refactoring. The criteria used for the selection of the tools here presented was their relevance in terms of associated publications or popularity.

#### The Smalltalk Refactoring Browser

The Smalltalk Refactoring Browser [38] is a tool that provides an interactive environment to perform many refactorings automatically to Smalltalk programs [39, 40].

The Smalltalk Refactoring Browser plugs into the host development environment<sup>1</sup> and operates over its parse trees. Refactorings are implemented via a rewriting system using tree pattern matching.

The Smalltalk Refactoring Browser also performs dynamic code analysis in order to carry out analysis which are either difficult or impossible to perform via static code analysis only, e.g., to determine the type of a particular variable (which is difficult due to Smalltalk dynamic typing) and to determine cardinality relationships between objects. The dynamic analysis relies upon Smalltalk reflective facilities.

#### HaRe – The Haskell Refactorer

HaRe – the Haskell Refactorer [41] – is a tool that provides support to refactor Haskell programs from within a program editor<sup>2</sup> [42, 43].

The refactorings operate on the AST and are implemented in Strafunski [44] – a Haskell centered program transformation system based on the notion of a functional strategy [45, 46], inspired on the Stratego language and the rewriting strategies paradigm.

---

<sup>1</sup>VisualWorks, VisualWorks/ENVY, or IBM Smalltalk

<sup>2</sup>VIM and Emacs

Refactorings preserve as much as possible the comments and white-space of the original source code. This is accomplished by representing the program both as a token stream and an AST, using the AST only as an auxiliary representation to guide the direct modification of the token stream.

### **Eclipse**

Eclipse [47] is a popular open-source *Integrated Development Environment* (IDE) for Java and other languages with a strong refactoring ability.

Refactorings are carried out by operating on an AST and writing back changes to the textual representation of the source code, therefore preserving user formatting and markers.

### **Other tools**

More refactoring tools are mentioned at <http://www.refactoring.com/tools.html>.

## **2.4 Review of related work**

Based on the work revised, the most important to this work are: [1], which provides the basic methodology for machine code decompilation; [7, 8], which provides an extensive description of processor semantics, necessary for decompiling; [34], which establishes a sound terminology and notation for describing a refactoring and provides an extensive refactoring catalog; and [48, 26, 49], which provides a system for complex program transformations, such as the ones involved for machine code decompilation. [11] is not so import, at least not in an initial stage, since most of its techniques are focused on automatic decompilation, where the decompilation analysis is continuous and over all program being decompiled, rendering them inadequate for interactive decompilation, where the analysis is short-lived, and over specific points of the program.



## Chapter 3

# Catalog of low-level refactorings

The decompilation of a program can be carried out as the sequential application of basic program transformations to that program, where every transformation increases the abstraction level of the code while retaining its semantics. So each transformation can be considered as a refactoring – a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [34].

This chapter proposes a catalog of refactorings for low-level [near Assembly] code for reverse engineering purposes. The refactorings here listed help on making the low-level code incrementally more intelligible. The combined and successive application of these refactorings can effectively bring a low-level machine code to a higher-level code, while retaining its semantics, as illustrated by fig. 3.1.

Some of the refactorings here listed are in all aspects identical to their higher-level languages counterparts, listed in [34]. Other refactorings are specific to the traits of Assembly code. Table 3.1 lists all refactorings of this proposed catalog.

The catalog is presented using the standard format described in [34, p. 103]: a short synopsis, an example, the motivations for its use, and the application mechanics. The examples are written in almost syntactically correct C code, with some exceptions in order to faithfully represent near Assembly code. Namely, statements can also appear in the global scope, besides of appearing in function bodies.

In section 5.4 it is presented the application of these refactorings to decompile a realistic example.

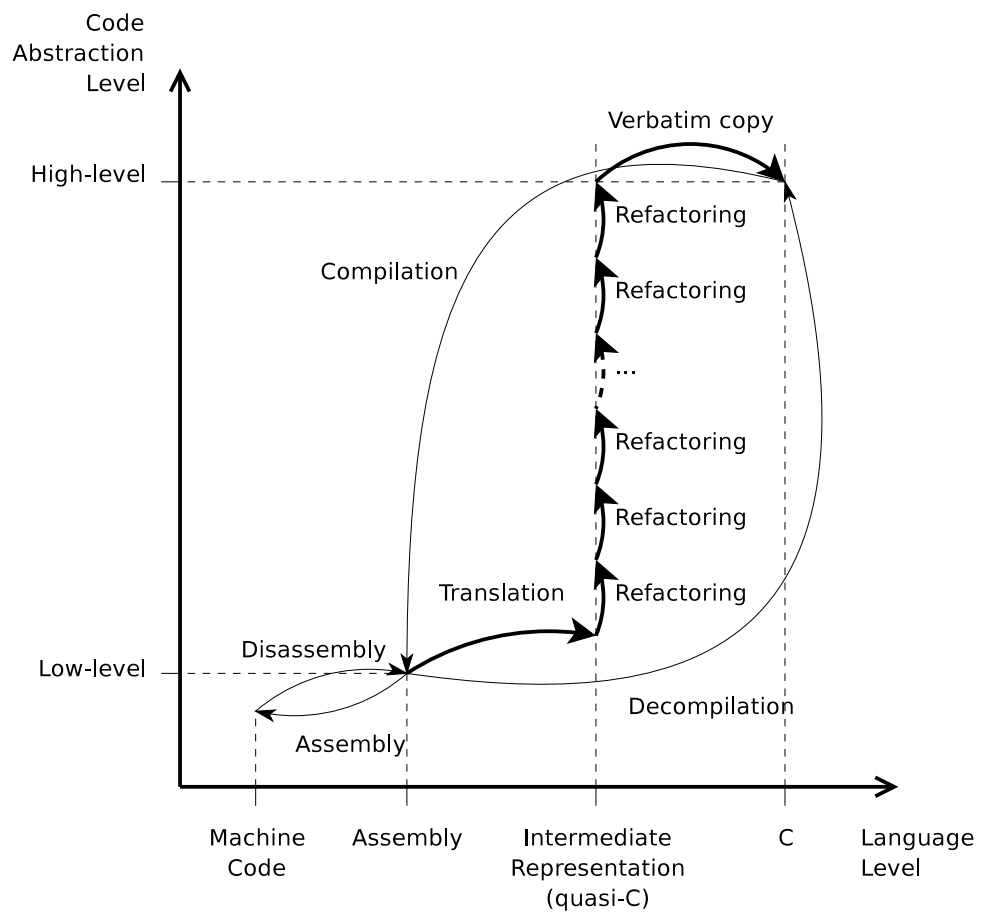


Figure 3.1: Decompilation as a sequence of refactorings

Table 3.1: Refactoring catalog

<b>Purpose</b>	<b>Name</b>	<b>Page</b>
Function prototyping	Extract Function	18
	Set Function Return	19
	Add Function Argument	20
Organizing data	Extract Local Variable	21
	Inline Temp	21
	Split Temporary Variable	22
	Replace Magic Number with Symbolic Constant	23
	Replace Data Values with Record	23
	Replace Type	24
	Dead Code Elimination	24
	Rename Symbol	25
Simplify Expression	25	
Structuring control flow	Structure <i>If</i> Statement	27
	Structure <i>If-Else</i> Statement	27
	Structure <i>Do-While</i> Statement	28
	Structure Infinite Loop	28
	Structure Continue Statement	29
	Structure Break Statement	29
	Structure While Statement – Form I	30
	Structure While Statement – Form II	30
	Structure While Statement – Form III	31
	Inline Return Statement	32
	Consolidate Boolean <i>And</i> Expression	32

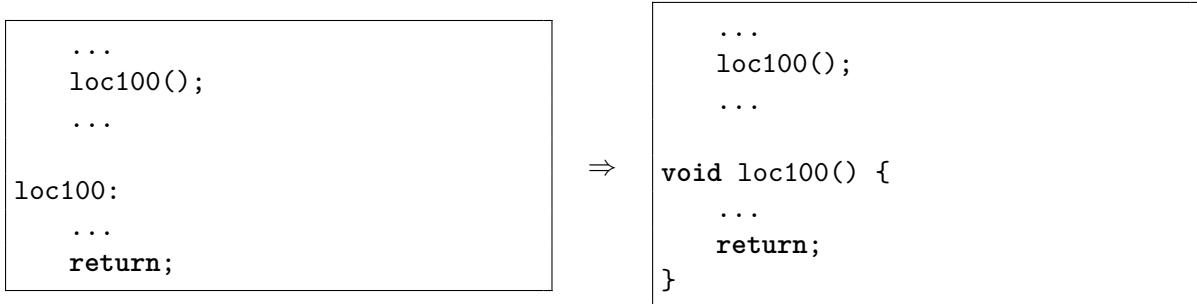
## 3.1 Function prototyping

As in most imperative languages, functions constitute the basic reusable unit of Assembly code, and are usually generated from the higher-level source code on an one to one basis during compilation. But the information about the function bodies, arguments, and local variables is not properly retained by the Assembly code. The following refactorings allow to incrementally lift the bodies, prototypes, and frames of functions.

### 3.1.1 Extract Function

You have a set of code fragments that constitutes an individual function.

*Turn the fragments into a function.*



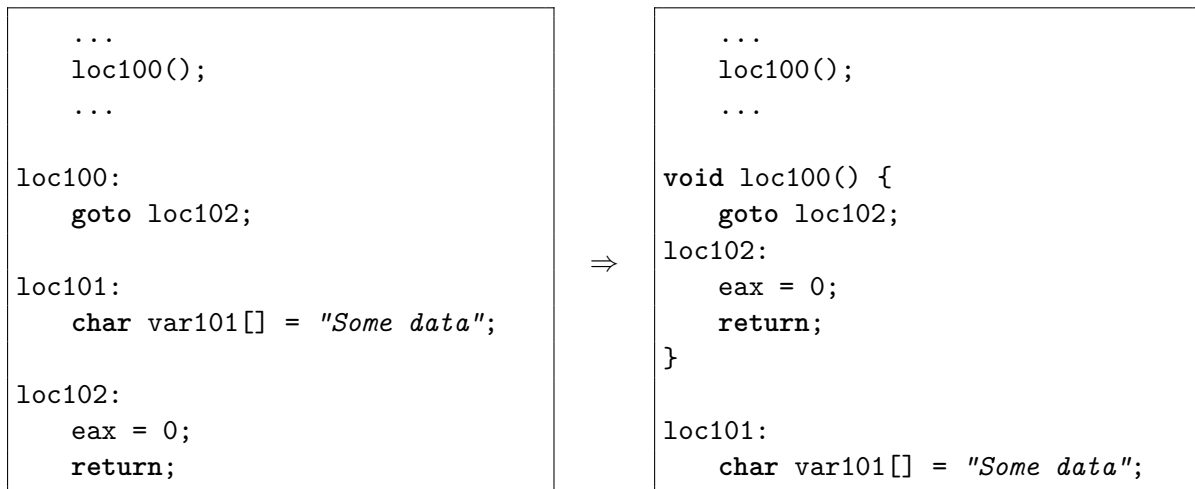
#### Motivation

The Assembly functions generated by compilers are not always clearly delimited in machine code. Moreover, the machine code corresponding to the function body may be interleaved with auxiliary data, such as initialization constants and jump tables.

So the basic step in reverse engineering the code is to aggregate the scattered code fragments in individual functions.

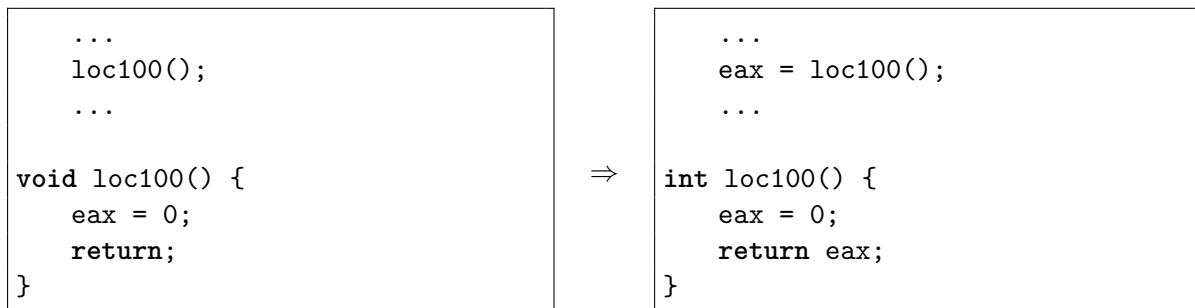
#### Mechanics

- Identify all code fragments belonging to a function.
- Shift all the code fragments together in order to make a single continuous code fragment.
- Wrap the fragment by a function enclosure.
- Scan the code for references for the function label and promote them to function calls.

**Example: noncontiguous code fragments****3.1.2 Set Function Return**

A register or the stack is used to pass the function return value.

*Define the function return type with the appropriate type, making explicit that such stack position or register is the return value.*

**Motivation**

Most processors have no built-in support for returning values in function calls, i.e., native function calls have no return type. It is a common function calling convention to use a particular processor register to pass the function return value to the caller for simple types and the processor stack for complex types. For example, in the Intel IA-32 architecture the `eax` register is used to pass integers. This is a platform-specific low-level detail, and not one of the source code, which should therefore be eliminated for proper code understanding.

**Mechanics**

- Change the function return type from `void` to the appropriate type.
- Post-assign all calls to the function to the specified register or stack position.

- Add the specified register or stack position to every return statement in the function body.

### 3.1.3 Add Function Argument

The stack or a register is used to pass an argument to a function.

*Define a new function argument with the appropriate type, making explicit that such stack position or register is used to hold the argument.*

<pre>...   eax = 1;   loc100();   ...  void loc100() {   ...   ... = eax;   ...   return; }</pre>	⇒	<pre>...   eax = 1;   loc100(eax);   ...  void loc100(int arg1) {   ...   ... = arg1;   ...   return; }</pre>
---	---	---

#### Motivation

In a similar fashion as the function return value, it is a common calling convention to pass function arguments in particular registers, the stack, or both. This is platform-specific detail, which should be eliminated for attaining proper code understanding.

#### Mechanics

- Add a new function argument with the appropriate type.
- Add the register or stack location as argument to all function calls.
- Replace all references to the register or stack location in the function body to the argument name.

## 3.2 Organizing data

During compilation all the data flow is mapped to accesses from/to the processor registers, stack, and global memory. The following refactorings incrementally transpose that data flow in terms of local and global variables. They operate mostly on a function level.

### 3.2.1 Extract Local Variable

A register or the stack is used to hold the value of a local variable.

*Define a new local variable with the appropriate type, and change all local references to that stack location or register into a reference to that variable.*

```
void loc100() {
    ...
    eax = ...;
    ...
    ...
    ... = eax;
    ...
}
```

⇒

```
void loc100() {
    int var1;
    ...
    var1 = ...;
    ...
    ...
    ... = var1;
    ...
}
```

#### Motivation

During compilation, local variables are mapped into registers or stack positions. This transformation must be reversed in order to improve code understanding.

#### Mechanics

- Declare a new variable in the function scope.
- Replace all references to the specified register or stack position by a reference to the newly created variable.

### 3.2.2 Inline Temp

You have a temporary variable that is assigned and used just once or a few times.

*Replace all references to that temporary value with the actual expression.*

```
...
eax = 0xff;
ecx = ecx & eax;
edx = eax | edx;
...
```

⇒

```
...
ecx = ecx & 0xff;
edx = 0xff | edx;
...
```

## Motivation

The expression depth of Assembly code is very shallow, as individual Assembly instructions can only perform basic arithmetic operations. This leads to an excessive use of temporary variables, as the compiler breaks down an expression into smaller sub-expressions. These temporary variables should be inlined so that the full expression may be recovered and easily understood.

Also, on many processor architectures a register access is quicker than storing an immediate constant value, or it produces a smaller code footprint, therefore optimizing compilers often allocate registers to hold frequently used constants (including addresses of frequently used functions). But it is no advantage for the code comprehension having temporary variables used as aliases for constants or functions.

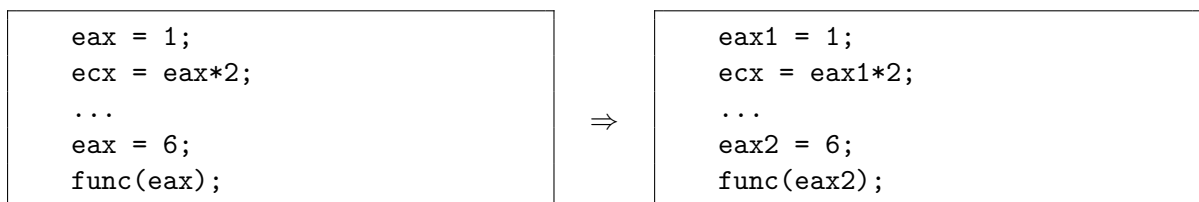
## Mechanics

- Replace the variable by its expression in the statements the value is used.
- Remove the variable assignment.
- Remove the variable declaration if the variable is no longer used.

### 3.2.3 Split Temporary Variable

You have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable.

*Make a separate temporary variable for each assignment.*



## Motivation

In compiled code, the registers are recurrently used as temporary variables, having more than one responsibility in the code, which is confusing to the reader. A different temporary variable should be used for each responsibility.

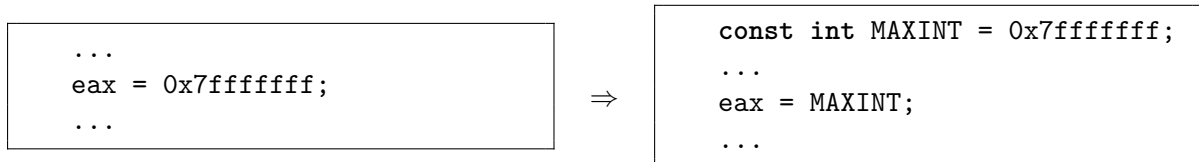
## Mechanics

- Declare a new variable for each assignment of the specified variable.
- Rename the variable at and after each assignment, using one of the newly defined variable names.



### 3.2.4 Replace Magic Number with Symbolic Constant

You have a literal number with a particular meaning.  
*Create a constant, name it after the meaning, and replace the number with it.*



#### Motivation

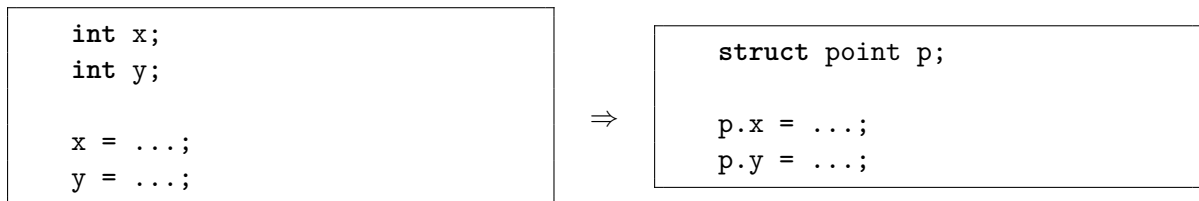
When compiling, all the semantic information concerning enumeration values, pre-processor macro definitions, and constants is lost. So all the symbolic constants must be specified during decompilation.

#### Mechanics

- Create a new constant with the number, named after the meaning.
- Replace the number with the newly created constant.

### 3.2.5 Replace Data Values with Record

You have a set of variables that altogether constitute a record.  
*Replace the variables by a single record and replace the variable references by references to the record members.*



#### Motivation

During the compilation the record unity is lost, as record members are replaced by mere memory offsets. It is necessary to re-join the record members back in order to recover the original records.

#### Mechanics

- Define a new record with the appropriate members, if one is not defined already.
- Replace the specified variables declarations by a single record declaration.
- Replace the variables by members of the newly declared record.

### 3.2.6 Replace Type

You are using integer variable to carry other kind of information.

*Replace the integer type with the appropriate type, and propagate it across the expressions the variable is used.*

```
void function() {
    int tmp;
    tmp = ...
    puts((char *)tmp);
}
```

⇒

```
void function() {
    char *tmp;
    tmp = ...
    puts(tmp);
}
```

#### Motivation

Integer types and floating point types (on processors with a *Floating Point Unit*) are about the only types natively supported by a common processor. Operations with all other high-level types (*enums*, *typedefs*, *structs*, and *unions*) are translated into operations with these native types. Pointers are translated to memory addresses, which are also indistinguishable from regular integers in most architectures. This refactoring allows to recover the original high level types by propagating the type information across the code.

#### Mechanism

- Define the new type, if not defined yet.
- Replace the variable type.
- Propagate type in the expressions it is used, adding or removing type casts as needed.

### 3.2.7 Dead Code Elimination

You have several variable assignments, whose value is not used.

