THE UNIVERSITY OF
WARWICK

AUTHOR: **Richard Warburton** DEGREE: **Ph.D.**

TITLE: **On the Generation and Analysis of Program Transformations**

DATE OF DEPOSIT: .....

I agree that this thesis shall be available in accordance with the regulations governing the University of Warwick theses.

I agree that the summary of this thesis may be submitted for publication.

I **agree** that the thesis may be photocopied (single copies for study purposes only).

Theses with no restriction on photocopying will also be made available to the British Library for microfilming. The British Library may supply copies to individuals or libraries. subject to a statement from them that the copy is supplied for non-publishing purposes. All copies supplied by the British Library will carry the following statement:

> "Attention is drawn to the fact that the copyright of this thesis rests with its author. This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's written consent."

AUTHOR'S SIGNATURE: .....

USER'S DECLARATION

1. I undertake not to quote or make use of any information from this thesis without making acknowledgement to the author.
2. I further undertake to allow no-one else to use this thesis while it is in my care.

DATE SIGNATURE ADDRESS

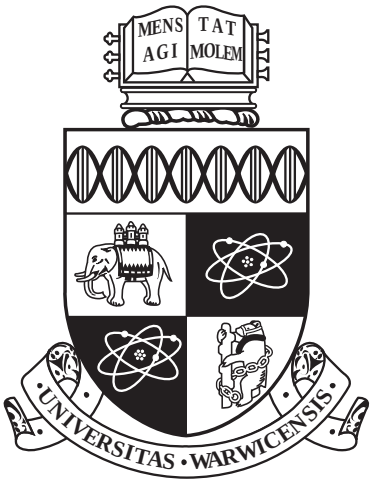
.....

.....

.....

.....

.....



On the Generation and Analysis of Program Transformations

by

Richard Warburton

Thesis

Submitted to the University of Warwick

for the degree of

Doctor of Philosophy

Department of Computer Science

August 2010

THE UNIVERSITY OF
WARWICK

Contents

Acknowledgments

I'd firstly like to thank my supervisor Sara Kalvala for her support, encouragement and advice during the course of my PhD. I'd also like to thank Ranko Lazic and Paul Curzon for being my examiners, and to Ranko especially for acting as a second adviser throughout my PhD. I'd like to thank David Lacey for initiating the TRANS project and for his research contribution to program transformations.

Within the Warwick academic community I'd also like to thank Simon Hammond, Matthew Leeke, Sadiq Jaffer, Hammad Qureshi, John Aldis, Thomas Worrall, John Oliver, Christopher West, Jessica Smith, Adam Chester and Timothy Monks for advice and discussion. My long suffering office-mates Sarah Lim, Elizabeth Hudnott and Jason Nurse deserve thanks for always providing friendly and insightful discourse and a welcome work environment.

Beyond work the moral and emotional support of my parents and family has been invaluable throughout my PhD and I credit any success I have in life to their upbringing. I'd like to thank Vic Smith, Benjamin Weber, Chris Lamb and briefly Bradley Smith for being great people to live with. I'd also like to thank Daniel Jones, Timothy Peach, Daniel Pellecchia, Amit Patel and Paul Wheeler for being solid as a rock. Finally I'd like to thank the University of Warwick Computing Society, and especially the DotA players, for being a fun group of people.

Declarations

The results and descriptions within this thesis are my own work, with the accepted contributions of my PhD Supervisor, Sara Kalvala. The underlying language combining temporal logic with rewrite rules that my results are based on was conceived by David Lacey, and was originally described in his thesis, [44].

Abstract

This thesis discusses the idea of using domain specific languages for program transformation, and the application, implementation and analysis of one such domain specific language that combines rewrite rules for transformation and uses temporal logic to express its side conditions. We have conducted three investigations.

- An efficient implementation is described that is able to generate compiler optimizations from temporal logic specifications. Its description is accompanied by an empirical study of its performance.
- We extend the fundamental ideas of this language to source code in order to write bug fixing transformations. Example transformations are given that fix common bugs within Java programs. The adaptations to the transformation language are described and a sample implementation which can apply these transformations is provided.
- We describe an approach to the formal analysis of compiler optimizations that proves that the optimizations do not change the semantics of the program that they are optimizing. Some example proofs are included.

The result of these combined investigations is greater than the sum of their parts. By demonstrating that a declarative language may be efficiently applied and formally

reasoned about satisfies both theoretical and practical concerns, whilst our extension towards bug fixing shows more varied uses are possible.

Chapter 1

Introduction

1.1 Challenges

Programming is an activity at the heart of Computer Science. The productivity of programmers has been hugely improved by the migration to high level languages and away from the assembly and machine level languages that have been used in the past. The compiler has been a key technological advance in this regard, as it allows higher level languages to be executed with improved efficiency. But in order to approach the efficiency of manually written assembly, compilers must optimize the programs that they are producing.

The abstract specification of program transformations is a unifying theme throughout this thesis. In the same way that a higher level of abstraction in programming languages allows for a higher programming productivity, abstractly specified and automatically applied program transformations increase the productivity of program transformation writers. These writers include the developers of compilers or bug fixing tools. Furthermore they make some tasks possible that were unfeasible or impractical before.

The optimization phase is an integral part of most real-world compilers, and significant effort in compiler development is spent in obtaining fast-running code. This effort must be balanced with the need to ensure that optimizations do not introduce errors into programs.

Inefficiencies in high level languages may be caused both by the nature of these languages and the programming methodologies that are in common use. For example it may be too laborious to apply an optimization by hand, or the level of abstraction used in the program may be designed to encourage maintainability, rather than efficient code. Manual optimization by source code rewriting may also take up the programmer's time, thus reducing the potential productivity advantage of a higher level language. The programming language may also be too high level to allow an optimization to be performed, or there may be some inefficiency within the translation of the higher level language down to machine code.

A framework that, in the context of real world languages, allows one to compile programs in a way that does not introduce bugs into them could address these issues. This must allow one to take a compiler optimization, in some form, and be able to apply it to a computer program as an optimization. It must also allow one to formally analyse its soundness, that is to say whether it introduces a bug into the program that it optimizes.

A related problem is that of transforming programs in order to fix bugs within them. Past research on static analysis and model checking has demonstrated the applicability of automated techniques for identifying bugs within computer programs. How to go from this information towards automatically fixing the bugs is a different and significant problem.

Bugs within computer programs are a constant challenge for software engineers.

The earlier within the development cycle that they can be identified and fixed the cheaper that fixing them can be, consequently findings bugs before the programs are tested and released may reduce software development time and costs.

The bugs that are automatically identified with the approach outlined in this thesis are those that are implicit within the source, since programs are commonly written without a formal specification of correctness. Consequently we focus on finding common bug patterns, for example the inability to release resources that have been acquired, or potential race conditions within programs.

1.2 Proposed Approach

The approach to program transformation described in this thesis follows a tradition of specifying compiler optimizations in a domain specific language. This approach uses *Temporal Logic* to describe paths through program control flow graphs upon which the program can be transformed. The programs are transformed by applying custom actions using the bindings that satisfy the temporal logic formula. The positions at which programs are transformed correspond to the nodes of the program's control flow graph.

The implementation presented within this thesis uses model checking as a method of performing static analysis over an intermediate representation of the program being optimized. Optimisation specifications are compiled into optimizing phases, which can be run within the context of an optimization framework. *Domain Specific Languages* (DSLs) are used within the implementation of our system in order to minimise the translation effort, and utilise existing research in the area of program analysis and optimization. Performance analysis is carried out using an industrially recognised benchmark.

Our bug fixing methodology is derived from the idea of using model checking

in order to identify bugs in programs. We use rewrite rules over fragments of program syntax in order to apply transformations to programs. In this sense it is a generalisation of the approach to transformations that we apply to compiler optimizations. This system also combines analysis of the source program's abstract syntax with a more fine grained semantic interpretation derived from the program's object code representation.

1.3 Overview

This thesis contributes to the state of the art in the following ways:

- It introduces an implementation that automatically generates optimizations from specifications and that can be practically used against a real world programming language. The construction of an efficient implementation has motivated the development of a novel intermediate representation of Java programs, using *Binary Decision Diagrams* (BDDs) to aid in symbolic model checking. It also introduces a method of interactively and visually rewriting the control flow graph (CFG) of Java programs. A case-study backed analysis of the performance ramifications of using our approach is also presented.

- Performing Formal Analysis is carried out on several optimizations that are commonly implemented in modern ahead of time compilers. This proves that the given optimization does not alter the semantics of the program that it is transforming.

- A methodology for transforming programs with the aim of fixing bugs is developed. A language to accompany the methodology, and a prototype tool for applying the language to `Java` programs are also described. This technique can integrate into existing programming methodologies and integrated development environments.

In the past people have shied away from large scale automated program transformations based on a domain specific language as a tool for the restructuring of existing computer programs. For example, compiler optimizations have been hand written implementations of dataflow analysis algorithms within a normal programming language, and attempts to automatically fix bugs in computer programs have been limited at best. By establishing a solid grounding for program transformations in terms of their specification, implementation and analysis this thesis completes the picture of their usage.

The rest of this thesis is structured as follows: Chapter 3 describes previous work in this area, briefly summarises background knowledge required for understanding the thesis and gives a description of the specification language used. Chapter 4 provides details of the implementation of Rosser, a system for generating compiler optimizations from the specification language that we outline. Chapter 5 focuses on the area of program transformation with the aim of automatically fixing bugs, describing the language used for transformation, examples of bugs that can be fixed using this approach and a prototype implementation. Chapter 6 contains formal semantics for the language used in Chapter 4 and example proofs that several optimizations preserve the semantics of programs that they are transforming. Chapter 7 discusses and summarises the work described in the thesis and compares results with related work.

Chapter 2

Background

2.1 Temporal Logic

Temporal logics extend *propositional logics* by introducing the notion that a proposition can be true or false depending on the state in which a system is in. A system is described by a model that determines in which states propositions are true, and in which states they are false.

There are a variety of temporal logics, characterised by the expressiveness with which they allow formulae to specify different states. For example there is the distinction between a linear time and branching time logic. In a linear time logic there is no distinction between different paths that are successors of a given state, whilst branching time logics allow this qualification.

This does not mean that all branching time logics are more expressive than linear time logics. For example the Linear Temporal Logic described by [84] can express some properties that Computational Tree Logic, a branching logic, cannot express and vice-versa. In branching time logics the model of the future is that of a tree, where one may

take multiple paths at any point.

Another point of differentiation is between path and fixed point logics. Fixed Point Logics allow one to define fixed points at which logical statements hold, whilst in a path logic the paths that can be referred to are limited to the connectives that are defined within the logic. For example the modal μ calculus introduced by [43] is a branching time fixed point logic, that has both most and least fixed points.

2.1.1 Computational Tree Logic

Computational Tree Logic [12] is a branching time path based temporal logic that is used extensively within the specification languages in Chapter 5 and Chapter 6. Computational Tree Logic is defined with respect to a *Kripke Structure* which forms its model.

Definition 2.1.1 *A Kripke Structure is a triple,* (S,$\rightarrow$,L) *of:*

- S, *A set of States*
- $\rightarrow$, *a binary relation over* S *representing transitions between states.*
- L, *A Labelling function mapping from elements of* S *to the set of atomic propositions that hold at a given state.*

One could think of a Kripke Structure as a directed graph with states annotated by atomic propositions. *Model Checking* is the problem of determining whether some model $\mathcal{M}$ at some state s satisfies a logical formulae ϕ, and usually denoted $\mathcal{M}$,$s \models \phi$.

In addition to classical predicate logic connectives such as $\wedge$, $\vee$ and $\neg$ CTL has several connectives that are defined with respect to time points. Each connective consists of a temporal quantifier and a path quantifier. Path quantifiers are either of

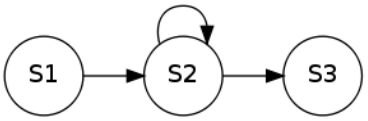


Figure 2.1: State transition system example

the form A ϕ, meaning that ϕ holds true for all paths at this point, or E ϕ, meaning that there exists a path where ϕ holds true.

There are four temporal quantifiers within CTL: neXt, Until, Future, and Global.

- *Next*—X ϕ holds true if ϕ holds true at the next state within the system.
- *Until*—ϕ U ψ holds true if ψ holds true at some state in the future, and ϕ holds true at every state until then.
- *Future*—F ϕ holds true if ϕ holds true at some future state.
- *Global*—G ϕ holds true if ϕ holds true on all future states.

Example 2.1.1 *Let* $\mathcal{M} = (\{S1,S2,S3\},\rightarrow,\mathsf{L})$. $\rightarrow$ *is shown in Figure 2.1. Let the labelling function* L *show that* $\{p\}$ *holds true at* $S1$, $\{q\}$ *at* $S2$ *and* $\{r\}$ *at* $S3$. *For this example model, the following CTL formulae hold true:*

1. $\mathcal{M},_{S1} \models p$
2. $\mathcal{M},_{S1} \models \mathtt{AX}\ p$
3. $\mathcal{M},_{S1} \models \neg\ \mathtt{AF}\ \neg p$
4. $\mathcal{M},_{S2} \models \mathtt{E}\ [\, q\ \mathtt{U}\ r\,]$
5. $\mathcal{M},_{S2} \models \mathtt{EG}\ q$

CTL is frequently presented with the additional connective *Release*. $\phi \, \mathtt{R} \, \psi$ holds true if ϕ is true on all states until ψ is true. Note that if ψ is never true, then ψ must hold true forever if $\phi \, \mathtt{R} \, \psi$ is to hold true. Another common extension is to allow backwards variants of connectives. In this instance, for a given temporal connective c the backwards operator $\overleftarrow{c}$ can be interpreted over the model $\mathcal{M} = (S, \rightarrow, L)$ as c interpreted over $\mathcal{M}'$ where $\mathcal{M}' = (S, \{(s, s') \mid (s', s) \in \rightarrow\}, L)$.

Model Checking a CTL formula for a given Kripke structure can be performed in linear time for the size of the formula and the number of states [13]. There is a strong correlation between expressiveness of a given temporal logic and the efficiency with which model checking can occur. For example the modal μ calculus—is more expressive than CTL, but requires an exponential time in the size of its input in order to perform model checking.

The ability to provide temporal quantification over paths is used extensively in the specifications that are presented in Chapter 3. This wouldn't be possible using a Linear Temporal Logic. The combination of being able to express useful properties and the efficiency in terms of model checking complexity are the two motivating reasons for choosing CTL as part of the transformation language described later in this thesis.

Chapter 3 provides a more rigorous and complete definition of CTL within the semantics of a program transformation language. Chapter 6 formally defines the semantics of a language using CTL in the context of specifying compiler optimization, within a mechanized theorem prover.

2.1.2 Applications in Computer Science

Model checking has been employed as a technique for verifying that certain properties hold of both hardware and software. More recently David Schmidt and Bernhard Steffen

recognised that there is a strong link between model checking and dataflow analysis. Dataflow analysis is used within the compiler optimization community to iteratively compute sets of values about a program. These sets can be used by the compiler in order to optimize the program that is being compiled [1; 55].

Schmidt and Steffen show that equations for dataflow analyses are expressible in modal μ Calculus in [73], and that dataflow analysis algorithms have been generated from modal logics, as described in [75].

The key to this technique is that models can be automatically refined from programs by using the Control Flow Graph of the program as the model. Each statement within the control flow graph of the program is used as a state, whilst the successor statement relation with the control flow graph corresponds to $\rightarrow$ within a Kripke Structure. Local propositions at each state can be referenced using the labelling function L. For example there might be a proposition that holds true at a state if the instruction at that state is an assignment.

This approach to program transformation is implemented in the DFA & OPT-Metaframe by [40], a toolkit designed to aid compiler construction by generating analyses and transformations from specifications, see Section 2.2.1.

2.1.3 Formal Semantics

A *Formal Semantics* aims to provide a rigorous and complete definition of the meaning of programs written in a given language. This is motivated by the concerns of both language users and implementers. A formal semantics can provide the basis for the formal analysis of many tools manipulating the language, or of formal analysis of programs written in the language itself.

The formal verification of and reasoning about metatheory surrounding language

semantics has been motivated by formal methods work in general. Challenges in this area have become less burdensome by the use of theorem provers. The POPLMARK Challenge [4] is a benchmark for proof systems in this regard.

Formal semantics are frequently characterised into different groups, based upon the manner in which they are defined.

1. **Operational:** an abstract machine is defined for a given language, and the execution of program fragments correspond to transitions between the states of this machine. The meaning of the whole program corresponds to the sequence of these transitions, often referred to as steps, between states. Operational Semantics have been used to formalise the meaning of Standard ML, as described in [54], and [59] describes a near complete subset of C in the HOL theorem prover.

2. **Denotational:** a program is described by constructing mathematical objects, referred to as *Denotations*, that represent the meaning of expressions. Denotational Semantics are referred to as being compositional in nature, meaning that the denotation of a phrase is constructed from the denotation of its subphrases. Denotational semantics is extensively discussed by [76].

3. **Axiomatic:** An axiomatic semantics is used for reasoning about programs. *Hoare Logic*, introduced in [32], provides logical axioms for each fragment of the language. These axioms are frequently assertions about the program's memory at a given state. The semantics of a program are whatever properties can be proved about it. Recent work in axiomatic semantics has provided a semantics for a significant subset of C, also by [59]. [69] reduces the burden of proofs about memory invariants through the introduction of Separation Logic.

There are other forms of language semantics, such as a *Categorical Semantics*,

however these are still not commonly defined and consequently not discussed in detail. Operational semantics is most relevant to this thesis.

Operational Semantics can be sub-categorised into *natural* or *structural* forms. A structural, or small-step, operational semantics describes the step-by-step changes to abstract machine states. A natural, or big-step, semantics defines a more abstract relationship in terms of the overall effect, rather than individual steps.

Most modern Operational Semantics presentations follow the style of [66]. The semantics are modelled as a transition relation between states. These are defined in terms of a series of inference rules that define the valid transitions for a fragment of syntax. Transitions for composite fragments of syntax, such as an `if` statement or `while` loop are defined in terms of the transitions of their component parts.

2.2 Transformation Systems

This section describes several transformation systems for computer programs that are relevant to the work in this thesis. Since Chapter 3 describes a specific transformation language, TRANS, that is used throughout the remainder of this thesis it contains a comparative analysis (Section 3.7) between these systems and TRANS. A critical review of the work in this thesis that also relates to other literature is included in Chapter 7.

2.2.1 DFA&OPT-Metaframe

The Metaframe system [40] is a toolkit for program analysis and transformation. The complete toolkit is a large-scale system that provides libraries and tools to aid the construction of industrial strength compilers. Part of this system is a transformation tool called the DFA&OPT-Metaframe toolkit. This toolkit in some respects bears a close relation to the system described here in that it also uses temporal logic as a specification

language.

There are, however, differences between the two systems. Metaframe does not use first order temporal logic to specify properties and communicate the results of the analysis to the transformation. The analyses are specified in propositional temporal logic and the resulting program analysis functions are called from a domain specific imperative programming language similar to Pascal. The user writes a temporal logic formula that is automatically converted into a program analyser, which can then be used in the writing of a compiler. During the conversion of a temporal logic formula into an analysis function a formula is partially evaluated to a model checker and optimized to produce an analysis routine similar to that in hand-written compilers (in particular with no loss of speed over the hand-written versions).

2.2.2 Genesis/GOSpeL

The Genesis system [88] implements specifications in the transformation specification language GOSpeL, where optimizations are specified through an ACTION component that transforms the program, while the TYPE and PRECOND components specific the safety conditions.

The ACTION component of the language contains operators for modifying, copying and removing statements of a program.

The GOSpeL system supports matching patterns of code on individual statements and detecting four different types of flow dependencies between nodes, identified in [61]. These dependancies are shown in Figure 2.2.

Dependency	Description
flow_dep(n,m)	a variable definition at m is used at n.
anti_dep(n,m)	a use of a variable at n is re-defined at m.
out_dep(n,m)	a variable definition at m is output at n.
ctrl_dep(n,m)	n is a conditional statement and m occurs after one of its branches.

Figure 2.2: Dependencies in GOSpeL

2.2.3 Optimix

Optimix is a graph rewriting system developed by Assmann [2; 3]. It can be used for many purposes including specifying some of the transformations described in this thesis. It is based on modifying the control flow graph of a program. Optimix analyses a program by repeatedly applying small rewrites to its graph. Each rewrite extends the graph with nodes that do not represent part of the program but capture information about the program. By using repeated application, each individual rewrite can be quite simple and succinctly specified but combined rewrites propagate quite complex information around the graph. This information, represented as extra nodes or edges can then be used to mark where a program is to be transformed.

The use of a general method makes Optimix very expressive and useful for a variety of transformation and analysis problems.

2.2.4 Rewriting

There are numerous rewriting-based transformation systems for functional languages, as typically there is no need for complex side conditions. An early implementation of an automatic transformation system can be found in the TAMPR system, which has been under development since the early '70s [8; 9]. TAMPR starts with a specification that is translated to pure lambda calculus, and rewriting is performed on pure lambda

expressions. OPTRAN is also based on rewriting, but it offers far more sophisticated pattern matching facilities [52]. TrafoLa is another system able to specify sophisticated syntactic program patterns [31]. The Glasgow Haskell compiler allows the programmer to add pragmas to code which allow extra rewrites to be performed on the program during the compilation process [35]. The system MAG [19] provides similar functionality but with more advanced mechanisms for resolving side conditions that are functional equalities.

2.3 Conclusions

Existing research on program transformation leaves open many research questions. Temporal logic specifications offer a concise approach to defining dataflow properties over control flow graphs, but no solution to fixing the bugs that are identified. The Metaframe system, described in Section 2.2.1, provides a succint specification language for program analyses but the elegance and ability to reason about it formally is limited by their use of a Pascal-like imperative programming language in order to change the program that is being optimised. The Genesis system that was introduced in Section 2.2.2 provides a more declarative system for program transformation, but reasoning about the properties of the transformations hasn't been investigated. It seems hard to reason about the Optimix system that was described in Section 2.2.3. The information that is annotated to graph nodes, that represents the results of program analyses, is the final computation of the confluence of repeatedly applied graph rewrites. This is in contrast to the nature of specifications used by Genesis, for example, where all the information about a transformation is clearly encoded into the specification.

The TRANS language is described in Chapter 3. It offers a viable approach to specifying program transformations that perform compiler optimizations, but it is

unclear to what extent languages such as TRANS can solve problems outside of the domain of compiler research.

Existing procedural transformation languages can specify a useful range of transformations, but are hard to reason about and prove useful properties of the transformations, for example whether they change the semantics of programs they are transforming. The semantics of the languages themselves is also a potentially unclear point, with a lack of machine checking or in some cases even formal definition of semantics.

Conversely declarative transformation languages have yet to be efficiently implemented. Chapter 4 addresses this issue — providing an efficient implementation for the TRANS language.

Chapter 3

A Program Transformation Language

This chapter defines the TRANS language, variants of which are used for program transformation throughout the thesis. A semantics for a small source language is presented, along with a semantics for the TRANS language specified with respect to this source language. An exploration of an alternative semantics is developed in the context of a Java like language within the Isabelle theorem prover in Chapter 6. This section also describes a catalogue of common optimizations used within modern optimizing compilers, by specifying them in the TRANS language.

Much of this work is a continuation of the research of David Lacey, who initiated the TRANS project. The TRANS language that we use is based on that outlined in [44]. The description provided in Chapter 3 was originally published in [36] as part of joint work with David Lacey and, although it has been improved and clarified by the author it should not be considered an original contribution of this thesis.

TRANS itself was originally designed for specifying and reasoning about compiler

optimizations. David Lacey explored these problems in the context of the L_0 language that is used in this chapter. Its extension and application to the real world `Java` language, as described in Chapter 4 is a specific contribution of this thesis.

[47] provides a simple technique for proving that TRANS optimizations are semantics preserving. These proofs are not mechanically checked in any kind of theorem prover. They also operate over the toy language used for descriptive purposes in Chapter 3. The formalisation of the language semantics is also very similar in terms of its overall approach. Their proofs establish a bisimilarity relation, as described in Chapter 6. The work in terms of formalising TRANS with respect to a more realistic language that is described in Chapter 6 was also not conducted previously by Lacey.

In Chapter 5 TRANS is extended to operate over source code. It can then be used to specify transformations that fix bugs within computer programs automatically. This usage wasn't conceived of within the original design of TRANS by Lacey.

Section 3.7 of this chapter contains a literature review that compares the work of Lacey from this chapter with other approaches. This builds on the descriptive literature review in Chapter 1 to include references to TRANS specifications. In Chapter 7 there is a further comparative and critical evaluation of other work in the research areas covered by this thesis and how they related to the research presented here.

3.1 The L_0 Programming Language

The methodology for specifying transformations in this thesis applies to a variety of languages; we introduce a simple imperative language L_0 to aid presentation. This toy language is standard in its behaviour; a formal semantics for L_0 can be found elsewhere [44]. The language is meant to exemplify compiler intermediate representations rather than programming languages, since that is the level at which optimizations are usually

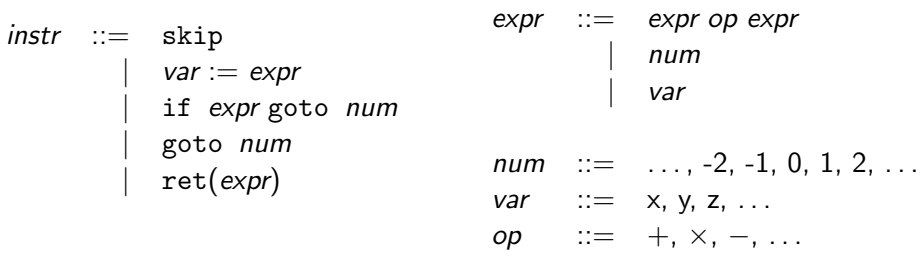


Figure 3.1: Grammar for instructions in the L_0 programming language

applied. Some features of even low-level programming languages, such as function calls, exceptions, input/output statements and pointers, have not been included in L_0.

Control structures are written using jump statements instead of looping constructs. We assume the instructions are labelled; if a `goto` or conditional statement refers to an instruction label greater than the length of the program then control jumps to the last instruction.

An L_0 program consists of a sequence of instructions where an instruction can take one of five typical forms.

Definition 3.1.1 *An instr is a single command of the form given in Figure 3.1.*

Definition 3.1.2 *A* program *of length n ($n \geq 1$) is a list $I_0, \ldots, I_{n-1}$ of instructions, where instruction I_{n-1} is the only instruction of the form* `ret`(e).

An example program in L_0, corresponding to a small `Java` program, which sets two lines of an array to be equal to the integer 1, is presented in Figure 3.2.

The control flow graph

The transformation approach introduced in this thesis uses representation of programs as *control flow graphs* (CFGs), where each node corresponds to an individual instruction

```
int gcd()
{                                   0: m := 252
    int m = 252, n = 105;           1: n := 105
    while(n != 0) {                 2: if (n == 0) goto 7
        int temp = n;               3: temp := n
        n = m % n;                  4: n := m % n
        m = temp;                   5: m := temp
    }                               6: goto 2
    return m;                       7: ret(m)
}
```

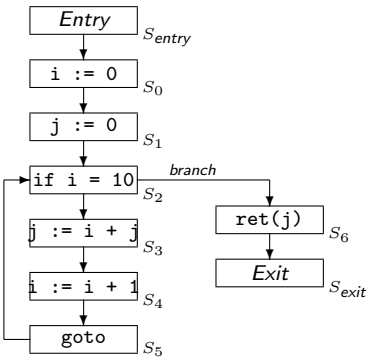
Figure 3.2: An example program in Java and its translation into L_0

in the program, except for designated *Entry* and *Exit* nodes. The edges between nodes represent *possible* steps in the program between the instructions. In order to facilitate transformations, labels are added to edges of CFGs: edges that result from the condition in a conditional statement being true are labelled $branch$, and all other edges are labelled seq to signify default sequential execution. Figure 3.3(b) shows an example CFG in which a node marked S_n corresponds to statement numbered n in the L_0 example source code. The $Entry$ and $Exit$ nodes don't have a source code equivalent.

Formally, a CFG is defined as a triple consisting of a set of nodes, an edge relation and a labelling function which labels each node with an instruction.

Definition 3.1.3 *A CFG for a program* $I_0, \ldots, I_{n-1}$ *is the tuple* $\langle Nodes,\ Edges \subseteq Nodes \times Nodes \times \{seq, branch\}, I : Nodes \to Instr \rangle$ *where:*

```
0:  i := 0
1:  j := 0
2:  if i = 10 goto 6
3:  j := i+j
4:  i := i+1
5:  goto 2
6:  ret(j)
```



(a) (b)

Figure 3.3: A program in L_0 and its control flow graph

$$
\begin{aligned}
Nodes &= \{Entry, N_0, \ldots, N_{n-1}, Exit\} \\
Edges &= \{(Entry, N_0, seq), (N_{n-1}, Exit, seq)\} \\
&\cup \{(N_i, N_{i+1}, seq) \mid I(N_i) \neq \texttt{goto } \_, 0 \leq i < n-1\} \\
&\cup \{(N_i, N_j, branch) \mid 0 \leq i < n, 0 \leq j < n, I(N_i) = \texttt{if } e \texttt{ goto } j\} \\
&\cup \{(N_i, N_{n-1}, branch) \mid 0 \leq i < n, j < 0 \vee j \geq n, I(N_i) = \texttt{if } e \texttt{ goto } j\} \\
&\cup \{(N_i, N_j, seq) \mid 0 \leq i < n, 0 \leq j < n, I(N_i) = \texttt{goto } j\} \\
&\cup \{(N_i, N_{n-1}, seq) \mid 0 \leq i < n, j < 0 \vee j \geq n, I(N_i) = \texttt{goto } j\} \\
I(N) &= \begin{cases} I_i & \textit{if } N = N_i, \\ \texttt{skip} & \textit{otherwise} \end{cases}
\end{aligned}
$$

All CFGs that are referred to within this chapter have the additional property of *recoverability*. A CFG is said to be *recoverable* if:

- only one node is associated with a `ret` instruction and this is the only node whose

successor is the $Exit$ node;

- the *Entry* node has no predecessors;
- the *Exit* node has no successors;
- any node associated with a conditional has exactly two successors, one edge of type seq and one of type $branch$;
- any node not associated with a conditional has exactly one successor connected by an edge of type seq (apart from the $Exit$ node).

Recoverability is a useful property since the transformation methodology described here performs transformations on the CFG rather than the program itself, and one would like to recover a program in L_0 from the generated CFG. Recoverability simplifies the process of converting a transformed CFG back into an L_0 program.

For the purpose of this chapter, recoverability is preserved by ensuring that a transformation is not applied if any of the above properties are violated by the resulting CFG. One consequently does not need to consider it when reading the example specifications in this chapter.

Chapter 4 describes how the implementation maintains the invariants of its compiler framework's intermediate representation, which equates to a recoverability proposition. Chapter 6 is conducted in the context of a language with exceptions. Therefore no notion of recoverability exists, since the exceptions provide multiple exit nodes for the method.

Paths through a control flow graph

The transformations introduced in Section 3.5 use side conditions which describe properties about *complete paths* through a program. Complete paths from a specified point

n are sequences of connected nodes through a CFG from point n to the *Exit* point. Accordingly, complete paths can be extracted from a CFG using the following definition:

Definition 3.1.4 *For a CFG $\mathcal{G}$, the set of complete paths of a system from a state n_0 is denoted CPaths$(n_0, \mathcal{G})$ and consists of all finite sequences $n_0 n_1 \ldots n_k$ such that $n_i \rightarrow n_{i+1}$, for all n_i with $i < k$ and such that there does not exist a n_{k+1} such that $n_k \rightarrow n_{k+1}$. The notation $x \rightarrow y$ is used when $\exists(n, m, \_) \in Edges \mid n = x, m = y$.*

Limitations of L_0

This chapter is restricted to a simple toy language in order to ease the initial language description. [44] describes hand written proofs of soundness of transformations over the L_0 language. More complex features, such as pointers and exceptions, are discussed in Chapter 6, where we present some exploration of how this language and methodology can be used with real programming languages.

3.2 The TRANS Specification Language

We have designed a language for expressing specifications, which we call TRANS, that captures the features described in the previous section. The syntax of TRANS is shown in Figure 3.4. Note that `op`, `literal` and `num` are L_0 elements from Figure 3.1. We overload logical binders and use standard CTL binding rules. The @ operator binds more weakly than operators on node conditions.

Throughout this chapter we description the semantics of the TRANS language with respect to L_0. The fundamental aspects of the language can be applied to many other languages. The syntax of the side conditions can remain the same, for example shows how it possible to incorporate aliasing conditions into side conditions. It is nec-

essary to change the syntax of the pattern matching component of TRANS in order to pattern match a particular programming language or intermediate representation.

3.2.1 Introductory Example

The language is simple, yet can express many standard compiler optimizations, particularly with the introduction of *strategies* in 3.3, which combine transformations to create more complex transformations. As an example of TRANS in use, the constant copy propagation transformation is written as:

$$replace\ n\ with\ x := e[c]$$

if

$$\mathit{stmt}(x := e[v]) \wedge \overleftarrow{A}(\neg \mathit{def}(v)\ U\ \mathit{stmt}(v := c))\ @\ n\ \wedge\ \mathit{conlit}(c)$$

The part of the transformation before the `if` statement is the *action* and after it is called the *side condition*. The action describes what changes are to be made to the program, whilst the side condition describes the conditions under which the program is to be transformed. This transformation is described in more detail on page 45.

Information is shared between the two parts through *metavariables* which are variables in TRANS that bind to parts of the program's abstract syntax. The metavariables in this specification are n, x, e, c and v. Within the TRANS language $e[c]$ denotes an expression containing c as an operand.

The transformation's meaning is dependent on the values that these variables represent. Informally, the meaning of a transformation is to perform the specified actions of the transformation using some valuation of the meta-variables such that the side condition is true.

This side condition can be more easily understood by splitting it into three distinct components. $\mathit{conlit}(c)$ requires that the metavariable c can only bind to a

literal	::=	*metavar* | *num* | exit | start | seq | branch
expr-pattern	::=	*literal*
	|	*expr-pattern op expr-pattern*
	|	*expr-pattern*[*expr-pattern*]
pattern	::=	`if` (*expr-pattern*)
	|	*metavar* := *expr-pattern*
	|	`skip`
	|	`ret`(*expr-pattern*)
node-condition	::=	*node-condition* $\vee$ *node-condition*
	|	*node-condition* $\wedge$ *node-condition*
	|	$\neg$ *node-condition*
	|	$\exists$ *metavar* . *node-condition*
	|	$[EX \mid AX \mid \overleftarrow{EX} \mid \overleftarrow{AX}]_{[literal]}$ (*node-condition*)
	|	$[E \mid A \mid \overleftarrow{E} \mid \overleftarrow{A}]$ (*node-condition* U *node-condition*)
	|	node(*literal*)
	|	stmt(*pattern*)
side-condition	::=	*side-condition* $\vee$ *side-condition*
	|	*side-condition* $\wedge$ *side-condition*
	|	$\neg$ *side-condition*
	|	$\exists$ *metavar* . *side-condition*
	|	*node-condition* @ *literal*
	|	*pred* (*literal*$_1$,. . . ,*literal*$_n$)
action	::=	replace *literal* with *pattern*$_1$;*pattern*$_2$;. . . ;*pattern*$_n$
	|	remove_edge (*literal*,*literal*,*literal*)
	|	add_edge (*literal*,*literal*,*edge-type*)
	|	add_edge (*literal*,*literal*,*metavar*)
	|	split_edge (*literal*,*literal*,*pattern*)
transform	::=	*action*$_1$,. . . ,*action*$_n$ if *side-condition*
	|	MATCH *side-condition* IN *transform*
	|	APPLY_ALL *transform*
	|	*transform* $\Box$ *transform*
	|	*transform* THEN *transform*

Figure 3.4: The grammar of TRANS

constant literal value, for example 1. *stmt*$(x := e[v])$ is a predicate that pattern matches statements, it only matches assignments. If it matches at a statement then x is bound to lval of the assignment, e to the rval, and v to any value within the expression e. Note this might also be the whole of e. $\overleftarrow{A}(\neg$*def*$(v)\ U$ *stmt*$(v := c))$ @ n means that the only definition of v that reaches n is $v := c$.

The actions in this case is very simple — to simply replace the node n with a new statement that it has constructed. The pattern syntax here is the same form as used by the $stmt$ predicate, but instead of matching existing statements in the program, it is used to reconstruct new statements.

3.2.2 Macros, syntactic sugar

Rewriting is an important tool in this framework. The traditional way to write conditional rewrites (where a node can be replaced by a sequence of nodes) is as follows:

$$\textit{literal}\text{:}\textit{pattern} \Longrightarrow \textit{pattern}_1\text{;}\textit{pattern}_2\text{;}\ldots\text{;}\textit{pattern}_n$$

This notation is supported in our framework as syntactic sugar and mapped into a *replace* action. For example, the conditional rewrite:

$$\begin{array}{l} n : p \Longrightarrow q_1; q_2; \ldots; q_m, \\ A_1, A_2, \ldots, A_k \\ \text{if} \\ \phi \end{array} \quad ,$$

where p is the pattern to be matched to the statement at n, is an alternate syntax for the action:

$$replace\ n\ with\ q_1; q_2; \ldots; q_m,$$
$$A_1, A_2, \ldots, A_k$$

if

$$\mathit{stmt}(p)\ @\ n \wedge \phi$$

A macro definition provides a way to name commonly used formulae. Macros are of the following general form

$$let\ p(\overrightarrow{x}) \triangleq \phi$$

The expression $p(\overrightarrow{\tau})$ in a formula represents the syntactic substitution $\phi[\overrightarrow{\tau}/\overrightarrow{x}]$, where each variable in $\overrightarrow{x}$ is replaced by the corresponding term in $\overrightarrow{\tau}$. This allows one to specify formulae that will be used in several different transformations. Free variables are used in some macros when all the uses of the variables have specific denotations, such as loop head or loop tail.

As examples, two macros which match nodes that are connected and strongly connected, respectively, to m can be written as: *e.g.*

$$let\ connected\_to(m) \triangleq E(True\ U\ \mathit{node}(m))$$
$$let\ strongly\_connected\_to(m) \triangleq E(True\ U\ node(m)) \wedge \overleftarrow{E}(True\ U\ node(m))$$

Macros support the definition of temporal operators *eventually* (F) and *forever* (G) in terms of *until* operators in the standard way:

$$let\ EF(\phi) \triangleq E(true\ U\ \phi)$$
$$let\ AF(\phi) \triangleq A(true\ U\ \phi)$$
$$let\ EG(\phi) \triangleq \neg AF(\neg\phi)$$
$$let\ AG(\phi) \triangleq \neg EF(\neg\phi)$$

3.3 Semantics

In this section we describe the semantics of TRANS. First the semantics of the side conditions will be described, then the semantics of the actions and finally the semantics of a complete transformation.

Semantic objects and valuations

Transformations are based on the interpretation of free variables which refer to objects in the program. There are several different sets of objects relating to a program.

Definition 3.3.1 *The semantics* TRANS *refers to the objects which are manipulated in programs in L_0, the main ones being:*

Instr	*The set of possible instructions*
Expr	*The set of possible expressions*
Var	*The set of program variables*
Num	*The set of numbers used in the program*
Op	*The set of operators on elements of Expr*
SynFunc	*The set of syntactic functions*

Note that *Var* $\subseteq$ *Expr* and *Num* $\subseteq$ *Expr*. We use the symbol $\oplus$ to denote members of *Op*. Members of *SynFunc* are functions of type *Expr* $\rightarrow$ *Expr* which denote simple syntactic substitution, for example $\lambda x.x + \mathrm{y}$ where y and $+$ are elements of L_0. A restriction that the bound variable only occurs once in the body of the function is made to ensure each function picks out only one part of a syntax tree.

Definition 3.3.2 *The set $\mathcal{O}$ of objects of a program with Control Flow Graph $\mathcal{G}$ is defined as*

$$\mathcal{O} = \mathit{Nodes(G)} \cup \mathit{Edges(G)} \cup \mathit{Instr} \cup \mathit{Expr} \cup \mathit{Op} \cup \mathit{SynFunc} \quad .$$

Transformations are defined through the use of free variables. We use MetaVar as the type of metavariables, and we use a, b,... as metavariables. The type MetaVar can bind to values within the set $\mathcal{O}$, defined in Definition 3.3.2. The type MetaVar is specifically used in the translation of side conditions into Binary Decision Diagrams described in Section 5.5.

The semantics of TRANS is given in terms of a *valuation* function for objects of L_0.

Definition 3.3.3 *A $valuation$ is a mapping from MetaVar to $\mathcal{O}$.*

Let $Valuation$ be the type of all valuations. A $valuation$ can convert a $Pattern$ (*i.e.* the *expr-pattern* non-terminal of TRANS in Figure 3.4) into a semantic object. This is, in effect, "substituting" into a pattern containing free variables.

Definition 3.3.4 *The partial function $subst$: Valuation $\times$ Pattern $\nrightarrow$ Expr is defined by*

$$
\begin{array}{lcll}
subst(\sigma, x) & = & \sigma(x) & \text{if } \sigma(x) \in \textit{Expr} \\
subst(\sigma, x\ o\ y) & = & subst(\sigma, x)\ \sigma(o)\ subst(\sigma, y) & \text{if } \sigma(o) \in \textit{Op} \\
subst(\sigma, e[d]) & = & \sigma(e)(subst(\sigma, d)) & \text{if } \sigma(e) \in \textit{SynFunc}
\end{array}
$$

Side conditions

The basis of the side conditions in TRANS are first order CTL formulae. We define the type $NodeCondition$ that intuitively corresponds to these connectives, and formally to the language defined by the *node-condition* non-terminal of TRANS. These are connected together using first order logical connectives. The truth of these formulae depend on a $valuation$, a node in the CFG and the CFG itself. Accordingly, we introduce

a semantic function $\lceil . \rceil$ which maps a $NodeCondition$ to its meaning:

$$\lceil . \rceil : NodeCondition \rightarrow ((Valuation \times Node \times FlowGraph) \rightarrow Bool)$$

The definition of $\lceil . \rceil$ follows the semantics of CTL. For convenience we define $\overleftarrow{\mathcal{G}}$ which is identical to the graph $\mathcal{G}$ but with direction on every edge inverted.

Definition 3.3.5 *The semantic function for node conditions is defined by*

$$\begin{array}{lcl}
\lceil \phi \vee \psi \rceil(\sigma, n, \mathcal{G}) & = & \lceil \phi \rceil(\sigma, n, \mathcal{G}) \textit{ or } \lceil \psi \rceil(\sigma, n, \mathcal{G}) \\
\lceil \phi \wedge \psi \rceil(\sigma, n, \mathcal{G}) & = & \lceil \phi \rceil(\sigma, n, \mathcal{G}) \textit{ and } \lceil \psi \rceil(\sigma, n, \mathcal{G}) \\
\lceil \neg \phi \rceil(\sigma, n, \mathcal{G}) & = & \textit{it is not the case that } \lceil \phi \rceil(\sigma, n, \mathcal{G}) \\
\lceil \exists x.\phi \rceil(\sigma, n, \mathcal{G}) & = & \exists v : v \in \mathcal{O} : \lceil \phi \rceil(\sigma.x- > v, n, \mathcal{G}) \\
\lceil \mathsf{node}(m) \rceil(\sigma, n, \mathcal{G}) & = & \sigma(m) = n \\
\lceil \mathsf{stmt}(p) \rceil(\sigma, n, \mathcal{G}) & = & I(n) \cong_\sigma p \\
\lceil EX_{[l]}(\phi) \rceil(\sigma, n, \mathcal{G}) & = & \exists m, n : (n, m, l) \in Edges(\mathcal{G}) : \lceil \phi \rceil(\sigma, m, \mathcal{G}) \\
\lceil AX_{[l]}(\phi) \rceil(\sigma, n, \mathcal{G}) & = & \forall m, n : (n, m, l) \in Edges(\mathcal{G}) : \lceil \phi \rceil(\sigma, m, \mathcal{G}) \\
\lceil E(\phi \; U \; \psi) \rceil(\sigma, n, \mathcal{G}) & = & \exists p : p \in CPaths(n, \mathcal{G}) : Until(p, \phi, \psi) \\
\lceil A(\phi \; U \; \psi) \rceil(\sigma, n, \mathcal{G}) & = & \forall p : p \in CPaths(n, \mathcal{G}) : Until(p, \phi, \psi) \\
\lceil \overleftarrow{EX}_{[l]}(\phi) \rceil(\sigma, n, \mathcal{G}) & = & \exists m, n : (m, n, l) \in Edges(\mathcal{G}) : \lceil \phi \rceil(\sigma, m, \overleftarrow{\mathcal{G}}) \\
\lceil \overleftarrow{AX}_{[l]}(\phi) \rceil(\sigma, n, \mathcal{G}) & = & \forall m, n : (m, n, l) \in Edges(\mathcal{G}) : \lceil \phi \rceil(\sigma, m, \overleftarrow{\mathcal{G}}) \\
\lceil \overleftarrow{E}(\phi \; U \; \psi) \rceil(\sigma, n, \mathcal{G}) & = & \exists p : p \in CPaths(n, \overleftarrow{\mathcal{G}}) : Until(p, \phi, \psi) \\
\lceil \overleftarrow{A}(\phi \; U \; \psi) \rceil(\sigma, n, \mathcal{G}) & = & \forall p : p \in CPaths(n, \overleftarrow{\mathcal{G}}) : Until(p, \phi, \psi)
\end{array}$$

The definition of the next operators (EX, AX etc.) includes the optional parameter $_{[l]}$, which indicates whether an edge in the graph is a *branch* or *seq* edge. If the statement holds true regardless of which type of branch is used, the parameter may be omitted.

We define *Until* in the following manner.

Definition 3.3.6 *Consider a path $p \in CPaths(n, \mathcal{G})$, for some n, such that $p = n_0 n_1 \ldots n_k$. The predicate $Until(p, \phi, \psi)$ holds if:*

$$\exists j : 0 \leq j \leq k : \lceil \psi \rceil(\sigma, n_j, \mathcal{G}) \wedge \forall i : 0 \leq i < j. \lceil \phi \rceil(\sigma, n_i, \mathcal{G})$$

In order to capture pattern matching, the specification of the side condition semantics makes use of the relation $\cong_\sigma$ which is a subset of $Instr \times Expr - Pattern$ and defined by:

$$\begin{aligned}
I \cong_\sigma x := e &= I = \sigma(x) := subst(\sigma, e) \\
I \cong_\sigma \texttt{if } (e) &= \exists n. I = \texttt{if } subst(\sigma, e) \texttt{ goto } n \\
I \cong_\sigma \texttt{skip} &= I = \texttt{skip} \text{ or } \exists n. I = \texttt{goto } n \\
I \cong_\sigma \texttt{ret}(e) &= I = \texttt{ret}(subst(\sigma, e))
\end{aligned}$$

The temporal logic formulae can be combined using logical connectives and the @ operator, creating side conditions of the type $Condition$ that do not depend on a particular node in the CFG. We define the semantics of these conditions by overloading the semantic function $\lceil . \rceil$:

$$\lceil . \rceil : Condition \rightarrow (Valuation \times FlowGraph \rightarrow Bool) \ .$$

The definition of this function is straightforward:

Definition 3.3.7 *The semantic function for side conditions is defined by*

$$
\begin{aligned}
\lceil \phi \vee \psi \rceil(\sigma, \mathcal{G}) &= \lceil \phi \rceil(\sigma, \mathcal{G}) \textit{ or } \lceil \psi \rceil(\sigma, \mathcal{G}) \\
\lceil \phi \wedge \psi \rceil(\sigma, \mathcal{G}) &= \lceil \phi \rceil(\sigma, \mathcal{G}) \textit{ and } \lceil \psi \rceil(\sigma, \mathcal{G}) \\
\lceil \neg \phi \rceil(\sigma, \mathcal{G}) &= \textit{it is not the case that } \lceil \psi \rceil(\sigma, \mathcal{G}) \\
\lceil \exists x.\phi \rceil(\sigma, \mathcal{G}) &= \exists \tau : \tau \in \mathcal{O} : \lceil \phi \rceil(\sigma.x- > \tau, \mathcal{G}) \\
\lceil \phi \ @ \ m \ \rceil(\sigma, \mathcal{G}) &= \lceil \phi \rceil(\sigma, \sigma(m), \mathcal{G}) \\
\lceil p(\overline{x}) \rceil(\sigma, \mathcal{G}) &= \widehat{p}(\sigma(\overline{x}), \sigma)
\end{aligned}
$$

Here, the logical combination of node conditions is standard. The last clause above refers to global predicates, described next.

Basic Predicates

The TRANS language allows a wide range of *predicates* to be defined, some of which are fundamental predicates, and some of which are simply commonly used macros. We make the distinction between those predicates that are global, and those that are parameterised by node.

Global predicates

Global conditions (such as *conlit*(c), which states that c is a literal constant) are defined by a family of global predicates (each in a type $GlobalPred_n$ where n is the arity of the predicate). The semantics of these expressions varies from predicate to predicate and is stipulated by a family of functions $\widehat{\cdot}$, such that for each arity n there is a function:

$$\widehat{\cdot} : GlobalPred_n \rightarrow (\mathcal{O}^n \times Valuation) \rightarrow Bool) \ .$$

In the example transformations described in this chapter, the following basic global predicates on L_0 are used:

conlit(x)	x is a constant literal
varlit(x)	x is a variable literal
freevar(x, e)	x is a free variable of the expression e
is(x, e)	the expression e can be statically determined and evaluates to x

Usually, we write "x is e" instead of *is*(x, e), in the style of Prolog. The formal definition of the above predicates is as follows

$$\begin{aligned}
\widehat{\textit{conlit}}(x)\sigma &= \sigma(x) \in Num \\
\widehat{\textit{varlit}}(x)\sigma &= \sigma(x) \in \textit{Var} \\
\widehat{\textit{freevar}}(v, e)\sigma &= \sigma(v) \in FV(subst(\sigma, e)) \\
\widehat{\textit{is}}(x, e)\sigma &= \sigma(x) = evalC(subst(\sigma, e))
\end{aligned}$$

The predicates return false if the function $subst$ or $evalC$ is undefined when called. The definition of the *is*(x, e) predicate uses the following auxiliary function:

$$\begin{aligned}
evalC(x) &= x && \text{if } x \in Num \\
evalC(x \oplus y) &= evalC(x) \ [\![\oplus]\!] \ evalC(y) && \text{if } \oplus \in Op
\end{aligned}$$

Within this definition $[\![op]\!]$ is the application of op to the surrounding arguments. It is also useful to have one global predicate *fresh*, which succeeds when its argument is bound to the next new variable *i.e.* a variable that has not been mentioned in the program previously.

Local predicates

Local predicates describe conditions at particular nodes during execution. The following local predicates are useful for the transformations presented in this chapter:

def(x) @ n	the statement at node n assigns to variable x
use(x) @ n	the statement at node n is an assignment, whose expression contains the sub-expression x
trans(e) @ n	the statement at node n does not assign to any of the free variables in the expression e

These predicates are defined in TRANS as macros.

$$let \quad \textit{def}(x) \quad \triangleq \quad \exists e.\ \textit{stmt}(x := e)$$

$$let \quad \textit{use}(x) \quad \triangleq \quad \exists v, e.\ \textit{stmt}(v := e[x]) \vee \textit{stmt}(\texttt{if}\ \ (e[x])) \vee \textit{stmt}(\texttt{ret}(e[x]))$$

$$let \quad \textit{trans}(e) \quad \triangleq \quad \neg\exists v, d.\ \textit{stmt}(v := d) \wedge \textit{freevar}(v, e)$$

Actions

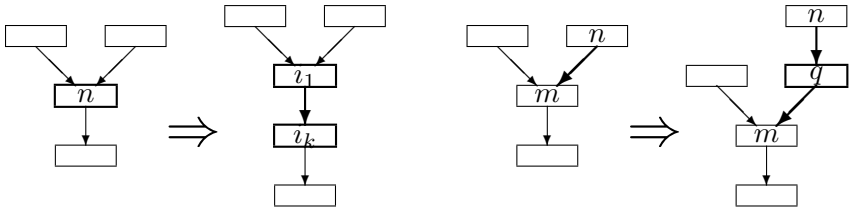
Side conditions stipulate which valuations allow the transformation to apply to a program (namely the valuations σ such that $\lceil\phi\rceil(\sigma, \mathcal{G})$ holds). Given such a valuation, an *action* specifies how to alter the program. An action is defined as a function on flow graphs—a partial function, since it could still fail due to a type mismatch. Accordingly, the semantics of different actions have type:

$$\lceil . \rceil : \textit{Action} \rightarrow (\textit{Valuation} \times \textit{FlowGraph} \nrightarrow \textit{FlowGraph})$$

There are four types of actions in TRANS; combinations of these actions allow nodes to be added or deleted at any position in the graph.

The *add_edge* action adds an edge of a particular type between two nodes. If an edge of the correct type already exists between the two nodes then the action has no effect.

Definition 3.3.8 *The action* $add\_edge(n, m, e)$ *is defined for the case that edges as metavariables as:*



(a) (b)

Figure 3.5: The replace and split_edge actions

$$\lceil add\_edge(n, m, e)\rceil(\sigma, \langle N, E, I\rangle) \;=\; \langle N, E \cup \{(\sigma(n), \sigma(m), \sigma(e))\}, I\rangle$$

In the case that an edge is represented by a literal edge-type, it is defined as:

$$\lceil add\_edge(n, m, e)\rceil(\sigma, \langle N, E, I\rangle) \;=\; \langle N, E \cup \{(\sigma(n), \sigma(m), e)\}, I\rangle$$

The *remove_edge* action removes an edge between two nodes. If no such edge exists then the action has no effect.

Definition 3.3.9 *The action* $remove\_edge(n, m, e)$ *is defined as:*

$$\lceil remove\_edge(n, m, e)\rceil(\sigma, \langle N, E, I\rangle) \;=\; \langle N, E \setminus \{(\sigma(n), \sigma(m), \sigma(e))\}, I\rangle$$

The *replace* action replaces a node with a sequence of nodes. This is illustrated in Figure 3.5(a). The action changes all three components of the graph. It adds the nodes necessary to create the new sequence of instructions, to be connected by a series of seq edges. The successor edges of the node being replaced are moved to the end of the block and the labelling function of the graph is altered to map the correct values of the instructions on the nodes in the replacement block.

Definition 3.3.10 *The action* $replace\ n\ with\ i_1; \ldots; i_k$ *is defined as:*

$$\lceil replace\ n\ with\ i_1; \ldots; i_k\rceil(\sigma, \langle N, E, I\rangle) \;=$$
$$\langle N \cup \{n_2, \ldots, n_k\}, E', I \rhd [n_1 \mapsto subst(\sigma, i_1), ..., n_k \mapsto subst(\sigma, i_k)]\rangle$$

where $n_1 = \sigma(n)$ *and* $n_2, \ldots, n_k$ *are nodes not occurring in the original graph and* $\triangleright$ *alters the labelling function* I *with new bindings. The edge relation* E' *is defined as:*

$$E' = \{remap\_succ(n_1, n_k, e) \mid e \in E\} \cup \{(\sigma(n_i), \sigma(n_{i+1}), seq) \mid 1 < i < k\}$$

and

$$remap\_succ(x, y, e) = \begin{cases} (m, s, t) & \textit{if } x = n,\ y = m,\ e = (n, s, t) \\ e & \textit{otherwise} \end{cases}$$

The $split\_edge$ action alters a graph by inserting a node between two existing nodes joined by a particular edge. This action is useful because it specifies which particular edge is used as placement for a node, in the case where its successor node has several predecessors. This is illustrated in Figure 3.5(b), which uses the same labelling function. In this definition n and m both represent nodes in the graph, whilst e is the edge between them that the transformation splits, and i is the new instruction to be inserted in the split graph.

Definition 3.3.11 *The action* $split\_edge(n, m, e, i)$ *is defined as:*

$$\begin{aligned} &\lceil split\_edge(n, m, e, i) \rceil (\sigma, \langle N, E, I \rangle) \ = \\ &\langle N \cup \{q\}, \\ &\ (E \setminus \{(\sigma(n), \sigma(m), \sigma(e))\}) \cup \{(\sigma(n), q, \sigma(e)), (q, \sigma(m), \sigma(e))\}, \\ &\ I \rhd [q \mapsto subst(\sigma, i)] \rangle \end{aligned}$$

where q *is a new node not occurring in the original CFG and* $\triangleright$ *overwrites a map with a new entry.*

It is possible to compose several actions in sequence. For example the action

$$\begin{aligned} &replace\ n\ with\ x := 4, \\ &split\_edge(n, m, e, y := x) \end{aligned}$$

will first perform the node replacement and then perform the edge split. It is straightforward to define how these actions are performed in sequence:

$$\begin{aligned} \lceil A_1, A_2, \ldots, A_k \rceil(\sigma, \mathcal{G}) &= \lceil A_2, \ldots, A_k \rceil(\sigma, \lceil A_1 \rceil(\sigma, \mathcal{G})) \\ \lceil \rceil(\sigma, \mathcal{G}) &= \mathcal{G} \end{aligned}$$

This completes the definition of the semantics of the action component of transformations. This formalisation, along with the semantics of the side conditions in the previous section, allows one to define the meaning of a complete transformation.

Transformations

This section presents the semantics of a transformation, drawing on the previously defined semantic functions. The overall form of a transformation is:

$$A_1, A_2, \ldots, A_n$$

if

$$\phi$$

where each A_i is some action and ϕ is the side condition. The transformation attempts to compute any possible valuations such that, for a given valuation σ, $\lceil \phi \rceil(\sigma, \mathcal{G})$ is true. It then uses these valuations to perform the actions A_1 to A_n in left to right order.

Since a side condition can be true under many different valuations, the meaning of a transformation is given by a *set* of functions that transform graphs. The meaning of a transformation depends on a *partial* valuation. A partial valuation is a partial function from metavariables to objects. The semantic function $\lceil . \rceil$ for transformations will be of type:

$$\begin{aligned} \lceil . \rceil : \quad & \mathit{Transformation} \\ \rightarrow \quad & (\mathit{PartValuation} \times \mathit{FlowGraph} \rightarrow \mathbb{P}(\mathit{FlowGraph} \rightarrow \mathit{FlowGraph})) \end{aligned}$$

We then define a transformation in TRANS to be the set of transformations we get courtesy of valuations that are compatible with the given partial valuation and make the side condition true.

Definition 3.3.12 *The semantic function on transformations is defined as:*

$$\lceil A_1, \ldots, A_k \textit{ if } \phi \rceil(\tau, \mathcal{G}) \; = \\ \{\lambda x. \lceil A_1, \ldots, A_k \rceil(\sigma, x) \mid \lceil \phi \rceil(\sigma, \mathcal{G}) \textit{ holds and } \sigma \upharpoonright dom(\tau) = \tau\}$$

where $f \upharpoonright D$ denotes the function f restricted to the domain D.

This definition of the semantics as a set of functions between CFGs makes explicit the non-determinism that stems from the fact that many different substitutions may satisfy the applicability condition of the transformation. This can be seen to correspond to the fact that the transformation may apply to many different parts of the CFG. The intended application for these transformations resolves the non-determinism by choosing one correct valuation, *i.e.* one *local* place to transform.

Strategies

In order to succinctly specify more complex transformations, we introduce *strategies*, which are operators that act upon transformations. In this section we describe four strategies which will be used to define optimizations in Section 3.6.

Matching Free Variables

The $\mathrm{MATCH} \ldots \mathrm{IN}$ strategy executes a transformation restricted to a valuation that satisfies a particular formula. This strategy is particularly useful in combination with other strategies, as in such cases the desired valuations cannot be simply added to the

side condition. For example, the transformation

$$\text{MATCH } \textit{stmt}(x := e) \text{ @ } n \text{ IN } T$$

specifies that the transformation T should be used only when a substitution that makes the formula *stmt*$(x := e)$ @ n true is found. Only the variables n, x and e are restricted; any other free variables in T are unaffected. Often T is a complex transformation made of strategies.

The MATCH...IN strategy is defined in terms of the semantic function for transformations.

Definition 3.3.13 *The* MATCH...IN *strategy is defined by the following extension to the semantic function on transformations:*

$$\lceil \text{MATCH } \phi \text{ IN } T \rceil(\tau, \mathcal{G}) =$$
$$\{f \mid \lceil \phi \rceil(\sigma, \mathcal{G}) \textit{ holds}, \ \sigma \restriction dom(\tau) = \tau, \ f \in \lceil T \rceil(\tau \cup (\sigma \restriction FV(\phi)), \mathcal{G})\}$$

where $FV(\phi)$ is the set of free variables occurring in the formula ϕ.

Global Transformation

With Definition 3.3.12 of the semantics of transformations as a set of functions between CFGs, the natural solution is to choose one particular instance to solve the nondeterminism. An alternative which is required for some program transformations is a global transformation that applies in every place, *i.e.* for every valuation that satisfies the side condition. This is achieved with the APPLY_ALL strategy.

Definition 3.3.14 *The* APPLY_ALL *strategy is defined by:*

$$\lceil \text{APPLY\_ALL}(T) \rceil(\tau, \mathcal{G}) = \{f_1 \circ f_2 \circ \ldots \circ f_n \mid f_i \in \lceil T \rceil(\tau, \mathcal{G}) \setminus \{f_1 \ldots f_{i-1}\}\}$$

where n is the (finite) number of elements of $\lceil T \rceil(\tau, \mathcal{G})$.

In other words, the $\mathrm{APPLY\_ALL}$ strategy applies all of the possible transformations in any order.