*Remove those variable assignments.*

```
/* Original Assembly
   instruction:
   * addw %eax, %ebx
   */
tmp = ebx; // original value
ebx = ebx + eax;
zf = ebx == 0; // zero flag
nf = ebx < 0; // negative flag
of = ... // overflow flag
cf = ... // carry flag
pf = ... // parity flag
```

⇒

```
/* Original Assembly
   instruction:
   * addw %eax, %ebx
   */
ebx = ebx + eax;
```

### Motivation

When compiling a program the compiler chooses the sequence of processor instructions that best mimic the behavior specified by the source code. Most processor instructions, however, have several hard-coded side-effects that are often unintended by the compiler. The best example is given by the flag registers, which are modified after every arithmetic processor instruction, but their value is typically only used on conditional jumps. Therefore a semantic translation of an arithmetic processor instruction often results in several unused flag assignments, which can and should be removed for better code understanding.

### Mechanics

- Traverse the code in the inverse direction of the control flow, removing all unused variable assignments.

#### 3.2.8 Rename Symbol

You have a symbol with a meaningless machine generated name.  
*Rename that symbol into some meaningful.*

<pre>void loc12345() {     ... }</pre>	⇒	<pre>void do_something() {     ... }</pre>
--	---	--

### Motivation

During compilation and linking, if debugging information is stripped then most information regarding symbol names is lost, with the sole exception of the exported symbols of dynamically linked libraries. Disassemblers normally just create arbitrary symbols names (composed with either a sequence number or the symbol memory address) whenever one is needed. To improve code understanding these symbols should be renamed to a meaningful name, which reflects the program semantics.

### Mechanics

- Ensure there is no name conflict.
- Change all occurrences of the symbol in the code to the new symbol name.

#### 3.2.9 Simplify Expression

You have a mathematical expression with unnecessary complexities.  
*Simplify that expression.*

<pre>eax = eax ^ eax;</pre>	⇒	<pre>eax = 0;</pre>
-----------------------------	---	---------------------

### Motivation

When generating machine code to evaluate mathematical expression the compiler frequently resorts to equivalent expression that are more complex (from a human point of view) yet evaluated faster by the targeted hardware. For example, if zeroing a register processor is pretended, it is often faster to perform a XOR of the register against itself than to move the immediate zero operand from the machine code into the register. These more complex expressions are rarely the ones idealized by the programmer, and they should be simplified in order to improve their understanding.

### Mechanics

- Apply the appropriate mathematical simplification rules to the expression in order to make it more simple.

### Example: equality comparisons

Equality comparisons are normally translated into comparisons against zero, to make efficient usage of the *zero flag* register, which exists in most processors.

<pre>if(eax - 2 == 0) ... </pre>	⇒	<pre>if(eax == 2) ... </pre>
----------------------------------	---	------------------------------

### Example: small (bitwise) integer constant multiplication

Integer multiplications of constants with a small number of non-zero bits are normally compiled as a sum of bitwise shifts, which take fewer processor cycles than a regular multiplication instruction.

<pre>eax = (ecx &lt;&lt; 3) + (ecx &lt;&lt; 1);</pre>	⇒	<pre>eax = ecx * 10;</pre>
---	---	----------------------------

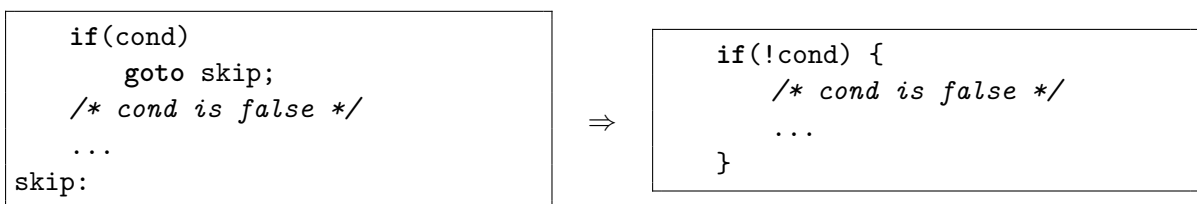
### 3.3 Structuring control flow

All high-level language control structures (*if*, *while*, and *for* statements) are translated into jumps and conditional jumps on Assembly language. These refactorings incrementally recover the high-level control structure that match the jumps control-flow graph.

#### 3.3.1 Structure *If* Statement

You have a conditional jump over consecutive statements.

*Negate the conditional expression and make statements the conditional clause.*



#### Motivation

*If-then* statements are translated by the compiler as a conditional jump over the statements that constitute the *then* clause. This refactoring recovers the original control structure.

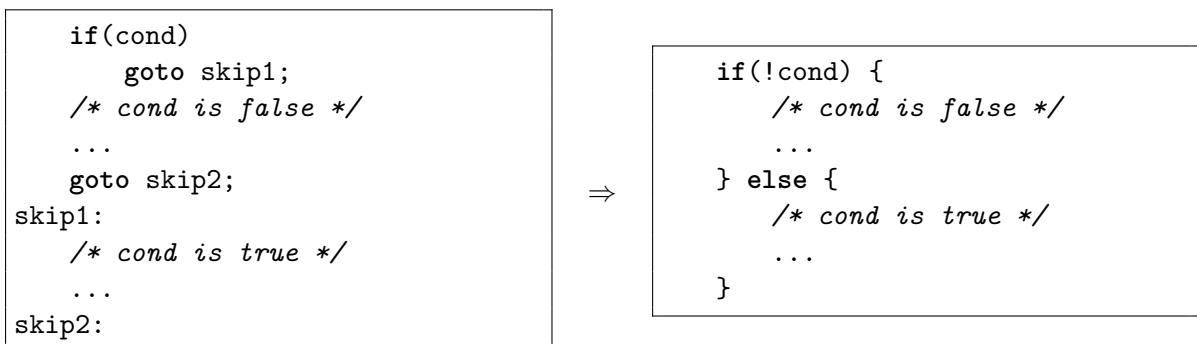
#### Mechanics

- Negative the *if* statement condition.
- Replace the jump by the skipped statements.
- Remove the jump label, if not longer used.

#### 3.3.2 Structure *If-Else* Statement

You have a conditional jump to two sets of consecutive statements.

*Make each set of statements a clause of the conditional statement.*



**Motivation**

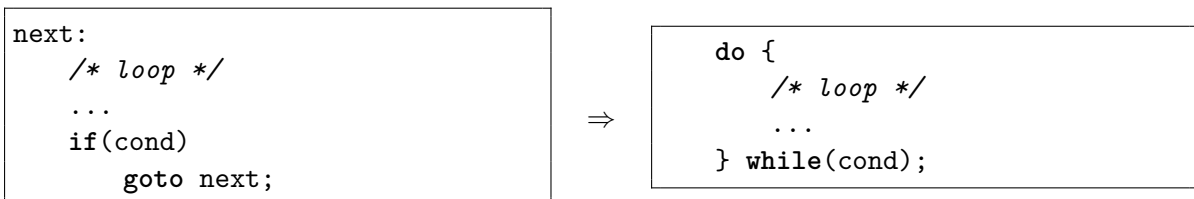
*If-then-else* statements are translated by the compiler as a conditional jump to two sets of consecutive statements. This refactoring recovers the original control structure.

**Mechanics**

- Negative the *if* statement condition.
- Make the first set of skipped statements the *then* clause.
- Make the second set of skipped statements the *else* clause.
- Remove any unused label.

**3.3.3 Structure Do-While Statement**

You have a conditional jump to a previous label.  
*Make the intermediate statements the body of a do-while loop.*

**Motivation**

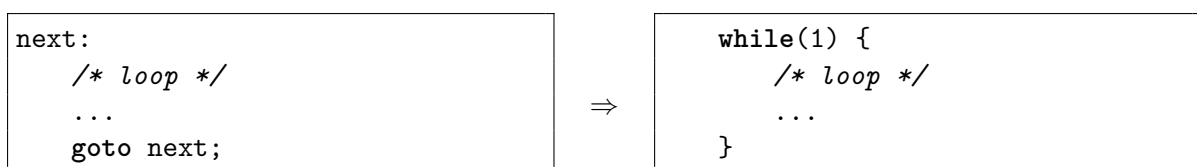
*Do-while* statements (and sometimes other kinds of loop statements) are translated as conditional back jumps. This refactoring recovers those original control structures.

**Mechanics**

- Make the set of skipped statements the body of the *do-while* statement.
- Make the jump condition the condition of the *while* clause.
- Remove the label, if no longer used.

**3.3.4 Structure Infinite Loop**

You have a jump to a previous label.  
*Make the intermediate statements the body of an infinite loop.*



**Motivation**

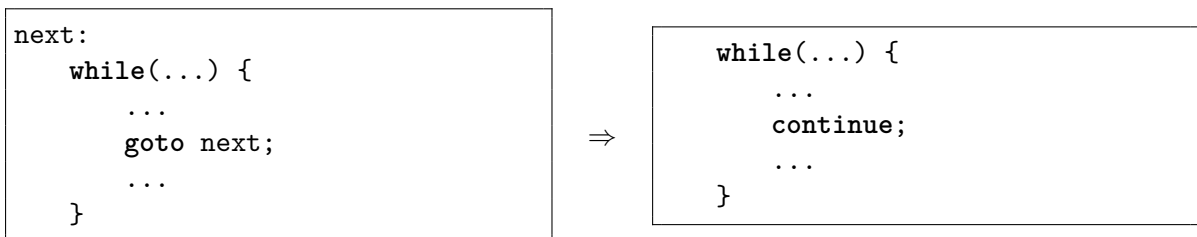
Infinite loops are not frequently used by programmers, nevertheless, compilers often use these unconditional loops together with `break` and `continue` jumps to translate ordinary *while* and *for* loops. This refactoring recovers those original control structures as an intermediate step.

**Mechanics**

- Make the set of skipped statements the body of the *do-while* statement.
- Remove the label, if no longer used.

**3.3.5 Structure Continue Statement**

You have a jump inside a loop to a label immediately preceding the loop.  
*Replace the jump by a continue statement.*

**Motivation**

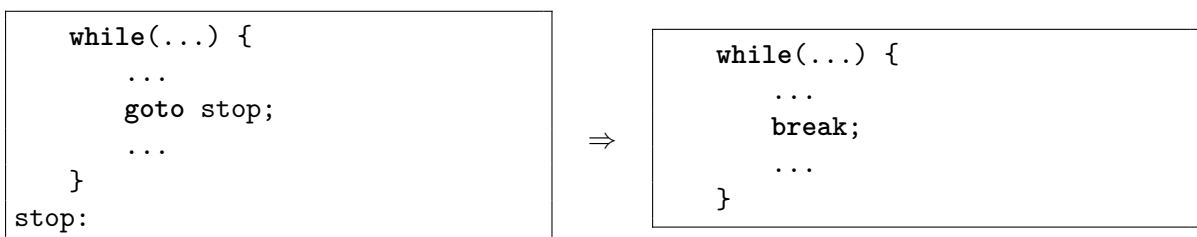
`Continue` statements are translated by compilers as simple jumps. This refactoring recovers the original statement. It requires that the refactoring corresponding to the loop structure is previously applied.

**Mechanics**

- Replace the jump by a *continue* statement.
- Remove the label, if no longer used.

**3.3.6 Structure Break Statement**

You have a jump inside a loop to a label immediately succeeding the loop.  
*Replace the jump by a break statement.*



**Motivation**

Break statements are also translated by compilers as simple jumps. This refactoring recovers the original statement. It requires that the refactoring corresponding to the loop structure is previously applied.

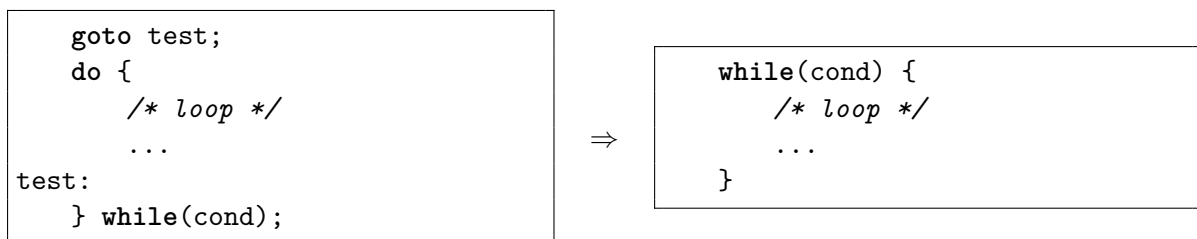
**Mechanics**

- Replace the jump by a *break* statement.
- Remove the label, if no longer used.

**3.3.7 Structure While Statement – Form I**

You have a *do-while* statement preceded by an unconditional jump to a label immediately before the statement condition.

*Make the do-while statement a while statement.*

**Motivation**

*While* statements are translated by compilers into one of the three following possible forms: the condition succeeds the loop body, precedes it, or both. This refactoring recovers the original control structures when the former form is used. It requires the Structure Do While Statement refactoring to be applied first.

**Mechanics**

- Remove the jump.
- Change the *do-while* statement into a *while* statement.
- Remove the label, if no longer used.

**3.3.8 Structure While Statement – Form II**

The loop block of an infinite loop starts with a conditional break.

*Make the infinite loop a regular while statement.*



```

while(1) {
    if(cond)
        break;
    /* rest */
    ...
}

```

⇒

```

while(cond) {
    /* rest */
    ...
}

```

### Motivation

As previously mentioned, *While* statements are translated by compilers into one of several possible forms. This refactoring recovers the original control structures where the condition precedes the loop body. It requires that the Structure Infinite Loop and Structure Break Statement refactorings are applied first.

### Mechanics

- Move the condition of the conditional jump into the *while* statement.

### 3.3.9 Structure While Statement – Form III

A *do-while* loop is nested inside an *if* statement with the same condition.  
*Make the loop a regular while statement.*

```

if(cond) {
    do {
        /* loop */
        ...
    } while(cond);
}

```

⇒

```

while(cond) {
    /* loop */
    ...
}

```

### Motivation

As previously mentioned, *While* statements are translated by compilers into one of several possible forms. This refactoring recovers the original control structures where the condition both precedes and succeeds the loop body. It requires that the Structure If Statement and Structure Do-While Statement refactorings are applied first.

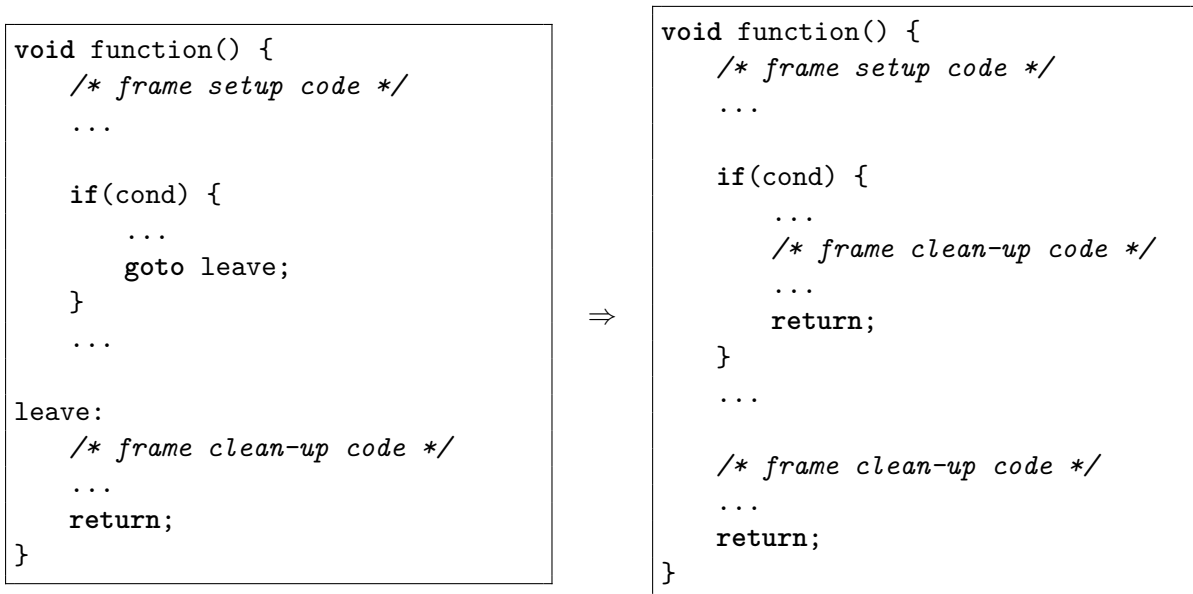
### Mechanics

- Remove the *if* statement.
- Change the *do-while* statement into a *while* statement.

### 3.3.10 Inline Return Statement

You have an unconditional jump to a set of consecutive statements ending with a return statement.

*Inline those statements.*



#### Motivation

When a function has multiple return statements the compiler unifies these statements in a single return statement, to avoid unnecessary duplication of the clean-up code. A jump, however, does not have the same semantic meaning of a return statement, being sometimes preferable the latter for better code comprehension.

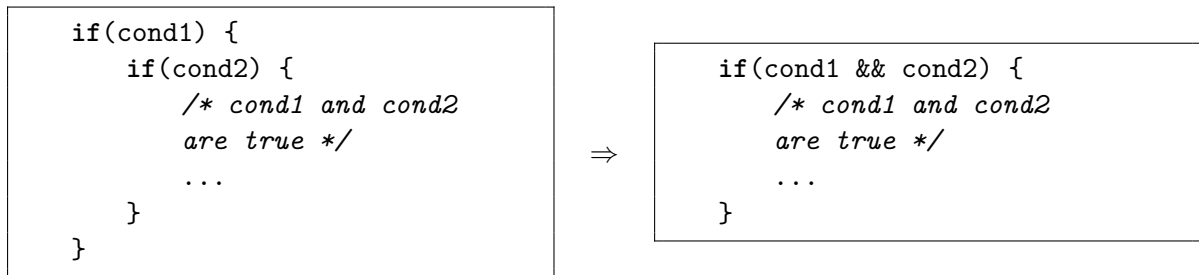
#### Mechanics

- Inline the statements between the label and the return statement instead of the jump.
- Remove the label, if no longer used.

### 3.3.11 Consolidate Boolean *And* Expression

You have two immediately nested *if* statements.

*Merge both conditional expressions with the Boolean and operator.*



### Motivation

The Boolean expressions of control-flow statements are usually translated by compilers as the composition of the simpler *if* statements, rather than being evaluated arithmetically. This refactoring allows to recover the original Boolean expressions. Boolean *or* expressions are translated via *and* expressions and negation.

### Mechanics

- Merge both conditional expressions with the Boolean *and* operator.

## 3.4 Example

An example of the application of these refactorings to decompile machine code is given in the interactive tool tutorial, in section 5.4.



## Chapter 4

# Design of the IDC tool

This chapter describes the design of the *Interactive Decompilation* (IDC) tool – the main forces behind the development and the rationale of the main design decisions.

### 4.1 Requirements

The main requirements defined for the IDC tool are the following:

- **Import Assembly code.** The main input to the tool is a program (or a program fragment) written in Assembly language. The generation of Assembly code from a binary executable is already the purpose of many available tools and is, therefore, beyond the scope of this work. At the moment the only targeted processors are those belonging to the Intel IA32 family [50, 51], but others may (and should easily) be supported in the future.
- **Visualize and export *quasi-C* language code.** The main output of the tool is the C language. The standard C is already a versatile language – it is able to represent code from low to high-level –, nevertheless, some small extensions to the standard C language will be required in order to faithfully represent the initial highly unstructured Assembly code.
- **Provide a context-sensitive refactoring browser.** The user interface should allow the user to browse the refactoring catalog listed in chapter 3, and choose which one to apply, and where. The list of shown refactorings should be context-sensitive, i.e., only the refactorings applicable at the user-specified point of the current program should be shown.
- **Visualize auxiliary information.** The user interface should also present to the user the visualization of useful derivative information of the current program, which may aid the user in his analysis of the underlying code and guide him through the decompilation process. The most notorious examples of such derivative information are *Control Flow Graphs* (CFGs), call graphs, and data cross-reference tables.

## 4.2 Design decisions

This section provides the rationale behind the most important design decisions. In spite of some of the decisions made being somewhat subjective, an honest attempt to justify them all is made here. Fig. 4.1 shows the main modules of IDC.