Nondeterminism

In certain cases it is useful to extend the nondeterminism of transformations when combining them. The operator $\Box$ on transformations corresponds to a nondeterministic choice which is made available for when valuations are matched.

Definition 3.3.15 *The $\Box$ operator on transformations is defined by the following extension to the semantic function on transformations:*

$$\lceil T_1 \Box T_2 \rceil(\tau, \mathcal{G}) = \lceil t_1 \rceil(\tau, \mathcal{G}) \cup \lceil t_2 \rceil(\tau, \mathcal{G})$$

Composition

Sequential composition involves performing one transformation directly after another. The composed transformation will only succeed if both component transformations succeed. Composition is done with the T_1 THEN T_2 strategy.

Definition 3.3.16 *The THEN strategy is defined by:*

$$\lceil T_1 \ \mathrm{THEN}\ T_2 \rceil(\tau, \mathcal{G}) = \{ f \circ g \mid f \in \lceil T_1 \rceil(\tau, \mathcal{G}), g \in \lceil T_2 \rceil(\tau, \mathcal{G}) \}$$

This strategy makes most sense intuitively if both T_1 and T_2 are deterministic *i.e.* both $\lceil T_1 \rceil$ and $\lceil T_2 \rceil$ contain only one element. It is worth noting that the cardinality of $\lceil T_1 \ \mathrm{THEN}\ T_2 \rceil(\tau, \mathcal{G})$ is the multiple of the cardinalities of $\lceil T_1 \rceil(\tau, \mathcal{G})$ and $\lceil T_2 \rceil(\tau, \mathcal{G})$. In this sense THEN can be said to introduce new non-determinism.

Four strategies have now been defined and although they are simple they allow transformations to be combined in a very flexible manner. Section 3.5 describes simpler

transformations, whilst Section 3.6 presents more sophisticated transformations that include strategies. But before introducing these examples, we examine how loops can be recognised within our methodology.

3.4 Identifying Loops via Dominators

Since our system operates over the control flow graph of the program, it does not naturally have information about the existence and positioning of loop structures. In this section, we show how loops can be recovered from unstructured graphs, and we define the macro $loop$ which is used extensively in our specification of loop optimizations in Section 3.6.

The key concept is that of *dominance*. A node n is said to dominate a node m if every path from the start of the program to node m must pass through node n. This can be expressed as a temporal logic formula, by stating that at the start node there does not exist a path that satisfies $\neg$*node*(n) until the path reaches m.

$$let\ \textit{dom}(n, m) \triangleq \neg E(\neg \textit{node}(n)\ U\ \textit{node}(m))\ @\ start$$

Conversely, we can define *post dominance*, when every backward path from the exit node to m must pass through n.

$$let\ \textit{pdom}(n, m) \triangleq \neg \overleftarrow{E}(\neg \textit{node}(n)\ U\ \textit{node}(m))\ @\ exit$$

For the purpose of this chapter, the general pattern for a reducible loop is depicted in Figure 3.6. It is generally the case that optimizing compilers do not deal with irreducible loops, and we consequently do not account for this case [78]. Some loops test their condition after their loop bodies, for example do-while loops within the Java programming language. We do not account for this case since it can be easily

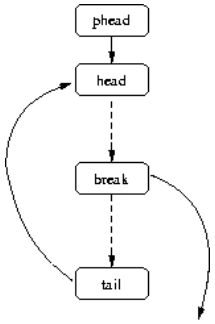


Figure 3.6: General pattern for a loop

converted into the form we look for by copying the loop body to a sequence of nodes ahead of the loop header.

Loops are characterised by certain key nodes: the *pre-header* ($phead$), the *head*, the *tail* and the *break* nodes. We specify the order of the key nodes in the loop with dominance relations. The pre-header must dominate the head (*dom*$(phead, head)$ must hold) and the break node must post-dominate the tail (*pdom*$(break, tail)$ must hold). In addition, the pre-header node must be the immediate predecessor of the head node:

$$AX(\textit{node}(head))\ @\ phead$$

A loop is identified by the existence of a *back-edge* between head and tail. The *back-edge* relation can be specified as an edge between two nodes whose source can reach its target in a backwards direction:

$$let\ \textit{back-edge}(tail, head) \triangleq (EX\,\textit{node}(head)) \wedge (\overleftarrow{E}F\,\textit{node}(head))\ @\ tail$$

It is worth noting that the *back-edge* predicate may hold true at positions other than those within the context of loop analysis (since we allow unrestricted *goto* with L_0). This is acceptable for our purposes, since the loop-related transformations refer to the *loop* definition, defined later in this section, which specifically binds the *back-edge*

definition to the loop tail and head nodes. We can now define the relation between the four key nodes that define a loop

$$let\ \textit{loop}(phead, head, break, tail) \triangleq \begin{array}{l} \textit{dom}(phead, head)\ \wedge\ \textit{pdom}(break, tail)\ \wedge \\ AX(\textit{node}(head))\ @\ phead\ \wedge \\ \textit{back-edge}(tail, head) \end{array}$$

Some optimizations such as our version of strength reduction apply only to *well-structured loops*, whose only entry is the head node and only exit is the break node. To identify such loops, nodes inside the loop and outside the loop are distinguished. Execution of nodes inside the loop lead eventually to the break node before either re-entering the loop or exiting the program. So all paths from nodes outside the loop reach either the exit or pre-header without going through the break

$$let\ \textit{out\_loop} \triangleq A(\neg \textit{node}(break)\ U\ \textit{node}(phead) \vee exit)$$

An illegal jump out of the loop requires the existence of a node that is not the break node but has a successor outside the loop.

$$let\ \textit{out\_jump} \triangleq \neg \textit{node}(break) \wedge EX(out\_loop)$$

An illegal jump into the loop can be defined in a similar manner.

$$let\ \textit{in\_jump} \triangleq \neg \textit{node}(head) \wedge \overleftarrow{EX}(out\_loop)$$

These predicates allow us to define well-structured loops:

$$let\ \textit{wsloop}(phead, head, break, tail) \triangleq \begin{array}{l} \textit{loop}(phead, head, break, tail)\ \wedge \\ A(\neg \textit{out\_jump}\ U\ \textit{out\_loop})\ @\ head\ \wedge \\ A(\neg \textit{in\_jump}\ U\ \textit{out\_loop})\ @\ head \end{array}$$

This definition of loops is used within the strength reduction and loop fusion transformations described in the following two sections. Intuitively, it captures loops

that have some block of sequential instructions up to a tail node, from which an edge goes back to the loop header. Somewhere within this loop there is a node that has a branch that allows one to break out of the loop, called the *break* node. A *pre-head* node is also matched, to support the hoisting of instructions to the place immediately preceding the loop.

3.5 Example Transformations

This section provides examples of common optimizing transformations that can be specified in TRANS, and which are amongst transformations that are found in the optimizing phase of many compilers. However, the presentation here may differ from standard presentations in the sense that each transformation may be only a *part* of a more complex optimization; the improvement in code will occur when it is combined with other transformations, such as dead code elimination. Specifying the transformations in this modular way supports experimenting with application of transformations in different orders to increase efficiency. The TRANS language also makes the transformation more amenable to formal analysis, for example proving that the transformation is semantics preserving.

Dead code elimination

Dead code elimination removes a definition of a variable if it is not used in the future. The rewrite simply removes the definition:

$$n : (x := e) \implies \texttt{skip}$$

The side condition on this transformation is that all future paths of computation should not use this definition of x or, more precisely, there does not exist a path with a node

$$n : (x := e) \implies \texttt{skip}$$
if
$$\neg EX(E(\neg \mathit{def}(x)\ U\ \mathit{use}(x) \wedge \neg \mathit{node}(n)))\ @\ n$$

Figure 3.7: Specification of dead code elimination

that uses x without a different instruction re-assigning to x first. You may recall from **??** that *def*(x) holds true at nodes where where `x` is written to, *use*(x) at nodes where `x` is read from and *node*(n) at nodes that `n` may bind to. The `@` allows the specification to restrict possible values of a node to a specific node condition.

This can be specified using the $E(\ldots U \ldots)$ construct, noting that x could be used at node n itself. So the paths that should be identified are those not using x until the formula *use*$(x) \wedge \neg$*node*(n) holds. The final specification of dead code elimination is shown in Figure 3.7.

Constant propagation

Constant Propagation is a transformation where the use of a variable is replaced with the use of a constant known before the program is run (*i.e.* at compile time). The standard method of finding out if the use of a variable is equivalent to the use of a constant is to find all the possible statements where the variable could have been defined, and check that in all of these statements, the variable is assigned the same constant value. The rewrite itself is simple:

$$n : (x := e[v]) \implies x := e[c]$$

This rewrite uses the term $e[v]$ to find an expression e with sub-expression v. Note that, as per Definition 3.3.1, a variable v matches against *one* occurrence of the sub-expression in e. The rewrite then replaces the occurrence that has been matched. So if

$$
\begin{array}{ll}
 & n : (x := e[v]) \Longrightarrow x := e[c] \\
\text{if} & \\
 & \overleftarrow{A}(\neg \mathit{def}(v)\ U\ \mathit{stmt}(v := c))\ @\ n\ \wedge\ \mathit{conlit}(c)
\end{array}
$$

Figure 3.8: Specification of constant propagation

e matches the expression $\mathtt{x} + (\mathtt{x} - 2)$, with v matching the left hand x and c matching 3, then the rewrite inserts the expression $3 + (\mathtt{x} - 2)$.

For the rewrite to be correct, v and c must be restricted so that v necessarily equals c at the given point. The idea is that if all backward paths from node n are followed, then the first definition of v encountered must be of the form $v := c$. As all such paths must be checked, the $\overleftarrow{A}(..U..)$ constructor is appropriate. To fit into the "until" path structure we can observe that requiring the first definition on a path to fulfil a property is equivalent to requiring that the path satisfies non-definition until a point where it is at a definition and the property holds. The full specification of constant propagation is given in Figure 3.8.

This transformation is equivalent to standard constant propagation as found in existing compilers. Sometimes it is useful to propagate other entities such as variables; the transformation specifications in these cases are almost identical.

There are several extensions to constant propagation that are quite specialised and require complex algebraic reasoning; a survey of their computational complexity can be found in [56]. Conditional constant propagation is presented in [86]. These extensions expose a current limitation of TRANS: it does not allow recursive pattern matching facilities, therefore expressions of arbitrary complexity cannot be folded. It only takes account of expressions of the form of $constant\ op\ constant$. Further extensions to TRANS could be used to capture these special cases.

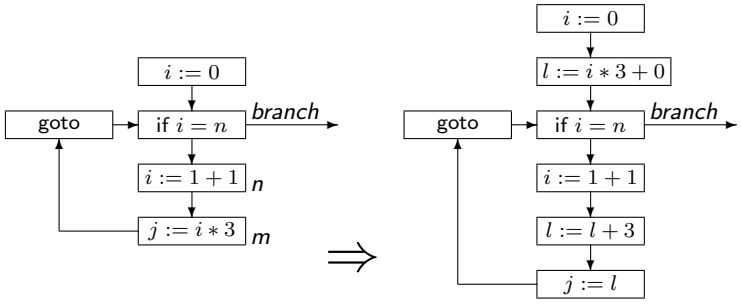


Figure 3.9: Strength reduction

Strength reduction

Strength reduction is a transformation that replaces multiplications within a loop structure with additions that compute the same value. This is beneficial since the computational cost of multiplication is usually greater than that of addition.

For strength reduction to apply, a basic induction variable and a derived induction variable dependent on it within the loop must be identified. Within a loop, a *basic induction variable* is one that has only one assignment of the form $v := v + c$. A *derived induction variable* of the loop is a variable that is linearly dependent on a basic induction variable v. The optimization is illustrated in Figure 3.9. We use n to denote the node at which the induction variable increments, and m to denote the node at which the derived induction variable is assigned to within the loop.

The following relation states that v is an induction variable that is incremented at program point m with increment c:

$$let\ \textit{basic\_induction\_var}(v, c, m) \triangleq \begin{array}{r} \neg \textit{out\_loop}\ \wedge (\textit{stmt}(v := v + c)\ @\ m\ \wedge \\ A(\textit{node}(m) \vee \neg \textit{def}(v)\ U\ \textit{out\_loop}))\ @\ \textit{head} \end{array}$$

A linearly derived induction variable is one that has only one assignment in the loop

assigning it to a linear function of the variable it depends on. This can be detected using the predicate *dependent_var*:

$$let\ \textit{linear\_def}(w, v, k, d) \triangleq (\textit{stmt}(w := v * k) \wedge d \text{ is } 0) \vee \textit{stmt}(w := v * k + d)$$

$$let\ \textit{dependent\_var}(w, v, k, d) \triangleq \begin{array}{c} \neg \textit{out\_loop}\ (\wedge \textit{linear\_def}(w, v, k, d)\ @\ n\ \wedge \\ A(\textit{node}(n) \vee \neg \textit{def}(w)\ U\ \textit{out\_loop}))\ @\ \textit{head} \end{array}$$

Finally, the dependent variable w is initialised before the start of the loop in the transformed program. In the first iteration of the loop the value of w between the head of the loop and its first definition will be different in the transformed program compared to the original program. To ensure the value difference does not matter the following invariant must be preserved:

$$A(\neg \textit{use}(w)\ U\ \textit{def}(w) \vee exit)\ @\ \textit{head}$$

In other words, either the variable w cannot be used between the head of the loop and its first definition within the loop or control flow leaves the loop if the variable w is not live.

When these conditions are satisfied, the strength reduction transformation introduces a fresh variable w' and replaces the assignment $w := v * k$ with $w := w'$. In order to maintain the correct value of w', it is necessary to add the assignment $w := w + \textit{step}$, where *step* is matched to the value of $c * k$ (using the *is* predicate) after execution of the assignment $v := v + c$. In addition w must be initialised to the correct value just after the pre-header of the loop. The overall transformation is specified as in Figure 3.10 .

Variations

The above transformation introduces a new variable and on its own only adds new calculations and does not make code any faster. This is a case where cleaning up

$$
\begin{array}{rcl}
phead : s & \Longrightarrow & s; w' := v * k + d, \\
n : (w := v * k) & \Longrightarrow & w := w', \\
m : (v := v + c) & \Longrightarrow & v := v + c; w' := w' + step
\end{array}
$$

if

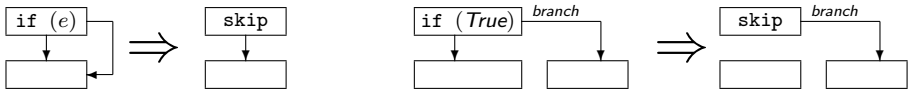
$loop(phead, head, break, tail)$ $\wedge$
$basic\_induction\_var(v, c, m)$ $\wedge$ $dependent\_var(w, v, k, d)$ $\wedge$
$A(\neg use(w)\ U\ def(w) \vee exit)$ @ $head$ $\wedge$
$conlit(k)$ $\wedge$ $conlit(c)$ $\wedge$ $step$ is $k * c$ $\wedge$ $fresh(w')$

Figure 3.10: Specification of strength reduction

transformations are needed. In particular, repeatedly applying variable propagation and dead code elimination removes the calculations involving the original dependent variable w.

Some loop strengthening algorithms (including the one in [1]) find more types of dependent variables. Specifically, dependent variables may exist that are a linear function of an induction variable but their definition is in terms of other dependent variables. Detecting this kind of dependent variable is appropriate when all loop strengthening is done in one monolithic transformation. In our framework, where small transformations are iterated, the complex approach is not necessary since repeated strengthening along with variable propagation and algebraic simplification will eventually strengthen variables such as *w* above.

A more complex version, combining strength reduction with code motion [41] needs the addition of TRANS strategies and is described in Section 3.6. There are even more sophisticated induction and dependent variable detection techniques that are beyond the scope of both this discussion and the TRANS language. For example see [83] where some advanced algebraic reasoning is required.



(a) (b)

Figure 3.11: Two cases of branch elimination

$$
\begin{array}{l}
n : (\texttt{if}\ \ (e)) \Longrightarrow \texttt{skip}, \\
remove\_generic(n, s), \\
add\_edge(n, s, seq) \\
\text{if} \\
AX(node(s))\ @\ n
\end{array}
\qquad
\begin{array}{l}
n : (\texttt{if}\ \ (True)) \Longrightarrow \texttt{skip}, \\
remove\_generic(n, t), \\
remove\_generic(n, f), \\
add\_edge(n, t, seq) \\
\text{if} \\
EX_{branch}(node(t))\ @\ n\ \wedge\ EX_{seq}(node(f))\ @\ n
\end{array}
$$

(a) (b)

Figure 3.12: Specification of two variants of branch elimination

Branch elimination

A jump statement such as depicted in Figure 3.11(a) where two branches of a conditional lead to the same node can be rewritten to a `skip` ; the two conditional edges are replaced by a sequential edge. The specification of this transformation is given in Figure 3.12(a).

Branch elimination can also disconnect an unused branch of a conditional, as shown in Figure 3.11(b). A pattern like this might occur after other transformations. The optimization is based upon two transformations: one for recognising 'always true' conditions and one for recognising 'always false' conditions. The 'always-true' case is specified in Figure 3.12(b).

3.6 Example Transformations using Strategies

Strategies provide a way of exploiting the non-determinism of matching within the side conditions of transformations. Here we describe transformations that use strategies to alter the control flow of a program.

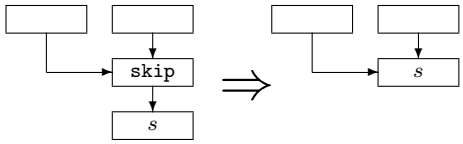


Figure 3.13: Skip elimination

Skip elimination

Some transformations, such as dead code elimination, may leave `skip` instructions in the program being optimized. As illustrated in Figure 3.13, these are unnecessary and can be removed.

A `skip` statement may have several predecessors but has only one successor. To remove the `skip` statement, it is necessary to remove all edges connected to it and add an edge between each predecessor to the successor. The strategy language is very suitable for such manipulations: a MATCH strategy locates a `skip` instruction and its unique successor, and the APPLY_ALL strategy performs the edge removal. The resulting specification of skip elimination is shown in Figure 3.14. It uses an action $remove\_generic$ which removes possibly existing edges between two nodes independently of their type (*i.e.* branch, seq). Recall that an APPLY_ALL strategy executes the transformation for every matching substitution of free variables (given that n and s are bound in the MATCH strategy).

Loop Fusion

Loop Fusion, as illustrated in Figure 3.15, is a control flow transformation which fuses two consecutive indexed loops into one. This often makes the code more time-efficient since it reduces the number of increment instructions to i and allows more opportunity

$let\ remove\_generic(n, m) \triangleq remove\_edge(n, m, seq); remove\_edge(n, m, branch)$.

MATCH
  $\mathit{stmt}(\texttt{skip}) \wedge AX(\mathit{node}(s))$ @ n
IN
  APPLY_ALL
    $remove\_edge(p, n, e)$
    $add\_edge(p, s, e)$
   if
    $\overleftarrow{EX}_e(\mathit{node}(p))$ @ n
  THEN
   $remove\_generic(n, s)$

Figure 3.14: Specification of skip elimination

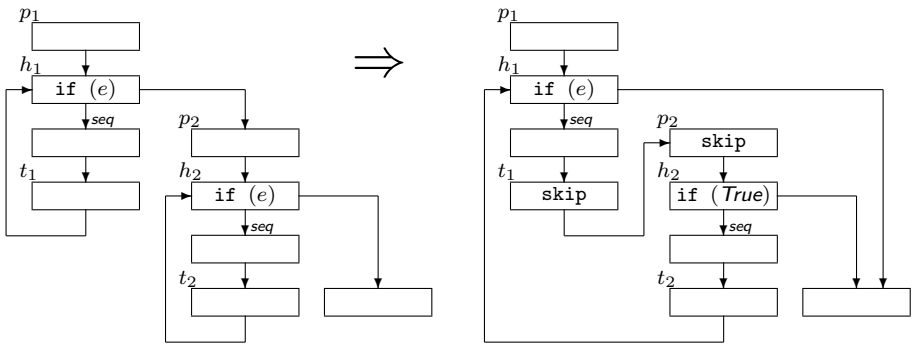


Figure 3.15: Loop Fusion

for instruction scheduling[1].

The specification of fusion must identify two loops. To make the transformation simpler, the specification shown here is restricted to loops that follow a `for`-like pattern, *i.e.* where the break node is the same as the header node and whose tail node increments the induction variable of the loop. The macro *loop* for identifying loops introduced in Section 3.4 is used twice here. This identifies the two loops that the specification

[1]However, this transformation may not produce faster code, especially as it can introduce worse cache behaviour.

transforms. The first has prehead p_1, head and break h_1 and tail t_1. The second has prehead p_2, head and break h_2 and tail t_2. These definitions specify the header and break nodes as the same node (since they are bound to the same meta-variable). Loops can be identified as consecutive by checking that all the second loop's pre-header's immediate predecessors are the break node of the first loop, *i.e.*

$$\overleftarrow{AX}(\mathit{node}(h_1)) \ @ \ p_2$$

The node $cont$ that follows the break in the second loop (*i.e.* where control leaves the loop) is detected with the following predicate:

$$EX(\mathit{node}(cont) \ \wedge out\_loop_2) \ @ \ h_2$$

The loops must be indexed in the same manner. Firstly, both the pre-header nodes (initialising the induction variable) and the break nodes must have the same instruction:

$$\mathit{same\_instr}(p_1, p_2) \wedge \mathit{same\_instr}(h_1, h_2)$$

where *same_instr* is defined by:

$$let \ \mathit{same\_instr}(n, m) \triangleq \exists s.\mathit{stmt}(s) \ @ \ n \ \wedge \mathit{stmt}(s) \ @ \ m \ \ .$$

Furthermore, both loops must have a common induction variable x (in the following formula $basic\_induction\_var_i$ is defined as on page 47 for either the first or second loop):

$$\mathit{basic\_induction\_var}_1(x, c, t_1) \wedge \mathit{basic\_induction\_var}_2(x, c, t_2)$$

Since the transformation is being applied to `for`-pattern loops, the specification stipulates that the basic induction variable is incremented at the tails of the loops. Finally, both exit conditions of the loops must leave the loop by a seq edge:

$$EX_{seq}(out\_loop_1) \ @ \ h_1 \ \wedge EX_{seq}(out\_loop_2) \ @ \ h_2$$

It is also necessary for initialisation, increment and exit values (i, c, and k respectively) to be unchanged between the two loops. In the transformation presented below they are restricted to be constant literals. For the loops to be successfully fused, we need to ensure that the statements in the second loop do not depend on the first loop. To this end, the predicate *ind_expr* holds of an expression whose components are not defined within the first loop:

$$let\ \mathit{ind\_expr}(e) \triangleq \neg \overleftarrow{E}(trans(e)\ U\ \neg trans(e) \wedge \neg out\_loop_1)$$

The predicate *independent* holds for nodes that do not depend on the first loop using the definition *ind_expr*(e).

let independent $\triangleq$

$$\texttt{skip} \vee \exists e.\ \mathit{ind\_expr}(e) \wedge (\mathit{stmt}(\texttt{if}\ \ (e)) \vee \mathit{stmt}(\_ := e) \vee \mathit{stmt}(\texttt{ret}(e)))$$

This expression can then be used to state that all the statements in the second loop are independent:

$$A(\mathit{independent}\ \ U\ \mathit{out\_loop}_2)\ @\ h_2$$

If these conditions are satisfied, the transformation will fuse the loops by connecting the tail of the first loop to the head of the second loop, the tail of the second loop to the head of the first loop and the break of the first loop to the $cont$ node. The macro *move_edge* is used to perform these connections as defined by:

$$let\ move\_edge(a, b, c) \triangleq \mathit{remove\_edge}(a, b, seq), \mathit{add\_edge}(a, c, seq)$$

The increment of the induction variable from the first loop and the initialisation and break from the second loop must be removed. However, it is simpler to replace the break in the second loop with the constant condition `if` $(True)$ and let branch elimination remove the edges later.

$h_2 : (\texttt{if}\ \ (x \oplus k)) \Longrightarrow \texttt{if}\ \ (True)$,
$p_2 : x := i \Longrightarrow \texttt{skip}$,
$t_1 : (x := x + c) \Longrightarrow \texttt{skip}$,
$move\_edge(t_2, h_2, h_1)$,
$move\_edge(h_1, p_2, cont)$,
$move\_edge(t_1, h_1, h_2)$,

if

$loop(p_1, h_1, h_1, t_1) \wedge loop(p_2, h_2, h_2, t_2) \wedge$
$\overleftarrow{AX}(\mathit{node}(h_1))\ @\ p_2 \wedge$
$EX(\mathit{node}(cont) \wedge out\_loop_2)\ @\ h_2 \wedge$
$same\_instr(p_1, p_2) \wedge same\_instr(h_1, h_2) \wedge$
$\mathit{basic\_induction\_var}_1(x, c, t_1) \wedge \mathit{basic\_induction\_var}_2(x, c, t_2) \wedge$
$EX_{seq}(out\_loop_1)\ @\ h_1\ \wedge EX_{seq}(out\_loop_2)\ @\ h_2 \wedge$
$A(\mathit{independent}\ U\ \mathit{out\_loop}_2)\ @\ h_2 \wedge$
$\mathit{conlit}(i) \wedge \mathit{conlit}(n) \wedge \mathit{conlit}(c)$

Figure 3.16: Specification of loop fusion

The complete specification of loop fusion is shown in Figure 3.16. Note that this version implements a restricted version of fusion, as the definition of *independent* does not capture all independent uses within the second loop.

Partial redundancy elimination

Partial redundancy elimination transforms cases like the one shown in Figure 3.17. The calculation of the expression `a + b` at node n will have already been computed if one path is taken but not if the other path is taken. The transformation adds the calculation of the expression to the other path as well, making the calculation at node n fully redundant. The idea is that after partial redundancy elimination, common subexpression elimination can remove the calculation at node n thus improving performance of the left branch of computation.

Expression e is said to be *available* at point p if there is some point on every

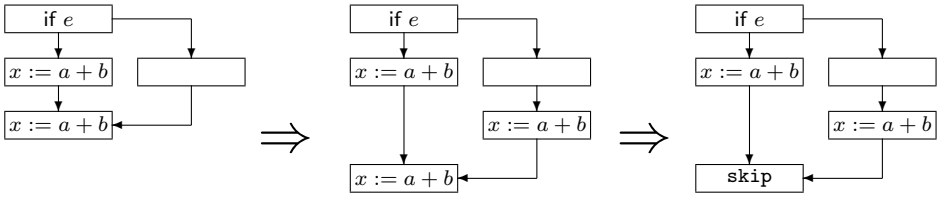


Figure 3.17: Partial redundancy elimination

program path to p where the expression is calculated and if the same expression were evaluated at p it would result in the same value. This is captured by specifying that for every path backwards from that point p a calculation of the expression is reached before any of the constituents of that expression is reached. This concept of availability, as well as the notion of an expression being available on *some* path, are captured by the definitions:

$$let\ \mathit{avail}(e) \triangleq \overleftarrow{A}(\mathit{trans}(e)\ U\ \mathit{use}(e))$$

$$let\ \mathit{avail\_one}(e) \triangleq \overleftarrow{E}(\mathit{trans}(e)\ U\ \mathit{use}(e))$$

These two notions can be combined to specify *partial availability*, defined as a point where on some paths the expression is available but not on all paths—the situation to be eliminated:

$$let\ \mathit{partial\_avail}(e) \triangleq \mathit{avail\_one}(e) \wedge \neg \mathit{avail}(e)$$

To eliminate the partial redundancy, calculations of an expression are placed at the point where they become unavailable—a point that has predecessors where the expression is available and predecessors where it is unavailable:

$$\overleftarrow{EX}(\mathit{avail}(e)) \wedge \overleftarrow{EX}(\neg \mathit{avail}(e))$$

However, these calculations should be placed where they will not cause extra calculations

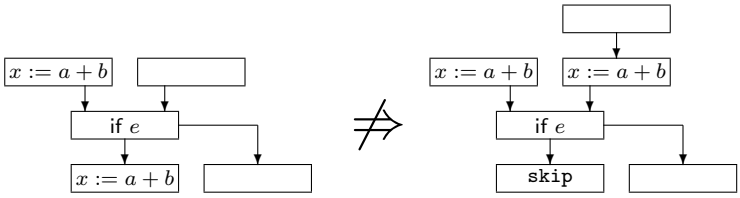


Figure 3.18: Inappropriate use of partial redundancy elimination

on other paths. Figure 3.18 illustrates a case where a calculation should not be moved, as there is a chance the result will never be used.

A safe place to add a computation of an expression to eliminate a redundancy at node n can be identified by showing that all paths at that place must lead to node n, not altering any of the constituents of e along the way. The predicate *trans*(e) holds true at nodes where e isn't altered. The fragment $A($*trans*$(e)\ U$ *node*$(n))$ satisfies this extra property. This also finalises the definition of a possible placement (pp) of a computation of expression e to eliminate a partial redundancy at node n:

$$let\ \textit{pp}(n,e) \triangleq \neg \textit{avail}(e) \wedge \overleftarrow{EX}(\textit{avail}(e)) \wedge A(\textit{trans}(e)\ U\ \textit{node}(n))$$

Partial redundancies are eliminated at points that satisfy three properties. An expression must be calculated there, which occurs for expression e in our full specification at node n. That expression is partially available, as denoted by the macro $partial\_avail(e)$. Finally every backward path either leads to a point where the expression is available or could be made available with a possible placement. The $\overleftarrow{A}(...\ U\ ...)$ construct is used in order to check backwards paths for temporal properties. We use the *trans*(e) predicate in order to ensure that none of the components of e can be changed. Finally *pp*$(n,e) \vee$ *avail*(e) ensures that the node is a possible placement or the expression

MATCH
  $(\mathit{use}(e) \wedge \mathit{partial\_avail}(e) \wedge \overleftarrow{A}(\mathit{trans}(e)\ U\ \mathit{pp}(n,e) \vee \mathit{avail}(e)))$ @ n
  $\mathit{fresh}(h)$
IN
  APPLY_ALL
    $\mathit{split\_edge}(p, m, z, (h := e))$
  if
    $\mathit{pp}(n,e)$ @ $m\ \wedge\ \overleftarrow{EX}_z(\mathit{node}(p)\ \wedge\ \neg\mathit{avail}(e))$ @ m

Figure 3.19: Specification for partial redundancy elimination

e is available there.

$$\mathit{partial\_avail}(e) \wedge \overleftarrow{A}(\mathit{trans}(e)\ U\ \mathit{pp}(n,e) \vee \mathit{avail}(e))\ @\ n$$

If there is such a node, the transformation shown in Figure 3.19 places the computation of e between each possible placement and each of its predecessors for which e is not available. This leads to conditions where common sub-expression elimination and dead code elimination are applied to remove the calculation at node n. Partial redundancy elimination optimizations generally perform what is known as critical edge splitting [41] which increases applicability of an optimization. We assume edge splitting has been performed before this transformation is applied.