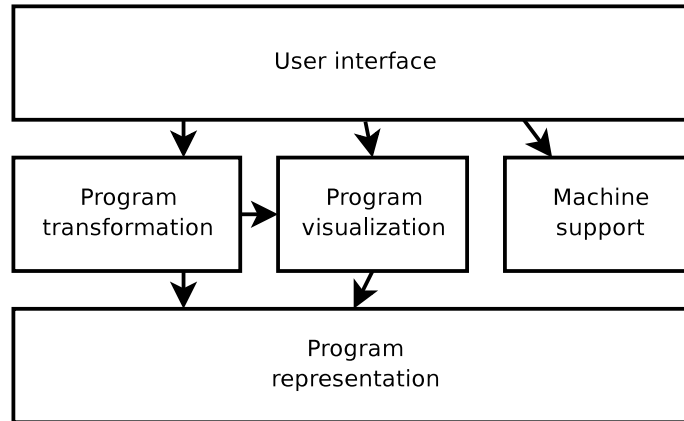


Figure 4.1: Main modules of the IDC tool

### 4.2.1 Programming language

The chosen programming language for implementing the IDC tool was Python – a dynamically typed, object-oriented language. Besides the object-oriented programming paradigm, Python also has good support for other programming paradigms, such as the imperative<sup>1</sup>, and functional<sup>2</sup> programming paradigms. The Python language is easily embeddable and extensible, allowing to progressively migrate the most time-crucial components into a more efficient foreign language (such as C or C++) if needed. Python also has excellent introspective abilities, which can be a huge code saver<sup>3</sup>. These characteristics, plus an extensive library and supporting tools (such as, parser generators, GUI designers) make it an excellent language for rapid application prototyping.

Another candidate language with similar characteristics would be the Ruby language. The Java language would also be a strong candidate, especially because some relevant third-party libraries are available in the Java language, and because choosing Java would open the door to write the tool as an Eclipse plug-in, allowing to reuse much of its code. Nevertheless, Python was the language that the author was more comfortable and more confident to provide the desired results within the available time frame.

### 4.2.2 Program representation and transformation

Between the input Assembly language code and the output C language code the program being decompiled must be represented throughout all its intermediate stages. This representation

<sup>1</sup>Python does not require to every function to be a class method or every variable to be a class attribute – it allows both regular functions and global variables.

<sup>2</sup>Python supports functional programming constructs such as `map`, `filter`, `reduce`, and list comprehensions.

<sup>3</sup>The use of Python introspective abilities within the interactive compilation tool was frequent, as is documented throughout this chapter.

will be referred as the *Intermediate Representation (IR)*<sup>4</sup>.

Different alternatives for representing the IR and its transformations were considered. These alternatives are summarized as a decision tree in fig. 4.2.

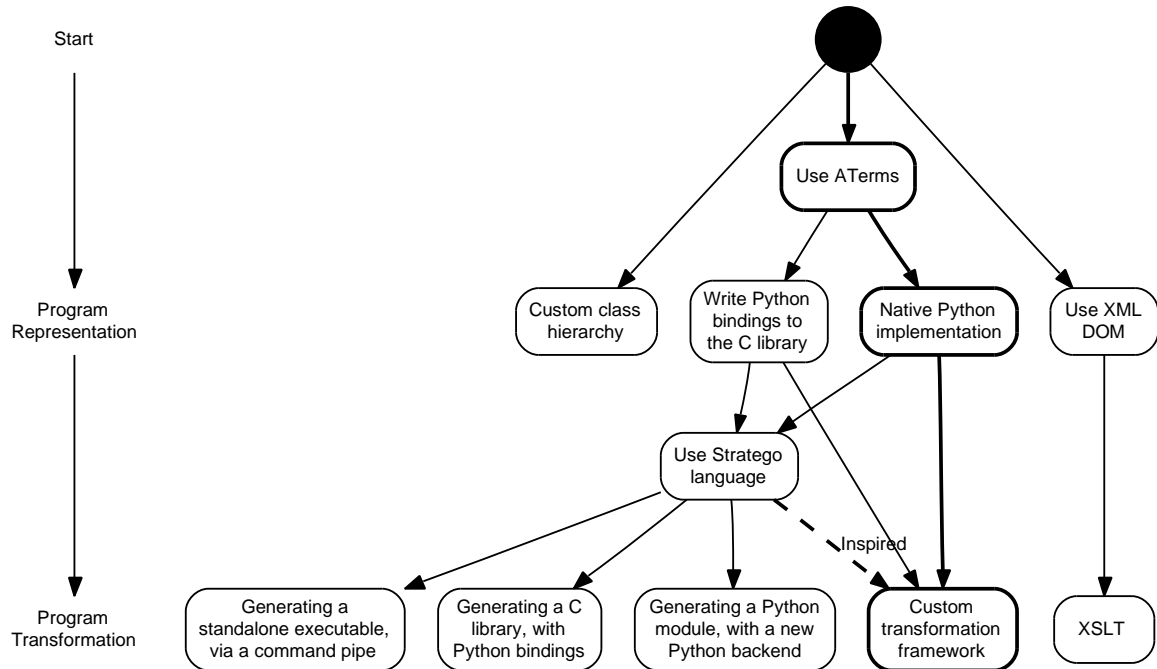


Figure 4.2: Decision tree for choosing the IR data type, and. The alternatives taken are indicated by thick lines.

### Choosing the appropriate data type for program representation

The most common representation of program is the *Abstract Syntax Tree (AST)*. Two existing tree-like data types were considered for storing the IR AST: the ATerm data-type and the XML *Document Object Model (DOM)*.

An ATerm (short for *Annotated Term*) is both an abstract data type designed for the representation and manipulation of tree-like data structures and a set of formats for exchanging this data between distributed applications [48]. An ATerm is immutable once created, and is in general represented internally as a *Direct Acyclic Graph (DAG)*, which saves both space and time by avoiding unnecessary duplications. The primary implementation of the ATerm library is in the C language, but there are implementations of the library in the Java and Haskell languages also.

The ATerm library is used by the Stratego/XT transformation tools. Stratego is a language for transformation of ASTs represented by terms, based on the paradigm of rewriting strategies [52]. Stratego/XT bundles the Stratego language with tools for parsing and code pretty-printing, among others.

XML is a format for tree-like data that has become ubiquitous in nowadays computing. Representing a program as a XML document brings the advantages of being able to use a

<sup>4</sup>Which is also the designation commonly used in compiler terminology.

wide set of well-established libraries and tools. Among the XML-related technologies is the XSLT language, which, in principle, could be used to describe program transformations.

Nevertheless, the generic-purpose XML and XSLT technologies have significant drawbacks when compared to more specialized technologies such as ATerm and Stratego. The most important drawback is that XML DOM elements are mutable and cannot be shared among several documents. Since many program transformations are difficult or even impossible to be applied in-place, the application of such transformations would imply making a modified copy of the *whole* program tree, which is inefficient. With ATerm this does not happen, being the reason why it was ultimately chosen for storing the IR.

### Integrating the ATerm library

The amount of work required to write and maintain a Python binding to the ATerm C library would likely be greater (or at least equiparable) to the amount of work required to write a native Python implementation. Also, the C ATerm library makes extensive use of pointer arithmetic, and is not very portable<sup>5</sup>. These reasons, plus the existence of implementations of the ATerm library in other programming languages, suggested that a native Python port of the ATerm library would be the best course of action.

### Integrating the Stratego language with an interactive tool

After settling with ATerm as the IR data type, integrating Stratego/XT with the interactive compilation tool was a promising possibility – the Stratego language provides the means to quickly code program transformations, and the associated literature includes several examples of how to accomplish (in Stratego) program transformations similar to some of the refactorings described in chapter 3. However, all attempts to integrate Stratego encountered major difficulties:

- **Via a command pipe:** The first difficulty is that Stratego/XT compiles transformations written in the Stratego language into programs written in C, which read an ATerm from standard input and write the transformed ATerm into standard output. But calling these Stratego generated executables from the interactive tool via a pipe was not an option, because the computational effort required for the double ATerm formatting/parsing would make the user interaction with any non-trivial program sluggish. Therefore an out-of-process integration with Stratego was out of question.
- **Via a C library:** The second difficulty is that Stratego/XT is designed to generate standalone programs. The Stratego compiler does have an option to generate libraries, but this option exists merely to support separate compilation – these libraries are meant to be used with another Stratego compiled program, and not with third party code. And modifying the Stratego C-backend to generate a standalone library (which would then be wrapped by Python bindings) seemed unfeasible for someone without deep knowledge of the Stratego C-backend internals.
- **Via a Python backend:** Although the C-backend is the most supported and complete Stratego backend, there is currently a draft of a Stratego Java backend, which generates Java classes (using the Java ATerm library) together with Java syntax definition and

---

<sup>5</sup>For example, the C ATerm library does not support 64-bit architectures at the moment.



Java code pretty-printers. So the option of constructing a similar backend for Python (using the Python ATerm library) was considered, but it was an endeavor exceedingly ambitious to be undertaken in the available time frame. Writing a new backend is, by itself, a big task, and writing a new backend for the Python language was aggravated by the fact that Python is not a context-free language – code indentation is meaningful – making its syntax specification for the Stratego/XT not a straightforward task.

From the previous points the choice of writing a custom (though Stratego inspired) program transformation framework in Python was taken.

### 4.2.3 GUI toolkit

There is a wide range of GUI toolkits available for the Python language. Narrowing to those that are cross-platform<sup>6</sup> the strongest candidates are the Tk, wxWidgets and GTK toolkits. The Tk toolkit, although part of the standard Python library, is neither a featureful or visually pleasant toolkit, when compared with the wxWidgets and GTK toolkits; being seldom used for writing graphical applications in Python. The wxWidgets and GTK toolkits are roughly equivalent in features. In the end, the choice was for the GTK toolkit. An important particularity of the GTK toolkit is its text-view widget, which allows to have marks in the text – a feature that helped significantly to solve the *pointing problem*, which is discussed in detail in section 4.3.8.

## 4.3 Architecture

This section describes the final architecture of the interactive decompilation tool.

### 4.3.1 Overall architecture

The main packages that compose the interactive tool, and their purpose are:

- **aterm** – library for representing a program AST (port of the ATerm library to Python);
- **transf** – library for creating program transformations (a loose reimplementaion of Stratego in Python);
- **ir** – IR supporting program transformations;
- **refactoring** – refactorings implementation package;
- **box** – program code pretty-printing via the Box language;
- **dot** – support for graph generation via the Dot language;
- **asm** – Assembly language parsing;
- **ssl** – parsing and translation of the Semantics Specification Language (SSL), a language for describing the semantics of processors instructions;
- **machine** – machine (processor) abstraction;

---

<sup>6</sup>Including at least Windows, Linux, and Mac OS platforms.

- ui – user interface.

Fig. 4.3 shows the package dependency diagram. These packages will be discussed in more detail below.

### 4.3.2 Program representation

The `aterm` package is an adaptation of the ATerm library [48] into the Python language.

As previously mentioned, an ATerm (short for *Annotated Term*) is both an abstract data type designed for the representation and manipulation of tree-like data structures (like program ASTs) and a set of formats for exchanging of ATerms between distributed applications.

An ATerm, in its textual representation, can be:

- an integer literal, such as 1 and -28;
- a real literal, such as 1.414 and 1E+10;
- a string literal, such as "x" and "Hello World!";
- a list of zero or more ATerms, such as [1, 0.2, "a"] and [ ];
- or a function application, consisting of a capitalized name and zero or more argument ATerms, such as `Plus(Var("x"), Int(1))`, and `True`;
- and can be optionally followed by one or more annotations ATerms, such as `Mult(1,4){Type(Int)}`, or `Sym("x"){Line(14), Col(5)}`.

ATerm annotations are non-structural information that is transparent to most operations. Annotations are kept in operations results but they do not affect the non-annotation part of the results, unless explicitly required. Annotations can be used for information such as source file name and line numbers, type checking, reference IDs, etc.

ATerms can also be matched against an ATerm pattern, or built from one. The syntax of ATerm patterns is the ATerm syntax extended with wildcards (`_`) and variables (lower-case names, such as `a` or `x`). Listings 4.1 and 4.2 illustrate how to use the Python ATerm library to make ATerms from patterns and match ATerms against patterns.

Listing 4.1: Making ATerms with the Python ATerm library

```
from aterm.factory import factory

# create a term using a pattern with wildcards
a = factory.make("Plus(_,_) ", 1, 2)
print a # it will print Plus(1,2)

# create a term using a pattern with variables
b = factory.make("Mult(x,y)", x=1, y=2)
print a # it will print Mult(1,2)
```

ATerms are immutable once created. From this it results that the manipulation of an ATerm must always be carried out by creating a derivative copy, but also that all unmodified

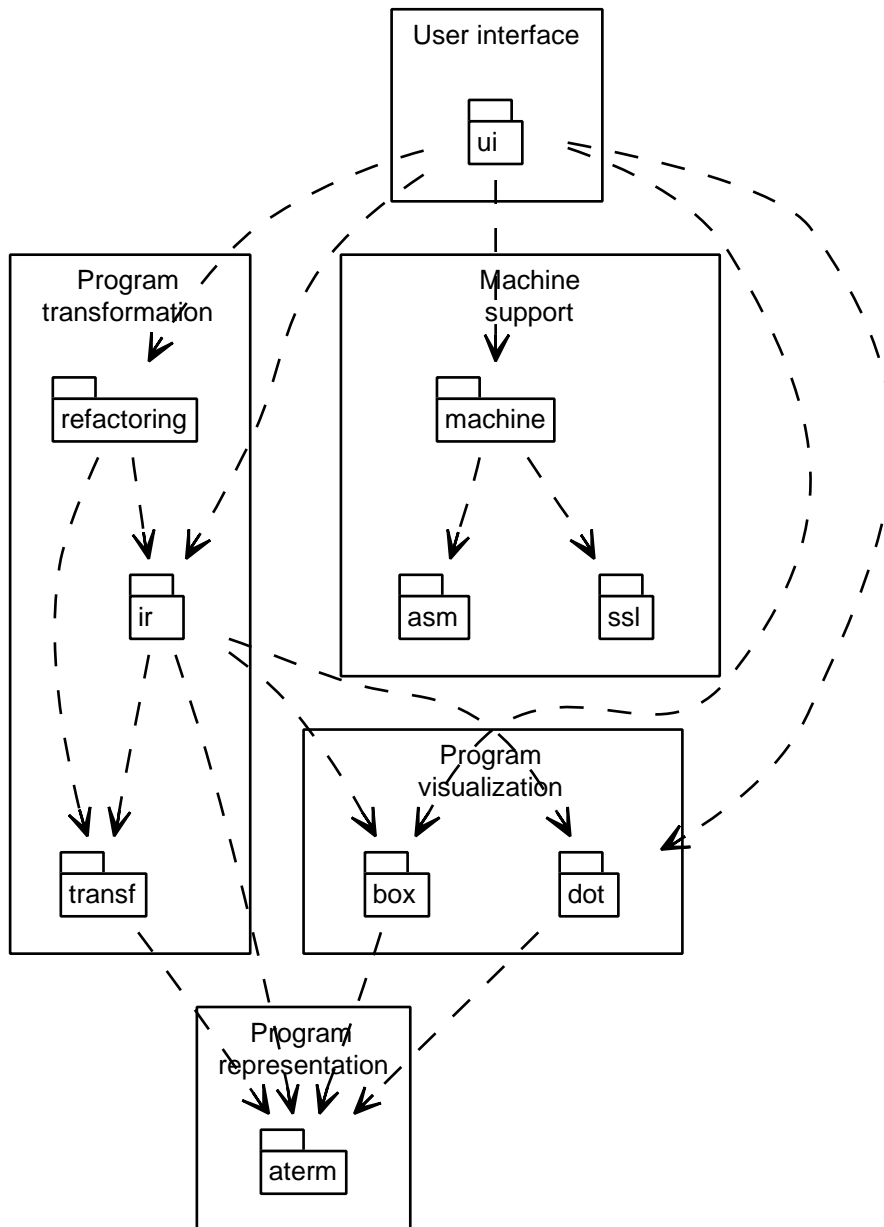


Figure 4.3: Package dependency diagram of the interactive decompilation tool

Listing 4.2: Matching ATerms with the Python ATerm library

```

from aterm.factory import factory

# create a term
t = factory.parse("Plus(Var("x"), Int(1))")

# match against a pattern with wildcards
x, y = t.rmatch("Plus(_,_)")
print x # it will print Var(x)
print y # it will print Int(1)

```

subterms can be reused in the resulting ATerm without copying them. This means that most ATerm transformations can be carried out fast and with reduced memory overhead.

The Python implementation of the ATerm library is largely inspired on the Java implementation [53], which is the only other object-oriented implementation of the ATerm library existing at the present. Nevertheless, it differs from it in a number of aspects:

- The pattern syntax is closer to the syntax used by the Stratego language (which will be discussed later), namely by supporting variables rather than just wildcards.
- Term annotations are not just label–annotation pairs, but any term.
- Internally, the Visitor design pattern [54] is extensively used, namely for comparing, matching, and making ATerms. This increases the code reuse and permits a richer set of ATerm operations. The Java ATerm library uses the Visitor pattern only for writing ATerms, while all other operations are implemented as virtual methods.
- There are no placeholder terms – an ATerm pattern is no longer an ATerm itself. Matching an ATerm against a pattern is done by combining ATerm visitors, and building an ATerm from a pattern is done by combining specialized ATerm factories. This yields a leaner ATerm data type, and eases the transformations of ATerms since the possibility of an ATerm being a placeholder during a transformation no longer exists.
- Subterm sharing is implemented, but maximal subterm sharing is not yet implemented. Although useful for large programs, the added complexity of its implementation was not justified for the simple academic examples planned for this work.

Fig. 4.4 shows the class diagram for an ATerm.

### IR schema

The ATerm data type is agnostic in respect to the meaning of a particular application term. It has no knowledge of what is the meaning of `Int`, or if the addition of  $1 + 1$  should be represented as `Add(1,1)`, `Plus(1,1)`, `Plus(Int(1), Int(1))`, or `Plus(One, One)`. That is a convention entirely up to the library caller.

For the IR it was developed a schema that could represent the program, in all its stages from near-Assembly language code to conventional C language code. Table 4.1 describes the

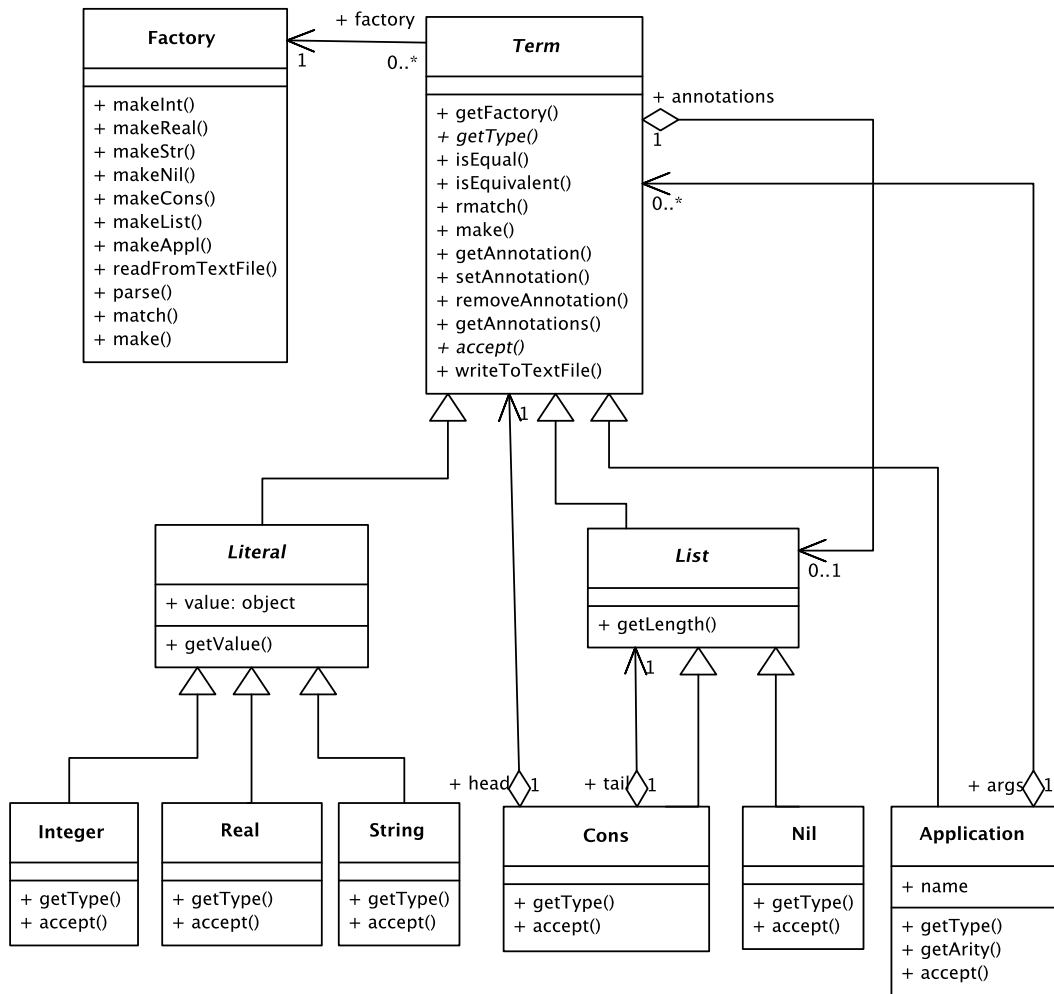


Figure 4.4: Class diagram of the aterm package

IR schema in the *Abstract Syntax Description Language* (ASDL)<sup>7</sup>. The ASDL is a domain specific language for describing the abstract syntax of compiler intermediate representations and other tree-like data structures [55], developed for the Zephyr<sup>8</sup> project.