Lazy Code Motion

Lazy code motion is another form of partial redundancy elimination, with a more global view of moving around the calculation of an expression. Rather than just finding one partial redundancy to eliminate, it finds the "best" places to calculate any expression calculated in the code. The transformation only moves the expression as far away from the original computation as needed to remove redundancies. This reduces any harm to

the performance of register allocation on the program. Our formulation of Lazy Code Motion very closely follows that of Steffen and Knoop [41].

The first property of interest to this transformation is *down safety*. A program point is down-safe with respect to an expression e if all paths from that point reach a calculation of e without redefining any of the constituents of e.

$$let\ \mathit{d\_safe}(e) \triangleq A(\mathit{trans(e)}\ U\ \mathit{use}(e))$$

Computations can be placed at down-safe points. The earliest computation point at which e must be computed is one at which there exists a path backwards that has no down-safe points until it reaches a point where one of the constituents of e is redefined, if it exists:

$$let\ \mathit{earliest}(e) \triangleq \overleftarrow{E}(\neg \mathit{d\_safe}(e)\ U\ (\neg \mathit{trans}(e) \vee start))$$

The points that are both down-safe and earliest provide sufficient criteria for where to place calculations of e to eliminate partial redundancy, but ideally the computation should be placed as close before the point of redundancy as possible. We can define places we could safely place a computation of e after an *earliest* placement. A *later* placement is one that on all backward paths from that point can find a down-safe and earliest place without going through a computation of e (in which case we have not gone too far).

$$let\ \mathit{later}(e) \triangleq \overleftarrow{A}(\neg \mathit{use}(e)\ U\ \mathit{d\_safe}(e) \wedge \mathit{earliest}(e))$$

The latest placement point is now defined as being a *later* placement point that either computes e or does not have later placement points on all of its successors.

$$let\ \mathit{latest}(e) \triangleq later(e) \wedge (\mathit{use}(e) \vee \neg AX(later(e)))$$

Computations of e are inserted at points that satisfy this *latest* predicate. These points cover all the computations of e but are as close to the original computations as possible. The transformation inserts the calculation of e at this point and store the result in some new variable h. All the other computations of e will then just use the result stored in this variable. However, this is not always ideal since a node that satisfies *latest* may calculate the result and put it in h only to use it straight away and never later on in the program. There is no need to introduce the new calculation when the only place it would be used is the node it was introduced. To avoid this situation, the *isolated* predicate identifies when a node will not pass on the use of a computation of e:

$$let\ \mathit{isolated}(e) \triangleq AX(A(\neg \mathit{use}(e)\ U\ \mathit{latest}(e)))$$

The transformation *insert*(h, e) inserts a calculation of e (storing it in variable h) after nodes that satisfy *latest*(e) but not *isolated*(e):

$$let\ \mathit{insert}(h, e) \triangleq m : s \Longrightarrow s; h := e \text{ if } (\mathit{latest}(e) \wedge \neg \mathit{isolated}(e))\ @\ m$$

Calculations of e can be removed at program points that satisfy *redundant*(e), that is points that are neither *latest* or *isolated*.

$$let\ \mathit{redundant}(e) \triangleq \neg(\mathit{latest}(e) \vee \mathit{isolated}(e))$$

The removal transformation depends on where the redundant calculation occurs, so it can be written in three variations: one for when the calculation occurs in an assignment, one for when it occurs in a conditional and one for when it occurs in a return statement.

$$\begin{aligned}
\mathit{let\ rem_assign}(h, e) &\triangleq m : x := c[e] \Longrightarrow x := c[h] \text{ if } redundant(e)\ @\ m \\
\mathit{rem_branch}(h, e) &\triangleq m : \texttt{if}\ \ (c[e]) \Longrightarrow \texttt{if}\ \ (c[h]) \text{ if } redundant(e)\ @\ m \\
\mathit{rem_return}(h, e) &\triangleq m : \texttt{ret}(c[e]) \Longrightarrow \texttt{ret}(c[h]) \text{ if } redundant(e)\ @\ m
\end{aligned}$$

MATCH
  partial_avail(e) @ n
  fresh(h)
IN
  APPLY_ALL (*insert*$(h, e) \,\square$ *remove*(h, e))

Figure 3.20: Specification of lazy code motion

Then *remove*(h, e) is the transformation that removes a redundant use of e (by using the variable h instead).

$$let\ \textit{remove}(h, e) \triangleq rem\_assign(h, e) \,\square\, rem\_branch(h, e) \,\square\, rem\_return(h, e)$$

The complete transformation, shown in Figure 3.20, finds an expression that is partially redundant and then performs all applications of both *insert* and *remove*.

Lazy Strength Reduction

Lazy strength reduction [42] is a transformation that combines strength reduction with code motion. Code motion recognises paths where an expression is available; with strength reduction the expression to be reduced is not directly available but can be made available by altering the paths leading up to its calculation.

The goal is to eliminate partially redundant calculations such as of the expression $v * c$ where v is a variable literal and c is a constant literal. Variable v is said to be *injured* at a node if v is redefined at this node solely by addition of a constant value. Formally:

$$let\ \textit{injured}(v) \triangleq \exists d.\textit{stmt}(v := v + d) \wedge \textit{conlit}(d)$$

The value of the expression $v * c$ after execution of a node at which v is injured can be found by adding on the constant $c * d$. Using the notion of *injured* nodes the

trans predicate can be redefined for the expression $v * c$ to say that either none of the constituents in the expression are redefined or that v is merely injured.

$$let\ \mathit{trans}_{sr}(v,c) \triangleq \neg def(v) \vee \mathit{injured}(v)$$

This adjusted trans_{sr} predicate allows the definition of the $\mathit{d\_safe}_{sr}$, $\mathit{earliest}_{sr}$, later_{sr}, latest_{sr} and $\mathit{isolated}_{sr}$ predicates, analogous to the predicates used for specifying lazy code motion:

$$\begin{array}{lll}
let\ \ \mathit{d_safe}_{sr}(v,c) & \triangleq & A(\mathit{trans}_{sr}(v,c)\ U\ \mathit{use}(v*c)) \\
let\ \ \mathit{earliest}_{sr}(v,c) & \triangleq & \overleftarrow{E}(\neg \mathit{d_safe}_{sr}(v,c)\ U\ \neg \mathit{trans}_{sr}(v,c)) \\
let\ \ \mathit{later}_{sr}(v,c) & \triangleq & \overleftarrow{A}(\neg \mathit{use}(v*c)\ U\ \mathit{d_safe}_{sr}(v,c) \wedge \mathit{earliest}_{sr}(v,c)) \\
let\ \ \mathit{latest}_{sr}(v,c) & \triangleq & \mathit{later}_{sr}(v,c) \wedge (\mathit{use}(v*c) \vee \neg AX(\mathit{later}_{sr}(v,c))) \\
let\ \ \mathit{isolated}_{sr}(v,c) & \triangleq & AX(A(\neg \mathit{use}(v*c)\ U\ \mathit{latest}_{sr}(v,c)))
\end{array}$$

Transformations $insert_{sr}(v,c,h)$ and $remove_{sr}(v,c,h)$ are defined analogously to the transformations used in the specification of lazy code motion. In addition, any node that injures the value $v * c$ must be altered to update the value h. The optimizer should do this only on *injured* nodes that have a path to a node that will use h *i.e.* a node satisfying $\mathit{redundant}(v,c)$. The $adjust$ transformation is therefore defined as:

$let\ \mathit{adjust}(v,c) \triangleq$

$m : v := v + d \Longrightarrow v := v + d; h := h + step$

if $step$ is $d * c$

$injured(v)$ @ m

$E(\neg \mathit{latest}_{sr}(v,c) \vee \mathit{isolated}_{sr}(v,c)\ U\ redundant_{sr}(v,c))$ @ m

The specification of lazy strength reduction, shown in Figure 3.21, is similar to lazy code motion but uses the new predicates and the additional *adjust* transformation.

This specification illustrates the strength of our approach, in which small transformations are defined independently—and checked for correctness—and then combined

MATCH
  $\mathit{partial\_avail}_{sr}(v * c) \wedge \overleftarrow{A}(\mathit{trans}_{sr}(e)\ U\ \mathit{pp}(n,e) \vee \mathit{avail}(e))$ @ m
  $\mathit{fresh}(h)$
IN
  APPLY_ALL $(insert_{sr}(v,c,h) \Box remove_{sr}(v,c,h) \Box adjust(v,c))$

Figure 3.21: Specification of lazy strength reduction

using strategies, resulting in the full optimizations. Each individual transformation may not result in improvement of code, but it is the composition, either through specific strategies or indeed an overall loop, which results in more efficient code. And each transformation can be re-used when devising new optimizations.

3.7 Comparison with other Transformation Languages

3.7.1 DFA&OPT-Metaframe

The system based on TRANS, as it stands, is not as suitable for compiler construction as the DFA approach described in Chapter 2. Due to the high-level and general nature of TRANS, the implementation will not necessarily be as fast as the more specific propositional temporal logic analysis found in Metaframe. In Chapter 4 we consider a performance evaluation of a TRANS implementation against hand written optimisations. Metaframe includes a Turing complete programming language, more expressive than TRANS, albeit more difficult to reason about. TRANS has some advantages over DFA. Firstly the approach being entirely declarative rather than imperative makes it easier to reason about formally. For example, previous work [46; 47] shows precisely how optimization specifications can be proved to be sound, that is, semantics preserving. Chapter 6 describes how this can be formalised in the context of a realistically

$$let \quad \textit{flow\_dep}(n,m) \quad \triangleq \quad \exists x.\textit{def}(x) \ @ \ \mathsf{m} \ \wedge \overleftarrow{A}(\neg \textit{def}(x) \ U \ \textit{node}(m)) \wedge \textit{use}(x) \ @ \ n$$
$$let \quad \textit{anti\_dep}(n,m) \quad \triangleq \quad \exists x.\textit{use}(x) \ @ \ \mathsf{n} \ \wedge \overleftarrow{A}(\neg \textit{def}(x) \ U \ \textit{node}(n)) \wedge \textit{def}(x) \ @ \ m$$

Figure 3.22: Flow and Anti Dependancies

Java like language and checked by a mechnised theorem prover. Secondly, having the side conditions and transformations combined into a single language makes it easier to understand the program transformations than in the DFA approach.

3.7.2 Gospel

Each of the dependencies listed in Figure 2.2 can be expressed in TRANS. For example flow and anti dependancy macros are listed in Figure 3.22. GOSpeL also allows these dependencies to be altered with *direction vectors*. For example, the dependency 'flow_dep(n,m,<)' states that an array element definition at n is indexed at a place before (under the order of some iterative loop) the index of an array element used at point m. Such predicates cannot be written in TRANS and the language would have to be extended to handle inequality constraints.

Another aspect of GOSpeL is that it can match patterns binding variables to whole blocks of code and then move and modify these to enable optimizations such as loop unrolling and inlining. Again, this is not currently possible in TRANS but conservative extensions of block matching operators have been investigated [45]. Some of the transformations described in this thesis (such as partial redundancy elimination) have not been investigated in the context of GoSPeL.

Figure 3.23 shows the specification of constant propagation in GOSpeL; we see that this specification can quite easily be converted into TRANS. The ACTION part of the specification states that the statement Sj is modified by replacing a sub-term with

```
TYPE
  Stmt: Si,Sj,Sl;

PRECOND
  Code_Pattern        /* Find a constant definition */
    any Si: Si.opc == assign AND type(Si.opr_2) == const;

  Depend        /* Use of Si with no other definitions */
    any (Sj,pos):flow_dep(Si,Sj,(=));
    no (Sl,pos):flow_dep(Sl,Sj,(=)) AND (Si != Sl)
        AND operand(Sj,pos) != operand(Sl,pos);

ACTION        /* Change use of Si in Sj to be constant */
  modify(operand(Sj,pos),Si.opr_2);
```

Figure 3.23: Specification of constant propagation in GOSpeL

a constant term, in TRANS this is specified as:

$$Sj : (y := e[x]) \Longrightarrow y := e[c]$$

The Code_Pattern in the specification binds a statement to Si which assigns a variable to a constant, which is simple to write in TRANS: $x := c$ @ $Si \ \wedge \mathit{conlit}(c)$

The DEPEND part of the GOSpeL statement contains two parts, the first indicating that Sj is flow dependent on Si, and the second that Sj is dependent on no statement other than Si. Flow dependence indicates whether a variable is used in a statement that has a defining instance at a point it is dependent on. It can be defined in TRANS in the following way:

$$let\ flow\_dep_{=}(x, n, m) \triangleq \mathit{use}(x) \ @\ m \ \wedge \mathit{def}(x) \wedge E(\neg \mathit{def}(x)\ U\ \mathit{node}(m)) \ @\ m$$

The complete direct translation into TRANS of the GOSpeL specification is:

$$Sj : (y := e[x]) \Longrightarrow y := e[c]$$

if

$$x := c \wedge \mathit{conlit}(c) \ @ \ Si$$

$$flow\_dep_{=}(x, Si, Sj)$$

$$\neg\exists Sl.\ flow\_dep_{=}(x, Sl, Sj) \wedge \neg \mathit{node}(Si) \ @ \ Si$$

The TYPE and PRECOND components of GOSpel transformations are together are equivalent to the condition part of TRANS specifications. Consequently there is some similarity between the overall structure of the specifications.

Overall, the philosophy of the TRANS approach is different from GOSpeL in that analyses in TRANS are broken down into smaller components, rather than development of specific (and potentially quite complicated) analyses. This has the advantage of increasing the expressiveness and allowing more uniform formal analysis. Nevertheless, some analysis has been done on transformations in GOSpeL, in particular an approach to prove (by pen-and-paper) that disabling interference does not occur between two transformations is provided in [89].

3.7.3 Optimix

An important difference between TRANS and the Optimix system is that Optimix uses the graph to store intermediate analysis information required for the transformations. The temporal logic formula that TRANS uses provides all this information in one step and abstracts away the detail of each step from optimization writer.

3.8 Conclusions

This chapter describes the semantics and applicability of past work on the TRANS language . Specific examples are shown that correspond to program transformations that are commonly used as compiler optimizations. The TRANS language is designed to be used for specifying different types of optimizations. The Lazy Code Motion and Lazy Strength reduction optimizations are inherently complex, however they can be specified in TRANS. Since the TRANS language operates over a control flow graph — a generic representation — it can be applied to a wide variety of different languages. For example Chapter 4 describes how the TRANS approach can work with `Java`.

Little work has been previously done on implementing an efficient compiler generator for TRANS. [44] outlines an interpretation algorithm over the aforementioned toy language. No empirical performance analysis was undertaken. Furthermore, the TRANS language was used only for compiler optimisations, and not program transformations in general. In contrast this thesis discusses how to extend the language to source code and apply it to automatically fix bugs in computer programs.

Chapter 4

Implementation of an Optimisation Generator

4.1 Introduction

The Rosser toolkit has been developed, during the course of this PhD, in order to allow optimizations to be specified in a domain specific language and then compiled and deployed towards optimizing object programs. The optimizers generated by Rosser exploit model checking to apply dataflow analysis to programs to find optimizing opportunities. We validate the technique by comparing the application of optimizers generated by our system against hand-written optimizations using the Java based Scimark 2.0 benchmark [68].

An optimization phase is an integral part of most real-world compilers, and significant effort in compiler development is spent in obtaining fast-running code. This effort must be balanced with the need to ensure that optimizations do not introduce errors into programs, and the desire to not worsen compilation time significantly. Rosser

allows the application of specifications of compiler optimizations to Java Bytecode. Optimisations are matched against programs using model-checking, and graph rewriting is used to actually modify the programs.

The contributions from the design of Rosser to the design of compilers include:

- An implementation that automatically generates optimizations from specifications and can be practically used against a real world programming language.
- A novel intermediate representation of Java programs, that uses BDDs to aid in symbolic model checking.
- A method of interactively and visually rewriting the control flow graph (CFG) of Java programs using the Rosser system.
- A case-study backed analysis of the performance ramifications of using model checking for dataflow analysis compared with hand-written analysers.

This chapter discusses the design of Rosser and provides some experimental results. We also consider our BDD variable approach and how this is influential in the performance of the generated compiler optimizations. This is validated with empirical analysis.

4.2 Binary Decision Diagrams

Much of the discourse within this thesis concerns the applicability of temporal logic derived languages for program transformation. Efficient implementation of such transformations demands efficient model checkers, that can be derived from temporal logic specifications. A technique that we use for representing the intermediate valuations within Section 5.5 is the Binary Decision Diagram, or BDD [11].

A BDD represents a Boolean Function, that is to say functions from $\{0,1\}^n$ to $0,1$. Binary Decision Diagrams are an extension of Binary Decision Trees. Binary Decision Trees are graphs, where every leaf node is either labelled 0 or 1. All other nodes are branches with two sub nodes, the edge to one node is labelled 1 and the other 0. In diagrams we adopt a convention of marking the 1 edge as a dotted line, and 0 as a dashed line. Each level within the tree is labelled with an index into the arguments of the boolean function.

A Binary Decision Diagram can be considered a generalised Binary Decision Tree. BDDs are normally considered in their reduced and ordered form, we now describe the optimizations that are applied in order to reduce the size of a BDD.

1. End Points—A single node for each of 0 and 1 is introduced, and all leaf nodes are pointed to the appropriate node.

2. Unnecessary Node Removal—Every node, *n*, whose successor edges both go to the same node, *m*, is removed. The incoming edges of *n* are redirected to *m*.

3. Removing subBDDs—a subBDD is a part of a BDD that occurs below a given node. If two subBDDs are identical then one of them is removed and all incoming edges are redirected to the other.

The size of BDD representations is often dependant upon the ordering of their variables. This is because different variable orderings give rise to more or less opportunity for the three optimizations above to be applied.

Reduced and Ordered Binary Decision Diagrams are used because of the existence of efficient algorithms for performing logical operations over them. We can apply a mathematical operation $\oplus$: 0,1 $\rightarrow$ 0,1 (such as $\wedge$ or $\vee$) to BDDs. Given BDDs `A` and `B` the efficiency of applying these operations is at most $\mathcal{O}(|\texttt{A}|.\ |\texttt{B}|)$ [34]. In practise it

Operation	Comment
$x = 0B$	Assigns the empty set to relation x
$x = 1B$	Assigns the set of all possible elements to relation x
$(x =>)\ r$	projects attribute x away from relation r
$(x => y)\ r$	renames attribute x, from relation r to y
$(x => x\ y)\ r$	copies attribute x, from relation r to y
$r1\ \&\ r2$	Intersection of relations $r1$ and $r2$
$r1 \mid r2$	Union of relations $r1$ and $r2$
$r1 - r2$	Set Difference of relations $r1$ and $r2$
$r1\{x\} >< r2\{y\}$	Joins relations $r1$ and $r2$ where x equals y, projecting y
$r1\{x\} <> r2\{y\}$	Joins relations $r1$ and $r2$ where x equals y, projecting x and y

Table 4.1: JEDD operations

is usually lower. Checking the equivalence of two BDDs is $\mathcal{O}(1)$. The use of ROBDDS in model checking is known as symbolic model checking.

BDD Implementation

The Soot compiler framework [81], that is used by the implementation described in this chapter, provides a toolkit for the exploitation of BDD based dataflow analyses within compiler optimizations. The use of Binary Decision Diagrams (BDDs), in data-flow analysis applications, has been shown by [51] to have the potential to significantly reduce memory consumption and improve runtime performance. The TRANS implementation described in this chapter uses an existing implementation for representing BDDs and operations over them called JEDD.

JEDD is an extension to Java that allows a higher level representation of BDDs developed by [50]. Relations are introduced as a primitive type within JEDD, and several operations, such as union, intersection, difference and comparison are defined over them. BDDs can be directly coded as relations. JEDD has been used as the basis for implementing inter-procedural data flow analysis [6]. The operators of JEDD are summarised in Table 4.1

4.3 Architecture

4.3.1 Architectural Overview

The Rosser compiler framework comprises three components. A meta-compiler, RosserC, translates TRANS specifications to produce the code for the optimizing phase. Every optimization specification is compiled into the general form of finding satisfying valuations for its side condition, by application of its side condition to the intermediate representation. The program generated (referred to as RosserS) is loaded into the runtime framework and applied to a program via the `Soot` framework.

The `Soot` framework provides an intermediate representation for `Java` programs called `Jimple`. In `Jimple` expressions are represented as trees, at a `Java`-like level, and control flow at a lower level utilising basic conditionals and goto statements [80]. `Soot` also provides an implementation for generating the `Jimple` Intermediate Representation from `Java Bytecode`. We introduce a new language called *Dimple*—a representation equivalent to `Jimple` in overall structure, but using BDDs instead of Plain Old `Java` Objects (POJOs) in order to implement the optimizations. *Dimple* represents the relations between parents and children as `Jimple` expression trees. The translation between `Jimple` and *Dimple* is done through the RosserF framework. Only parts of the program that are relevant to the optimizations are translated, since only some components of the program need to be pattern-matched. For example since we specify no inter-procedural optimizations there is no representation of the class hierarchy in our implementation. These interactions are illustrated in Figure 5.19.

The RosserC compiler for TRANS specifications comprises a series of phases. After a specification has been lexed and parsed it is converted into an intermediate representation which closely follows the base structure of TRANS, with a different `Java`

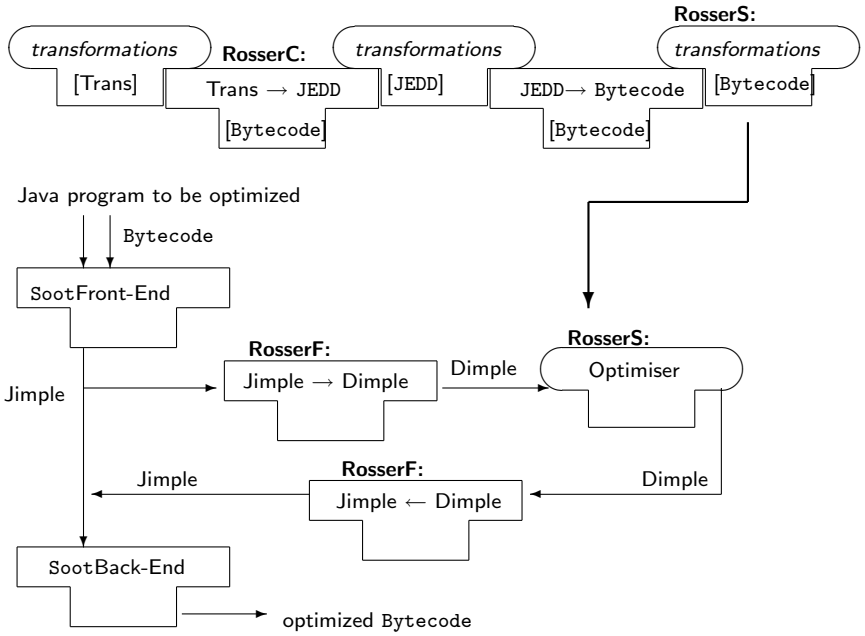


Figure 4.1: Architecture of the Rosser framework

`class` to represent each different construct within the language. There are, however, several additional intermediate representation elements that allow small components of the language to be refined into a lower level intermediate representation. These are carried out ahead of the output of object code in order to allow a modular implementation of the compiler architecture.

The architecture of many modern compilers, for example GCC and `javac`, follows a design pattern known as the *visitor pattern* [25]. Orthogonal modularity between data structures and algorithms is enabled, by splitting data into different *kinds* and algorithms into different *interpretations*. In this way algorithms and data are orthogonally modular. Within our compiler a kind is one element of intermediate representation and an interpretation is a phase with the compiler.

The design of RosserC uses a general *Visitor* interface that implements refinement and output phases, while the use of dynamic class loading allows one to configure which phases are run at runtime. The order in which phases are executed therefore depends on a configuration file, which also allows setting of global options. The 'Phases' section of the configuration file takes a list of class names to execute. Each class must implement the Visitor interface and be within the class-path of RosserC at execution. The 'Properties' section takes a list of global properties that are passed to each phase as a map. This, combined with the Intermediate Representation, defines the environment in which each phase executes.

The refinement of CTL is implemented by *CTLVisitor*, and the use/def refinements by *MayMustVisitor*. The `MayMustVisitor` code performs translations from `use` and `def` into their may/must equivalents, whilst `mayuse`, `mustuse`, `maydef`, `mustdef` are examples of components existing in the Intermediate Representation that are not defined in TRANS.

4.3.2 Use/def analysis

The first phase in translation is the refinement of `use` and `def` predicates. These predicates hold true at statements where the variable that they refer to is either read or written to. In `Java`, as with other programming languages, it is possible for several variables to alias the same heap object. This affects `use` and `def` predicates because it requires a refinement of the semantics to *variables that alias heap objects* that are either read or written to. Since assignment may differ depending on what path through the program's control flow graph was taken, the aliasing relationship depends on this path.

As with traditional dataflow analyses [1], the approach taken in Rosser is to divide relationships into 'must' and 'may' forms. For example, `MustUse(x) @` n, indicates that for all execution paths at node n, x is used in the computation that occurs at node n. If `MayUse(x) @` n, then there exists an execution path such that, at node n, x is used in a computation. The situation is symmetric for `MayDef` and `MustDef`. TRANS specifications, however, do not use these conditional variants of the predicates and must be refined accordingly by RosserC.

In order to be a sound refinement of the predicates in TRANS it is necessary for the may/must variants to conservatively approximate their behaviour. That is to say, they must never enable an optimization that would otherwise be disabled. The underlying idea is to differentiate between predicates that enable or disable transformations.

Consider `use(x)`, Rosser needs to choose to replace `use` with either `mayuse` or `mustuse`. `mayuse` holds true at a super set of nodes where `use` holds true, whilst `mustuse` holds true true at a subset of these nodes. If the side condition of a transformation is only `use(x)` then that transformation will be applied at more places if there are more places where `use(x)` holds true. So if the refinement were to substitute `use` for `mayuse` then it would enable the transformation at unsound places in the program,

however mustuse won't cause any unsound transformations.

The inverse of this example holds true if we consider a transformation with a side condition that consists entirely of $\neg$ use(x). Here if the refinement algorithm substituted mustuse for use it would cause the transformation to be applied at unsound locations, since $\neg$ mustuse(x) holds true at locations that $\neg$ use(x) does not. mayuse is the sound substitution in this circumstance. Its undesirable to have to 'hard code' every possible side condition as to whether it should substitute the mayuse or mustuse.

It is thus necessary to develop an approach to substitution that never enables a transformation where it the use specification doesn't require it be enabled. In the two examples above it was determined that mustuse was a sound substitution in the case of use(x) and that mayuse was a sound substitution in the case of $\neg$ use(x). In order to generalise these substitution rules to use predicates arbitrarily nested within CTL formulae we introduce the concept of *polarity*.

A CTL formula can be viewed as a tree, with its outermost connective at the top, and each leaf consisting of a predicate or true or false. The polarity of predicate p is positive if there is an even number of negations on the path through the tree from the top most element to p. It is negative otherwise.

If a predicate has a positive polarity then nodes where it holds true are increasing the set of possible points at which a transformation can be applied. The use predicate in the example use(x) has a positive polarity, since there are 0 $\neg$ symbols. If the polarity is negative, then the predicate holding true disables possible transformations, a generalisation of the $\neg$ use(x) case.

Since the must variant of a predicate holds true at a subset of nodes where the predicate holds true it is always a sound substitution for a predicate with a positive polarity. The may variant of a predicate is a sound substitution for a scenario when the

use of a predicate in a side condition has a negative polarity.

This approach to refining use/def predicates means that specifications written in the TRANS language are portable over both languages that allow and disallow aliasing. This has the added advantage of facilitating prototyping of optimizations in simple contexts (for example against Local primitives, which are pass by value) and then be able to apply them in more complicated situations. This underlies one of the design principles of Rosser: to move the burden of compiler development away from the optimization specification and into the framework.

4.4 Representation

The *Dimple* representation introduced in this thesis offers a novel approach to the intermediate representation of programs. Whilst BDDs have been used as the basis of representing sets of data during dataflow analysis [6], they have not been used to represent entire programs before.

Information about the program being optimized is split into several domains. Each domain contains a numbered set of elements that can be used within the `JEDD`system. These domains define the type system of Dimple.

OP consists of all operators represented in `Jimple`, for example addition, and negation.

ET is the edge type domain and contains three possible values: sequential, branch and exception, and is used in pattern matching different edges.

Node lists every node in the control flow graph.

Call references every method invocation.

Value contains an entry for every possible value in the program, for example the expression $x + 1$, contains an entry for x, 1 and $x + 1$.

Name	Type	Comment/Example
Nodes	⟨Node⟩	All nodes in CFG
Skips	⟨Node⟩	All Noop instructions
Edges	⟨Node,Node,ET⟩	x = 1; y = x →⟨x = 1, y = x, SEQ⟩
ReturnValues	⟨Node,Value⟩	return x; →⟨return x, x⟩
Assign	⟨Node,Value,Value⟩	y = z + 1 →⟨y = z + 1,y,z + 1⟩
IfStmt	⟨Node,Value⟩	if (x==3) →⟨if (x==3), x==3⟩
Expr	⟨Value,Value,Value,OP⟩	x + y →⟨x + y, x, y, +⟩
UExpr	⟨Value,Value,OP⟩	!x →⟨!x, x, !⟩
Conlit	⟨Value⟩	All Constants, eg ⟨1⟩
Varlit	⟨Value⟩	All Variable literals, eg ⟨x⟩
CallSites	⟨Value,Call⟩	Relation between call sites and values
MustDef	⟨Node,Value⟩	At x = 3; ⟨x = 3,x⟩

Table 4.2: Representing programs in *Dimple*.

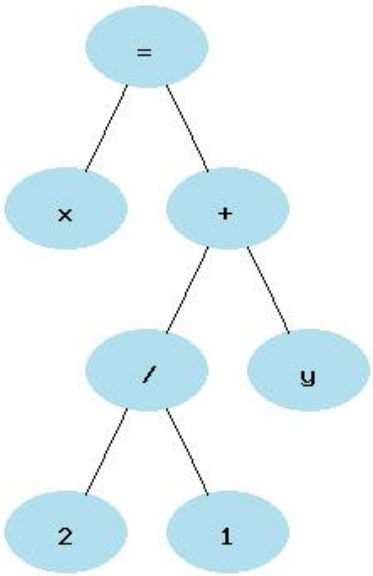
Table 4.2 summarises the translation of the syntactic components of programs in Jimple. Recall that use and def are predicates that hold true if their argument (a variable) is read from or written into at a given node. This fragment of the translation presents the MustDef relation, while similar treatment applies to MayDef, MayUse and MustUse relations, which all store information about different use/def chains in the same format. The next section describes these relations in more detail.

Figure 4.2 shows both representations of a simple statement.

4.5 Optimiser Generation Strategy

4.5.1 Refinement and Type-Checking

Before specifications are applied to programs they undergo a series of refinement steps. The first step of refinement is to rewrite the CTL formulae. This reduces the possible connectives that can be used in side conditions in order to simplify the output phases.