Table 4.1: Schema of the Intermediate Representation

<pre> <i>module</i> = <b>Module</b>(<i>stmt</i>*)    <i>stmt</i> = <b>Asm</b>(<i>string</i> opcode, <i>expr</i>* operands)           <b>Assign</b>(<i>type</i>, <i>optExpr</i> dest, <i>expr</i> src)           <b>Label</b>(<i>string</i> name)           <b>GoTo</b>(<i>expr</i> addr)           <b>Break</b>           <b>Continue</b>           <b>Block</b>(<i>stmt</i>*)           <b>If</b>(<i>expr</i> cond, <i>stmt</i>, <i>stmt</i>)           <b>While</b>(<i>expr</i> cond, <i>stmt</i>)           <b>DoWhile</b>(<i>expr</i> cond, <i>stmt</i>)           <b>Ret</b>(<i>type</i>, <i>optExpr</i> value)           <b>Var</b>(<i>type</i>, <i>string</i> name, <i>optExpr</i> value)           <b>Function</b>(<i>type</i>, <i>string</i> name, <i>arg</i>*, <i>stmt</i>* body)           <b>NoStmt</b>    <i>arg</i> = <b>Arg</b>(<i>type</i>, <i>name</i>)    <i>expr</i> = <b>Lit</b>(<i>type</i>, <i>object</i> value)           <b>Sym</b>(<i>string</i> name)           <b>Cast</b>(<i>type</i>, <i>expr</i>)           <b>Unary</b>(<i>unOp</i>, <i>expr</i>)           <b>Binary</b>(<i>binOp</i>, <i>expr</i>, <i>expr</i>)           <b>Cond</b>(<i>expr</i> cond, <i>expr</i>, <i>expr</i>)           <b>Call</b>(<i>expr</i> func, <i>expr</i>* params)           <b>Addr</b>(<i>expr</i>)           <b>Ref</b>(<i>expr</i>)    <i>optExpr</i> = <i>expr</i>               <b>NoExpr</b>    <i>unOp</i> = <b>Not</b>(<i>type</i>)            <b>Neg</b>(<i>type</i>)    <i>binOp</i> = <b>And</b>(<i>type</i>)            <b>Or</b>(<i>type</i>) </pre>
---

<sup>7</sup>The schema does not make use of ASDL optional arguments because there is no such thing as a *null* ATerm – optional arguments need to be made explicit, such as the case with **NoStmt** and **NoExpr**.

<sup>8</sup><http://www.cs.virginia.edu/zephyr/>

Table 4.1: (continued)

	<b>Xor</b> ( <i>type</i> )
	<b>LShift</b> ( <i>type</i> )
	<b>RShift</b> ( <i>type</i> )
	<b>Plus</b> ( <i>type</i> )
	<b>Minus</b> ( <i>type</i> )
	<b>Mult</b> ( <i>type</i> )
	<b>Div</b> ( <i>type</i> )
	<b>Mod</b> ( <i>type</i> )
	<b>Eq</b> ( <i>type</i> )
	<b>NotEq</b> ( <i>type</i> )
	<b>Lt</b> ( <i>type</i> )
	<b>LtEq</b> ( <i>type</i> )
	<b>Gt</b> ( <i>type</i> )
	<b>GtEq</b> ( <i>type</i> )
	<i>type</i> = <b>Void</b>
	<b>Bool</b>
	<b>Int</b> ( <i>int</i> size, <i>sign</i> )
	<b>Float</b> ( <i>int</i> size)
	<b>Char</b> ( <i>int</i> size)
	<b>Pointer</b> ( <i>type</i> )
	<b>Array</b> ( <i>type</i> )
	<b>Compound</b> ( <i>type</i> *)
	<b>Union</b> ( <i>type</i> *)
	<b>FuncPointer</b> ( <i>type</i> , <i>type</i> *)
	<b>Blob</b> ( <i>size</i> )
	<i>sign</i> = <b>Signed</b>
	<b>Unsigned</b>
	<b>NoSign</b>

This schema is similar to what a C language AST schema would look like, but with minor modifications in order to be able to represent low-level Assembly code:

- A *goto* statement (**GoTo**) accepts an arbitrary address expression, rather just a label, in order to represent Assembly instructions such as jump tables.
- There is a special **asm** statement (**Asm**) for inlining Assembly code directly in C code<sup>9</sup>, in order to represent the code before the Assembly instruction translation.

### 4.3.3 Program transformation

Program decompilation and program refactoring are particular cases of program transformation. From the choice of using ATerms for the IR it follows that program transformations are

<sup>9</sup>Actually, most C compilers have a similar extension.

indeed ATerm transformations.

The `transf` package is an object-oriented framework that allows to create complex term transformations from simple blocks. It was inspired on the Stratego language, but adapted to the object-oriented paradigm<sup>10</sup>.

The basic block is a transformation – an object that attempts to transform a term within a specified context and returns the transformed term on success or raises an exception on failure. Its class diagram is show in fig. 4.5. The most basic transformations are the identity and failure transformations. The former always return the input term unmodified, while the latter always raises a failure exception.

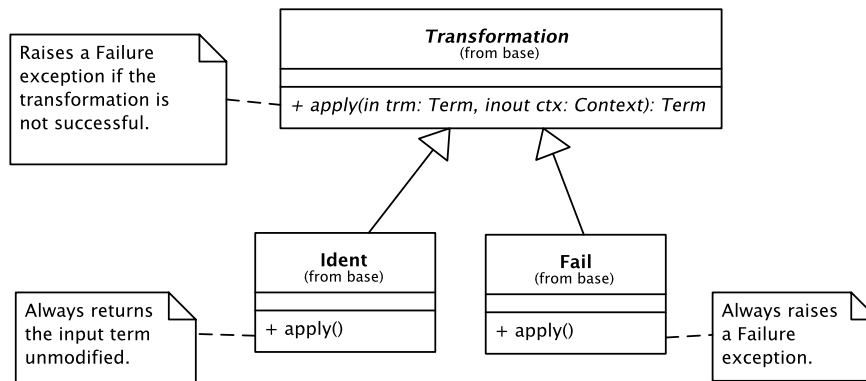


Figure 4.5: Transformation class diagram

A context is a nested, mutable dictionary-like object used to implement variable scopes, mapping variable names to variable values. Its class diagram is show in fig. 4.6. During variable lookup, variables not found in the local context scope are searched recursively in the ancestor context scopes.

A transformation can modify its context. A transformation, however, should not have side-effects other than context changes, e.g., given the same term and context a transformation should *always* give the same result<sup>11</sup>. In practice, this implies that a transformation object should be re-entrant and should hold no state. This constraint means that the information that is visible to and modifiable by a transformation is easily predictable and controllable, which is crucial to allow the bottom-up construction of huge and complex transformations from simple transformations blocks.

The most important combinators are described in table 4.2, and their class diagram is shown in fig. 4.7. The constructors for these combinator classes are protected<sup>12</sup> – the combinators are constructed by public functions that attempt simple optimizations before combining the transformations, as shown in listing 4.3.

For term manipulation there are three set of transformations: term matching, term building, and congruent term transformation. Term matching transformations will either pass terms with certain characteristics unmodified, and fail for other terms. Term building transformations will always create a new term, regardless of the term they are applied to. For

<sup>10</sup>The Stratego compiler is implemented in the Stratego language itself.

<sup>11</sup>There is actually one exception to this rule, that is a small set of transformations that ask for user input, used by refactorings.

<sup>12</sup>Note that in the Python language encapsulation is not enforced by the language, but by coding conventions instead.



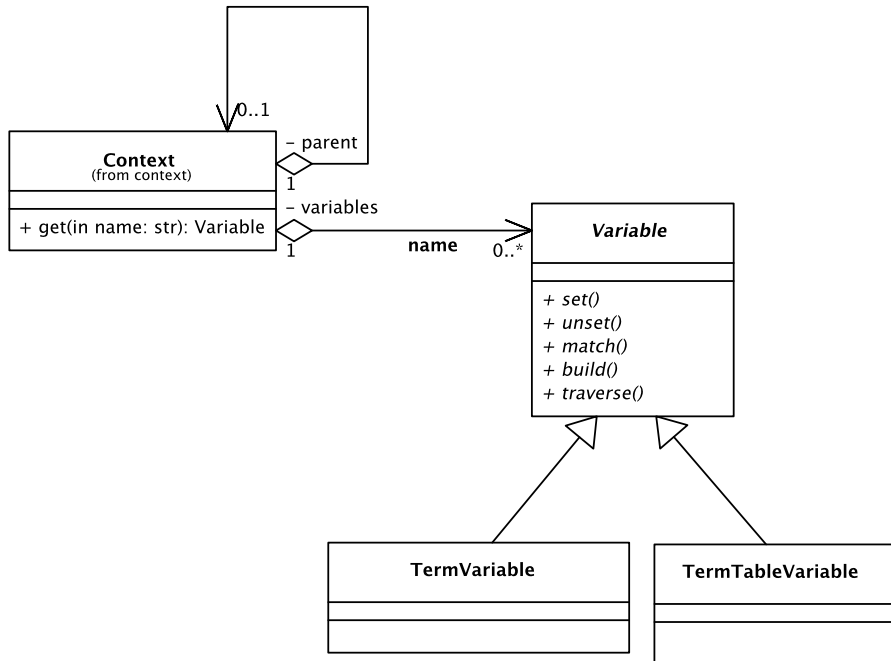


Figure 4.6: Transformation context class diagram

Table 4.2: Transformation combinators

Name	Description
Try( <i>t</i> )	Attempt the operand transformation <i>t</i> returning its result on success, or the unmodified input term on failure.
Not( <i>t</i> )	Return the unmodified input term if the operand transformation <i>t</i> fails, or fails otherwise.
Composition( <i>t</i> <sub>1</sub> , <i>t</i> <sub>2</sub> )	Apply the operand transformation <i>t</i> <sub>2</sub> to the result obtained after applying <i>t</i> <sub>1</sub> .
Choice( <i>t</i> <sub>1</sub> , <i>t</i> <sub>2</sub> )	Attempt to apply the operand transformation <i>t</i> <sub>1</sub> , falling back to <i>t</i> <sub>2</sub> if it fails.
GuardedChoice( <i>t</i> <sub>1</sub> , <i>t</i> <sub>2</sub> , <i>t</i> <sub>3</sub> )	Apply the operand transformation <i>t</i> <sub>1</sub> ; if it succeeds then <i>t</i> <sub>2</sub> is applied, otherwise <i>t</i> <sub>3</sub> is applied.

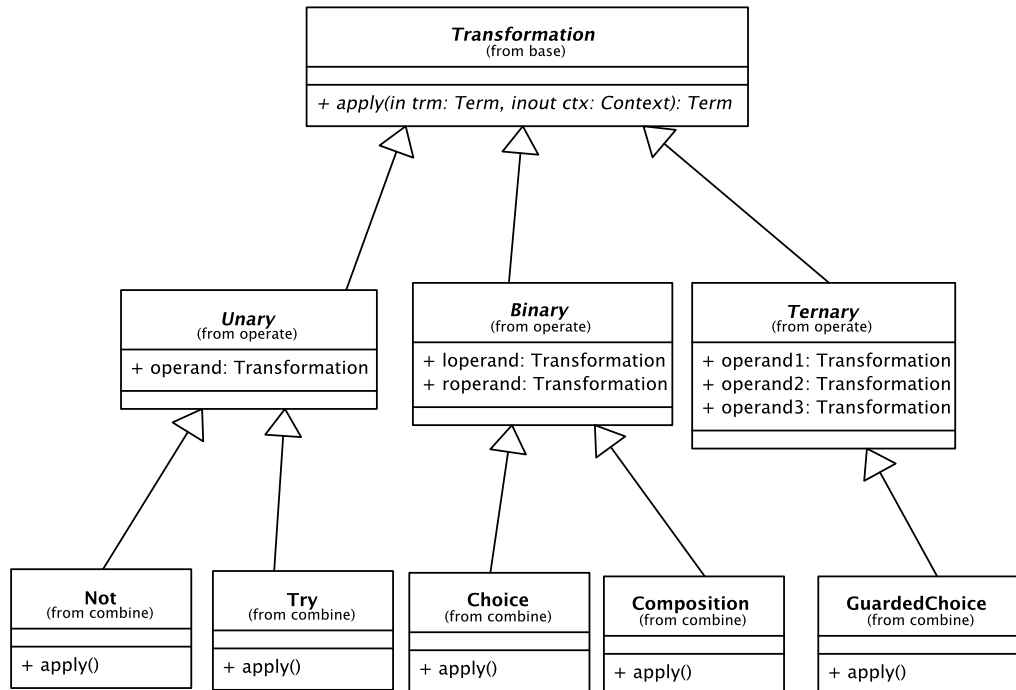


Figure 4.7: Transformation combinators class diagram

Listing 4.3: Implementation (in Python) of the Not transformation combinator

```

class _Not(operate.Unary):
    # Not combinator class
    def apply(self, trm, ctx):
        try:
            self.operand.apply(trm, ctx)
        except exception.Failure:
            return trm
        else:
            raise exception.Failure

def Not(operand):
    # Attempt to simplify the resulting transformation
    if operand is base.ident:
        return base.fail
    if operand is base.fail:
        return base.ident
    if isinstance(operand, _Not):
        return operand.operand
    return _Not(operand)
  
```

example:

- `match.Int(7)` will create a transformation that passes integer terms with value 7 unmodified, failing for every other terms, i.e., that *matches* such terms;
- `match.nil` is the transformation that matches the empty list term;
- `match.Appl("True", ())` will create a transformation that matches the `True` term application with no arguments;
- `build.Real(1.5)` will create a transformation that always returns a real term with value 1.5, regardless of the input term, i.e., that *builds* such term;
- `build.Cons(build.Str("s"), build.nil)` will create a transformation that builds the `["s"]` list term.

Congruent term transformations mix both the match and build behaviors, and exist for list terms and application terms. For example:

- `congruent.Cons(thead, ttail)` will create a transformation that matches a list construction term, and applies `thead` and `ttail` to the head term and tail term, respectively;
- `congruent.Appl("Plus", (targ1, targ2))` will create a transformation that matches `Plus` application terms, and applies `targ1` and `targ2` to the first and second argument terms, respectively;
- `congruent.ApplCons(tname, targs)` will create a transformation that matches any application terms, and applies `tname` and `targs` to the application name and the application arguments term, respectively.

Note that congruent transformations also preserve the term annotations.

Term traversal transformations are built from three base transformations:

- `All(t)` will create a transformation that applies `t` to all immediate subterms of the input term, failing if `t` ever fails;
- `One(t)` will create a transformation that applies `t` to the first immediate term it succeeds, returning immediately;
- `Some(t)` will create a transformation that applies `t` to as many immediate subterms as possible, but at least one;

These transformations use internally the congruent transformations, in order to preserve the annotations. Full tree traversals can be defined from these. For example, listing 4.4 shows the implementation of the `BottomUp` and `TopDown` traversal transformation factories, which create a transformation that applies an operand transformation to every subterm in a term, from bottom-up and top-down orders, respectively. Many more traversals are defined in the `transf.traverse` subpackage.

Another implicit concept is the transformation factory – any callable that takes a combination of terms, transformations, and other transformations factories and returns a transformation. Due to Python's dynamic typing there is no need for transformation factories

Listing 4.4: Implementation (in Python) of the BottomUp transformation traverser

```

def BottomUp(operand):
    bottomup = util.Proxy()
    bottomup.subject = combine.Composition(All(bottomup), operand)
    return bottomup

def TopDown(operand):
    topdown = util.Proxy()
    topdown.subject = combine.Composition(operand, All(topdown))
    return topdown

```

to derive a particular class. Therefore any transformation class or transformation yielding functions can be used. To help distinguish between transformations and transformation factories, the names of transformations instances start with a lower-case letter, while the names of transformations factories start with an upper-case letter.

There are higher-order transformations that take transformations factories as arguments. One example is the `Foldr` transformation of the `transf.unify` subpackage, whose implementation is shown in listing 4.5. `Foldr` takes as argument `Cons`, which is a transformation factory whose result will be recursively applied to every list construction term in a list. For example,

```
unify.Foldr(build.Int(0), arith.Add, build.Int(1))
```

will create a transformation that counts the elements in a list: it will build the zero valued integer term at the list tail, the one valued integer term at every element, and pass the intermediate results to the arithmetic addition transformation factory, `arith.Add`, which takes two integer yielding transformations as arguments.

Since manually writing transformations in Python code is exceedingly verbose, a parser for a program transformation language quite similar to Stratego was implemented, which allows to create transformations with less typing, as illustrated by fig. 4.8. Thanks to the extensive Python introspective abilities (more specifically, the ability to access a caller's global and local namespace) it is possible to seemingly mix transformation definitions with regular Python, as shown in listing 4.6. The function `parse.Transfs` parses the transformation definition given in the string argument, and adds them to the caller's local namespace. Having exactly the same effect than defining in Python.

Despite being quite similar, the implemented transformation language does differ from the Stratego language, mainly due to convenience and incompleteness. The main differences are:

- There are no term tuples. Since Stratego language is implemented in Stratego itself, it uses term tuples to pass more than one term as a transformation input; but packing and unpacking terms into term tuples would adversely impact overall performance. Therefore the transformation library was designed to not use tuples at all, using transformation factories instead. For example, while Stratego's `add` strategy takes a term tuple `(x,y)` as input and produces a term with its output, here it is supplied an `Add` transformation factory that accepts two operand transformations, whose results will then be added.

Listing 4.5: Python implementation of the FoldR transformation factory

```

def Foldr(tail, Cons, operand=None):
    if operand is None:
        operand = base.ident
    # transformation proxy for allowing recursiveness
    foldr = util.Proxy()
    foldr.subject = combine.Choice(
        # apply the 'tail' transformation at the empty list tail
        combine.Composition(match.nil, tail),
        # or call the Cons transformation factory
        Cons(
            # applying the operand to the list head
            project.head * operand,
            # and apply itself recursively to the list tail
            project.tail * foldr
        )
    )
    return foldr

```

With the transformation language

```
simplifyPlus = parse.Rule("Plus(Int(x), Int(y)) -> Int(arith.Add(!x, !y))")
```

⇕

Without the transformation language

```

simplifyPlus = scope.Local(("x", "y"),
    combine.Composition(
        match.Appl("Plus", (
            match.Appl("Int", match.Var("x")),
            match.Appl("Int", match.Var("y"))
        )),
        build.Appl("Int", (
            arith.Add(
                build.Var("x"),
                build.Var("y")
            ),
        ))
    )
)

```

Figure 4.8: Example of transformation language

Listing 4.6: Mixing the transformation language in Python code

```

def matchPair(x, y):
    return match.Cons(x, match.Cons(y, match.nil))

# matchPairOfLiterals symbol is not yet defined...
parse.Transfs('''
matchPairOfLiterals(x, y) = matchPair(?Lit(<x>), ?Lit(<y>))
''')
# ... but now it is!

matchPairOfZerosLiterals = matchPairOfLiterals(match.Int(0), match.Int(0))

```

- While Stratego's *switch-case* statement allows the use of arbitrary transformations for the cases, here it is required that the cases are regular terms. This allows for the *switch-case* to be implemented as a single hash table lookup, instead of a linear walk over all cases, substantially improving performance.
- Congruence transformations require a tilde (~) prefix. In Stratego, congruence transformations are identified with an upper-case initial, however in the implemented transformation language these are also needed for transformation factories, hence the need to distinguish between them by another prefix.
- Stratego dynamic rules are not yet implemented – a term hash table is implemented, which is sufficient for the current transformation needs.

#### 4.3.4 Machine instruction semantics

In order to decompile Assembly code it is necessary to know the meaning of each instruction of the machine's processor. A similar problem arises in machine emulation and binary-translation, and it is usually addressed with machine description languages, such as *Register Transfer Lists* (RTLs). The problem also arises in cross-platform compilers, but to a lesser extent, since a full semantic description of a machine is not usually required for the Assembly code emission.