Node	Variable	Value
n	x	0

Index	Val-1	Val-2	Op
0	1	y	+
1	'2'	'1'	/

The Node & Quad-Values Relations

Figure 4.2: The statement `x = 2 / 1 + y` as a `Jimple`tree and Dimple tuples

The rewrites used are listed in Table 4.3 and are applied exhaustively.

Before	After
AG p	¬ E (true U ¬ p)
A (p U q)	¬ (E (¬ q U (¬ p ∧ ¬ q)] ∨ EG ¬ q))
AX p	¬ EX ¬ p

Table 4.3: Temporal Logic Refinements

Rewrite rules are refined to a pattern matching component, which becomes part of the side condition, and a TRANS action. In the case of dead code elimination, this is the *replace* action, which swaps an existing node bound to a meta-variable, inserting an Intermediate Representation element generated from variable bindings in its place.

RosserC also performs type checking. The goal is to statically identify the types of all the meta variables within the TRANS specification. This is beneficial for two

replace `n with skip`

if

`stmt(x := e) @ n` $\wedge \neg$ `EX (E (` $\neg$ `maydef (x) U mustuse (x)` $\wedge \neg$ `node(n) )) @ n`

Figure 4.3: Refined specification of dead code elimination

reasons. Firstly the output code is statically typed, and so type checking TRANS formulae helps generate object code. Secondly it is helpful in order to reduce the number of accidental or transcription errors within TRANS formulae. If a meta-variable has to bind to a structure of one type in a certain place within the specification and a different type in another part, then it is clearly not a well-formed TRANS specification. Consider the hypothetical specification:

$n : x := e \Rightarrow$ skip *if* conlit(n)

This specification fails type checking because the metavariable n has to be a node in its use on the left hand side, and a constant literal if it is an argument to `conlit`.

Figure 4.3 shows the effect of refinement on the specification of dead code elimination shown in Section 3.5. Here the pattern matching has become part of the side condition and the use/def predicates have been refined.

4.5.2 Code Generation

The RosserC compiler outputs `JEDD` code, where for each optimization a corresponding class is generated. The side condition is compiled into a method called `condition`, whose return type is a relation, with an attribute for each metavariable within the specification, its only parameter being the method to be optimized. A transformation is applied through method `transformation`, which in turn calls the `condition` method and then iterates over all the values within the resulting valuation set. Generating the

`condition` method body proceeds by recursion of the structure of the now refined TRANS side conditions.

Figure 4.4 shows the compiled pattern matching for `stmt(x := e) @ n` from the specification of dead code elimination. First, a temporary attribute `x1` is introduced into the valuation to designate the current node. This can be seen in the type of the variable `x2` on line 1. Line 2 restricts this attribute to nodes. In lines 3 and 4 the variables e, n and x are restricted to the right hand side, result variable and node of assignments, respectively. Lines 5 and 6 show the temporary node being equated to n and then projected away.

The `JEDD` code shown in Figure 4.5 illustrates predicates being compiled. Line 2 shows the restriction of `mustuse` to a local finite domain. In line 4 the temporary attribute `x6`, that fulfils the same purpose as `x1` in the previous example is unified with the attribute `n`. Lines 3 and 4 calculate the set of valuations where the current node is n. Line 5 implements the $\neg$ operator, calculating valuations where the current node is not n. Finally we take the intersection of the subcomponents, in order to satisfy the $\wedge$ in the example. Note that literals after colons, for example N6, refer to physical domains that are used by the BDD implementation. The first letter is the same as the corresponding logical domain, for example N refers to Node. Since there may be multiple physical domains for each logical domain they are numbered.

```
<e,n,x,x1:N6> x2 = 1B;
x2 = x2{x1} >< meth.Nodes{n};
<e,n,x> x3 = meth.Assign;
x2 = x2{e,n,x} >< x3{e,n,x};
x2 &= (x1 => x1,x1 => n)((n => )(x2));
<e,n,x> x4 = (x1 => )(x2);
```

Figure 4.4: Compilation of `stmt(x := e) @ n`

```
<e,n,x,x6:N7> x7 = 1B;
x7 = x7{x6,x} >< meth.MustUse{n,x};
<e,n,x,x6:N8> x8 = 1B;
x8 &= (x6 => x6,x6 => n)((n => )(x8));
x8 =  1B  - x8;
x8 = x7 & x8;
```

Figure 4.5: Compilation of `mustuse(x)` $\wedge \neg$ `node(n)`

The code generation algorithm used in Rosser generates standard imperative code, using `JEDD` as its object language. The return type of the `condition` method is a relation, containing one attribute that corresponds to a TRANS metavariable in the original specification. At every stage, intermediate variables that are generated are typed as the same type as the return type. When generating conditions for node conditions, a temporal part of the condition, there is additionally an attribute that represents the current node of the specification.

Figure 4.6 describes how side conditions are compiled. The function cs compiles side conditions, whilst ct compiles the sub-expressions within side conditions that have some temporal aspect. There are a few common attributes about the way different components within the side condition introduce new temporary variables. Basic predicates, such as `node(n)`, create new temporaries. TRANS unary operators, such as $\neg$, depend on the result of their inner expression, stored in a single temporary, referred to in the definition as `pred`, while binary operators, such as $\wedge$ depend on two temporaries, `pred1` and `pred2`. All expressions store the result of their component of the model checking in a variable, referred to as `res` in the definition. Variables called `t1`, `t2` etc. refer to temporary variables within the object code of inner components. In the generated code, all these variables have disjoint names to each other, however, this is abstracted from the following section for reasons of readability. The function `cp` emits code to

pattern match an expression with a sequence of nodes. Its first parameter is the pattern to match, and the second is the node to match it at. This is also omitted from the presentation for reasons of brevity. The definition provides a mapping from TRANS Intermediate Representation to a list of JEDD instructions.

4.5.3 Action Code Generation

Figure 4.7 gives the example action for code elimination, `x := e => skip`. In the code `_it` is the name of the iterator for the results set of the analysis. Line 1 shows the loop condition over this set. Line 2 shows that each element of this set is represented by an array of elements. Line 4 constructs the replacement `skip` instruction. Line 5 replaces the old instruction with the `skip` inside of `Jimple`. Line 6 replaces it within *Dimple* by renumbering the elements. Note that `_f` is a factory class for new Pattern instances.

The replacement becomes inherently simple due to the way pattern matching is refined into the side condition. Additionally to rewriting, Rosser supports the insertion and deletion of new nodes and edges. These are all implemented similarly, iterating over the elements of the finite set of valuations and replacing each element. By renumbering elements of the underlying domain that are being rewritten in simple cases such as this, rewrite rules do not need to alter the structure of the CFG at all, and thus the CFG does not need to be recalculated.

4.5.4 Side Condition Code Generation

We now describe the implementation of the code generation algorithm in the *SootVisitor*. It is possible to re-implement the Visitor interface and simply load that class in at runtime, thus allowing a flexible redirection of the RosserC infrastructure to a different compiler architecture.

```
cs true          =  [res = 1B]
cs False         =  [res = 0B]
cs conlit(v)     =  [t1 = 1B, res = t1{v} >< Conlit{c}]
cs varlit(v)     =  [t1 = 1B, res = t1{v} >< Varlit{v}]
cs ¬ φ           =  cs φ @ [res = 1B - pred]
cs φ @ n         =  cs φ @ [res = (at =>) pred{n,at}
                    <> pred{at,n}]
cs φ ∧ ψ         =  cs φ @ cs ψ @ [res = pred1 & pred2]
cs φ ∨ ψ         =  cs φ @ cs ψ @ [res = pred1 | pred2]

ct true          =  [res = 1B]
ct False         =  [res = 0B]
ct node(n)       =  [t1=1B, res=t1{n,at} >< t1{at,n}]
ct stmt(p)       =  cp p at
ct ¬ φ           =  ct φ @ [res = 1B - pred]
ct EX[e] φ       =  ct φ @ [t1 = Edges{et}
                    >< new{et => e}{et}, res =
                    (to=>at) pred{at} <> t1{from} ]
ct EX φ          =  ct φ @ [t1 = (et=>)Edges,
                    res = (to=>at) pred{at} <> t1{from} ]
ct E[ φ U ψ ]    =  ct φ @ ct ψ @ until pred1 pred2
ct φ ∧ ψ         =  ct φ @ ct ψ @ [res = pred1 & pred2]
ct φ ∨ ψ         =  ct φ @ ct ψ @ [res = pred1 | pred2]
```

where the `until` function is defined as:

```
until pred1 pred2 = [ t1 = (et=>) Edges,
                      acc = pred2,
                      do { prev = acc;
                           t2 = (from=>) pred1{at} <> t1{to};
                          acc |= pred2 & t2
                        } while(prev != acc),
                      res = acc  ]
```

Figure 4.6: Side Condition compilation

```
while ( _it.hasNext() ) {
  Object[] _val = (Object[]) _it.next();
  try {
    Unit _x = _f.SkipPattern();
    units.swapWith(((Unit) _val[1]), _x);
    ObjNumberer.patch(((Unit) _val[1]), _x);
  } catch(Throwable t) {
    System.err.println(t);
  }
}
```

Figure 4.7: JEDD code for the *replace* action

An example of this operation is given in Section 4.5.2. This generates standard imperative code, using JEDD as its object language. The return type of the `condition` method is a relation, that contains one attribute that corresponds to a TRANS metavariable in the original specification. At every stage, intermediate variables that are generated are typed as the same type as the return type. When generating conditions for node conditions, a temporal part of the condition, there is additionally an attribute that represents the current node of the specification.

Figure 4.6 describes how side conditions are compiled. The function cs compiles side conditions, while ct compiles the sub-expressions within side conditions that have some temporal aspect. There are a few common attributes about the way different components within the side condition introduce new temporary variables. Basic predicates, such as `node(n)`, create new temporaries. TRANS unary operators, such as $\neg$, depend on the result of their inner expression, stored in a single temporary, referred to in definition as `pred`, while binary operators, such as $\wedge$ depend on two temporaries, `pred1` and `pred2`. All expressions store the result of their component of the model checking in a variable, referred to as `res` in the definition. Variables called `t1`, `t2` etc. refer to temporary variables within the computation inner components. In the generated code, all these variables have disjoint names to each other, however, this is abstracted from the following section for reasons of readability. The function cp emits, code to pattern match an expression with a sequence of nodes. This is omitted from the presentation for reasons of brevity. The definition provides a mapping from TRANS Intermediate Representation to a list of JEDD instructions.

4.6 Interactive and Batch Mode

The RosserF runtime framework can be operated in one of two main modes: interactive or batch. The interactive mode is designed to allow the user to develop new optimization specifications, while batch mode is a traditional compiler process that applies a list of optimizations sequentially. The interactive mode has been developed on the principle that the development of new ideas is informed by experiment. Building on this principle, interactive mode allows one to develop a specific method to apply to the program being optimized.

The interactive mode provides a *conditional* sub-view and a *transformational* sub-view. The *conditional* view provides the user with a view of the control flow graph of the selected method. The user can then enter a side condition, with which to model check the program. This then generates a set of valuations for the given program, and a visual representation of the valuations on the control flow graph. The *transformational* view allows the user to apply complete TRANS transformations to the selected method and visually see the results, in the form of before and after control flow graphs.

The ability to allow the user to test out the effectiveness of different compiler optimizations improves the productivity of developing an effective optimization strategy. The approach of specifying optimisations by way of a domain specific language enables the user of the system to more easily apply an optimization, than one could with a hand-written optimisation.

4.7 Performance Analysis

Since this approach to generating compiler optimizations involves generating compiler code indirectly from a specification, it raises questions about its practical applicability.

We compare Rosser with hand-written optimizations in the mature Soot framework [80], which is arguably a very high standard against which the performance of generated optimizations can be measured.

We use the Scimark scientific computing benchmark [68] to compare the performance of optimization phases. The performances are compared in terms of *effectiveness* (the extent to which the performance of the program being optimized is improved) and *efficiency* (how long it takes to apply a transformation to a program). The benchmarking was all performed on a 2Ghz Core 2 Duo with 2GB of RAM.

The performance of three optimizations is compared: lazy code motion, common subexpression elimination and dead code elimination. In both frameworks these optimizations are applied in this order. We chose only to compare these three optimizations since they are all commonly known compiler optimisations, and are used extensively in most compilers and therefore have a large effect on performance of compilers. In the direct comparison presented below no other optimisations were used since the optimisations that were have written didn't have an intraprocedural equivalent in Soot. We have experimented with other complex optimizations, but have no direct point of comparison with hand written optimizations.

The hand written optimizations that are being compared against are distributed with the Soot framework. They are written by contributors to the Soot project and not artificially constructed for this particular benchmark. The Soot project has been developed by Sable compiler group at McGill University, whose track record of publications denotes them as domain experts.

4.7.1 Effectiveness

The graphs in Figure 4.8 and Figure 4.9 show the running times of the Scimark 2.0 benchmark on two different virtual machines. The three columns for each program show runtimes without any static optimization in blue, optimized by `Soot` in red, and optimised by Rosser in yellow. The Scimark benchmark consists of five different benchmarks, in the graphs these are listed by their abbreviations. The FFT stands for a Fast Fourier Transform over 4000 complex numbers. SOR solves several finite difference applications over a 100x100 grid using Jacobi Successive Over-relaxation. Monte Carlo uses a Monte Carlo integration to approximate the value of Pi. SparseMatrixMult performs calculations over a matrix that uses a compressed row format to store 5 non-zeros in each row, in a 1000x1000 matrix. LU calculates the LU factorization of a dense 100x100 matrix.

Some modern `Java` virtual machines, for example the SUN JVM used in Figure 4.8 already incorporates many of the optimizations that are being applied. These optimizations are applied during the runtime of the program by the Virtual Machine itself. The SUN JVM is included in Figure 4.8 as a realistic benchmark, since it is the reference and most commonly used JVM. The Sable VM is a JVM that performs very little runtime optimization by comparison and is thus included as a better control for the comparison.

Using the SUN JVM the speedup generated by Rosser is 13.5%, comparable to `Soot` which improves performance by 14.5%. Since the implementation demonstrates that this approach to optimization works, rather than comparing the relative merits of ahead of time and runtime optimization, this is not a convincing argument against our approach to optimizer generation, as the performance of Rosser is comparable to the hand-written `Soot` optimizations for this benchmark. In both cases the program with

best improvement is the SOR benchmark, and in both cases it is the lazy code motion optimization that makes the impact, since the other two optimizations are performed by the SUN JVM anyway.

The numbers on SableVM are more flattering to both Soot and Rosser, due to the more simplistic optimizations performed by the SableVM. Here Rosser improves performance by an average of 25%, while Soot achieves 42%. Again our generated optimizations perform slightly worse than hand-written optimizations. When taking into account the closer performance that was obtained on the SUN JVM the overall effectiveness isn't much worse than hand written optimizations. Rosser is still improving the performance over an unoptimized bytecode run in both cases.

The Soot implementation of lazy code motion performs critical edge splitting before applying its optimization, while Rosser does not. This might explain the difference in effectiveness.

4.7.2 Efficiency

The Soot system applied its optimizations to Scimark in 15 seconds, while Rosser took approximately 270 seconds. This does not seem very encouraging, but more detailed analysis revealed that two methods in Scimark were responsible for a large amount of the time used by Rosser. It's important to note that when referring to a method here, it is being used in the Java sense, as opposed to a computation kernel (eg SOR). For the other 131 methods, the Rosser system only took 30 seconds, and the corresponding Soot time was 14 seconds. Over a weighted average of the 131 methods the Rosser optimizer was $2.143\times$ slower than the hand-written Soot optimizer. The following section discusses an investigation into removing the pathological cases from the implementation.

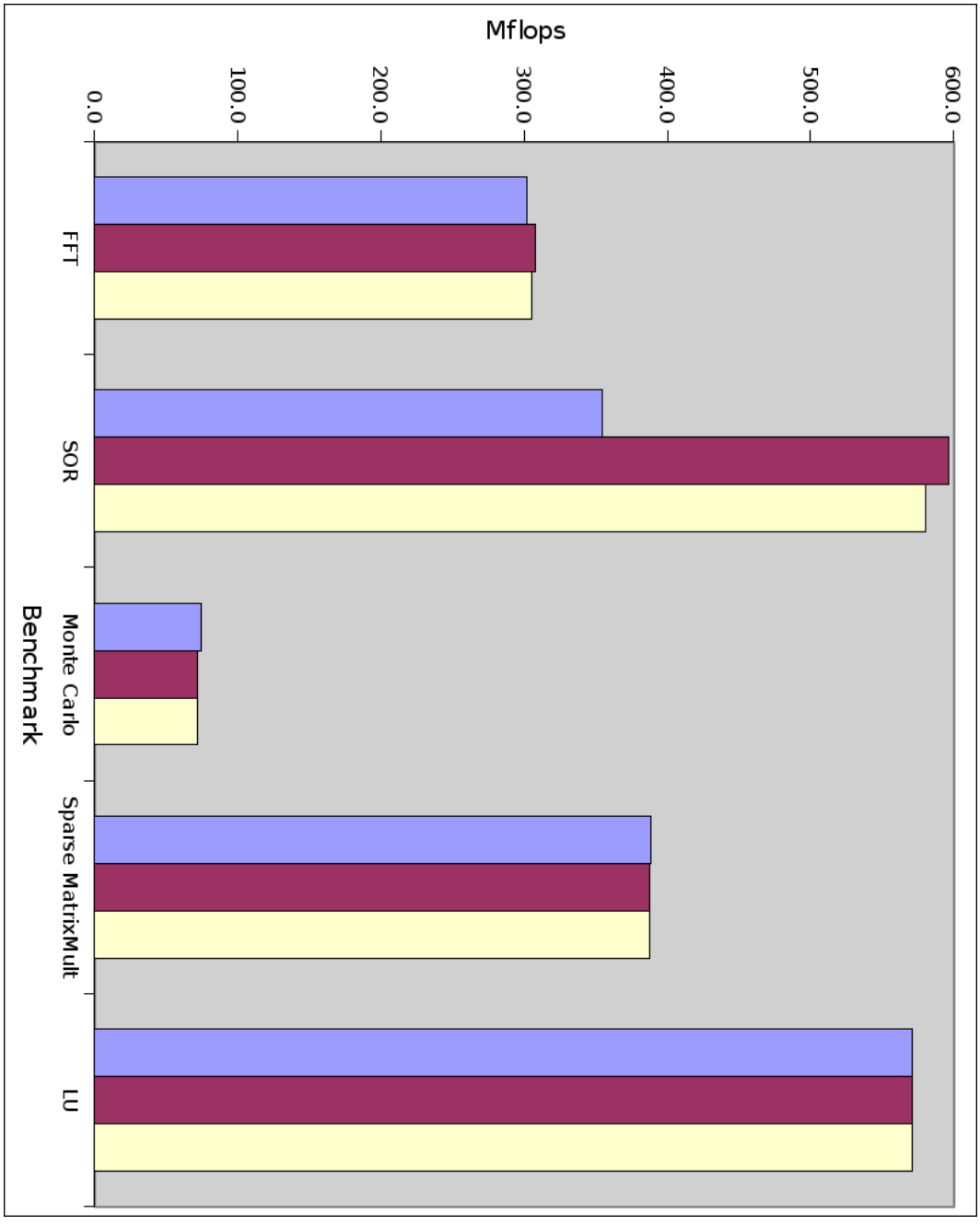


Figure 4.8: Scimark 2.0 on SUN "Hotspot" JVM 1.6, Blue is No Optimisation,Red is Soot's Optimisation,Yellow is Rosser

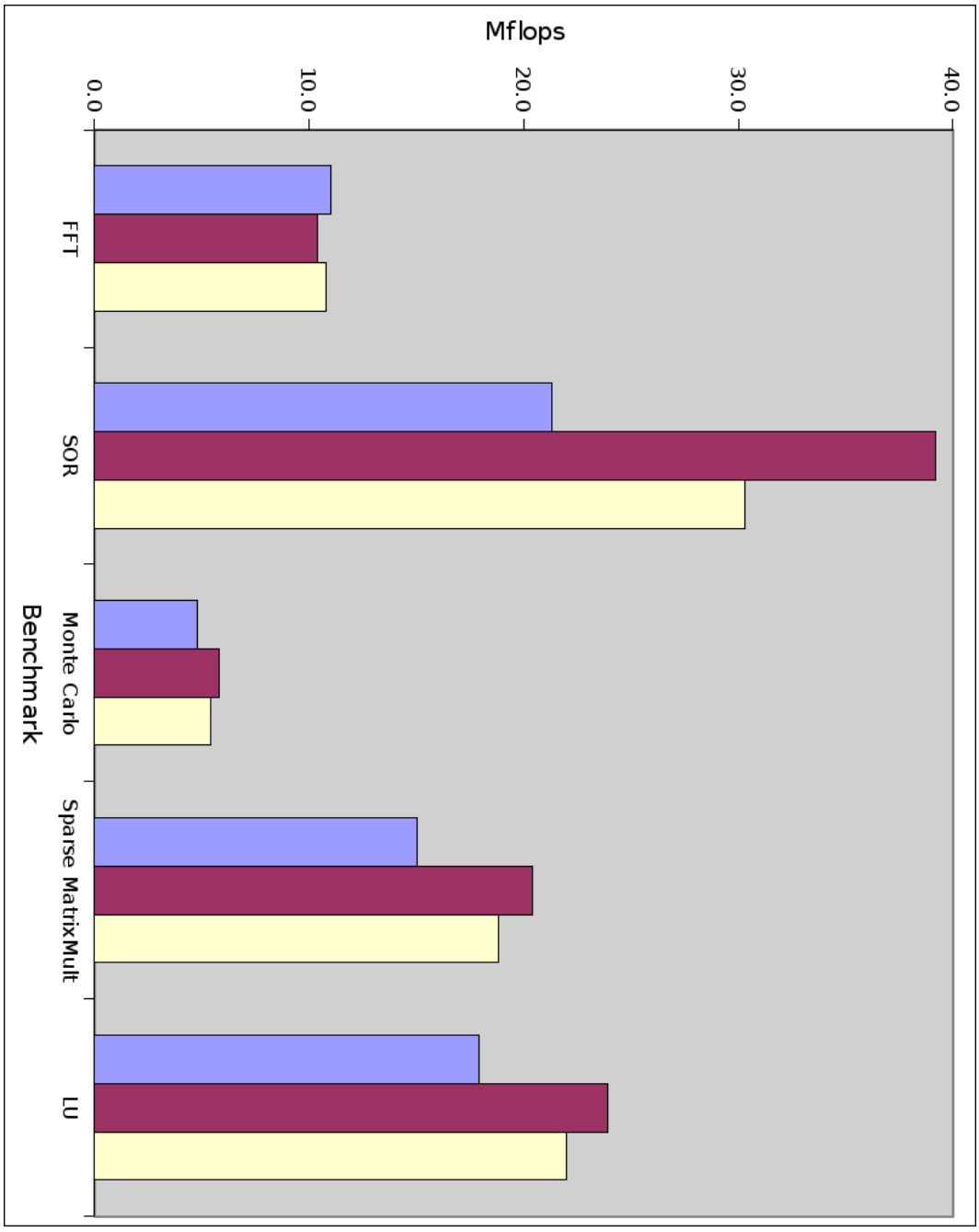


Figure 4.9: Scimark 2.0 on SableVM 1.13, Blue is No Optimisation,Red is Soot's Optimisation,Yellow is Rosser

4.7.3 Removing Pathologies

It is well known that the size of BDDs used in model checking can be highly influenced by the ordering of the variables within their construction. It was consequently investigated as a potential cause of the pathological cases discovered within Rosser's runtime performance. Several simple ordering algorithms have been investigated with the aim of reducing the negative influence of these pathological performance cases on the runtime performance of optimizations generated by Rosser.

[11] observed that variable ordering significantly alters the size of the BDD. He showed that for a boolean function, one variable ordering may yield a BDD that is exponential in the number of variables, while a different ordering may yield a BDD of polynomial size. Since the time complexity of operations over BDDs are parameterised by their size, the ordering is influential in the efficiency of the Rosser system. [7] proved that finding an optimal variable ordering is an NP-Complete problem, which is why most systems for variable ordering use a heuristic based approach.

Bits within BDD variables can be composed through interleaving them or sequencing them. In a sequential ordering, all bits of the first variable are placed first, followed by all bits of the second variable, and so on. Interleaving places the first bit of every variable, then the second bit of every variable, and so on. These compositions can be generalised, so that the composition of multiple variable can be substituted as a variable, for example with variables x_1,x_2,x_3,x_4 an ordering could be produced that interleaves x_1 and x_2 and then sequences this with x_3 and x_4.

A desirably property of an ordering would be that it can be computed from some property of the TRANS specification being compiled. The problem of variable ordering within the JEDD code generated by Rosser is quite a domain specific problem. Our methodology follows from past applications of BDDs to pointer analysis that makes use

```
<x:V1, n:N1, n2:N2, e:ET1> foo = 1B;
foo = foo{n,n2,e} >< meth.Edges{src,dest,et};
foo = foo{x} >< meth.Conlit{c};
<x:V1, x1:V2> clone = (n =>)(n2=>)(e=>)(x => x1) foo;
```

Figure 4.10: JEDD snippet used in ordering examples

of a simple static ordering for a given analysis, as described in [50].

Implementation and Orderings

The ordering of variables within BDDs is modelled within the JEDD relational algebra system as the ordering of physical domains. Rosser generates multiple physical domains for certain logical domains, one for each variable that is typed by the given logical domain. These are numbered sequentially, for example V1,V2... denotes physical domains associated with the Value logical domain, Figure 4.11 gives a complete listing and thus is useful when reading the examples, see Section 4.4 for explanation of the different logical domains. Within the following algorithms physical domains are either interleaved or sequentially ordered. All example orderings are given with respect to the snippet of JEDD code in Figure 4.10.

The approach of testing out different orderings allowed a resolution to the problem of the performance pathologies within the Rosser system. The different ordering approaches tried are explained in the sections below. Figure 4.12 gives a comparison of performances with different ordering schemes used. Within this table the averages are measured as slowdown over the time it took `Soot` to perform these optimizations. A higher number is worse. The *Average without outliers* column shows the average slowdown over `Soot` for optimizing when methods whose slowdown was over 2x the standard deviation are removed. Whilst comparing the overall performance should always include

Domain	Physical Domains
Value	V1,V2 ...
Node	N1,N2 ...
ET	ET1,ET2 ...
OP	OP1, OP2 ...
Call	C1, C2 ...

Figure 4.11: Relations between names of logical and physical domains

Ordering	Average Slowdown	Average Slowdown without Pathologies.	Pathological Cases
Default	18.01	2.14	2
Interleaved	26.32	5.4	5
Sequentially Ordered with Grouping	7.50	2.62	4
Sequential Ordering of interleaved Groups	2.31	1.78	1

Figure 4.12: Performance Comparison with different BDD Orderings

outliers within analysis, this column is useful for identifying whether an ordering has performed badly in a few instances or overall, and thus how to refine the given ordering.

Interleaved

The set of physical domains, as ordered by their first use in the declaration of a relation, are all interleaved. This resulted in worse overall performance, and more pathologies.

```
Jedd.v().setOrder(
    new Interleave(
        V1.v(),
        N1.v(),
        N2.v(),
        ET1.v(),
        V2.v()));
```

Figure 4.13: Example of Interleaved ordering

Sequentially Ordering with Grouping

In this approach all the domains are sequentially ordered, rather than interleaved. Physical domains that below to the same logical domain are adjacent to each other in the ordering. For example all the physical domains belonging to the `Value` logical domain are adjacent within the sequence. Overall performance significantly improved, and even though there were slightly more pathological cases, they were faster than the default case.

```
Jedd.v().setOrder(
    new Sequence(
        V1.v(),
        V2.v(),
        N1.v(),
        N2.v(),
        ET1.v()));
```

Figure 4.14: Example of Sequentially Ordering with Grouping

Sequential Ordering of Interleaved Groups

This is a similar case to the previous one, except that physical domains within the groups are interleaved, rather than being totally sequential. This resulted in a reduced number of pathologies, a better average case, and the single remaining pathology being less extreme in terms of performance.

```
Jedd.v().setOrder(
    new Sequence(
        new Interleave(V1.v(),V2.v()),
        new Interleave(N1.v(),N2.v()),
        ET1.v()));
```

Figure 4.15: Example of Sequential Ordering of Interleaved Groups

Ordering Improvements

Since the benefits of choosing the Sequential Ordering of Interleaved Groups applies to both the average slowdown, and the average slowdown without pathologies it seems to resolve more than just the pathological cases of a few methods in this benchmark. So it is likely to be an improvement to other cases as well. Future performance analysis of other benchmarking suites can also be conducted more easily after this work, since the implementation described in this chapter is able to generate the different BDD Ordering sequences through a commandline option.

Existing work on general ordering heuristics could be applied within Rosser, with the hope of finding a suitable ordering approach that would work generally. Different ordering heuristics could be experimentally evaluated within the existing Rosser system in order to decide which was the most suitable for BDDs given a representative set of optimizers generated from TRANS.

The JEDD also allows more exotic ordering schemes to be used, for example reversing the bits within an ordering scheme, or, performing a bit shift on existing orders. These have not been experimented with so far due to the complexity of understanding their impact on the size of BDDs.

4.8 Conclusions

The Rosser system described here applies compiler optimizations specified formally to Java programs within standard program development environments: the optimizations are mechanically translated into running code, and applied to given object programs within the `Soot` environment using a simple model checker for matching side conditions of optimizations to object code.

There is of course a performance price to be paid by not programing the optimizer directly. This cost is minimised by actually compiling the optimizations into JEDD rather than interpreting them, and the benefits of a declarative approach outweigh the performance cost, as sophisticated optimizations are often applied only when the code is ready for release—which is usually not a good time to find that the optimizer has introduced new bugs. The use of a formal notation has other benefits: it aids the interactive development of new optimizations and the explanation of the optimizations to third parties.

By using the Java language as the basis of the implementation questions, rather than a toy language all the difficulties of practical compiler development have been undertaken. By compiling TRANS Rosser offers a superior level of performance to specification interpretation systems. Not only does the Rosser system provide a Domain Specific Language for program transformation, its implementation layers on top of the work of the existing JEDD system in order to minimise implementation effort.

If program transformations are to be used as the basis of building practical compilers that can be used in professional programming, questions over the performance of such DSLs must be addressed. This chapter tackles these questions head on—Rosser and analysis of Rosser demonstrate that their usage can be practical.

This work also has a great deal of synergy with Chapter 5 where an approach for finding and fixing bugs automatically is developed along with a tool that implements this approach. The benefits of performance lessons from the Rosser system can be applied in such circumstances.

Chapter 5

Automated Bug Detection and Removal

5.1 Introduction

Bugs within Java programs often fall within well-known motifs, usually arising from misunderstood APIs or language features that encourage buggy corner cases. Existing software development tools can detect some of these situations, and integrated development environments may attempt to suggest automated fixes for some of the simple cases. We present a language for specifying program transformations paired with a novel methodology for identifying and fixing bug patterns within Java *source code*. We propose a combination of source code and bytecode analyses: this allows for using the control flow in the *bytecode* to help identify the bugs while generating corrected *source code*. The specification language uses a combination of syntactic rewrite rules and dataflow analysis generated from temporal logic based conditions. We introduce a prototype implementation that allows application of these transformations automatically

to programs, and discuss correctness issues within the context of such program transformations. Finally we discuss other possible areas of application for this methodology, including generating refactoring operations from specifications and application to other imperative languages.

The underlying TRANS language can be modified in small but important ways in order to operate over a source code language, rather than a compiler's intermediate representation. This allows us to consider new areas of application where the permanent change of source code is the goal of program transformation. This chapter applies program transformations specified in a variant of the TRANS language that is named $\text{TRANS}_{\text{fix}}$ to the problem of automatically fixing some common bugs within programs.

Our specifications for bug detection and removal differ from compiler optimizations in that they are not semantics preserving. Consequently there is not the need for formal analysis of semantics preservation. Our program transformations can be characterised as fixing bugs in the following general schema: pattern matching and temporal logic side conditions detect bug patterns, whilst replacement rules and actions fix them.

Debugging existing programs, while maintaining the *intent* of the programmer, is an unavoidable but difficult task, which can take significant effort in the software development lifecycle. Some existing tools can detect some of the commonly repeated bugs in particular programming languages, and some integrated development environments (IDEs) attempt to suggest automated fixes for some simple cases. For example in the Eclipse `Java` IDE if you refer to a class that hasn't been imported into the file it will offer to add an import statement for you. However, as far as we are aware, there is no general tool for specifying bug detection mechanisms that also offers suggested fixes based on the specifications.

Traditional application of abstract interpretation and static analysis is focused

around checking a specified property of a specified program. In this work we seek to find bugs in large families of programs by facilitating the coding of common bug patterns and then detecting instances of those bug patterns. Each instance of a bug pattern is a potential bug and each pattern has one or more resolutions associated with it, that can be instantiated for a given potential bug.

We consider some concurrency bugs, since they require more than simple syntactic pattern matching to be identified yet are amenable to temporal analysis. We use `Java` as our example platform, though our methodology is applicable to many imperative languages. Our approach to considering temporal control flow properties is to syntactically match specific threading library calls, as one would with normal literals.

Our contributions in this chapter are as follows:

1. We simplify the construction of tools for static analysis of bug patterns.
2. We propose a method to automatically fix a larger class of bugs than previous tools.
3. We show how to codify common bug patterns within a formally defined language.

In Section 5.2 we place our work in context, by identifying the kind of bugs which we consider and also the approach to software development for which our approach is particularly suited. We then describe, in Section 5.3, the language $\text{TRANS}_{\text{fix}}$ which can be used for both identifying bugs and implementing the transformations which correct the bugs. The prototype implementation FixBugs which applies bug fixes written in $\text{TRANS}_{\text{fix}}$ to `Java` programs is described in Section 5.5.

```
Lock l = ...;      l.lock();
try {
    // do something
} finally {
    l.unlock();
}
```

Figure 5.1: Pattern for correct locking

5.2 Methodology and Application

5.2.1 Example Bug Patterns and Categories

We use as starting point the classification of common Java bugs due to Hovemeyer and Pugh [33], which are used in the description of the FindBugs tool. Many of the bugs identified by Hovemeyer and Pugh are simple and their identification requires merely a syntactic pattern matching system. We consider a few non-trivial examples.

Method does not release lock on all paths This bug arises when a method acquires a lock, but there exists a path through the method where the lock is not released. Figure 5.1 illustrates a corrected situation, in which the call to the `unlock` method is wrapped inside a `finally` block. This ensure that it gets called no matter whether an exception is thrown, or the method returns within the `try` block.

Races over Collections As with many implementations of collection systems, those in the Java standard library have variants that lock their underlying data structure in order to avoid data races in concurrent circumstances. Due to the inefficiency of this locking in a sequential environment there also exist versions that do not lock the underlying data store. The use of the latter in an environment which requires concurrent access is nearly always a bug that can be fixed by using the concurrent variant.

```
try {
    conn.setAutoCommit(false):
    ....
    conn.commit();
} catch(java.sql.SQLException e) {
    if(conn != null) {
        try { conn.rollback(); }
        catch (java.sql.SQLException e) { // handle error }
    } }
```

Figure 5.2: JDBC Commit and Rollback Pattern

Failed database transactions may not be rolled back JDBC, a `Java` library for database connections, models the begin, committing and ending of transactions through explicit calls to methods. A common bug pattern is a failure to check whether a transaction needs to be rolled back if its commit fails. Figure 5.2 illustrates a situation in which, if a problem occurs with the database transaction, ie a `SQLException` is thrown, the program calls the `rollback` method of the connection. Conversely if there is no problem, the `try` block is finished with a call to `commit`. Another common problem is the failure to ensure that all paths either end in a commit or a rollback.

5.2.2 Placing Debugging within Software Development

In general, a good approach to tooling the fixing of bugs is to not entirely automate the application of transformations to the users' programs, since fixes may not always be semantics preserving, as they may change not only the way in which a program operates, but also its overall input/output function. Since the automated tool may not be designed to consider the specification of the program, there is the risk of introducing new bugs into a currently working system. Bug patterns usually identify scenarios that are likely to be a bugs, rather than being guaranteed to be so. In this context, the conservative approach is to not alter the program, but simply suggest bug fixes to the

user.

It may be at times difficult to instrument a bug-finding/fixing tool, and ideally potential users should be assumed to have little experience or understanding of the system in order to productively use it. Their existing development tools may incorporate some way of reporting suggested improvements to code, and these should still be supported. When using a more sophisticated bug-fixing tool, the user could simply see contextual and appropriate descriptions of the transformations, rather than their formal specification. In this context, the tailoring and deployment of bug-fixing techniques would be an activity undertaken by a few key team members, rather than necessarily every developer.

The inclusion within the development cycle of phases dedicated to improving code quality, such as the refactoring phases promoted by some agile methodologies, provides bug fixing program transformations with a suitable hook on which to integrate themselves to existing methodologies. Within a more traditional, waterfall, development model such an approach could be useful during a testing phase, after the program has been mainly written, but before it is shipped to customers.

The bug-fixing methodology described in this thesis fits in particularly well with modern agile software engineering methodologies, such as Extreme Programming, which have increased focus on the quality of the code itself. Application of best practises, unit testing, many eyes reading code through pair programming, etc. all attempt to reduce bugs cropping up within the program being developed by reducing the likelihood of the programmer writing bugs. Whilst these developments have been of positive benefit to programmers, experience shows that bugs still occur.

Our implementation, described in Section 5.5, uses the Eclipse toolkit's intermediate representation to perform program transformation. This enables the production

of source code that is formatted according to users' preferred style guidelines and integrates into the context in which programs are being developed, and ensures that the generated code requires no further formatting.

While we have incorporated a few common bugs into FixBugs, the aim is to provide a *framework* in which more bugs can be accounted for. The designing of new transformations is eased compared to traditional static analysis systems since the programmer does not have to implement a detailed static analysis and transformation toolkit in order to achieve their specific goal. Since the program transformations themselves are merely syntactic substitutions, it should be relatively natural for any experienced programmer to tailor the system to common bugs in their application area.

The FixBugs approach is not intended to subsume traditional debugging techniques such as testing, or traditional formal analysis techniques such as static analysis and model checking. Its integration into existing tools and techniques should complement their usage, allowing automated FixBugs sweeps of the code to be made in order to offer potential improvements to the code base. Bugs can be found as early as possible through these automated tools, rather than being identified later through failing test cases, at a much higher cost.

5.3 A Language for Detecting and Fixing Bugs

5.3.1 From TRANS to $\text{TRANS}_{\text{fix}}$

We describe a variant of the TRANS language, called $\text{TRANS}_{\text{fix}}$, suitable for specifying the transformation of Java source code with the aim of correcting bugs that may appear within programs. In contrast to the TRANS language for optimizations, where the goal is to produce optimized *low-level* code, $\text{TRANS}_{\text{fix}}$ is used to produce source code, since

the goal of debugging is usually to maintain reusable and readable source code, for the developers of the software to continue working on. Rather than operating on the low-level code which is used as input for the temporal logic side conditions, rewrite rules must operate on the *source program* itself.

$\text{TRANS}_{\text{fix}}$ specifications consist of actions and conditions: if the condition holds true then the action is applied. Many actions consist of replacing statements with other statements, although they can also include adding new methods to classes. Actions are applied if side conditions hold true.

A BNF for the $\text{TRANS}_{\text{fix}}$ pattern matching language is provided in Figure 5.6. Interesting aspects of $\text{TRANS}_{\text{fix}}$ are its use of metavariables, the new actions and strategies, and the type system as discussed in the subsequent sections.

5.3.2 Metavariables and wildcards

The core syntax of the rewrite rules is based on standard programming constructs (assignment statements, while statements, if statements, etc) which we assume are well understood. The syntax is expanded with constructs to support meta-variables, representing either syntactic fragments of the program or nodes of the CFG.

The language for transformations is a `Java` statement grammar, extended with metavariables that can bind to different program structures, and wildcards that can match any statement or sequence of statements. For example, the pattern for matching an integer assignment to an addition expression, that is later followed by re-assignment to that variable, is shown in Figure 5.3. Figure 5.4 gives a code snippet which matches to that pattern, and metavariable bindings that show how the pattern is matched.

The language for code reconstruction is the same as pattern matching. Its application is fundamentally different, however. In reconstruction metavariables are sub-

```
n: int x = l + r;
....
m: x = e;
```

Figure 5.3: $\text{TRANS}_{\text{fix}}$ Pattern Matching

```
int z = y + 5;
System.out.println(x);
z = z + 1;
```

Metavariable	Binding
x	z
l, r	y, 5
e	z + 1

Figure 5.4: Sample Java Code Listing

stituted with a statement, expression or type that has been bound to the metavariable during pattern matching, and model checking. Each statement in the syntax tree isomorphically corresponds to a node within the CFG, which enables the use of the results of model checking the side conditions in the code reconstruction.

A consequence of the desire to produce source code is the necessity of incorporating *scoping*; while scoping does not exist within methods at a bytecode level, it is a necessary part of the transformation language of $\text{TRANS}_{\text{fix}}$. This allows us to match programming language constructs such as `try` and `catch` blocks.

The $\text{TRANS}_{\text{fix}}$ language contains a wildcard operator "`....`" that matches against any statement or sequence of statements, including no statements. Since a wildcard statement is a normal pattern matching statement, it can also be bound using a label, allowing the matching or arbitrary blocks of code in strategic locations. In order to facilitate the writing of specifications that are intuitive to programmers, we also allow wildcards to be used in the reconstruction of statements. This is syntactic sugar for

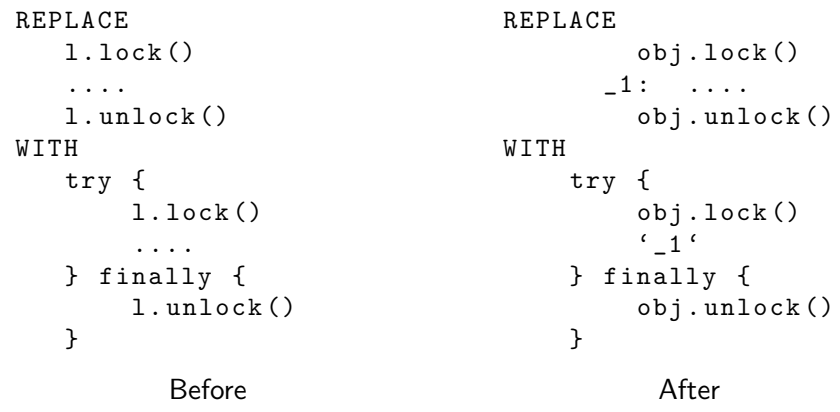


Figure 5.5: Removing Wildcard Reconstruction Syntactic Sugar

binding the wildcard statements to metavariables using labels, and then substituting in metavariable references within the reconstruction pattern. Figure 5.5 gives an example translation. Wildcard substitutions are indexed, so the nth wildcard block in pattern matching is substituted into the nth wildcard position in reconstruction.

5.3.3 `Java` Types

$\text{TRANS}_{\text{fix}}$ provides pattern matching for `Java` types as well. The syntax `:: m` is used to bind any type to the the metavariable `m`. One can explicitly refer to primitive types, such as `int` or object types, such as `java.util.Vector`. One can also match arrays. The two `new` calls within the expressions grammar allow pattern matching array initialisers specifically.

5.3.4 Actions

A simple rewrite merely replaces code snippets with new code; however, many transformations must actually change the structure of the `class` or apply rewrites at multiple places. These structural changes are supported by additional actions.

The `ADD_METHOD` action takes the return type of the method, its name, arguments

and a statement to act as the body. This is then added to a class, specified through a *metavar*. This is our primary method of transforming classes.

Combining uses of actions has many applications, for example one could rewrite a block of code into a method, and replace it with a call to this method, by using a `REPLACE` composed with an `ADD_METHOD`.

5.3.5 Strategies

As in the TRANS language, *strategies* are operators for combining different transformations. The $\mathrm{MATCH}\ \phi\ \mathrm{IN}\ T$ strategy restricts the domain of information in the transformation T by the condition ϕ. The $T_1\ \mathrm{THEN}\ T_2$ strategy applies the sequential composition of T_1 and T_2. When actions are applied normally, ambiguity with respect to what node actions and rewrites are applied to are automatically resolved. In other words, if there are several bindings that have the same value for a node attribute that is being used in a rewrite rule then only one of them is non-deterministically selected. The $\mathrm{APPLY\_ALL}\ T$ strategy uses all of the valuations within transformation T, without this restriction.

The `DO ... THEN` strategy performs sequential composition on the two actions that it is passed as arguments and forms a new atomic transformation. Note that these actions are both disabled if the side condition does not hold true for a given set of `metavar` bindings. In other words for a transformation in a `DO` chain to be applied all previous transformations in the chain must have been successful.

A non-deterministic choice strategy, called `PICK ... OR`, is used when the same analysis might suggest more than one possible fix. This fits in with the methodology of debugging we propose since the user must confirm the application of a transformation, thus they may be in a better position to make that choice.

Since strategies combine different transformations, they allow more expressive transformations to be developed. They can also allow one to generalise transformations. For example the `PICK` construct is useful if there are multiple scenarios where that one might want to apply a similar transformation, but operating over slightly different code patterns.

5.3.6 Syntactic Comparison with TRANS

Since $\text{TRANS}_{\text{fix}}$ is a modification of the TRANS language it shares many similarities. The *side-condition* and *node-condition* elements of the BNF in Figure 5.6 are near identical to TRANS, the only addition being the `type` predicate. *statement* and its subdefinitions, *expr-pattern* and *type-pattern* perform the same function as their TRANS equivalents but over source code instead of bytecode. An additional action has been added in order to add a method to classes.

5.3.7 Type System

$\text{TRANS}_{\text{fix}}$ is endowed with a simple type system that ensures that programs transformed by a $\text{TRANS}_{\text{fix}}$ specification are syntactically valid `Java` programs. For example, anything nested at an expression level is an expression. It does not guarantee that the programs output are well typed `Java` programs. We cannot ensure output programs are correctly typed because strategies (transformational combinators), such as `THEN`, may be used to combine a transformation that fixes an incorrect program.

In order to differentiate types of meta-variables being used in transformations from the types of `Java` variables, we refer to the former types as *kinds*. The kind system provides guarantees that can be used in our implementation, see Section 5.5. There are three Kinds within the kind system:

```
type-pattern   ::=  :: metavar
                |   primitive-type
                |   " class name "
                |   type-pattern []

expr-pattern   ::=  metavar "(" expression, expression ... ")"?
                |   % metavar "(" expression, expression ... ")"
                |   expression op expression
                |   unop expression
                |   (type-pattern) expression
                |   new type-pattern "(" expression , expression ... ")"
                |   new type-pattern []
                |   expression instanceof type-pattern

statement      ::=  metavar: statement
                |   ....
                |   ;
                |   ‘ metavar ‘
                |   type metavar = expression
                |   if  expression statement statement
                |   while  expression statement
                |   try  "{"  statement* "}"
                        ( catch "(" type-pattern metavar ")" "{" statement* "}" )*
                        ( finally statement )?
                |   return expression ;
                |   expression ;
                |   { statement* }
                |   return expression ;
                |   throw expression ;
                |   synchronized (expression) { statement  }
                |   for (expression*, expression, expression*)
                        { statement  }
                |   switch (expression) { statement*  }
                |   case expression: statement ;
                |   default ;
                |   assert expression ;
                |   continue metavar ;
                |   break metavar? ;
                |   this ( expression, expression ... );
                |   super ( expression, expression ... );
```

Figure 5.6: BNF for $\text{TRANS}_{\text{fix}}$

node-condition ::= `true` | `false`
| *node-condition* $\wedge$ *node-condition*
| *node-condition* "|" *node-condition*
| $\neg$ *node-condition*
| (`E` | `A` | `<E` | `<A`) *path-condition*
| `node` "(" *metavar* ")"

path-condition ::= (`X` | `F` | `G`) [*node-condition*]
| [*node-condition* U *node-condition*]

side-condition ::= `True` | `False`
| *side-condition* `or` *side-condition*
| *side-condition* `and` *side-condition*
| $\neg$ *side-condition*
| *node-condition* `@` *metavar*
| *pred* "(" *metavar*$_1$,...,*metavar*$_n$ ")"
| `type` *metavar* `is` *type-pattern*

action ::= REPLACE *statement** WITH *statement**
| ADD_METHOD *type metavar* "("
type metavar, ... ")" *statement* TO *metavar*

transform ::= *action* (`WHERE` *side-condition* |`ALWAYS`)
| `MATCH` *side-condition* `IN` *transform*
| `APPLY_ALL` *transform*
| `PICK` *transform* `OR` *transform* `OR` *transform* ...
| `DO` *transform* `THEN` *transform* `THEN` *transform* ...

Figure 5.7: BNF for $\text{TRANS}_{\text{fix}}$ side conditions

`Type Kind` for metavariables that bind to `Java` types

`Expression Kind` for metavariables used for `Java` expressions

`Statement Kind` for statements and blocks.

The kind system guarantees two important properties:

1. that no metavariable may bind to, or substitute into a position that requires more than one Kind.

2. that no metavariable may be used in a substitution, if it is not bound before hand.

A relatively simple algorithm is used to check these properties. A pass is made of the syntactic replacement rules and side conditions, keeping note of what context a metavariable is used in. If a metavariable is used in a context which implies it would need to be of more than one Kind, then kind checking fails. If there exist metavariables referred to in the substitution part of replacement that is not bound by either the pattern matching, or the side condition then it also fails.

5.4 Specification Examples

This section presents a series of specification examples that use $\text{TRANS}_{\text{fix}}$ in order to fix potential bugs within computer programs.

Several of the following bug patterns come from the Findbugs tool [33]. Some of the bug patterns that their tool finds have obvious solutions, and some don't. Furthermore some of the kinds of bugs that their tool identifies aren't specifiable within the $\text{TRANS}_{\text{fix}}$ language. For example their bug detector that tries to identify null derefences requires a form of interprocedural analysis. Some of the examples in this section don't

correspond to bugs at all, but merely quality issues in code. These kind of issues might be commonly made be programmers who are learning, and in this respect $\text{TRANS}_{\text{fix}}$ may be use to recommend program transformations that help students learn.

The different bug patterns that are presented in this section cover a wide variety of different types of bugs. The first three examples all correct bugs that could potentially cause a program to crash. The bug patterns described in Section 5.4.4 and Section 5.4.5 correspond to bugs that involve race conditions — a type of bug that is hard to identify through testing due to its non-deterministic nature. The bug that is fixed in could cause a program to crash. The specifications described in all correspond to code quality issues.

5.4.1 Method Does Not Release Lock On All Paths

JSR 166 introduced a series of concurrency libraries to the `Java` programming language. These include several `Lock` implementations that allow a program to acquire a lock by calling a `lock` function and release a hold on the lock by calling an `unlock` function. If the application using the `Lock` object never releases the lock it is impossible for any other threads waiting at the `lock` call to continue executing. This can cause a form of deadlock.

The specification presented here attempts to fix a scenario where a method locks a `Lock`, but doesn't necessarily unlock it. The specification is provided in Figure 5.8. Position `l` within the program matches the point at which the lock is locked, and `u` at the position where its unlocked. The side condition holds true when the lock that is released can sometimes unlock if you have locked, but not on every path. The replacement rule moves the unlock statement within a finally clause, ensuring that the lock gets executed on all paths through the method.

Since the `....` statement matches against patterns of statement sequences, it

```
REPLACE
     l: l.lock();
        ....
     u: l.unlock();
WITH
    try {
       l.lock();
       ....
    } finally {
       l.unlock();
    }
WHERE
```

$$\texttt{EF}(u) \wedge \neg\texttt{AF}(u)@\,\texttt{l}$$

Figure 5.8: Transformation to ensure lock released on all paths

doesn't matter if `l` and `u` are arbitrarily apart. If there are multiple `lock` and `unlock` statements in the method this specification will match all combinations that meet the side condition's requirement. So for example if there are two `lock` and `unlock` pairs then an application will suggest introducing a `try-finally` bug fix between each of these pairs, and also between the initial `lock` call and the final `unlock` call.

5.4.2 Database Transactions

Figure 5.9 shows a specification for ensuring that transactions are surrounded by the correct catch pattern for `SQLException` instances. The pattern matching of a call to the `setAutoCommit` method, matches the beginning of the transaction. The wildcard binds to anything between that and the `commit` call, which corresponds to a transaction. This block of code is then replaced with another block, surrounded by a `catch` statement, which rolls back the transaction in case of a database failure.

The `type` predicate constrains the `conn` variable to be an SQL Connection, in order to ensure that the transformation is not misapplied. The side condition also states that the `commit` call can never be ($\neg$EF) followed by a `rollback`.

```
REPLACE
        conn.setAutoCommit(false):
        ....
commit: conn.commit();
WITH
    try {
        conn.setAutoCommit(false):
        ....
        conn.commit();
    } catch(java.sql.SQLException e) {
        if(conn != null) {
            try {
                conn.rollback();
            } catch (java.sql.SQLException e) {
                e.printStackTrace();
            }
        }
    }
WHERE
type(conn,'java.sql.Connection') ∧¬EF(stmt(conn.rollback();))@ commit
```

Figure 5.9: Correction for JDBC Commit and Rollback Pattern

5.4.3 Unclosed File Handles

Problem This bug occurs when a method creates an IO stream object but does not assign it to any fields, pass it to other methods that might close it, or return it, and does not appear to close the stream on all paths out of the method. This may result in a file descriptor leak. Good programming discipline requires the use of a `finally` block to ensure that streams are closed.

Solution The definition in Figure 5.10 specifies the rearrangement of the closing mechanism for file handles. It matches the type of the stream object into the metavariable `streamtype` thus ensuring this is a stream. The other component of the side condition ensures that the close method throws an exception, by checking there is a path between the catch block and the node where the exception throw happens.

```
REPLACE
    ::streamtype stream = null;
    try {
        ....
        throw: stream.close();
    } catch (ex e) {
        c: ....
    }
WITH
    ::streamtype stream = null;
    try {
        ....
    } catch (ex e) {
        ....
    } finally {
        if(stream != null) {
            try {
                stream.close();
            } catch('IOException' e) {
                e.printStackTrace();
            }
        }
    }
WHERE
            subtype(streamtype,'java.io.OutputStream') ∧
                        EF (node(c)) @ throw
```

Figure 5.10: Closing File Handles

```
REPLACE
    cons: ArrayList x = new ArrayList()
WITH
    Vector x = new Vector()
WHERE
subtype(this,'java.lang.Thread') ∧
method('run') ∧
stmt(x.add(e);)  @ inthread and
¬ ( subtype(this,'java.lang.Thread') ∧ method('run') @ cons )
```

Figure 5.11: Adds List Synchronisation

The specification uses wildcard matching to keep the body of the `try` block within a `try` block, whilst moving the `close` call at the end of the method within a `finally` block, thus ensuring that it always gets called.

5.4.4 Correcting Races over Shared Collections

As shown in Figure 5.11, writes to an `ArrayList` inside of a `Thread` can be detected and the `ArrayList` replaced by a `Vector`, which has its accesses synchronised. This specification presents a simple case of a write-after-write dependency over one collection type; in order to detect many race conditions, one would write a variety of these specifications, in analogous forms.

The specification detects the current class to be a type a Thread, and for the method that the code is within to be the `run` method, which is the method that is executed in the separate Thread, in this context it checks that the `List` is written to, by checking whether the `add` method is called. This detects that there is a concurrent write race. We then check that the object is constructed somewhere else, other than the local Thread—in other words that it is data that is shared between multiple Threads.

```
REPLACE
  cond: if (inst == null) {
    sync: synchronized(x) {
      if (inst == null) {
        inst = e;
      }
    }
  }
WITH
    `sync`
WHERE
    <A [ ! def(inst) U stmt(::t inst;) ] @ cond
```

Figure 5.12: Removing Double Condition Checked Locking

5.4.5 Double Condition Checked Locking

Double condition checked locking is a pattern for lazy object instantiation in a concurrent environment that tries to minimise the cost of synchronization [74]. The principle is that, since the race condition is only on the first instantiation, one can avoid synchronization on all future references. The pattern wraps a synchronized instatiation block with another `if` statement. This approach is unsound in the context of the `Java` Memory Model [5] since the JVM runtime environment is free to optimise the order in which memory allocation, reference allocation and constructor calling occur. Specifically if the JVM allocates memory and calls the constructor before an object is assigned to a reference then two objects may be created.

Figure 5.12 shows a $\text{TRANS}_{\text{fix}}$ specification for removing the double condition checked locking instantiation pattern. The outer `if` statement of the pattern is matched by block labelled `cond`. The inner synchronization block, labelled `sync` is the normal lazy synchronized initialization pattern. The pattern replaces `cond` with `sync`, thus removing the outing layer of checking that causes the pattern to be unsound.

The side condition can be read as checking that the variable `inst` is never

defined between cond and the point at which it is declared. This is important since if fixbugs is dealing with an already existing inst variable then it is no longer the double condition checked locking pattern. In other words this transformation is conservative in circumstances where the programmer's intent is uncertain.

5.4.6 Resultset Reusage

The Java Database Connectivity (JDBC) API provides facilities for Java programmers to access databases. The results of queries to a database are returned as instances of the interface ResultSet. This is a resource that can be closed, and an analysis can be written in the style of Section 5.4.2 that automatically fixes potentially unclosed result sets. Another aspect to buggy usage of ResultSet is trying to access within objects that have already been closed.

In order to fix this bug we move the close call after the final usage of ResultSet. This is specified in $\text{TRANS}_{\text{fix}}$ in Figure 5.13. The assignment labelled usage matches the use of the rs variable. The first .... matches the body of the try statement, the second matches any catch blocks, whilst the third matches code between the try block and usage.

In order to fix the third block and assignment labelled usage are moved within the try block. Note that since the specification reorders .... statements it is necessary to refer to them through their desugared metavar names, for example _2.

The side condition ensures three properties. The property:

```
!  AF [ use(rs) ∧ use(rs) @ usage]
```

ensures that the variable rs is used at usage and that there are no further uses of rs. The implements predicate ensures that rs is actually an instance of ResultSet.

```
REPLACE
close: try { .... }
       ....
       finally {
         if(rs != null) {
           % rs.close();
         }
       }
       ....
usage: x = e;
WITH
       try {
         ....
         `_3`
         x = e;
       }
       `_2`
       finally {
         if (rs != null) {
           % rs.close();
         }
       }
WHERE
    ! AF [use(rs)] ^ use(rs) @ usage
    and implements(rs,'java.sql.ResultSet')
```

Figure 5.13: Move `ResultSet` closing around all uses

```
REPLACE                          REPLACE
    if ( expr ) {                    if ( expr ) {
        return true;                     return false;
    } else {                         } else {
        return false;                    return true;
    }                                }
WITH                             WITH
    return expr;                     return expr;
ALWAYS                           ALWAYS
```

Figure 5.14: Removing Redundant Return

```
REPLACE                          REPLACE
decl:   ::t x;                   decl:   ::t x;
        ....                             ....
ifn:    if ( expr ) {            ifn:    if ( expr ) {
            x = true;                        x = false;
        } else {                         } else {
            x = false;                       x = true;
        }                                }
WITH                             WITH
        ....                             ....
        ::t x = expr;                    ::t x = expr;
WHERE                            WHERE
A[ ! use(x) U node(ifn) ] @ declA[ ! use(x) U node(ifn) ] @ decl
```

Figure 5.15: Removing Redundant If pattern

5.4.7 Simplification

A common source of confusion for programmers is unnecessarily complex conditional code. By simplifying conditional statements we seek to make the source code easier to comprehend and therefore maintain. This kind of source code has, annecdotally, been commonly found amongst undergraduates.

The listing in Figure 5.14 removes a redundant `if` condition around a return statement through a simple syntactic transformation. There are cases for both the

```
REPLACE
decl:    x = expr;
         ....
ret:     return x;
WITH
     ....
     return expr;
WHERE
     A [ trans(expr) ^ ! def (x) U node(ret) ] @ decl
```

Figure 5.16: Removing unnecessary assignments before a return statement

`true` and `false` scenarios. Another example of a similar overcomplication is using an `if` statement to wrap boolean assignments. Figure 5.15 shows a specification that tidies this scenario. Figure 5.16 removes an unnecessary variable assignment that is immediately returned. This would generate a 'dead store' in the terminology of the findbugs tool. This is yet another example of a transformation that would be complicated to manually implement, but can be simply and comprehensibly expressed using the $\text{TRANS}_{\text{fix}}$ language.

5.5 Prototype Implementation

The approach proposed in this chapter, based on specifying bug fixes with $\text{TRANS}_{\text{fix}}$ and matching the specifications against low-level program representations, has been prototyped in the implementation we call FixBugs. This implementation takes a `Java` program in both source and `Bytecode` form and applies transformations to it, outputting a series of programs representing possible bug-fixed variants of the program.

5.5.1 Architecture

As shown in Figure 5.17, the FixBugs system comprises several components:

- the Core parses $\text{TRANS}_{\text{fix}}$ specifications, and calls into various components as required;
- the Pattern Matcher produces bindings to metavariables from source code and a pattern;
- the Model Checker produces bindings to metavariables that satisfy the side condition formulae; and
- the Generator alters the program itself, given bound metavariables, according to the actions.

The Model Checker relies on the ASM bytecode library [21] in order to generate the control flow graph of the program, as explained in Section 5.5.3. The `Java` program's source code is parsed using the Eclipse [20] project's `Java` developer tools. These provide a standardised intermediate representation for the programs. This representation is also used by the Generator, which manipulates this representation directly, and concrete syntax is generated from this abstract syntax.