Such machine description language is the *Semantics Specification Language* (SSL) [8], originally designed for binary translation inside the UBQT project [7], but later reused for decompilation in the Boomerang project [10]. The source distribution of either project includes the specifications of the several processors, among them the specification for Intel Pentium-class processors. Listing 4.7 shows the specification of the Intel ADDL instruction in SSL.

Integrating SSL within the tool consisted in: a) writing a SSL parser based on the existing Flex++/Bison++ parser; and b) translating into IR term patterns. Since parsing a full processor specification can be excessively time-consuming, doing so at run-time would create unnecessarily long start-up times for the tool, therefore the parsed instructions semantics are compiled into Python code, more specifically into an instruction lookup table that for each instruction name it gives a tuple containing:

- the operand names,

Listing 4.7: SSL specification of the Intel IA-32 ADDL instruction

```

ADDFLAGS32(op1, op2, result) {
  *1* %CF := ((op1@[31:31]) & (op2@[31:31]))
            | (~(result@[31:31]) & ((op1@[31:31]) | (op2@[31:31])))
  *1* %DF := ((op1@[31:31]) & (op2@[31:31]) & ~(result@[31:31]))
            | (~(op1@[31:31]) & ~(op2@[31:31]) & (result@[31:31]))
  *1* %NF := result@[31:31]
  *1* %ZF := [result = 0?1:0]
};

ADDL dst, src
  *32* tmp1 := dst
  *32* dst := dst + src
  ADDFLAGS32(tmp1, src, dst);

```

- the used temporary variable names,
- an ATerm pattern with a list of the IR statements.

Listing 4.8 shows an excerpt of that table for the Intel MOV instructions.

Listing 4.8: Excerpt of the compiled instruction lookup table for the Intel Pentium

```

insn_table = {
  ...
  'MOVB': ([ 'dst', 'src' ], [], '[Assign(Blob(8),dst,src)]'),
  'MOVL': ([ 'dst', 'src' ], [], '[Assign(Blob(32),dst,src)]'),
  'MOVW': ([ 'dst', 'src' ], [], '[Assign(Blob(16),dst,src)]'),
  ...
}

```

During Assembly translation, at run-time, each `Asm` statement term in the IR is looked up in the table, and replaced accordingly. Fig. 4.9 illustrates the whole Assembly loading and translating process.

### 4.3.5 Code pretty-printing

Code pretty-printing is done via the Box language [56], where the formatted code is represented as horizontal and vertical stacks of boxes, expressed in ATerms. The schema of the Box language is given in table 4.3, and an illustration of the whole process is given in fig. 4.10.

The use of Box as an intermediate representation allows to easily support both many input languages and output formats.

Some simplifications and additions were made to the original Box language:

- There is no spacing between boxes – all boxes are contiguous, as this is the most frequently desired case. Spaces must be specified literally.

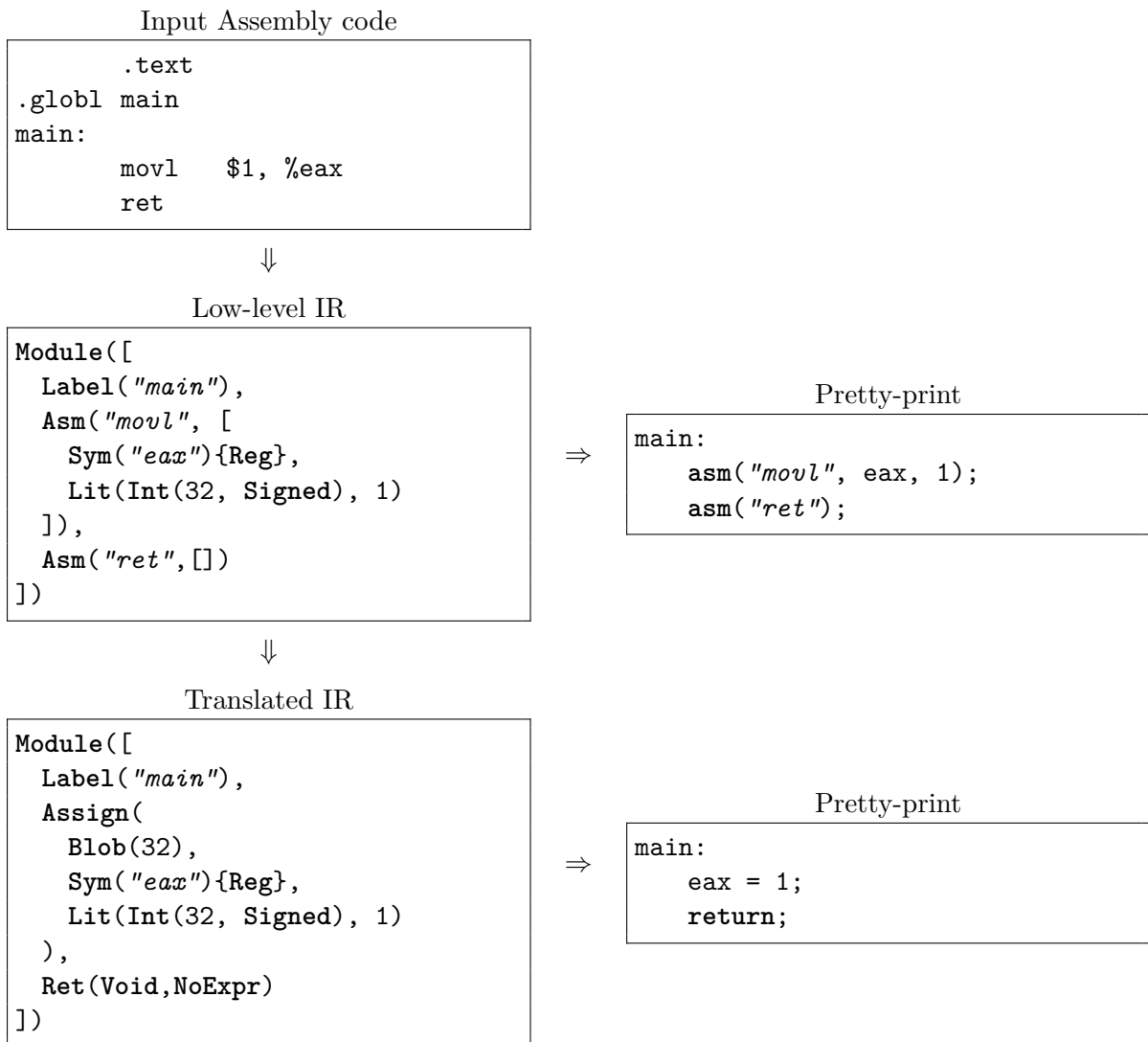


Figure 4.9: Example of the Assembly loading and translation process

Table 4.3: Schema of the Box representation

<i>box</i>	=	<i>string</i>	-- literal string
		<b>H</b> ( <i>box</i> * boxes)	-- horizontal composition
		<b>V</b> ( <i>box</i> * boxes)	-- vertical composition
		<b>I</b> ( <i>box</i> )	-- indent
		<b>D</b> ( <i>box</i> )	-- dedent
		<b>T</b> ( <i>string</i> name, <i>object</i> value, <i>box</i> )	-- tag



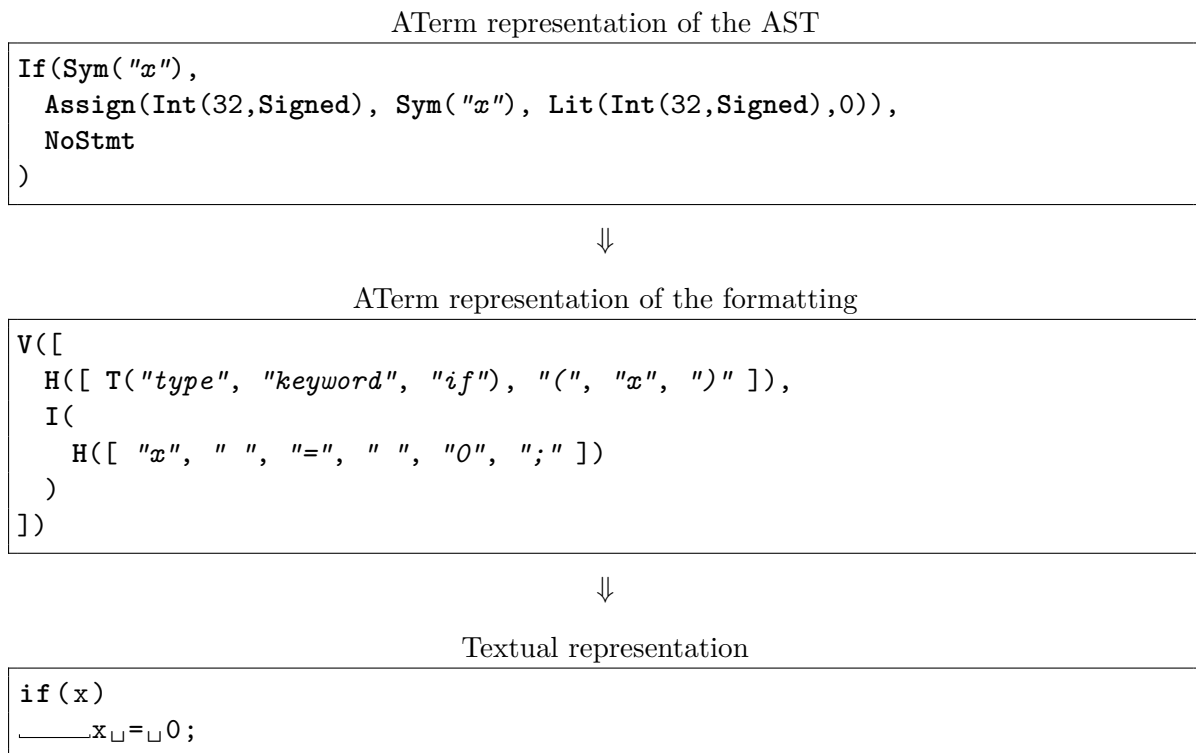


Figure 4.10: Example of code pretty-printing via Box representation

- There is a *dedent* construct, used for reducing the code indentation, which is useful for formatting C labels.
- There is a *tag* construct, used for tagging the output code with miscellaneous, transparent<sup>13</sup> information, used for transmitting syntax highlight hints, and references to the original code.

### 4.3.6 Transforming ATerms into non-ATerms

The transformation framework previously described addresses the problem of transforming ATerms into ATerms, but there is frequently the need to convert ATerms into other kind of data, such as writing them to a file stream, or converting into a concrete class hierarchy. The Visitor design pattern can be used for that end, but is not flexible enough, since a visitor provides the means to code a (single) generic function. The concept of a *Tree Walker*, widely used in parsing tools such as ANTLR is more flexible.

An ATerm walker is a class aimed to transform an ATerm as it traverses the tree. Each method of an ATerm walker behaves as a visitor, matching/transforming a subterm, while calling other methods or itself recursively in the process.

The coding of an ATerm walker is greatly simplified by the use of Python's introspective abilities and descriptors. These are present in a dispatcher – a method that takes a method

<sup>13</sup>Transparent in the sense it can be safely ignored.

prefix, and introspectively calls the walker method more adequate to handle a term, at run-time. For example, if the prefix is "foo", and the term is `Plus(1,1)` it will attempt to call the method "fooPlus"; if the term is "a text string", it will attempt to call the method "foo\_Str"; and so on.

Listing 4.9 shows an example of an ATerm walker – an excerpt of the writer for the Box language mentioned before. There is one dispatcher for each grammar production, therefore the Box language writer uses only one dispatcher. The walker for the transformation language translation uses many more.

A dispatcher also uses internally a Visitor to accomplish its task of determining the appropriate method to handle a term.

### 4.3.7 Refactoring

A refactoring has four responsibilities:

1. To identify itself via name, for use in UI menus.
2. To determine whether it can be applied to a given program, and to a particular point of the program (the current user selection in the UI).
3. To ask the user more for auxiliary input (such as new symbol names).
4. And finally, to apply itself.

Fig. 4.11 shows the refactoring class diagram. The last three responsibilities are actually implemented via ATerm transformations.

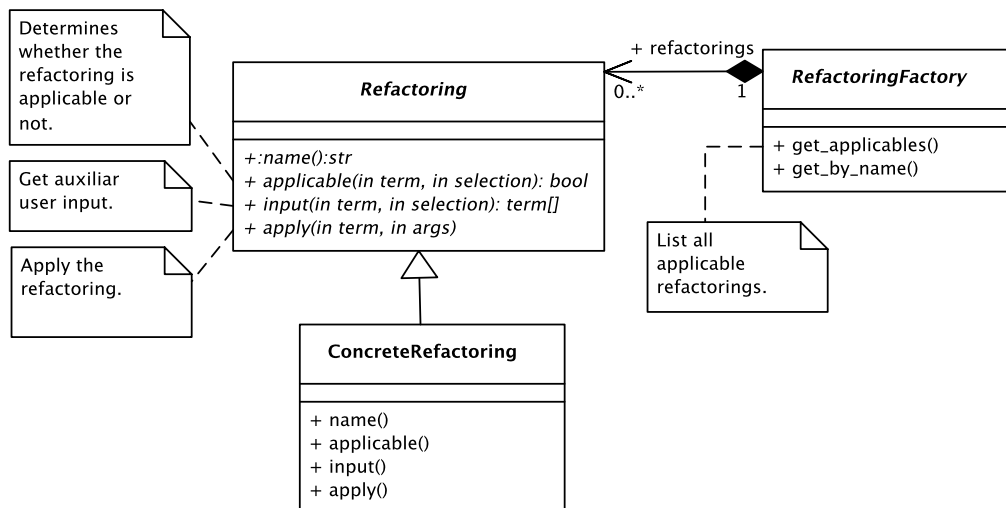


Figure 4.11: Refactoring class diagram

Input is separate from application, so that it is later possible to reproduce the refactoring without further user intervention – useful for more advanced undo mechanisms<sup>14</sup>.

The Dead Code Elimination refactoring (section 3.2.7) was implemented in a fashion similar to the example given in [26, sec. 7.2], by traversing the IR in the reverse direction

<sup>14</sup>Not yet implemented.

Listing 4.9: ATerm walker example – Box language writer

```
class Writer(walker.Walker):
    '''Writes boxes through a formatter.'''

    def __init__(self, formatter):
        walker.Walker.__init__(self)
        self.formatter = formatter

    # dispatch the term to one of the "write*" methods
    write = walker.Dispatch('write')

    def writeV(self, boxes, mode):
        if mode == HORIZ:
            raise Warning('vbox inside hbox', boxes)
        else:
            mode = VERT
        for box in boxes:
            self.write(box, mode)

    def writeH(self, boxes, mode):
        if mode == VERT:
            self.formatter.write_indent()
        for box in boxes:
            self.write(box, mode = HORIZ)
        if mode == VERT:
            self.formatter.write_eol()

    def writeI(self, box, mode):
        self.formatter.indent()
        self.write(box, mode)
        self.formatter.dedent()

    def write_Str(self, s, mode):
        if mode == VERT:
            self.formatter.write_indent()
        self.formatter.write(s)
        if mode == VERT:
            self.formatter.write_eol()

    ...
```

of control flow while keeping track of the needed variables in a hash table. In order to cope with *goto* statements, however, it was also necessary to keep a table of the needed variables at each label, and perform multiple passes into the AST until the list of needed variables at every label reaches a fixed point.

The Inline Temp refactoring (section 3.2.2) was implemented by removing the specified assignment statement and traversing the IR in the control flow direction, while replacing the specified variable until either its value is overridden or the last statement is reached. The encounter of a branch in the traversal would require a further pass from the label position, and so on, until the set of affected labels reaches a fixed point.

The refactorings for structuring control flow (section 3.3) were surprisingly simple to implement, as their implementation consist mostly of term matching and term building transformations.

### 4.3.8 User interface

The basic operation of the interactive decompilation is the successive transformation of a low-level input program into a higher-level output program. The main activity diagram is shown in fig. 4.12.

The user interface is designed around one model and multiple views (fig. 4.13). It is similar to the Model-View-Controller architecture pattern [57] but where the controller responsibilities are distributed by the model and the views. Each view (which graphically maps to a window or a widget) handles its own events, and modifies the model. Also each view registers itself with the model to receive notification signals of parts of the model that are relevant to it (such as the IR ATerm, the current selection, etc.); and whenever the model is changed the view updates itself.

#### Pointing problem

To allow the user to interact with the code, not only is necessary to visualize the code but also know which part of the code is associated with an user generated event (such as mouse-click).

The approach taken to solve this problem was to annotate the IR subterms with their respective path on the term tree, and channel this information to the Box formatter via tags. During the code box rendering this information is used to create click-sensitive areas that are cross-linked to the original terms which produced them (fig. 4.14).

Paths were chosen to identify terms because, besides uniquely identifying a term, it also allows to know whether a subterm is an ancestor, a child, before or after the term pointed by a given path, which can be helpful when implementing the refactorings.

Although this was not strictly necessary, it was chosen to annotate the subterms with their paths, since keeping track of term paths as they are being transformed would be exceedingly complex. The disadvantage is that annotating the subterms with their paths destroys the term sharing. To reduce this undesired effect, only the necessary terms are annotated: statement, expression, and type terms.

The same method was used to allow user interaction with the graphs generated by the Dot tool – by channeling the paths via the URL attribute of the Dot language and reading it back from the Dot generated image-maps.

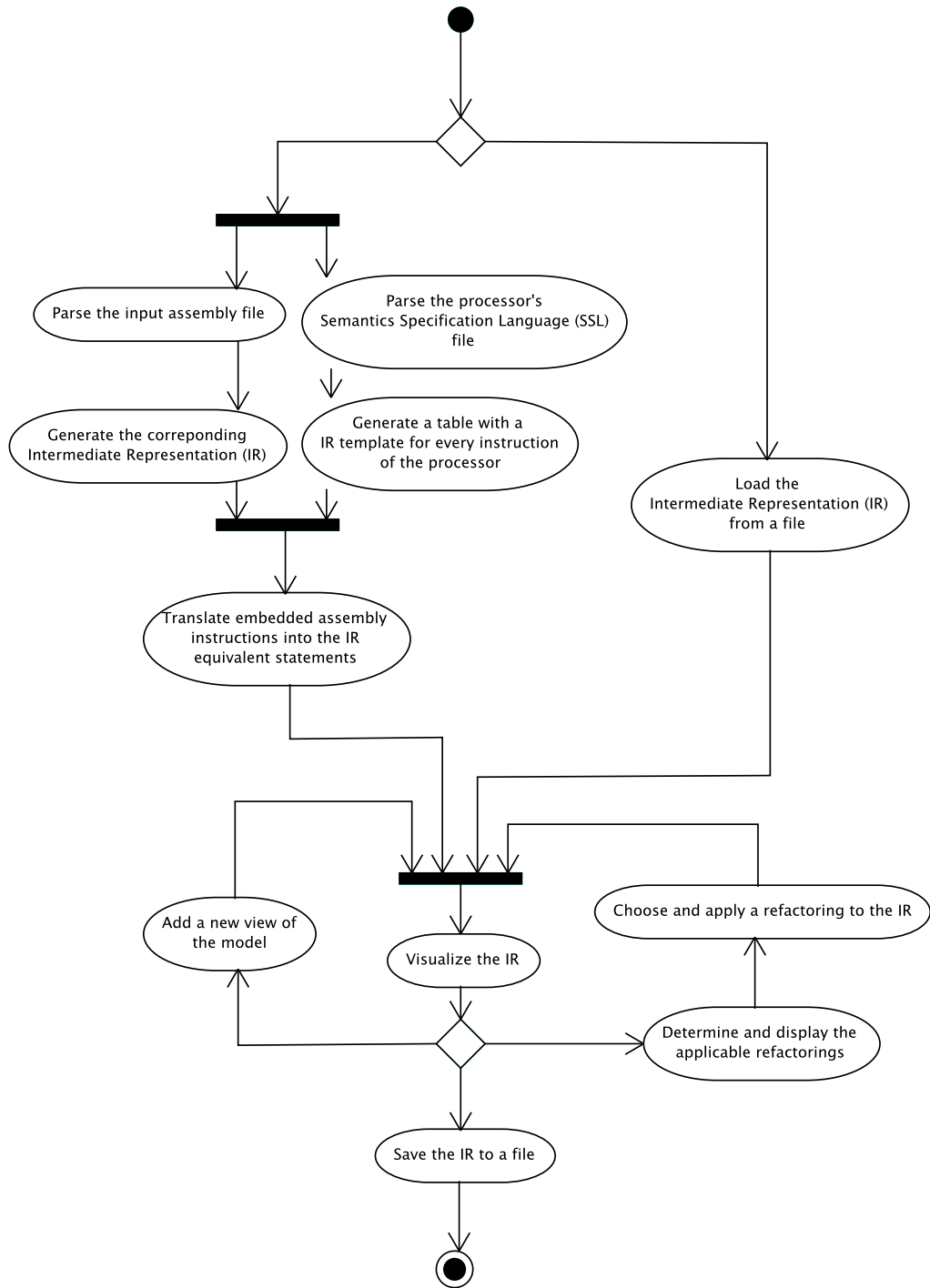


Figure 4.12: Main activity diagram

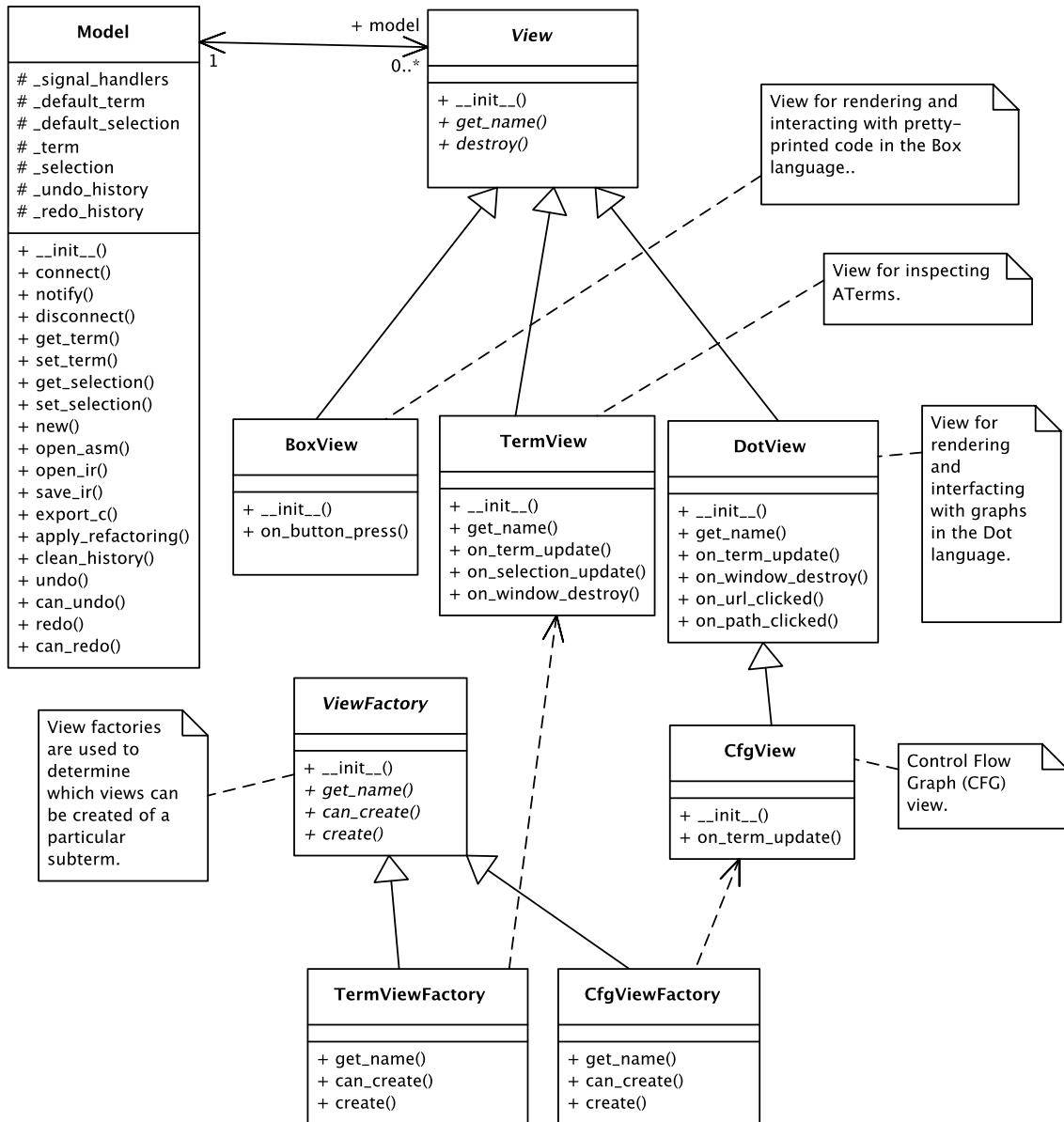


Figure 4.13: Model-View class diagram

## Initial IR

```

If(
  Sym("x"),
  Assign(
    Int(32, Signed),
    Sym("x"),
    Lit(Int(32, Signed),0)
  ),
  NoStmt
)

```

↓

## Path annotated IR

```

If(
  Sym("x"){Path([0])},
  Assign(
    Int(32, Signed){Path([0,1])},
    Sym("x"){Path([1,1])},
    Lit(Int(32, Signed), 0){Path([2,1])}
  ){Path([1])},
  NoStmt{Path([2])}
){Path([])}

```

↓

## Path annotated Box representation

```

T("path", [],
  V([
    H([ T("type", "keyword", "if"), "(", T("path", [0], "x"), ")" ]),
    I(
      T("path", [1],
        H([ T("path", [1,1], "x"), " ", "=", " ", T("path", [2,1], "0"), ";" ]))
    )
  ])
)

```

↓

## Click sensitive UI

```

if()
   =  0;

```

Figure 4.14: Path annotation for the pointing problem

**Undo**

The limited time frame of this work only allowed to implement a simple undo mechanism, whereby undo/redo is done by popping/pushing into a history stack of all intermediate versions of the IR, kept in memory.



## Chapter 5

# The IDC tool

This chapter now describes the resulting interactive decompilation (IDC) tool based on the design decisions presented in chapter 4, and how to use it with the help of an example.

### 5.1 About

The IDC tool is an interactive decompiler, where the user starts with an almost literal translation of Assembly code in C language, which he progressively decompiles by the successive application of low-level refactorings, ultimately leading to high-level C code.

### 5.2 Features

The current main features of IDC are:

- Import Intel IA32 Assembly code, in the AT&T syntax<sup>1</sup>.
- Visualize and export *quasi-C* language code.
- Provides a context-sensitive refactoring browser to the low-level refactorings listed in chapter 3.
- Visualize and manipulate the *Control Flow Graph* (CFG) and the *Abstract Syntax Tree* (AST) of the program.

### 5.3 Availability

The IDC tool source code, installation instructions, and examples are available from the IDC website, at <http://paginas.fe.up.pt/~mei04010/idc/>.

### 5.4 Tutorial

This section gives a tutorial on how to use the interactive decompilation tool to decompile a very simple Assembly program.

---

<sup>1</sup>The AT&T Assembly syntax is the one normally generated by the `gcc` compiler for the Intel IA32 architecture

### 5.4.1 Main window

The main program of the interactive compilation is the `idc.py` file. After starting it the main window appears (fig. 5.1). The tool can load either Intel IA32 Assembly files (`*.s`), or previously saved *Intermediate Representation* (IR) in textual ATerm format (`*.aterm`).

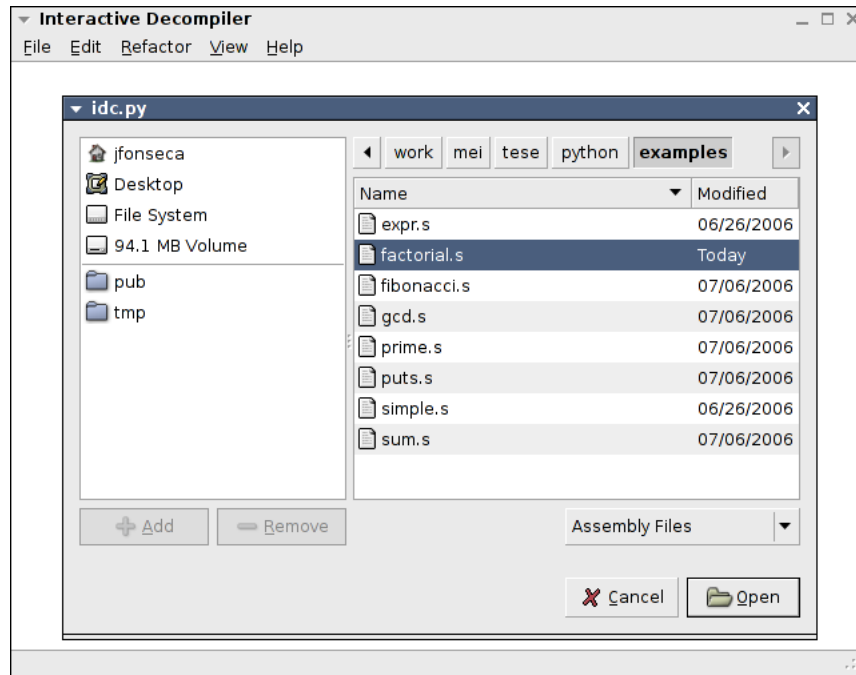


Figure 5.1: Tool main window

In the `examples` subdirectory of the source distribution are included sample Assembly files generated from C sources via the `gcc` compiler. From the *File* menu, let's open the `factorial.s` Assembly file (listing 5.1). The Assembly file is parsed and translated into the IR, and a pretty-printed view of the IR with syntax highlighting appears on the main window (fig. 5.2).

Either from the *Refactor* top-level menu, or from right-clicking on the code, a context-sensitive menu with a list of possible refactorings will appear (fig. 5.3).

Other views of the IR available from the *View* menu – at the moment, the *Control Flow Graph* (CFG) view (fig. 5.4) and the internal term view are available (fig. 5.5). Both views are *linked* with the main view, i.e., clicking in a CFG node or a term will select the respective code in all views. It is also possible to right-click on a CFG and access the *Refactor* pop-up menu from the CFG view.

### 5.4.2 Extract function prototype

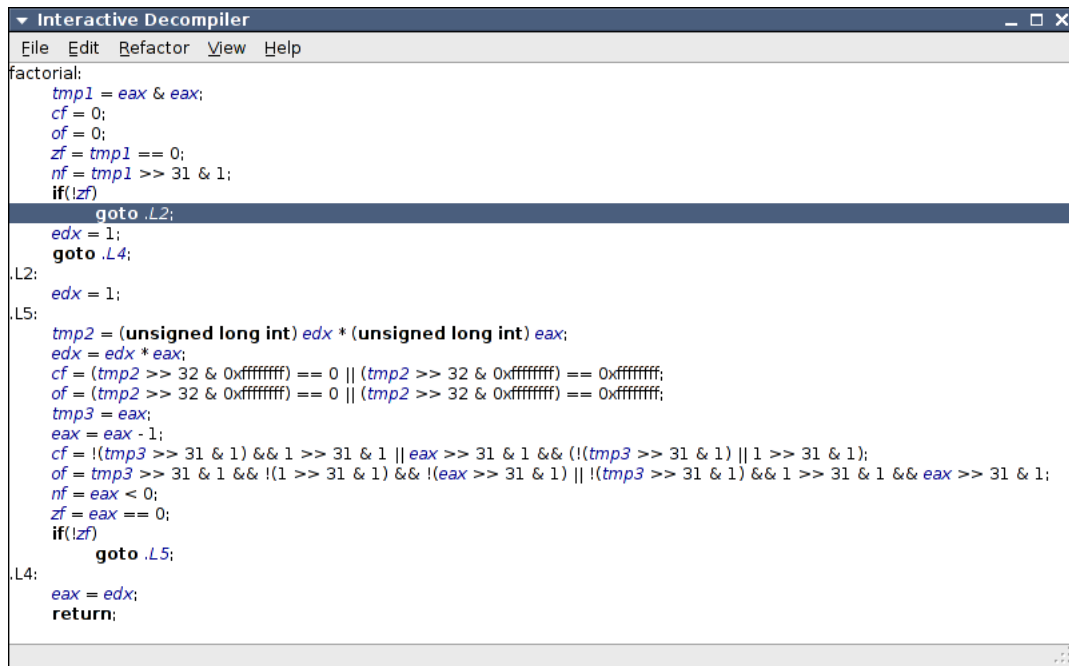
The first step to reverse engineer `factorial.s` is to extract the function. Many refactorings operate on a function scope, so it is imperative for the function signature to be reversed engineered by then. This can be accomplished by right-clicking on the `factorial:` label and choosing the Extract Function refactoring (p. 18). A function named `factorial` containing the statements between the label and the return statement (fig. 5.6).

Listing 5.1: Example input Assembly code (`factorial.s`)

```

.file    "factorial.c"
.text
.globl factorial
.type   factorial, @function
factorial:
    testl   %eax, %eax
    jne     .L2
    movl    $1, %edx
    jmp     .L4
.L2:
    movl    $1, %edx
.L5:
    imull   %eax, %edx
    decl    %eax
    jne     .L5
.L4:
    movl    %edx, %eax
    ret
.size    factorial, .-factorial
.ident   "GCC: (GNU) 4.1.2 20060715 (prerelease) (Debian 4.1.1-9)"
.section .note.GNU-stack,"",@progbits

```



```

Interactive Decompiler
File Edit Refactor View Help
factorial:
    tmp1 = eax & eax;
    cf = 0;
    of = 0;
    zf = tmp1 == 0;
    nf = tmp1 >> 31 & 1;
    if(!zf)
        goto .L2;
    edx = 1;
    goto .L4;
.L2:
    edx = 1;
.L5:
    tmp2 = (unsigned long int) edx * (unsigned long int) eax;
    edx = edx * eax;
    cf = (tmp2 >> 32 & 0xffffffff) == 0 || (tmp2 >> 32 & 0xffffffff) == 0xffffffff;
    of = (tmp2 >> 32 & 0xffffffff) == 0 || (tmp2 >> 32 & 0xffffffff) == 0xffffffff;
    tmp3 = eax;
    eax = eax - 1;
    cf = !(tmp3 >> 31 & 1) && 1 >> 31 & 1 || eax >> 31 & 1 && (!(tmp3 >> 31 & 1) || 1 >> 31 & 1);
    of = tmp3 >> 31 & 1 && !(1 >> 31 & 1) && !(eax >> 31 & 1) || !(tmp3 >> 31 & 1) && 1 >> 31 & 1 && eax >> 31 & 1;
    nf = eax < 0;
    zf = eax == 0;
    if(!zf)
        goto .L5;
.L4:
    eax = edx;
    return;

```

Figure 5.2: Pretty-printed view of the Intermediate Representation

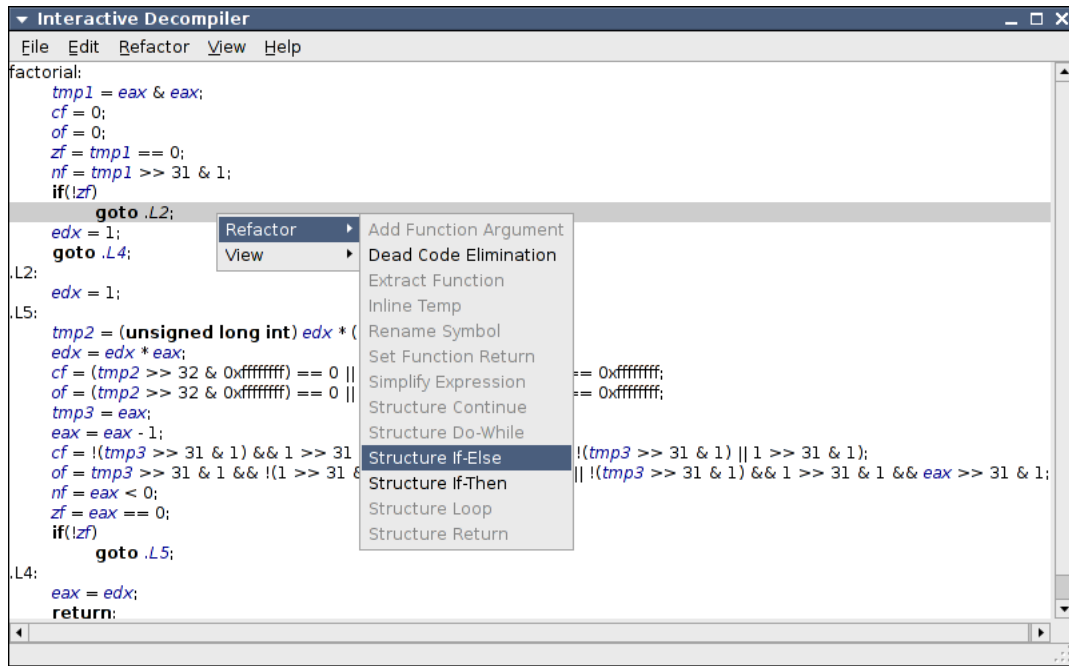


Figure 5.3: Context sensitive refactoring menu

The statement immediately before the return statement is an assignment to the `eax` register – the `eax` register is being used to pass the function return value to the caller. This is a common calling convention in code compiled for the Intel IA32 architecture. To make this explicit, and update the function return type, apply the Set Function Return (p. 19), specifying `eax` as the return symbol (fig. 5.7). `eax` will be added to the return statement, and the function return type will change from `void` to `signed int` (fig. 5.8).

The first statement inside the `factorial` function reads the value of the `eax` register – the `eax` is being used to pass an argument to the function. To make this explicit, and update the function prototype, apply the Add Function Argument refactoring (p. 20), specifying the `eax` register as the argument symbol.

Passing arguments in registers is not the most common calling convention in Intel IA32 code – usually function arguments are passed exclusively in the processor stack –, but some compilers for the IA32 architecture (such as Microsoft, Borland, and Watcom C++ compilers) have a *fastcall* option to use some registers to pass the first arguments of a function in registers, as that usually yields faster code. Other compilers (such as the `gcc` compiler), allow to completely customize the calling convention. This was intentionally the case for this Assembly file, as the current implementation of the Add Function Argument refactoring in the interactive decompilation tool does not yet support function arguments passed in the stack.