5.5.2 Representation

An important issue in writing static analysis systems is the representation over which the analysis is performed, notably whether at source code level, object code level or some intermediate representation. In order to bug-fix the programs themselves (rather than a low-level representation) it is necessary to perform the transformation at the *source code* level. Many existing systems for detecting bugs perform analysis at the *bytecode* level, and thus have difficulty incorporating automatic fixes to programs. There are many advantages, however, to performing analysis at a lower level: for example, it is easier

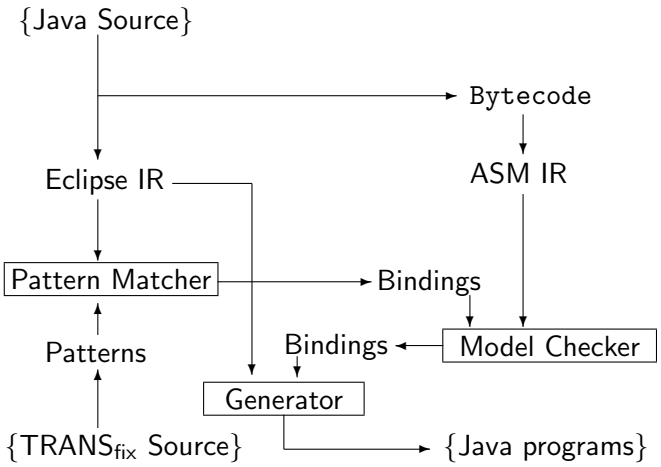


Figure 5.17: FixBugs Architecture

to extract the control flow graph from a language whose control flow is represented by conditional goto statements, rather than loops.

We attempt to blend the best of both worlds with our approach to analysis. We perform syntactic analysis against the source code of the program, whilst performing semantic analysis on a bytecode representation. We use the standard debugging information from the `Java Bytecode` format in order to correlate the results from the source and `Bytecode` analyses.

5.5.3 Silhouettes

One line of `Java` source code is compiled into one or more lines of `Java Bytecode`. Consequently there is a subtle impedance mismatch between the two systems when using the debugging information to bridge the analysis results of these two representational levels. We unify these levels within FixBugs through the concept of a *silhouette*. The silhouette of a line of source code is the corresponding set of lines of its bytecode. This

concept is reflected within all aspects of the analysis. For example the control flow graph silhouette of a source code line is the subgraph within the control flow graph that corresponds to that source code line. Every edge within the control flow graph of the program's source code has a corresponding edge within the bytecode control flow graph, but the inverse relation does not hold.

Silhouettes consequently partition the `Bytecode` control flow graph into several overlapping subgraphs. The edges between these subgraphs fall into three categories.

Definition 5.5.1 *An edge* (from,to) *is* inbound *with respect to some silhouette S if the* to *node, but not the* from *node, is a member of S.*

Definition 5.5.2 *It is* outbound *if the* from *node is a member of S, but not the* to *node.*

Definition 5.5.3 *If both* from *and* to *are within S we say that the edge is* contained *within S.*

Definition 5.5.4 *We can say that a graph (G) is minimal with respect to a set of silhouettes (S) iff there is no edge within G that is contained by a member of S.*

The relation between source code and bytecode CFGs is illustrated in Figure 5.18. We can minimise the `Java` control flow graph from the `Bytecode` representation very simply with the following steps:

1. extract `Bytecode` control flow graph (G) using ASM.
2. compute line numbering function (L) using ASM.
3. coalesce (G) to form (G').

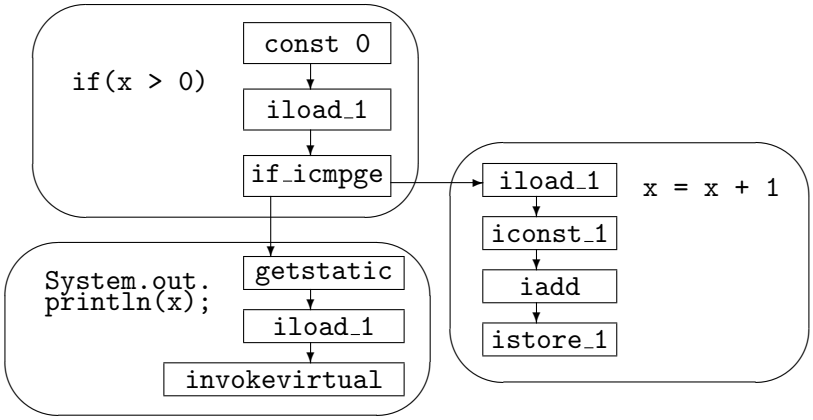


Figure 5.18: CFG Coalescing

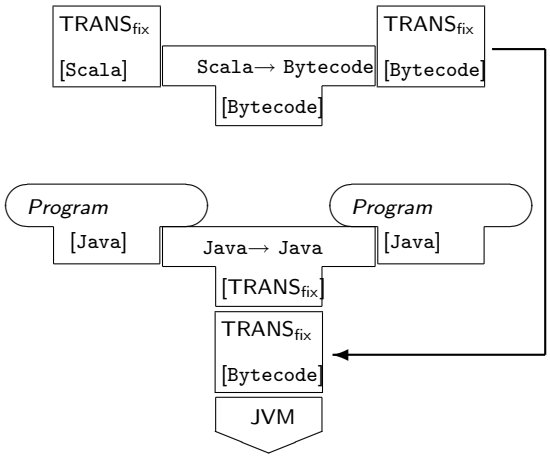


Figure 5.19: Transformational Diagram for fixbugs

Within the FixBugs implementation we represent the successor function of G as a map from integers onto sets of integers, and L as an array of integers. In order to calculate G' we therefore replace every edge *(from,to)* in G with an edge *(L(from),L(to))*. This ensures all inbound and outbound edges are replaced accordingly. We then remove all edges whose *from* and *to* nodes are identical, since they represent contained edges that do not exist within the source code control flow graph G'.

The existence of `Bytecode` analysis libraries, such as ASM makes it easier to extract the control flow graph and coalesce than to write a custom source code analysis. It also allows us to integrate other information more easily extracted at a `Bytecode` level, and then relabel it onto the `Java` control flow graph accordingly.

5.5.4 Implementation Details

Most of the software is written primarily in `Scala`, chosen because of its support for a functional style of programming, combined with the plentiful libraries that are available on the `Java` platform. Specification files are parsed using the parser combinators in `Scala`'s standard library, and disjoint union datatypes, modelled using case classes provide an intermediate representation for $\text{TRANS}_{\text{fix}}$ specifications. `Scala`'s pattern matching can then be used in order to bind $\text{TRANS}_{\text{fix}}$ metavariables to elements of `Java` source code, represented using Eclipse's Intermediate Representation. This development approach is described in Figure 5.19.

Being a prototype, the current implementation does not provide support for all the features of the $\text{TRANS}_{\text{fix}}$ language, such as strategies and class-level actions. It is the author's opinion that strategies would be easy to implement here without changing any of the underlying transformation implementation. This is based both on the past experience of implementation in Chapter 4 and because strategies work like functional

combinators it is relatively easy to implement them in a way that is oblivious to the implementation of the underlying transformation operations. The class level actions take the form of the standard action, so adding them shouldn't compromise any of the existing implementation but simply require more programming effort.

5.5.5 Performance

The performance of this prototype implementation in practise has been acceptable in anecdotal situations. For example applying a bug fixing transformations usually takes in the order of seconds for even a java class of several thousand lines of code.

Computational Tree Logic is polynomial time checkable in the size of the system times the length of the formula [14]. These correspond to the number of statements in the program being transformed, and the side condition of the transformational specification. Our pattern matching, and reconstruction implementations are both linear in the size of the pattern plus the size of the method.

We provided a thorough investigation of the performance of several common compiler optimizations specified in TRANS and compared it to existing hand written dataflow analyses in Section 4.7. In general, generated optimizations are 2x slower than hand written optimizations to apply to `Java` bytecode programs, however, some pathological cases exist that cause worse performance.

There are several differences between the TRANS implementation in Section 4.7 and the $\text{TRANS}_{\text{fix}}$ implementation in this chapter. The TRANS implementation compiles specifications, rather than interpreting them, and it also uses Binary Decision Diagrams in order to symbolically represent the state space of the analysis, rather than the explicit model checking we perform. These differences reflect the prototype nature of the implementation described here compared to the relative completeness of the TRANS

implementation. The use of TRANS in optimization must consider more states than the proposed use of $\text{TRANS}_{\text{fix}}$, since reducing silhouettes to a source code CFG reduces the number of nodes within the graph, as several bytecode statements may correspond to one source code statement.

5.6 Analysis

5.6.1 Stability

When many bug patterns specific to $\text{TRANS}_{\text{fix}}$ fix a bug within the source code then, since the bug is fixed, any future attempts to apply it to the source code will not change it. We refer to transformations that, having been applied to a program, are not re-applicable when analysed again as stable. In other words, the pattern matching or side condition of the transformation are not enabled at a second analysis of the transformation. This condition intuitively makes sense in many conditions: if a transformation fixes a bug then the bug is fixed after its application and a second application would not fix it further.

A transformation being stable also provides a reliable guarantee to practical usage scenarios. If bug fixing transformations are being repeatedly applied to a codebase during its development then they should not introduce another set of safety conditions at each application, since that would degrade code quality over time.

Optimisations are considered to be stable through an informal analysis of the side condition and pattern matching. An algorithm analysing transformations would be preferable and is a potential possibility, but has not been investigated within the constraints of this thesis. This informal analysis can be easily conducted for any transformation.

```
REPLACE
     l: l.lock();
        ....
     u: l.unlock();
WITH
    try {
       l.lock();
       ....
    } finally {
       l.unlock();
    }
WHERE
                  EF(u) | ¬AF(u)@ l
```

Figure 5.20: Transformation to ensure lock released on all paths

Consider the correct lock pattern checking pattern specifications, recalled in Figure 5.20. After the transformation is applied from any potential call of `lock()` the corresponding `unlock()` call is wrapped within a `finally` block. Consequently there is always a path from a `lock()` call to an `unlock` call. The ensures that the side condition fragment AF(u)@ `l` holds true, and the side condition is disabled.

5.6.2 Correctness Issues

In many cases it is not possible to necessarily determine whether an instance of a bug pattern is a genuine bug, but is frequently possible to determine that, if the scenario is an instance of the bug in question, that the given fix is a genuine fix. In other words all program traces that are within the desired specification, but not the actual specification are removed, and all program traces that fall within the actual specification, but not the desired specification are removed.

It is possible that the program itself might be correct, and accordingly the transformations should not be applied automatically. Additionally the bug finding patterns that we focus on correspond to behaviours that are generally considered bugs within a

program, for example deadlocks.

We would like to extend our methodology in order to be able to ensure that we are soundly applying transformations, rather than simply leaving the choice of whether to apply these transformations to the user of the tool. The required soundness properties could be annotated onto the program. For example our specification for ensuring that locks are released on all paths is sound if and only if the user of the system wishes a lock to be in a released state as a post-condition of the method. Information of this nature can already be added to Java programs using the existing annotations framework, that has been recently extended by [22]. There are already existing tools for invariant detection about partially annotated Java programs. [23] infers properties about nullness of variables.

Another element of such an extension would be the ability to automatically infer the soundness of transformations with respect to given pre and post conditions. Progress has been made towards the inverse goal. For example [72] provides a system for automatically inferring dataflow analyses from facts. Unlike compiler optimizations, transformations applied to fix bugs are not semantics preserving. The very aim of the transformation is to alter the program semantics in order to remove a bug. Consequently one is assuming that the program itself is incorrect according to some specification, but can be corrected to match this specification.

5.6.3 Optimizations

Some existing compiler optimizations can be used to remove potential bugs, or unclear code within programs, for example dead assignment removal, or unreachable code elimination. These can be specified within the existing TRANS system, of which $\text{TRANS}_{\text{fix}}$ is an extension.

Since these would be semantics preserving optimizations, there is less concern about applying them automatically. However user feedback might still be beneficial here, since a user may have written a method, and wish to keep it within their codebase, but may not have started to use it within their code. Consequently, removal of dead code, even if semantics preserving, should be applied with care.

Other optimizations specified in TRANS include lazy code motion, constant propagation, strength reduction, branch elimination, skip elimination, loop fusion, and lazy strength reduction; further details of these can be found in Section 3.5 and Section 3.6.

5.6.4 Further Applications

Other elements of IDE and language analysis tool construction can also benefit from the source transformation language we outline here. Refactoring operations are an interesting example of such a transformation.

A refactoring operation attempts to change the structure of a program internally, in order to improve readability or maintainability, without altering the observable functional behaviour of the program, for example by extracting some block of code into a named method. In this context existing formal analysis of TRANS-like languages could be useful for ensuring observational equivalence.

Refactoring operations require further information from the user of the IDE that $\text{TRANS}_{\text{fix}}$ does not provide. For example in the *Extract Method* refactoring operation one would need to know what the name of the method is. In order to automatically apply such transformations the concept of schematic variables is introduced.

A schematic variable is a variable that is not bound in the pattern matching or temporal constraints. We syntactically differentiate schematic variables by prefixing them with a ? symbol, such as `?x`. We can use the type system described in Sec-

tion 5.3.7 and underlying syntactic structure of the transformation to infer the type of any schematic variable. When applying the transformations we can use the schematic variables to display appropriate user interface dialogs in a given IDE.

Refactoring operations usually correspond to a refinement relation between programs. That is to say that all possible behaviours of the source program, are maintained in the transformed program, but the inverse relation may not hold. For example by extracting a public method from a sequence of statements we change the public API for a library. We can formalise this correctness criteria by stating that "There's no path through the program that is removed." This does not hold true of non-refactoring operations.

5.6.5 Applicability to Other Languages

Whilst our implementation is focused on the `Java` language, many of the ideas about automated bug fixing can be applied to other high level languages. The C and C++ programming languages, due to their similarity to `Java`, can have the same approach applied, but with the syntax of the program transformation language modified to suit their abstract syntax trees. Existing work has already been conducted on model checking and static analysis for the C programming language and in many ways this is an easier task than `Java` since the lack of classes and objects make the function dispatch semantics simpler.

In recent years a complex templating system has been added to C++ [77] that allows turing complete generation of code at compile time. The template free subset of C++ can be treated much like `Java`, albeit with a more sophisticated alias analysis algorithm that accounts for the presence of pointers. Incorporating $\text{TRANS}_{\text{fix}}$ into the templating system would require a modification of $\text{TRANS}_{\text{fix}}$ to allow the `type` predicate

to express the structural sub-typing relation that templates employ. For example instead of being able to express *variable x is an instance of class Foo* it would need to be able to express properties such as *variable x has method foo whose first argument is an Int*.

Strongly, dynamically, typed programming languages such as Ruby and Python are harder to adapt to. The open classes of Ruby — classes that can be spread over multiple files and imported in different orders — make any static analysis more difficult. For example it is impossible to statically determine whether a method being called even exists in the receiving object. Nonetheless the approach of transforming the abstract syntax tree could be applied, and basic control flow graph information can be extracted.

Functional languages such as SML or Haskell generally avoid program state, so the expressiveness of temporal logic side conditions is less useful. Furthermore the lazy evaluation of Haskell leaves the order of evaluation less well defined, and the very ordering of program states is thus also less well defined. Nonetheless few projects are written in functional programming languages, and the aforementioned problems are also true of existing use of model checkers and static analyses in the context of functional languages.

5.6.6 Further Work

The implementation could be further improved in various ways.

1. Improve the performance, by implementing a symbolic model checker, or backing onto a SAT solver.

2. Complete the implementation of language features, for example schematic variables and strategies.

3. Integrate into IDEs, in order to be able to use the tool effectively, rather than to

simply experiment with $\text{TRANS}_{\text{fix}}$.

The idea of using Silhouettes to unify information about a program between different representations of that program could be extended in many areas of program analysis. For example a sophisticated way of reducing register pressure during the register allocation phase of a compiler is using liveness and alias analysis information about variables in the program. This is frequently hampered by the nature of different intermediate representations in compilers. For example the GCC compiler has more precise and sophisticated alias analysis algorithms implemented over its high level *gimple* representation, but it only perform register allocation over the low level *Register Transfer Language* based representation. If one could unify the structure of the two representations using Silhouettes it might be possible to transfer the information collected about variables from the high level to the low level representation.

An investigation into the decidability of the stability property of transformations could provide benefit to the users of a $\text{TRANS}_{\text{fix}}$ system. The ideal goal here would be to provide an algorithm that decides whether a transformation is stable or not. Even a conservative or optimistic approximation that can decide for a subset of specifications whether they are stable or not would help a program transformation specifier understand their specifications better.

5.6.7 Conclusions

The TRANS language for program transformation specifications has been extended to encompass source code programs. This allows the specification of transformations that automatically remove bug patterns in computer programs. Whilst the transformations that perform compiler optimizations aim to preserve the semantics of the programs that they are optimizing—an issue considered further in Chapter 6—bug fixing transforma-

tions explicitly aim to subtly alter the semantics of programs they are applied to.

We describe a tool that allows the automated application of these transformations to programs and how its use can be integrated into existing development methodologies. Our implementation uses a novel technique for combining source code and object code analysis through *silhouettes*—a technique for unifying information annotated onto a control flow graph. This provides the same underlying model as the $\text{TRANS}_{\text{fix}}$ specification language for transformations.

In future this approach could be combined with partial program specifications in order to place the automated fixing of bugs through formally specified program transformation on a sound semantic footing. The codifying of common bug patterns in itself helps programmers to understand and appreciate the art of computer programming, through the subtleties of its science.

Chapter 6

Formal Analysis of Optimization Soundness

While the extension of TRANS to a source code transformation language in the previous chapter required little in the way of formal analysis, its usage in specifying compiler optimizations enables us to analyse the effect that these program transformations have on the program they are optimizing. Compiler optimizations are a special class of program transformation in that they are altering the way in a which program works, but not what the program itself does. We refer to compiler optimizations as being *semantics preserving*.

This chapter discusses a formal analysis of the TRANS specifications in the Isabelle theorem prover. It introduces some notions of semantics preservation and expounds on how a simple framework for proving semantics preservation is explored in the environment of the Isabelle theorem prover, over the `jinja` language that offers `Java`-like semantics. We discuss proofs of two example optimizations specified in the framework.

Proofs that have been conducted in Isabelle are referred to directly, so the theorems that you see, are those that have been proved. The Isabelle version used is Isabelle 2008, and the `jinja` [39] version used is that submitted to the Archive of Formal Proofs [26].

The example proofs of optimizations for Constant Propagation and Loop Invariant Code Motion prove in Isabelle a series of properties about the optimizations. There's a by-hand analysis that these properties imply soundness of the transformation. Furthermore these properties correspond to complex elements of proofs about TRANS defined over L_0 published in [44] and [47].

Specifically compiler optimizations that transform or refine a program within an abstract representation are considered. The convention that S refers to the source program, in other words before transformation, and that T refers to the transformed program is used.

6.1 Semantics Preservation

A transformation is considered to be semantics preserving if the set of properties that can be observed to hold of the source program also hold of the transformed program. The set of properties that may hold of a given program is specific to the semantics of the language and the nature of the definition of a language. We discuss these considerations in the context of the `jinja` language in Section 6.2.

6.1.1 Extensional Equivalence

A simple notion of equivalence between two functions is that of extensional equivalence, this considers two functions to be equivalent if the same result for given arguments.

Definition 6.1.1 *Two functions, S and T are extensionally equivalent iff $\forall x.Sx = Tx$*

In this approach a transformation is sound if it preserves extensional equivalence about all functions within the program.

This approach is not subtle enough to cover some scenarios about programs being transformed. Firstly many programs perform interaction with their environment that may constitute observable behaviour. For example in `Java` calling `System.out.println`, or in `C` `printf`. This is a normal function call.

Another issue ignored is potential non-determinism of functions, which restricts the applicability of this approach in concurrent environments. Our approach to program transformation is applicable for imperative languages that have program state, and extensional equivalence ignores the nature of state within the program.

6.1.2 Bisimulation

Bisimulation is a more sophisticated form of equivalence relation between state transition systems than extensional equivalence. If we describe two state transition systems, S and T to be bisimilar we mean there exists some bisimulation relation that holds between S and T. The bisimulation relation matches up states from S and T so that if there is a transition between states in S there is a corresponding transition between states in T and vice-versa [63].

Definition 6.1.2 *Given two programs, represented by state transition systems, $S = (N_S, \rightarrow_S)$ and $T = (N_T, \rightarrow_T)$, we say that S and T are* **bisimilar** *if there is a bisimilarity relation R and the follow properties hold of R:*

(a) $R \subset N_S \times N_T$

(b) if sRt and $s \rightarrow_S s_1$ then $t \rightarrow_T t_1$ and $s_1 R t_1$

(c) if sRt *and* $t \rightarrow_T t_1$ *then* $s \rightarrow_S s_1$ *and* $s_1 R t_1$

Part (a) of this definition denotes that the bisimulation relation holds between states. Part (b) expresses what it means for S to simulate T and Part (c) for T to simulate S.

Bisimilarity allows us to account for equivalence between concurrent programs. Since bisimilarity incorporates the state transition relation into equivalence, it is possible to incorporate program input/output into it as well.

Bisimilarity relations embody a strong notion of equivalence. They require that each state transition in S is matched by one and only one transition in T. This may not be an appropriate notion of equivalence in some compiler optimization cases. Consider the case of an optimisation that removes a loop that is guaranteed to terminate after a finite number of iterations and does not alter program state other than its loop counter. Informally the user of a program may consider this to be an optimization that preserves semantics, since there is no observable change in the behaviour of the program, other than in terms of its performance characteristics. The before and after programs would not be bisimilar, however, since the after program has only one transition that corresponds to the before programs' many transitions in a loop.

6.1.3 Weak Bisimulation

Bisimulation can be generalised to weak bisimulation by removing the constraint that when a step is simulated, it is simulated by a single step. Note that in Definition 6.1.3 $\rightarrow^*$ abbreviates the reflexive and transitive closure of the relation $\rightarrow$.

Definition 6.1.3 *Given two programs, represented by state transition systems,* $S =$

(N_S,$\rightarrow_S$) and $T = (N_T,\rightarrow_T)$, we say that S and T are **weakly bisimilar** *if there is a weak bisimilarity relation R and the follow properties hold of R:*

(a) $R \subset N \times N'$

(b) if sRs' and $s \rightarrow_S s_1$ then $s' \rightarrow_T^* s'_1$ and $s_1 R s'_1$

(c) if sRs' and $s' \rightarrow_T s'_1$ then $s \rightarrow_S^* s_1$ and $s_1 R s'_1$

6.1.4 Theorem Provers

Theorem provers, or proof assistants as they are sometimes known, are computer programs that provide an interactive environment in which to verify mathematical theorems. Theorem provers provide a language in which to specify some mathematical property and script a series of deductive steps that prove the property.

The proof language can be an existing programming language and the type system of this language is used to ensure correct application of rules—this is true of HOL Lite, described by [30]. In other cases, such as the Isabelle [64] and Coq [16] theorem provers the tool has a parser and intermediate representation for the language itself.

The proof metalanguage may semantically be a form of logic, either higher or first order, or some other form of algebra. For example the Coq theorem prover uses the Calculus of Inductive Constructions. This extends a higher order lambda calculus, by allowing the construction of inductive data types and types as first class values.

The LCF Family of theorem provers are a well known group of theorem provers, including the LCF [28], HOL [27], and Isabelle theorem provers. The logical core is a library within the programming language that the theorem prover is implemented in, usually an ML variant. New theorems can only be introduced via library functions,

which correspond to primitive inference rules within the proof logic. The abstract type discipline ensures that no invalid theorems may be proved, if the library functions are correctly written. Consequently LCF Theorem Provers are said to have a small 'trusted core' that is built upon within its proof language. This minimises the chance that programming errors may have introduced logical incorrectness.

6.1.5 Isabelle

Isabelle is a successor to the HOL Theorem Prover. The primary advance of Isabelle is that the core proof logic is not specific to a single logical formalisations [65]. Consequently it introduces a meta-logic with a few simple primitives, into which different object logics can be encoded. Proofs are then conducted within the object logics.

Proofs are primarily constructed through higher order unification. In this approach two terms can be unified if there is a unifying term that can substitute for free variables in both terms. Applying individual proof rules manually is augmented by tactics, that allow automated reasoning and application of proof rules. Isabelle's standard distribution contains several object logics, for example First Order Logic, Higher Order Logic and Zermelo Frankel Logic. We focus on the Higher Order Logic embedding for the remainder of this discussion.

6.1.6 Isabelle/HOL

Higher Order Logic is the most commonly used object logic within Isabelle, and its formalisation within, and use combined with, Isabelle is referred to as Isabelle/HOL. As well as traditional logical connectives, such as $\wedge$ it offers an environment for defining functional programming-like specifications. Custom datatypes can be defined, and pattern matching defined over them. All functions are total within this formalism, and

Isabelle can discharge termination proofs for many functions automatically.

Isabelle allows for sets to be defined inductively, and co-inductively. This is of particular use when defining formal semantics for programming languages. Transition relations between states can be defined as set membership, and inference rules within the formal semantics can be defined as inductive set membership rules.

6.2 `jinja`

We base our model of program semantics on the `jinja` language [39], which is a `Java` like system, entirely mechanized in Isabelle/HOL. It has been published within the Archive of Formal Proofs [58]. The theory itself presents big-step and small-step operational semantics for the source language, a proof of equivalence between these semantics, bytecode semantics and a multi-stage compiler from the source language to the bytecode language. It also contains a notion of programs being well formed, and a proof that the compiler correctly translates well formed programs. The `Bytecode` definition includes a simple generic dataflow framework and a `Bytecode` verifier. The `Bytecode` verifier performs an operation similar in to type checking. It is implemented using the dataflow framework.

Our use of the `jinja` framework focuses on the small-step operational semantics for the source program. The TRANS language's expression pattern matching works at a higher level than bytecode. For example, there being little structural restriction within bytecode, assuming some bytecode sequence passes bytecode verification, one can introduce goto instructions in the middle of integer arithmetic. Additionally arithmetic is compiled into several stack manipulating operations. TRANS makes assumptions about the structure of the language that make bytecode unsuitable. Proofs using TRANS are frequently constructed by proving that some property about program state is implied

by a side condition and that that property implies the soundness of a rewrite. This makes small-step semantics more appropriate than big step semantics, since it makes the internal state of the program at some point in its execution more explicit.

In `jinja` the only concept below a method is that of an expression. A statement is treated as an expression that is of the `Unit` type. Expressions are denoted as being *final* if they are either a value—`Val v`—or throw an exception—`throw (Addr a)`. In this case the `Addr a` refers to an address in the heap. The small step semantics define an inductive set that is syntactically sugared so that $P \vdash \langle e, s\rangle \rightarrow \langle e', s'\rangle$ denotes that for program P, expression e in state s is reduced by one step to expression e' in state s'. Program traces are denoted by $P \vdash \langle e, s\rangle \rightarrow *\langle e', s'\rangle$, which is the reflexive, transitive closure of the reduction relation.

`jinja` partially replicates the memory model of `Java`. In `jinja` $vname$ denotes a variable name, $cname$ denotes a class and val denotes a value.

Definition 6.2.1 *State in* `jinja`

$state = heap \times locals$

$locals = vname \rightharpoonup val$

$heap = addr \rightharpoonup obj$

$obj = cname \times fields$

$fields = vname \times cname \rightharpoonup val$

`jinja` does not completely represent `Java`; for example, whilst `Java` has multiple primitive datatypes, such as `float` and `short`, `jinja` only has an `int` type. Furthermore the only numerical operations defined within `jinja` are addition and comparison. This somewhat limits certain optimizations, for example the strength reduction of multiplication operations into addition operations within loop inductive variables. This can

be formally specified within TRANS specifications, but the Isabelle formalisation does not support it, since there is no multiplication.

6.3 Mechanisation and Proofs

There is ongoing discussion concerning the merits and issues of deep and shallow embedding within theorem provers [90]. In a deep embedding, the syntax of the language to be embedded is modelled as an abstract datatype. In a shallow embedding the logical formulae are written using HOL definitions, predicates and functions. The approach outlined below follows a shallow embedding.

We translate transformations into functions that change expressions within `jinja`. The overall style of embedding is modelled upon the idea of a combinator library, within functional programming. Consequently all language primitives get translated into functional combinators, and the writing of a TRANS specification consists of writing several functions that combine these language primitives.

6.3.1 Refinement

The TRANS specifications undergo some manual refinement, from the form that they are written in, to the form that they are used in Isabelle. The rewrite rules are converted into a pattern matching component that forms part of the side condition, and a replacement rule that becomes the new action. The replacement function simply replaces one expression with another in the `jinja` expression. It is consequently simpler to reason about, and is discussed in Section 6.3.5. Additionally macros are expanded.

6.3.2 Expression Reduction and Local Equivalence

This section provides some general definitions that will be used to denote transformational soundness for different TRANS optimisations. The definitions are not specific to any one transformation.

Definition 6.3.1 *The function* `replace` *relates expresions to their replacement. Its first argument* `init` *is the initial overall expression,* `from` *is some subexpression of* `init` *that is to be replaced with its third argument* `to`*. The function returns the replaced expression.*

fun *replace* :: *expr* ⇒ *expr* ⇒ *expr* ⇒ *expr*

The `replace` function is used to express the fact that we want methods in a program to be semantically preserved. It replaces a sub-expression within the method's body with another expression. This also allows us to prove The Sound Replacement lemma.

Lemma 6.3.1 *If we replace a subexpression* `from` *with another expression* `to` *in a method body, and under any circumstance* `to` *evaluates to the same final value in the same state as* `from`*, given the same initial state then overall method body evaluates to the same final value, in the same state. This is called SoundReplacement*

lemma *SoundReplacement*: !! *s*. [|*prog* ⊢ ⟨*init*, *s*⟩ →∗ ⟨*e′*, *s′*⟩;*prog* ⊢ ⟨*e*, *s*⟩ →∗ ⟨*e′*, *s′*⟩|]
==> *prog* ⊢ ⟨ (*replace init from e*), *s*⟩ →∗ ⟨*e′*, *s′*⟩

This lemma supports the fact that we only need to establish a local soundness condition for a given transformation. It is local in the sense that it suffices to prove that merely the expression being replaced evaluates to the same final value in the

same state.

Definition 6.3.2 *Two expressions are state equivalent if they both reduce to the same final value and state given the same initial state.*

definition *equivSt* :: *J-prog* => *expr* => *state* => *expr* => *state* => *bool* **where**
equivSt prog e1 s1 e2 s2 ==
  EX e' s'. prog ⊢ ⟨*e1*, *s1*⟩ →* ⟨*e'*, *s'*⟩ = *prog* ⊢ ⟨*e2*, *s2*⟩ →* ⟨*e'*, *s'*⟩

Lemma 6.3.2 *If expression* e *can reduce to* e' *then* e *and* e' *are state equivalent.*

lemma *EquivalentIfReduceable*: *prog* ⊢ ⟨*e*,(*h*,*l*)⟩ →* ⟨*e'*,(*h'*,*l'*)⟩ ==>
  equivSt prog e (*h*,*l*) *e'* (*h'*,*l'*)

6.3.3 Predicates

Predicates within TRANS can be considered in two ways.