### 5.4.3 Dead code elimination

At this point the function prototype is complete, and data flow analysis can be safely performed. We can now apply the Dead Code Elimination refactoring (p. 24), to eliminate all those assignments to unused flag registers and temporaries (fig. 5.10). The Dead Code

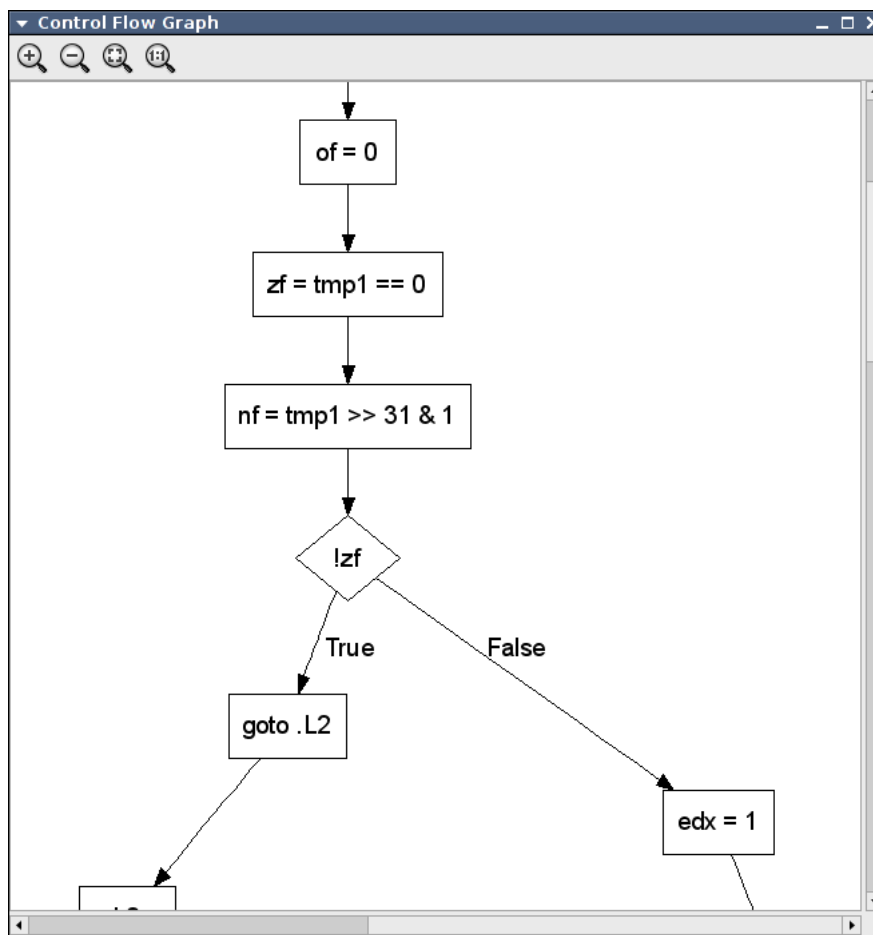


Figure 5.4: Control Flow Graph view

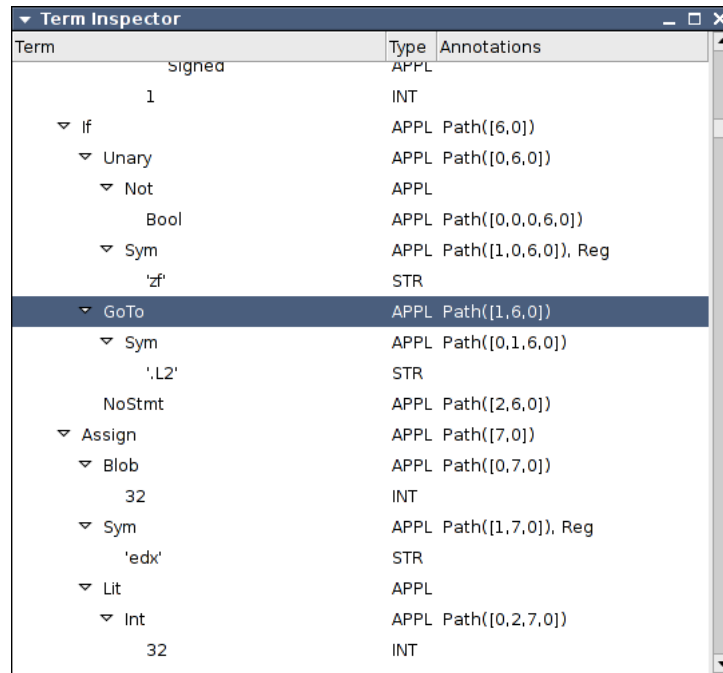


Figure 5.5: Term Inspector view



Figure 5.6: Code after applying the Extract Function refactoring

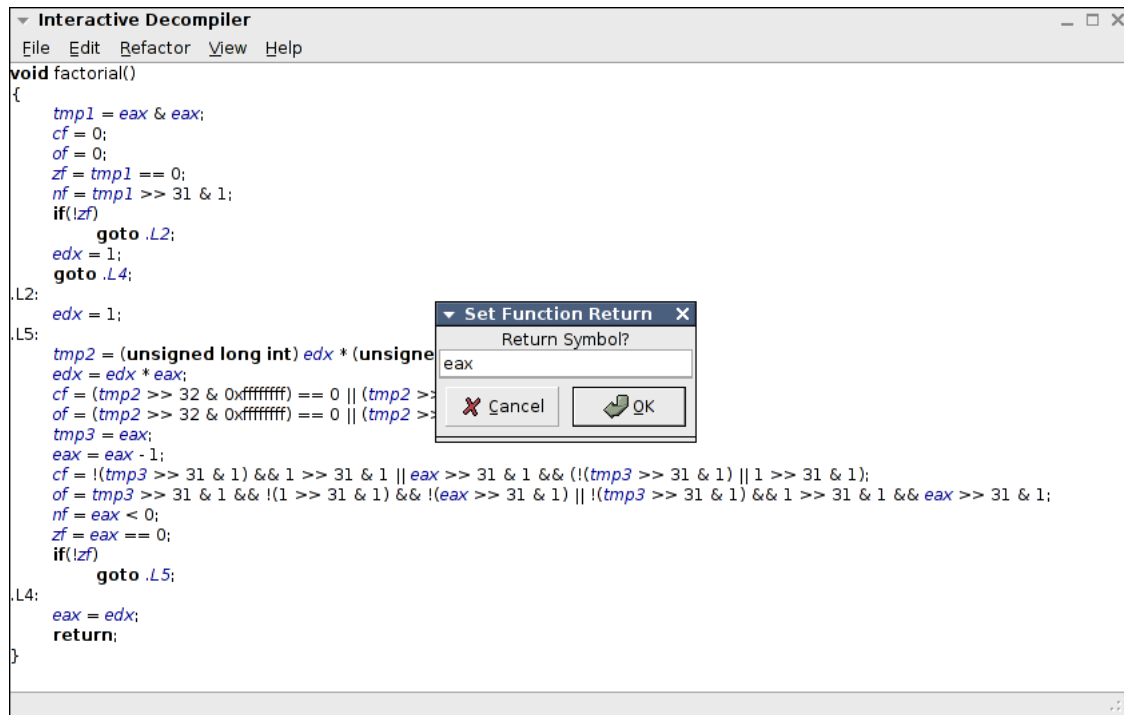


Figure 5.7: Specifying the return symbol for the Set Function Return refactoring

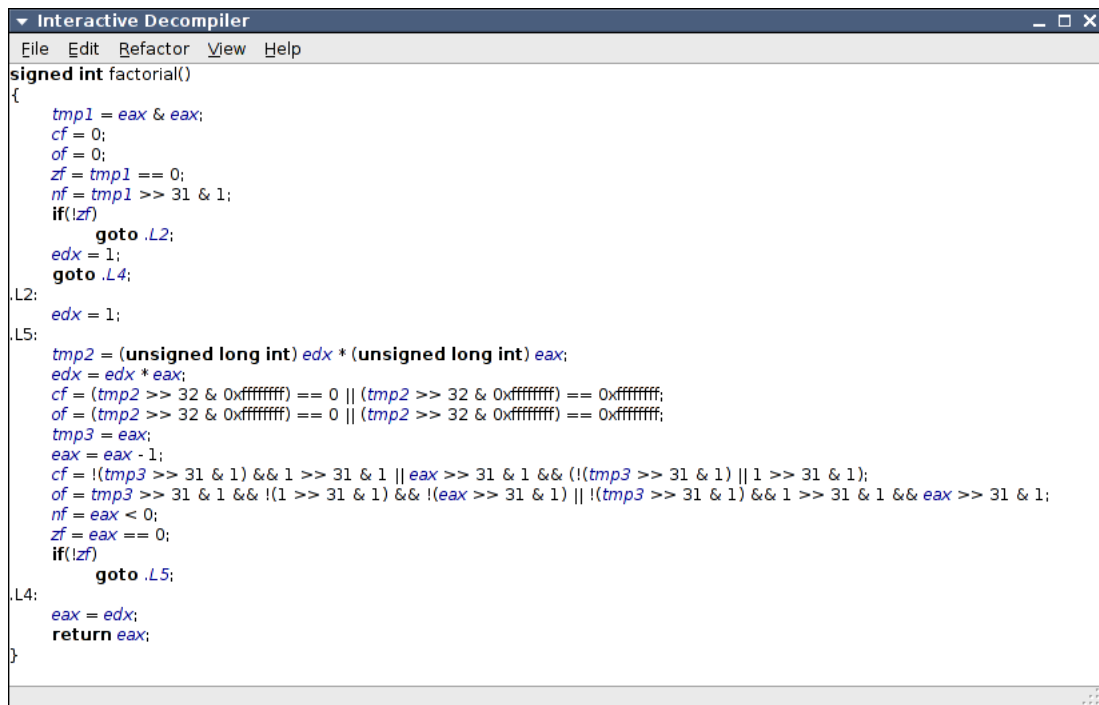
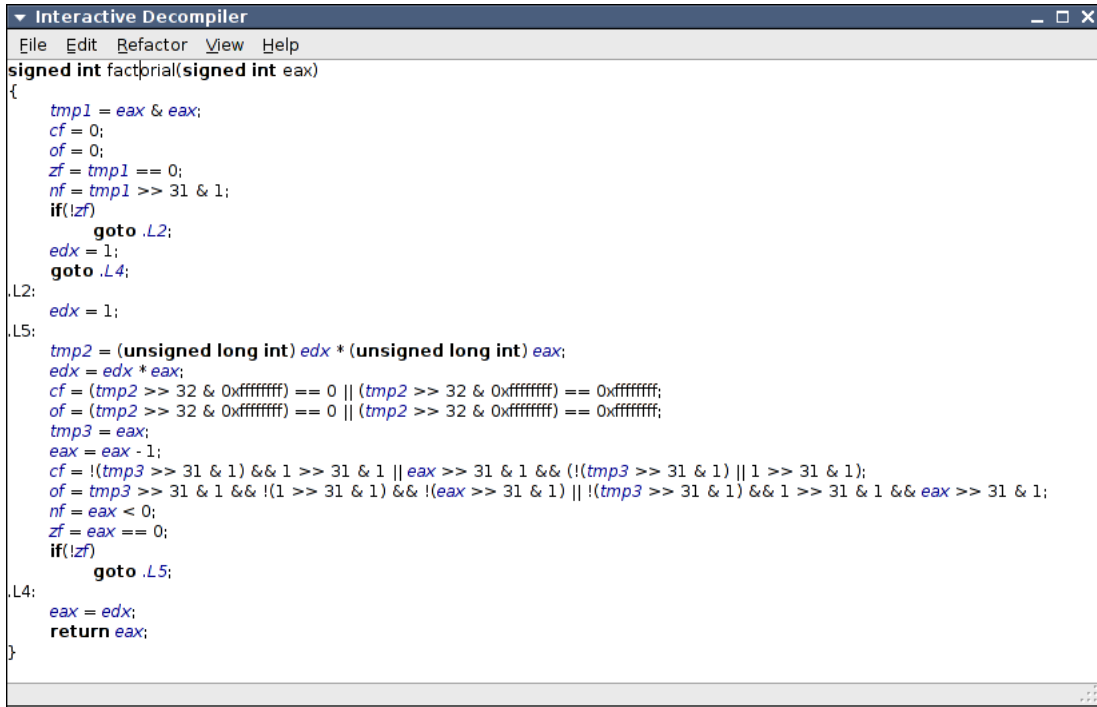


Figure 5.8: Code after applying the Set Function Return refactoring



```

Interactive Decompiler
File Edit Refactor View Help
signed int factorial(signed int eax)
{
    tmp1 = eax & eax;
    cf = 0;
    of = 0;
    zf = tmp1 == 0;
    nf = tmp1 >> 31 & 1;
    if(!zf)
        goto .L2;
    edx = 1;
    goto .L4;
.L2:
    edx = 1;
.L5:
    tmp2 = (unsigned long int) edx * (unsigned long int) eax;
    edx = edx * eax;
    cf = (tmp2 >> 32 & 0xffffffff) == 0 || (tmp2 >> 32 & 0xffffffff) == 0xffffffff;
    of = (tmp2 >> 32 & 0xffffffff) == 0 || (tmp2 >> 32 & 0xffffffff) == 0xffffffff;
    tmp3 = eax;
    eax = eax - 1;
    cf = !(tmp3 >> 31 & 1) && 1 >> 31 & 1 || eax >> 31 & 1 && (!(tmp3 >> 31 & 1) || 1 >> 31 & 1);
    of = tmp3 >> 31 & 1 && !(1 >> 31 & 1) && !(eax >> 31 & 1) || !(tmp3 >> 31 & 1) && 1 >> 31 & 1 && eax >> 31 & 1;
    nf = eax < 0;
    zf = eax == 0;
    if(!zf)
        goto .L5;
.L4:
    eax = edx;
    return eax;
}

```

Figure 5.9: Code after applying the Add Function Argument refactoring

Elimination could not have been applied sooner – applying before the Set Function Return refactoring would eliminate important code, as the refactoring would assume that the function had no return value, hence all assignments leading the final `eax` value would be erroneously eliminated.

#### 5.4.4 Control flow simplification

The code is now less denser and easier to follow, but the existence of `goto` statements is a hindrance to the code flow understanding. The CFG view (fig. 5.11) helps to realize the existence of an *if-then-else* statement in the first decision node (represented in the CFG by a diamond), and a loop after the second decision node.

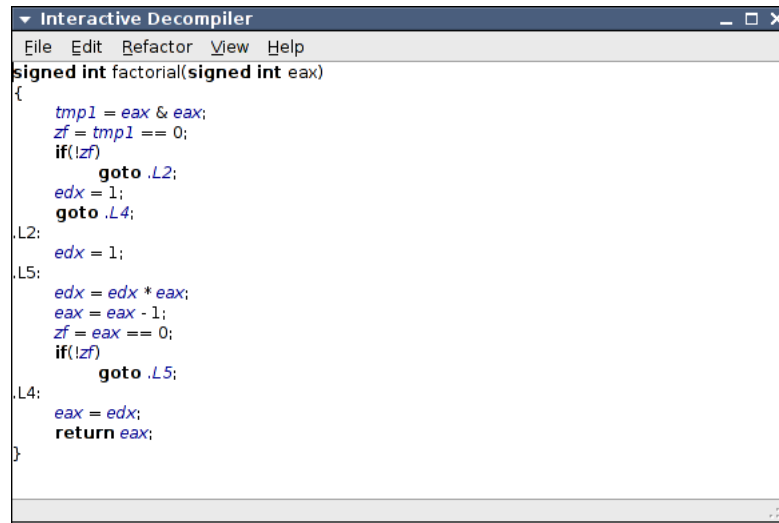
Right-clicking on the `goto .L2` statement presents the choice of structuring a *if-then* or a *if-then-else* statement. From the previous CFG inspection we will opt for the latter. Right-clicking on the `goto .L5` statement presents only the choice of structuring a *do-while* statement. After structuring these control flow statements, no more `goto` statements will remain (fig. 5.12).

#### 5.4.5 Data flow simplification

Although the control flow is now evident, the data flow is still unnecessarily complex, with an excessive use of temporary variables. These temporary variables can be eliminated with the application of the Inline Temp refactoring (p. 21) on the respective assignments (fig. 5.13).

The expressions are now more condensed, but there are some expressions resulting from compiler idiosyncrasies that can obviously be further simplified, such as the `eax & eax` into





```
Interactive Decompiler
File Edit Refactor View Help
signed int factorial(signed int eax)
{
    tmp1 = eax & eax;
    zf = tmp1 == 0;
    if(!zf)
        goto .L2;
    edx = 1;
    goto .L4;
.L2:
    edx = 1;
.L5:
    edx = edx * eax;
    eax = eax - 1;
    zf = eax == 0;
    if(!zf)
        goto .L5;
.L4:
    eax = edx;
    return eax;
}
```

Figure 5.10: Code after applying the Dead Code Elimination refactoring

simply `eax`, and `!(eax == 0)` into simply `eax != 0`. These simplifications can be performed by applying the Simplify Expression refactoring (p. 25) on the respective expressions (fig. 5.14).

#### 5.4.6 Variable renaming

Now that both the control flow and data flow are clear it is easier to understand the role of the variables, and name them. Even if the name of the function hasn't hinted, it is clear now that the purpose of this function is to compute the factorial of an integer. Using the Rename Symbol refactoring (p. 25) let's rename the argument `eax` into `n`, and the accumulator variable `edx` into `f` (fig. 5.15).

#### 5.4.7 Variable renaming

See the side-by-side comparison of the final code against the original C source from which `factorial.s` was compiled, shown in fig. 5.16.

Unfortunately it is not possible to apply the Structure While Statement refactoring due to the existence of the `f = 1` statements inside the `if` statements. The compiler duplicated this statement in both `if` branches, and it could be safely factored out, yielding the original source code, however such refactoring is not yet devised nor implemented.

## 5.5 Current limitations

As the previous tutorial shows, the interactive tool is still in a *proof of concept* state. It is not yet ready for the reverse engineering real life applications, suffering from some limitations:

- The current implementation of the Extract Function refactoring does not cope with functions split in non-contiguous code fragments.

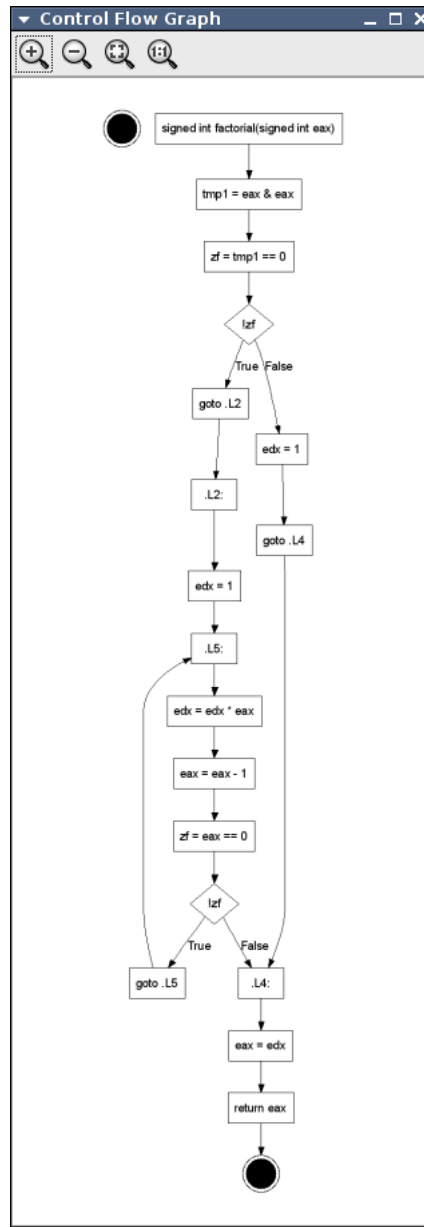
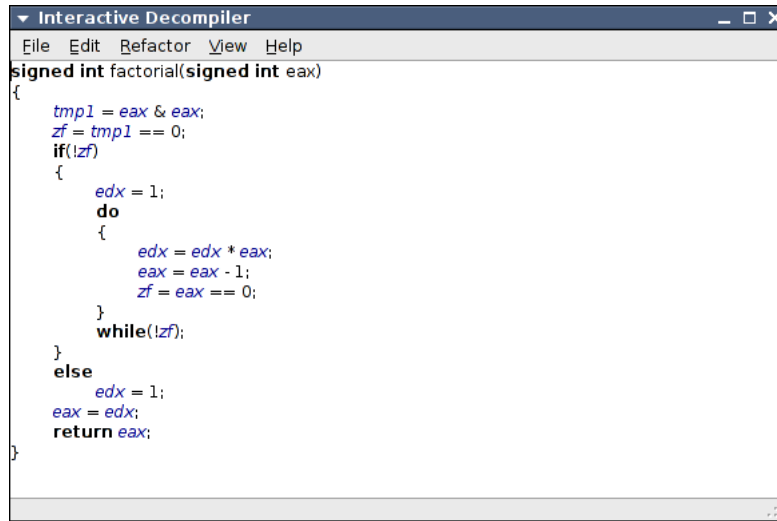
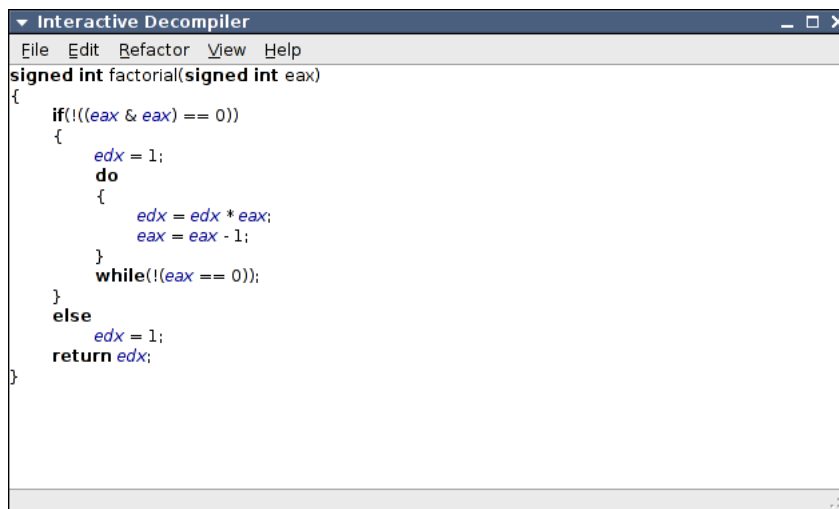


Figure 5.11: CFG after applying the Dead Code Elimination refactoring



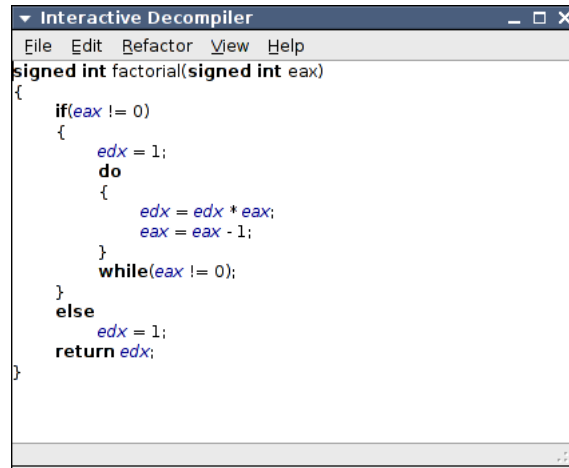
```
Interactive Decompiler
File Edit Refactor View Help
signed int factorial(signed int eax)
{
    tmp1 = eax & eax;
    zf = tmp1 == 0;
    if(!zf)
    {
        edx = 1;
        do
        {
            edx = edx * eax;
            eax = eax - 1;
            zf = eax == 0;
        }
        while(!zf);
    }
    else
        edx = 1;
    eax = edx;
    return eax;
}
```

Figure 5.12: Code after applying the Structure If Statement and Structure If-Else Statement refactorings



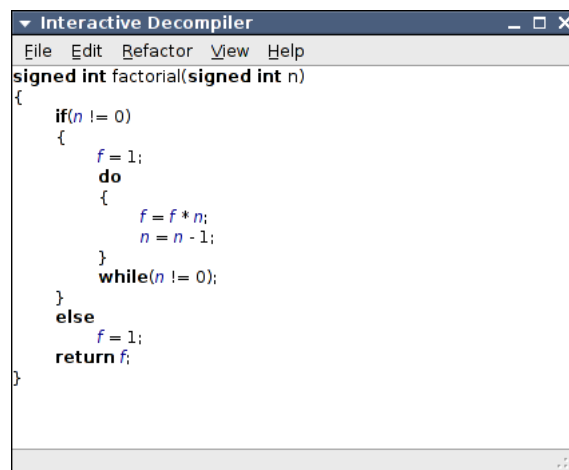
```
Interactive Decompiler
File Edit Refactor View Help
signed int factorial(signed int eax)
{
    if(!((eax & eax) == 0))
    {
        edx = 1;
        do
        {
            edx = edx * eax;
            eax = eax - 1;
        }
        while(!(eax == 0));
    }
    else
        edx = 1;
    return edx;
}
```

Figure 5.13: Code after applying the Inline Temp refactoring to the temporary variable assignments



```
Interactive Decompiler
File Edit Refactor View Help
signed int factorial(signed int eax)
{
    if(eax != 0)
    {
        edx = 1;
        do
        {
            edx = edx * eax;
            eax = eax - 1;
        }
        while(eax != 0);
    }
    else
        edx = 1;
    return edx;
}
```

Figure 5.14: Code after applying the Simplify Expression refactoring to the redundant expressions



```
Interactive Decompiler
File Edit Refactor View Help
signed int factorial(signed int n)
{
    if(n != 0)
    {
        f = 1;
        do
        {
            f = f * n;
            n = n - 1;
        }
        while(n != 0);
    }
    else
        f = 1;
    return f;
}
```

Figure 5.15: Final code after applying the Rename Symbol refactoring

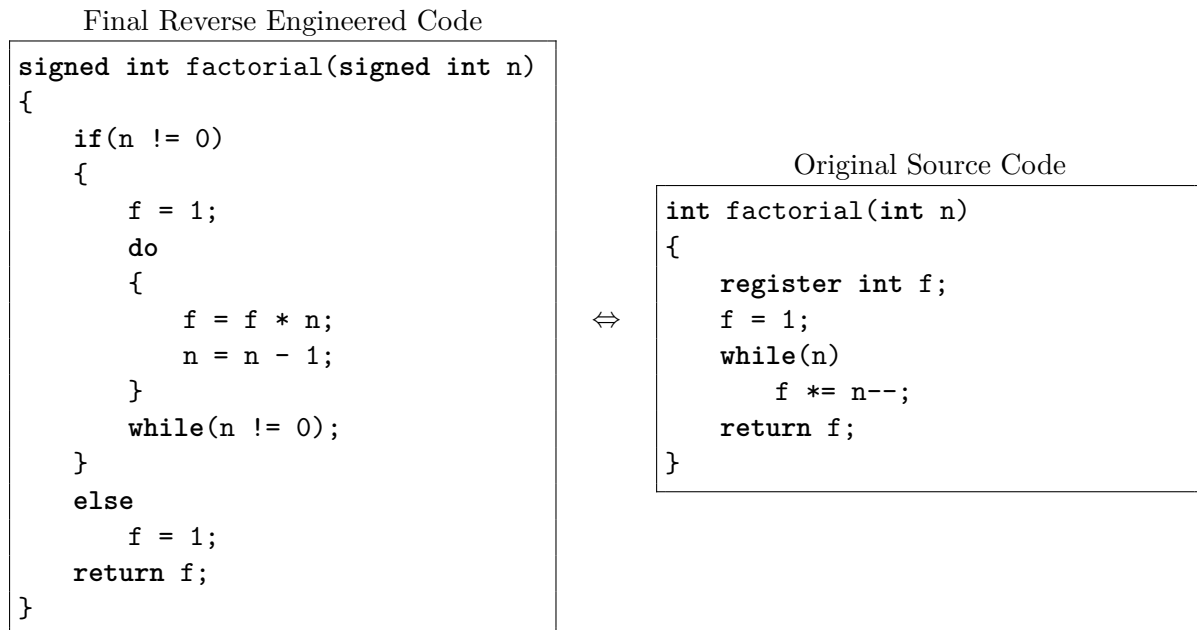


Figure 5.16: Side-by-side comparison of the example source code

- The data flow analysis performed in the refactorings focus only on register variables. This in turns means that:
  - The recovery of function arguments, function return values, and local variables from the processor stack is not yet supported by the current implementation of the concerning refactorings. Such recovery requires a data flow analysis of the stack and frame base pointer registers, which has not yet been coded. Therefore only code using function arguments, function return values, and local variables as registers is supported.
  - Variable declarations are neither produced nor observed, as everything is considered a register.
  - Global data in the input Assembly file is discarded.
- The implementation of several of the refactorings described in Chapter 3 is still missing due to lack of time.

At any rate, these limitations resulted from the limited amount of time available for the realization of this work, and never from the used methodology *per se*.

## 5.6 Extending the tool with new refactorings

This section explains how to add a new refactoring to the tool. As an example, we will create a new refactoring to globally rename a symbol.

A refactoring is constituted by three transformations: **applicable**, **input**, and **apply**.

The `applicable` transformations will tell the tool whether this refactoring can be applied or not, given the current selection and program. It should fail if it is not applicable – any other result is interpreted as the refactoring being applicable. Some refactorings can be globally applicable. For those, a simple identity transformation

```
applicable =
  id # identity transformation
```

will suffice, here shown in the transformation language. For the symbol renaming example, we will require that the current selection is a symbol, by projecting the current selection, and matching against the `Sym` term application, as

```
applicable =
  ir.path.projectSelection ; # project the current selection
  ?Sym(_) # is it a symbol?
```

The `input` transformation will ask the user for further information, and produce a list term, with all necessary arguments for applying the refactoring. If no arguments are required, a simple

```
input =
  ![] # build an empty list
```

will suffice. For the symbol renaming example, we will have the original and new symbol names as refactoring arguments, as

```
input =
  with src, dst in # declare some variables
    # get the original symbol name from the selection
    ir.path.projectSelection ; ?Sym(src) ;
    # ask the new symbol name from the selection
    lib.input.Str(!"Set Function Return", !"Return Symbol?") ; ?dst ;
    # build the argument list
    ![src, dst]
  end
```

The `apply` refactoring does the actual job of applying the refactoring to the program. It receives the original program as input, and the refactoring arguments in the `args` variable. For the symbol renaming example, it will be

```
apply =
  with src, dst in
    # extract the original argument list
    Where(!args; ?[src, dst]) ;

    # top-down traversal of all program subterms
    AllTD(
      # traverse symbol, matching its name against 'src', and replacing it by
      # 'dst'
      ~Sym(<?src; !dst>)
    )
```

```
end
```

Now it is just a matter of wrapping it up with some standard boiler plate code, and putting it in a file inside the `refactoring` subdirectory. Listing 5.2 shows the complete refactoring implementation in the Python language, with embedded transformations in the transformation language.

Listing 5.2: Complete Rename Symbol refactoring example

```
import refactoring
from refactoring._common import CommonRefactoring
import transf as lib
import ir.path

lib.parse.Transfs(r'''

    applicable =
        ir.path.projectSelection ;
        ?Sym(_)

    input =
        with src, dst in
            ir.path.projectSelection ; ?Sym(src) ;
            lib.input.Str(!"Set Function Return", !"Return Symbol?") ; ?dst ;
            ![src, dst]
        end

    apply =
        with src, dst in
            Where(!args; ?[src, dst]) ;
            ALLTD(
                ~Sym(<?src; !dst>)
            )
        end

''')

renameSymbol = CommonRefactoring(
    "Rename Symbol",
    applicable, input, apply
)
```





## Chapter 6

# Conclusions

As mentioned before, reverse engineering techniques in general (and machine code decompilation in particular) can be useful for software development and maintenance. The main focus of this work was to show how the incorporation of human interactivity – an aspect largely underused in the decompilation domain – in the decompilation process can substantially increase its usefulness.

### 6.1 Contributions

This work presented a novel and promising approach to program decompilation. By bringing together human interaction and refactoring, this approach has the potential to make decompilation a more useful and effective process.

A catalog of refactorings for low-level (near Assembly) C code was defined, where each refactoring helps making the low-level code incrementally more intelligible. So the combined and successive application of these refactorings can effectively bring a low-level machine code to a higher-level code, while retaining its semantics.

An interactive decompilation tool was developed. It automates the application of the low-level refactorings while providing an effective visualization of the program being decompiled. The use of the interactive decompilation tool and the low-level refactoring catalog was illustrated on an example that, albeit being very simple, demonstrates how the combined and successive application of the low-level refactorings can effectively recover high-level code from low-level Assembly code.

As side product of this work, a Python version of the ATerm library was developed, as well as program transformation system inspired on the Stratego language.

### 6.2 Directions for future work

Due to the limited time frame for the realization of this work, its deliverables are mostly a proof of concept.

Future work may include:

- Implement the remaining refactorings, and overcome the other current limitations of the interactive tool, described in 5.5.

- Annotate the IR with its *Static Single Assignment* (SSA) form. In the SSA form each variable is assigned exactly once. This property can simplify the implementation of several refactorings that handle variables.
- Visualize the *Program Dependency Graph* (PDG). The CFG represents the control flow relationships of a program, however, an undesirable property of a CFG is the fixed sequencing of operations that needs not to hold. Unlike the CFG, the PDG makes explicit both the data and the control dependencies for each operation in a program, without the unnecessary sequencing between operations [14, 15].
- Visualize program slices. A program slice consists of all statements and predicates of that program which might affect the value of a given variable at a given point of the program [16]. Program slicing can help the programmer understand complicated code.
- Make the interactive tool a generic refactoring browser. The interactive decompilation tool architecture is versatile enough that it can be easily adapted to become a generic refactoring browser for languages than C and other purposes than just code compilation. The missing pieces are parsers to those other languages, and to devise a way to keep the code layout and comments after refactoring.
- Target the C++ language instead of plain C. By extending the low-level refactoring catalog with refactorings to reverse engineer classes and methods it should be possible to decompile programs written in the OOP paradigm rather than just program written in the imperative paradigm.
- Improve the transformation library documentation, crystallize the transformation language syntax, and make it available as a stand alone library. The library is already generic and useful for program transformation tools other than decompilers or refactoring browsers, but some rough edges must be trimmed before it is releasable to third-parties.
- Improve the undo mechanism, allowing to undo a earlier refactoring without undoing the intermediate ones, as suggested in [40].

# Bibliography

- [1] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, Department of Computer Science, July 1994. URL <http://www.itee.uq.edu.au/~cristina/dcc.html#thesis>.
- [2] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.
- [3] The Program Transformation Wiki contributors. The program transformation wiki: Why decompilation?, 29 Apr 2005. URL <http://www.program-transformation.org/Transform/WhyDecompilation>.
- [4] The Program Transformation Wiki contributors. The program transformation wiki: Legality of decompilation, 10 Nov 2005. URL <http://www.program-transformation.org/Transform/LegalityOfDecompilation>.
- [5] Cristina Cifuentes, Mike Van Emmerik, and Norman Ramsey. The design of a resourceable and retargetable binary translator. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 280–291, October 1999. URL <http://www.itee.uq.edu.au/~cristina/wcre99.ps>.
- [6] Cristina Cifuentes and Mike Van Emmerik. The dcc decompiler. <http://www.itee.uq.edu.au/~cristina/dcc.html>.
- [7] UQBT – a resourceable and retargetable binary translator. <http://www.itee.uq.edu.au/~cristina/uqbt.html>.
- [8] Cristina Cifuentes and Shane Sendall. Specifying the semantics of machine instructions. Technical Report Technical Report 422, Department of Computer Science, The University of Queensland, December 1997. URL <http://www.itee.uq.edu.au/~cristina/papers/tr422.ps>.
- [9] IDA Pro Disassembler. <http://www.datarescue.com/idabase/>.
- [10] Boomerang decompiler. <http://boomerang.sourceforge.net/>.
- [11] Mike Van Emmerik and Trent Waddington. Using a decompiler for real-world source recovery, 2004. URL [http://www.itee.uq.edu.au/~emmerik/experience\\_long.pdf](http://www.itee.uq.edu.au/~emmerik/experience_long.pdf). This is an extended version of a paper published at the 2004 Working Conference on Reverse Engineering (WCRE 2004).

- [12] Boomerang: Still to be done. <http://boomerang.sourceforge.net/tobedone.php>, 22 Aug 2006.
- [13] CodeSurfer. <http://cayuga.grammatech.com/products/codesurfer/>, 2006.
- [14] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [15] Susan Horwitz and Thomas W. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411, May 1992. URL <http://www.cs.wisc.edu/wpis/papers/icse92.ps>.
- [16] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990. URL <http://www.cs.wisc.edu/wpis/papers/pldi88.retrospective.pdf>.
- [17] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 19, pages 11–20, December 1994. URL <http://www.cs.wisc.edu/wpis/papers/fse94.ps>.
- [18] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 20, pages 41–52, 1995. URL <http://www.cs.wisc.edu/wpis/papers/fse95b.pdf>.
- [19] CodeSurfer/x86. <http://cayuga.grammatech.com/research/cs-x86/>, 2006.
- [20] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. CodeSurfer/x86 – a platform for analyzing x86 executables. In *Proceedings of the International Conference on Compiler Construction*, April 2005. URL <http://www.cs.wisc.edu/wpis/papers/cc05-tool-demo.pdf>. (tool demonstration paper).
- [21] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23, 2004. URL <http://www.cs.wisc.edu/wpis/papers/cc04.pdf>.
- [22] The Program Transformation Wiki contributors. The program transformation wiki: Machine code decompilers, 4 Aug 2005. URL <http://www.program-transformation.org/Transform/MachineCodeDecompilers>.
- [23] Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005. doi: 10.1016/j.jsc.2004.12.011. Special issue on Reduction Strategies in Rewriting and Programming.
- [24] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. J. van Leeuwen, ed., 1990.
- [25] Sergio Antoy and John D. Gannon. Using term rewriting to verify software. *Software Engineering*, 20(4):259–274, 1994. URL <http://web.cecs.pdx.edu/~antoy/homepage/publications/tse94/paper.pdf>.

- [26] Martin Bravenboer, Arthur van Dam, Karina Olmos, and Eelco Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69:1–56, 2005.
- [27] Terence J. Parr. An overview of SORCERER: A simple tree-parser generator, 1994. URL <http://wwwantlr.org/papers/sorcerer.ps>. Poster paper in the International Conference on Compiler Construction 1994.
- [28] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.
- [29] D. E. Knuth. The genesis of attribute grammars. In *Proceedings of the international conference on Attribute grammars and their applications*, pages 1–12, 1990.
- [30] Matthijs Kuiper, Doaitse Swierstra, Marteen Pennings, Harald Vogt, and João Saraiva. Lrc: A purely functional, higher-order attribute grammar based system. <http://www.di.uminho.pt/~jas/Research/LRC/lrc.html>.
- [31] João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999. URL <ftp://ftp.cs.uu.nl/pub/RUU/CS/phdtheses/Saraiva/thesis.pdf>.
- [32] The Program Transformation Wiki contributors. The program transformation wiki: Transformation systems, 05 Jul 2005. URL <http://www.program-transformation.org/Transform/TransformationSystems>.
- [33] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992. URL <ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z>.
- [34] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 2000.
- [35] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004. ISBN 0321213351.
- [36] Eli Tilevich and Yannis Smaragdakis. Binary refactoring: improving code behind the scenes. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 264–273, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-963-2. doi: <http://doi.acm.org/10.1145/1062455.1062511>. URL <http://www.cc.gatech.edu/~yannis/binary-refactoring04.pdf>.
- [37] Kyle Brown. Design reverse-engineering and automated design-pattern detection in Smalltalk. Master’s thesis, North Carolina State University at Raleigh, Raleigh, NC, USA, 1996. Available from <http://www.ncstrl.org/>.
- [38] Refactoring browser. <http://st-www.cs.uiuc.edu/users/brant/Refractory/>.
- [39] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997. URL <http://st-www.cs.uiuc.edu/~droberts/tapos.pdf>.

- [40] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1999. URL <http://st-www.cs.uiuc.edu/~droberts/thesis.pdf>.
- [41] HaRe – the Haskell refactorer. <http://www.cs.kent.ac.uk/projects/refactor-fp/hare.html>.
- [42] Huiqing Li, Claus Reinke, and Simon Thompson. Tool support for refactoring functional programs. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 27–38, New York, NY, USA, 2003. ACM Press. URL <http://www.cs.kent.ac.uk/projects/refactor-fp/publications/tool-support-for-rfp.pdf>.
- [43] Huiqing Li, Simon Thompson, and Claus Reinke. The Haskell Refactorer: HaRe, and its API. In *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*, April 2005. URL <http://www.cs.kent.ac.uk/projects/refactor-fp/publications/HaRe-and-its-API.pdf>.
- [44] Ralf Lämmel and Joost Visser. <http://www.cs.vu.nl/Strafunski/>.
- [45] R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In *Proceedings of the Practical Aspects of Declarative Programming PADL 2002*, volume 2257 of *LNCS*, pages 137–154. Springer-Verlag, January 2002. URL <http://www.cwi.nl/~ralf/padl02.pdf>.
- [46] Ralf Lämmel and Joost Visser. Design Patterns for Functional Strategic Programming. In *Proceedings of the Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October 5 2002. ACM Press. URL <http://www.cwi.nl/~ralf/dp-sf.pdf>. 14 pages.
- [47] Eclipse project. <http://www.eclipse.org/eclipse/>.
- [48] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000. URL <http://www.cwi.nl/projects/MetaEnv/aterm/doc/at.pdf>.
- [49] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. *Stratego/XT Tutorial, Examples, and Reference Manual (latest)*. Department of Information and Computing Sciences, Universiteit Utrecht, Utrecht, The Netherlands, 2006. URL <http://nix.cs.uu.nl/dist/stratego/strategoxt-manual-unstable-latest/manual/>.
- [50] *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. Intel Corporation, 2001.
- [51] *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel Corporation, 2001.
- [52] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, June 2004. URL <http://www.cs.uu.nl/research/techreps/UU-CS-2004-011.html>.
- [53] H.A. de Jong and P.A. Olivier. Aterm java api. <http://www.cwi.nl/projects/MetaEnv/aterm/doc/aterm-javadoc/index.html>, 2003.

- [54] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.
- [55] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–227, Santa Barbara, October 1997. URL <http://www.cs.princeton.edu/~danwang/Papers/dsl97/dsl97-abstract.html>.
- [56] Mark G. J. van den Brand and Eelco Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996. URL <http://www.cs.uu.nl/people/visser/ftp/BV96.ps.gz>.
- [57] G. Krasner and S. Pope. A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988. URL <http://www.create.ucsb.edu/~stp/PostScript/mvc.pdf>.
- [58] Cristina Cifuentes. Structuring decompiled graphs. In *Proceedings of the International Conference on Compiler Construction*, pages 91–105, April 1996. URL <http://www.itee.uq.edu.au/~cristina/papers/cc96.ps>.
- [59] Agner Fog. Calling conventions for different c++ compilers and operating systems, July 2006. URL [http://www.agner.org/optimize/calling\\_conventions.pdf](http://www.agner.org/optimize/calling_conventions.pdf).
- [60] The Program Transformation Wiki contributors. The program transformation wiki: Is decompilation possible?, 29 Dec 2005. URL <http://www.program-transformation.org/Transform/DecompilationPossible>.
- [61] Paul Morris, Ron Gray, and Robert Filman. GOTO removal based on regular expressions. *Journal of Software Maintenance*, 9(1):47–662, 1997. URL <http://ic.arc.nasa.gov/people/filman/text/invision/nogo.pdf>.
- [62] H.A. de Jong and P.A. Olivier. Generation of abstract programming interfaces from syntax definitions. Software Engineering (SEN) SEN-R0212, Centrum voor Wiskunde en Informatica (CWI), July 2002. URL <http://www.cwi.nl/ftp/CWIreports/SEN/SEN-R0212.pdf>.