The term *semantic property* refers to the weakest condition that is implied about a program statement by the predicate in question holding true. This corresponds to the information about the program being transformed that the predicate denotes.

Secondly, there is the stronger function that a compiler developer would write in order to implement the corresponding predicate within their compiler. This implementation function should always imply the semantic property of the predicate, but due to some limitation, or design choice, it may also restrict the program additionally. For example we refer to `use` and `def` predicates, but a compiler implementor

in a language that has some form of variable aliasing would have to conservatively approximate such properties using a `may-use` and `may-def` predicate.

The embedding defines the semantic properties for predicates, in order to simplify the theorem proving effort and abstract from any specific implementation details. A more complete system that also proves the implementation correct would define an implementation function that corresponds to each predicate, and then prove that the function implies the semantic property.

For some predicates we define the negated predicate separately from the predicate. For example the `notdef` definition:

abbreviation

notdef :: *vname* ⇒ *J-prog* ⇒ *expr* ⇒ *heap* ⇒ *locals* ⇒ *expr* ⇒ *heap* ⇒ *locals* ⇒ *bool* **where**

notdef var prog e h l e′ h′ l′ ≡ *prog* ⊢ ⟨*e*,(*h*,*l*)⟩ → ⟨*e′*,(*h′*,*l′*)⟩ & *l var* = *l′ var*

Here the semantic property implied by the definition is that as expression `e` for a given heap `h` and local variables `l` evaluates to `e'` in (h', l') the local variable `var` refers the same value within the local variable environment before the evaluation and after. It additionally enforces that these expressions do indeed evaluate from `e` to `e'`.

6.3.4 Temporal Operators

Temporal operators within TRANS all correspond to functions within Isabelle/HOL. These hold true for a given temporal formulae ϕ if ϕ is satisfied for a list of expression, state pairs. This relates to a possible path of evaluation for the expression to take. By using recursion over lists, Isabelle's standard induction tactics provide much help for proving properties about the temporal operators, which fits our shallow embedding approach. Additionally we can use HOL's Existential and Universal quantification, and

```
a = 3;          a = 3;
b = a;          b = 3;
```

Figure 6.1: Code snippet before and after Constant Propagation

associated lemmas for reasoning about the corresponding quantification over paths in CTL.

6.3.5 Actions

For each TRANS language primitive there is a corresponding function, that performs the general action. Each action is consequently dealt with separately. The example proofs given for Constant Propagation and Loop Invariant Code Motion, in the subsequent sections, both express their actions directly in Isabelle/HOL.

6.4 Constant Propagation

Constant propagation is a transformation where the use of a variable is replaced with the use of a constant known at compile time. An example application to a trivial program is given in Figure 6.3.

Hence, where a variable x is assigned to a variable v, replace it with an assignment of a constant c to the variable x. We consequently specify the rewrite rule as:

$$n : (x := v) \implies x := c$$

The side condition needs to check that x will always be assigned to v at that program point and that v is always assigned to c on a path through the control flow graph where this optimization is applied. It is formulated by looking backwards on all

$$n : (x := e[v]) \Longrightarrow x := e[c]$$
$$\text{if}$$
$$\overleftarrow{A}(\neg \mathit{def}(v)\ U\ \mathit{stmt}(v := c))\ @\ n\ \wedge\ \mathit{conlit}(c)$$

Figure 6.2: Specification of constant propagation

paths through the program until it finds the point at which v is assigned to c, and checking that v is not redefined between that point and program point n. Additionally we need another predicate to ensure that c is genuinely a constant. A recap of the specification from Figure 3.8 is given in Figure 6.2.

The proof of soundness for constant propagation is structured as follows:

1. Prove in the out state of `v := c` that the variable v holds the value of the constant c
2. Prove that if at the beginning state of a path, variable v holds the value c, and v is not redefined then it will still hold that value at the end of the path.
3. Prove if for all paths entering an assignment `x := v` that if v has the value c at the in state, and x is assigned to v that it results in the variable x holding the value c, in other words that `x := c` evaluates to the same state as `x := v` when $\mathtt{v} = \mathtt{c}$

We now sketch some details of how this is translated to Isabelle. Note that within the following section, containing Isabelle theorems, v is referred to as *sub-var*, x as *ass-var* and c as *const*.

Definition 6.4.1 *init abbreviates the initial conditions of the until operator, in other words, that sub-var is assigned a constant.*

abbreviation *init* :: *vname* ⇒ *val* ⇒ *J-prog* ⇒ *expr* ⇒ *state* ⇒ *expr* ⇒ *state* ⇒ *bool* **where**
init sub-var const prog e s e′ s′ ≡ (*e* = *sub-var* := (*Val const*)) & *prog* ⊢ ⟨*e*,*s*⟩ → ⟨*e′*,*s′*⟩

Lemma 6.4.1 *The side condition initialisation lemma shows that init implies that sub-var has been assigned to a value of const within its successor state, which corresponds to step (1).*

lemma *SideConditionInit*: *init sub-var const prog e* (*h*,*l*) *e′* (*h′*, *l′*) ==>
l′ sub-var = (*Some const*)

Definition 6.4.2 *side-cond defines the side condition of the Constant Propagation specification in Isabelle/HOL. It is a translation of the condition* $\overleftarrow{A}(\neg\mathsf{def}(v)\ U\ \mathsf{stmt}(v := c))$.

fun *side-cond* :: *vname* ⇒ *val* ⇒ *J-prog* ⇒ *expr* ⇒ *state* ⇒ (*expr* ∗ *state*) *list* ⇒ *bool* **where**
side-cond sub-var const prog e s [(*e′*,*h′*,*l′*)] = *init sub-var const prog e s e′* (*h′*,*l′*) |
side-cond sub-var const prog pe ps ((*e*,*h*,*l*)#(*e′*,*h′*,*l′*)#*es*) =
(*notdef sub-var prog e h l e′ h′ l′* & *side-cond sub-var const prog pe ps* ((*e′*,(*h′*,*l′*))#*es*))

Being just a standard Isabelle/HOL function we can use standard inductive tactics to reason about the until's predicates set of states.

Lemma 6.4.2 *We show that Definition 6.4.2 implies that the state at the final node in the side condition chain is the required state. This is step (2) of the proof.*

lemma *SideConditionPropagation*: [| *side-cond sub-var const prog e s es*; *es* ≠ [] ;
(*e*,*h*,*l*) = (*last es*) |] ==> *l sub-var* = (*Some const*)

This proof proceeds by induction over the *side-cond* function.

Formulating step (3) corresponds to the following Isabelle Lemma. The assumption states that *sub-var* is equal to `const` in the the initial state.

```
i = 0                i = 0
while(i < 10)        k = 3*4
    k = 3*4          while(i < 10)
    i = i + k            i = i + k
```

Figure 6.3: Code Snippet before and after Loop Invariant Code Motion

lemma $[|lcl\ s\ sub\text{-}var = Some\ const\ ;\ s = (h,l)|] ==>$

$(P \vdash \langle ass\text{-}var{:=}(Var\ sub\text{-}var),\ s\rangle \rightarrow* \langle unit,\ (h,l(ass\text{-}var \mapsto const))\rangle)$

6.5 Loop Invariant Code Motion

Loop invariant code motion is an optimization that moves statements out of a loop that do not need to be executed every iteration.

We denote the expression in its initial position as 'before' and having been moved to the loop pre-head as `after`. The variable (in the example `k`) is denoted `invar`. The expression being assigned to `invar` (eg `3*4`) must be transparent on all paths between before and after and is denoted `expr`.

The proof of soundness for Loop Invariant Code Motion can be decomposed into three steps.

1. Show state transformation of `invar = expr` is the same at `after` as at location `before`. This implies subsequent reductions result in same final state and expression.

2. Show state between `after` and `before` is the same except for introduction of 'invar'. This implies expressions between are semantically equivalent in terms of their effects.

3. Show `expr` being transparent implies it computes the same value at before as after.

Loop Invariant Code Motion transforms the program at two different points: the position where it removes the assignment statement, and where it introduces the assignment. This is because we specify the movement of a statement as rewriting from a skip instruction and rewriting to a skip instruction. The `replace` function we use to specify the overall program transformation only replaces a single statement. This is not a problem for multiple statement changing optimizations in the context of `jinja` however, since every statement is an expression. Since statement composition is just another form of expression we can simply refer to the transformation of the entire program sequence changed by Loop Invariant Code Motion as being the replacement of a single expression. For example Figure 6.3 shows both the before and after snippets for a `jinja` expression.

Definition 6.5.1 *The Local State Mutation Function — this results from the reduction of the invariant expression and its assignment to* `invar`.

definition *mutate* :: *vname* => *val* => *state* => *state* **where**
mutate invar v s = (*let* (*h*,*l*) = *s in* (*h*,*l*(*invar*↦*v*)))

Lemma 6.5.1 *The local state with* `expr` *removed evaluates to the same state as before, but with* `invar` *removed from the set of local variables. Step (1) of proof.*

lemma *SoundStatementRemoval*: [|(*h*,*l*) = *s*; *prog* ⊢ ⟨*exp*,*s*⟩ → ⟨*Val v*,*s*⟩;
(*s*′ = *mutate invar v s*)|] ==> (*equivSt prog* (*invar*:=*exp*) *s unit s*′)

Definition 6.5.2 *A mutation is the side effects to program state of a sequence of steps of reduction.*

definition *mutation* :: *expr* => *J-prog* => *state* => *expr* => (*state* => *state*) => *bool* **where**

mutation e prog s e′ f = *prog* ⊢ ⟨*e*, *s*⟩ →∗ ⟨*e′*, *f s*⟩

The purpose of the following definition is to show that the function `f` forms the same relationship between states as executing expression `e` in state `s` does.

Definition 6.5.3 *An expression is ambivalent to a pair of states if it results in the same mutation and reduced expression if it is reduced in either state.*

definition *ambivalent* :: *expr* => *J-prog* => *state* => *state* => *bool* **where**

ambivalent e prog s1 s2 == *EX f . ALL e′. mutation e prog s1 e′ f* & *mutation e prog s2 e′ f*

Definition 6.5.4 *An expression is transparent with respect to another expression if it is ambivalent to its before and after states.*

abbreviation *trans* :: *expr* => *J-prog* => *expr* => *state* => *expr* => *state* => *bool* **where**

trans transp prog e s e′ s′ == (*prog* ⊢ ⟨*e*, *s*⟩ → ⟨*e′*, *s′*⟩) −−> (*ambivalent transp prog s s′*)

Lemma 6.5.2 *We show that* `invar := expr` *statement results in the mutation that* `invar` *is defined in the current state.*

lemma *prog* ⊢ ⟨*exp*,(*h*, *l*)⟩ → ⟨*Val v*,(*h*, *l*)⟩ ==>

mutation (*invar*:=*exp*) *prog* (*h*,*l*) *unit* (*mutate invar v*)

Definition 6.5.5 *The side condition of the Loop Invariant Code Motion specification into Isabelle, as a recursive function is called* `sc`.

fun *sc* :: *J-prog* => *vname* => *expr* => *val* => (*expr*∗*state*∗*expr*∗*state*) *list* => *bool* **where**

sc p invar exp v [(*e*,*s*,*e*′,*s*′)] = (*s*′ = *mutate invar v s* & *e* = *e*′) |

sc p invar exp v ((*eb*,(*hb*,*lb*),*ea*,(*ha*,*la*))#(*eb*′,(*hb*′,*lb*′),*ea*′,(*ha*′,*la*′))#*ess*) =

((*notdef invar p eb hb lb eb*′ *hb*′ *lb*′) &

(*notuse invar p eb eb*′ *hb lb hb*′ *lb*′ *ha la ha*′ *la*′) &

trans exp p eb (*hb*,*lb*) *eb*′ (*hb*′,*lb*′) &

p ⊢ ⟨*eb*,(*hb*,*lb*)⟩ → ⟨*eb*′,(*hb*′,*lb*′)⟩ &

p ⊢ ⟨*ea*,(*ha*,*la*)⟩ → ⟨*eb*′,(*ha*′,*la*′)⟩&

sc p invar exp v ((*eb*′,(*hb*′,*lb*′),*ea*′,(*ha*′,*la*′))#*ess*))

This allows us to show that if invar is not used and not redefined then the state relationship establishedin step (1) of the proof holds over the entire `(expr * state)` sequence. This corresponds to step (2) of the proof. The final lemma for step (3) requires us to show that an expression `expr` being transparent with respect to another expression `e` means that it calculates the same value before or after it. Note that we use the `mutation` definition in order to ensure that it provides the same state change function

lemma *trans expr prog e s e*′ *s*′ & *prog* ⊢ ⟨*expr*,*s*⟩ → ⟨*expr*′,*s-expr*′⟩ & *mutation expr prog s expr*′ *f* ==>

prog ⊢ ⟨*expr*,*s*′⟩ → ⟨*expr*′,*f s*′⟩

6.6 Alternative Language Definitions

Even though the definitions in this chapter have been built on top of the `jinja` formal semantics the ideas contained within can be applied to other programming languages. Even though undertaking the work of formalising the ideas against other programming

languages is outside the scope of this thesis, we do consider the C programming language henceforth.

The *Cholera* system, introduced in [59] provides a formal operational and axiomatic semantics for a large subset of the C programming language in the HOL theorem prover. Transferring the definition of TRANS to such a system would require changing the underlying definitions in a variety of ways.

The control flow graph would have to be defined, however, since there is a clear next state relation for C programs this is an achievable task. The abstract nature of the definitions provided for the `use` and `def` predicates makes it easier to incorporate pointers into the language. One of the most difficult challenges, however, would be accounting for the ambiguous nature of the manner in which expressions are reduced in C and the sequence point system.

A more general, albeit complicated, approach one could undertake would be to prove that program transformations are semantics preserving with respect to some abstract program semantics, and then to prove that a certain language meets the abstract program semantics. The Isabelle theorem prover provides some support for this form of definition in the form of its Axiomatic Type Classes [87]. An axiomatic type class allows the definition of a Haskell style type class, a form of abstract type, along with the assertion of a series of axioms about the type. Any instance of the type class has to provide witnesses that the type class's axioms all hold.

This would allow the structuring of proofs against an axiomatic type class, with a language providing an instance of that type class. There are some disadvantages for this approach however—notably that the language type class would have to be highly generic, making any proofs reliant on its definitions harder than for a specific language.

An alternative approach for abstracting proofs between different languages is

inspired by [17], where low level microcode is transformed into a high level language and then verified. Languages such as `Java` and `C` can be embedded into a high level language, about which proofs of transformational soundness can be extracted. The embedding would have to show that properties proved about the control flow graph of the higher level language still held of the embedded languages.

6.7 Conclusions

Exploratory work has been undertaken for creating a framework for proving that TRANS optimizations are semantics preserving, using a language similar to the real world `Java` language. Two optimizations have been proved to be semantics preserving. Furthermore there are underlying definitions and lemmas that are reusable for different TRANS optimizations.

There is much work yet to be done. Some elements of the formalisation are incomplete, such as the embedding of all of the remaining components of the TRANS language, and more optimizations need to proved sound, including the list cited in Chapter 3. A significant incomplete element of the framework is the definition of program strategies. Since the rest of the framework is defined in the style of functional combinators, and strategies form a kind of combinator for transformations they should fall into the structure of the overall definition quite well.

The manner in which program transformations in this section are defined follows the TRANS model specifically. It is possible to take a TRANS specification and convert it systemically into a series of Isabelle function calls that correspond to the definition of the same TRANS specification in Isabelle. Consequently the proofs about these functions correspond to proofs about a TRANS specification rather than an abstract transformation.

[53] describes the implementation of several TRANS strategies and how they were changed in order to be acceptable by Isabelle. Notably the strategies are altered to ensure that transformations are never applied to graphs on which their side condition does not hold. In order to achieve this the `APPLY_ALL` definition becomes recursive. If we were to extend our formalisation in order to include strategies then such changes are to be expected, but could be undertaken without serious implications for the TRANS language.

Chapter 4 described how an efficient implementation could be derived from a specification, whilst this chapter completes that picture, defining some of the language's semantics in a theorem prover. The TRANS language provides a positive influence in structuring the definitions that are needed to verify compiler optimizations, and whilst Chapter 4 shows how the language is useful in a practical context this chapter shows how it is useful in a theoretical one.

Chapter 7

Discussion

7.1 Discussion

So far this thesis has focused on discussing research results and accomplishments. There are some remaining limitations in the research discussed. While it is the author's belief that these are not severe, they are worth considering.

7.1.1 The Specification Language

The expressiveness of side conditions within our language is a concern. Whilst we offer a significant set of common compiler optimizations that can be specified using simple CTL, moving to modal μ calculus would improve matters, at the expense of readability. In the case of compiler optimizations being specified with TRANS, faint code elimination is an example optimization that could be specified with the addition of μ calculus. In the case of $\text{TRANS}_{\text{fix}}$ checking whether variables are definitely `null` or not would be an easy property to specify with the addition of μ calculus in the side conditions.

All optimizations are specified as intra-procedural optimisations, and there is

currently no support for inter-procedural analysis. It might be possible to extend the specification language to work over the inter-procedural control flow graph. This would allow the specification of inter-procedural optimizations. The practicalities of this have not been thoroughly investigated however.

Another area in which the transformational language is currently limited is with respect to Object Oriented Programming language features. For example, differentiating between static and virtual methods in `Java` is unsupported. In order to enable a wider range of transformations to be specified, for example a Polymorphic Inline Cache, it would be necessary to differentiate between such properties. Another example of the type of analysis its currently impossible to specify is class hierarchy analysis. Much of the necessary primitive information could be introduced by allowing further predicates into the side condition language. In the same manner that we introduce aliasing information through the `use` and `def` predicates.

7.1.2 Optimisation Generator

As discussed in Section 5.5 the optimization phases generated by the Rosser system are not as efficient as those offered by hand written optimizations. Fundamentally our approach of representing all domains of information within the system using BDDs will lead to some efficiency trade offs, as some information is better represented—both in terms of speed of processing and of memory consumption—using other methods such as bitfields. It may be possible to offer some method of allowing the optimization specifier to account for the different in-memory representations. This is a feature omitted from our implementation however.

Since the original work on optimization generation there has been an improved algorithm for model checking Computational Tree Logic with Variable Bindings [10].

This algorithmic improvement could be incorporated into our system in order to improve its efficiency.

7.1.3 Bug Fixing

The primary concern with our approach to automated bug fixing is the potential for introducing bugs into the program accidentally. By applying a methodology in which the automated transformations are an aid to human programming, rather than being naively applied, one minimises the risk of this occurring, but it is still a concern. We introduced the concept of stability in Section 5.6.1 in order to describe a circumstance where repeatedly applying a bug fixing transformation would introduce a new bug into the program's source code, or at least degrade it in quality by some measure, for example by introducing some duplicate variable check that reduces performance. This notion could be strengthen by introducing an automated check for stability. One could also seek out other, similar, properties about bug fixing transformations that could be used to rule out classes of accidental bugs.

Another issue is the limited set of bugs that can be fixed automatically. In practical terms there will always be bugs that require hand written code in order to identify and fix. It is possible to minimise this set of bugs however. Firstly more bug fixing transformations can be written, this is a task that our domain specific language makes considerably easier than if we were using a standard framework. Secondly we could extend our analysis and side condition framework to work inter-procedurally. This is especially important in the context of Java, whose code style encourages many short methods, and splitting up logic in reusable Classes. In order to achieve this method the underlying model for the temporal logic to check could be changed to the interprocedural control flow graph [57]. Thirdly additional transformational primitives could be

introduced, such as those generating entire classes. This would again allow more bug fixes to be automatically expressed.

The interactive approach to bug fixing, combining tool usage and hand rewrites, necessitates fast analysis of program source and bytecode. This limits the amount of analysis that can be automatically done. For example many interprocedural dataflow analyses are inherently NP-Hard [79]. This underlying tension forces a tradeoff between retaining a fast, interactive, tool and a more sophisticated set of transformations that take time to apply. Good IDE interaction could help minimise this concern by allowing the potential fixes to be applied one by one. Within a development methodology this would manifest itself as a sustained period of automated analysis followed by interactively applying or rejecting suggested transformations.

7.1.4 Formal Analysis

Our approach to analysing compiler optimizations can be used to prove that the optimization does not introduce bugs into programs being optimized, however, it is a time consuming and manual technique, despite the use of proof assistant. The complete language has not been formalised, so not all optimizations written in the language can be expressed in the theorem prover.

7.2 Related Program Transformation Systems

Several systems have been developed for implementing program transformations from specifications, and each presents a different balance between somewhat conflicting goals: richness of the language to express transformations, support for formal reasoning, and efficiency of application to real-world programs. This section details some of these systems and compares them to our approach.

7.2.1 TTL

Kanade's Temporal Transformation Logic (TTL) [37; 38] is a system similar to TRANS, that uses a CTL based proof technique. Kanade's focus is automatic verification of the soundness of the transformations themselves. Accordingly, instead of using the generic rewriting technique that TRANS uses, TTL has a set of transformational primitives. Each primitive represents a common element used within compiler optimizations, for example replacing an expression with a variable. Each of the transformational primitives has an associated soundness condition that, if satisfied, implies the soundness of the transformation. The soundness of transformations within TTL can be proved using the PVS system, which also supports validation of a trace from an instrumented compiler.

The primary difference between TTL and TRANS is that transformation primitives in TTL are less general. Whilst Kanade is able to show that TTL allows automated soundness proofs of some sophisticated optimizations, such as optimal code placement, other common optimizations, for example constant propagation, have not yet been specified, due to the data structures used to implement TTL. The use of specific transformational primitives also raises questions about how general his system is, most notably over the need to introduce further primitives in future. TTL is also seen as a specification language, for other compiler implementations, whilst TRANS can be refined and executed as the optimization stage of a compiler.

7.2.2 Cobalt and Rhodium

Lerner [48] describes the Cobalt system, which supports automated provability and executable specifications. Transformations in Cobalt are similar to TRANS in that they combine rewrite rules for an action and a form of temporal query as a side condition. The Temporal queries are of a more restrictive form that CTL. This has the disadvantage

of reducing the overall expressiveness of the language. It does allow a general proof to be given that if the predicates inside these query patterns imply certain properties, then the specified transformation is sound. These proofs can be conducted once for the entire framework, and are useful across different optimizations. The optimisation specific proof obligations can be discharged automatically using a theorem prover, since they require no inductive heuristics.

The specific nature of Cobalt's temporal conditions, though common to the dataflow analysis approach, is limited when compared to the generic model checking that TRANS performs with its CTL side conditions. This is one of the main motivations given for developing Rhodium [49], which is another domain specific language for developing compiler optimizations. Rhodium consists of local rules that manipulate dataflow facts. This is a significant departure in approach from TRANS, since it uses more traditional, data flow analysis based specifications rather than temporal side conditions.

7.2.3 Coccinelle

Coccinelle [60] is a system for applying semantic patches to existing codebases for the purpose of expressing collateral evolutions. A collateral evolution is a change in some part of a codebase that has an impact on other aspects of the codebase. A motivating example is operating system device drivers, where changing an API function in the core device driver API may require changes in all device drivers that call that API functions. Coccinelle uses a language similar to TRANS that combines temporal logic and rewriting in order to express these semantic patches.

The project also contributes to the broader research area of temporal logic derived dataflow analysis. [10] describes the manner in which CTL is extended with Free Variables in order to provide a side condition language equivalent to the one used in

TRANS. An efficient bottom-up model checking algorithm is provided for this language.

7.2.4 SSA Based Verification

[53] takes the semantics of the L_0 language introduced by Lacey [44] and extends it with features that a realistic programming language intermediate representation would have—notably including the ϕ nodes necessary to expression a Single Static Assignment (SSA) Form. They also define a transformation into SSA form, using the Strategy combinators from the TRANS language, and prove it correct. Their notion of correctness is an extensional equivalence between two CFGs. They also show that their algorithm implies every variable is assigned to only once. Their work does not reason about the correctness of any program optimizations.

7.2.5 Other Program Transformation Systems

The APTS system of Paige [62] describes program transformations as rewrite rules, with side conditions expressed as boolean functions on the abstract syntax tree, and data obtained by program analyses. These analyses also have to be coded by hand. Other transformation systems that suffer the same drawback include Khepera [24] and TxI [15]. Datalog-like systems express program analyses as logic programs [18]. A more modern system is Stratego [85], which has sophisticated mechanisms for building transformers from a set of labelled, unconditional rewrite rules. The Ctadel system [82] is a program transformation system that has been used, amongst other things, to transform code used in the mathematical modelling of meteorological phenomena. This was achieved through generating Fortran code that was then compiled into the HIRLAM weather forecast system. Ctadel can generate efficient numerical code from higher level, mathematical, specifications, but is specific to the domain of Partial Differential

Equations and not general programs. The Vista system [91] is a system of interactive program transformation aimed at the optimization of programs for embedded systems. It has a fixed selection of hard coded optimizations but has a very advanced user interface for iteratively transforming, viewing and testing code.

7.2.6 Translation Validation

Other approaches to checking the soundness of optimizers include Translation Validation [67] and Credible Compilation [70]. Both these approaches check individual runs of a compiler, comparing the input and optimized versions of the program for semantic equivalence. This is affected by establishing certain checkable criteria that the compiler must meet after optimization and instrumenting the compiler to perform the checks. This can demonstrate that a program has been correctly compiled, but cannot check the correctness of the compiler itself. This offers considerable improvement over testing as an implementation debugging technique, but does not offer the user of the compiler that many safeguards. Since such an approach requires considerable instrumentation of the compiler in order to provide the checking of semantic equivalence it carries a computational burden at runtime.

7.2.7 Automated Bug Fixing

FindBugs is a system for detecting bugs within `Java` programs [33]. It introduces the concept of a bug pattern, which is a common construct within a program that commonly causes errors. Misunderstood API features, and difficult language features are good examples of bug patterns. Findbugs detects these patterns through static analysis, but does not attempt to fix them. Its bug detection mechanisms are hand written in Java.

UCDetector[1] is an Eclipse plugin, a commonly used Java IDE, plugin that finds unnecessary code within a project. Its detection mechanism is a custom dead code static analysis. It can also detect when the visibility of a method can be restricted, for example from public to private. It can automatically fix the dead code issues that it detects, but only performs limited analysis of the programs.

The Netbeans IDE will in future have a system for migrating users away from deprecated method calls, by automatically applying a source code transformation that rewrites a call to a deprecated method into a different method call. These transformations are specified in an annotation to the method definition. The transformation language is simplified, allowing constraints on argument types, and a few specialised metavariables for substitution, for example $0 represents the object that is calling the method.

Samanta et al. present an algorithm to generate automated repairs for programs with only boolean variables in [71]. Their specification for the program is a Hoare Triple where the pre condition and post condition of the program are annotated. Their algorithm generates a set of changes to the program that make the Hoare Triple true.

Their algorithm propagates the Hoare Triple throughout the program, so that each statement is annotated with pre and post conditions, and then suggests modifications for statements that ensure the pre and post conditions hold true. This iteratively back propagates the weakest pre-condition for a given post condition, and forward propagates the strongest post condition of a given pre condition.

Repairs are applied to individual statements, which are considered in some order. If the back propagation of a weakest pre-condition is aborted, then the successor statements to the statement at which it fails are considered for repair. If the forward

[1] http://www.ucdetector.org/

propagation of a strongest post-condition fails, then the predecessor statements to the failing statement are considered for repair. The algorithm queries possible statements for repair, and then attempts a local synthesis of a correct statement.

This approach requires Hoare Triple annotations for a given program. The algorithm's propagation only works for terminating programs, since generating the strongest postcondition of a non-terminating trace is impossible. It is limited to repairing bugs that can be specified by Hoare Triples over the entire program, and the specification of a suitable Hoare Triple is required by the algorithm. Their algorithm is of tractable complexity, and avoids considering all potential repairs in the program. Their algorithm is proved sound and complete for their repair model, which is a strong guarantee. Their repair model however, is limited to individual statement changes, for example if it was necessary to repair a series of sequential statements collectively in order to ensure that the Hoare Triple held true the repair algorithm would be unable to do so.

[29] describes an approach to automatically finding and fixing bugs in C programs. They create a *Boolean Program* from their C program, which is an automatically generated abstraction where variables can either be true or false. This abstraction is then model checked in order to identify bad states, these are states that violate some assertion that is being model checked. A repair replaces a statement, such that the bad state can never be reached.

Establishing the correct repair is achieved by formulating the problem as a game to be solved, in which the game is won if the protagonist can find a repairing statement. Their implementation uses Binary Decision Diagrams in order to symbolically represent the possible game states.

Since there may be multiple implementations of the repairing statement that solve the constraint the system provides all possible implementations to the user. The

programmer intervention is required because the statements provided are guaranteed to solve the fault, by removing reaching paths to the faulty state, however, they may not represent the intended semantics of the program and may potentially violate other assertions about the program.

Since the boolean program is a conservative abstraction of a C program, their repairs to the boolean program correspond to repairs to a C program. It is established that even though repairs to the boolean programs require no memory or stack resource usage, their corresponding C repairs do. Their application to C has been used to identify and fix bugs within a Windows Parallel Printer Port driver. The example bugs presented included ensuring that an incomplete connection is closed.

This system bears many similarities to that described in Chapter 5 — it presents potential fixes to the user and relies on a model checking based approach to identify bugs. It does not require the specification to tell their system how to transform the program in order to fix the bugs in question. However their notion of a repair is limited to one statement and to ensuring that fault states are not reached. The system also requires manual intervention in order to ensure that the bug fixes are appropriate.

7.3 Summary

Chapter 1 motivates the aims of the thesis. Chapter 2 described the temporal logic that this work is based upon, and past attempts at specification languages. Chapter 3 provides a description of the transformation language, clarified from David Lacey's work.

Chapter 4 described an implementation that generates compiler optimizations for a programming language that is commonly used. This system includes a novel intermediate representation suitable for compilers based on temporal logic and model checking derived dataflow analysis. Empirical analysis of the performance properties of

this implementation over an industrial strength benchmarking suite is described.

Chapter 5 introduced an idea for automatically fixing certain bugs within source code programs. In order to demonstrate its applicability a set of example bug fixing specifications are given. These specifications also provide an insight as to how to apply the technique to common bug fixing scenarios. The implementation of a bug fixing tool that allows automated application of patterns for fixing bugs allows these ideas to be validated by experiment. This implementation utilises a novel combination of source and bytecode analysis in order to blend the information most easily obtained from either means.

Chapter 6 considered several approaches to the formal analysis of compiler optimizations. The overarching goal being to prove that they do not introduce bugs into the programs that they are analysing in a mechanised theorem prover.

7.4 Final Remarks

This thesis offers an argument for the use of a domain specific languages for program transformation. It shows that, by compiling an optimization phase from a transformation specification and using BDDs, it is possible to approach hand written optimizations in terms of efficiency and effectiveness.

The thesis argues that program transformations can be used in order to fix bugs in programs. This use is achieved by specifying a pattern and side condition that identifies a bug, and using rewriting in order to fix the bug. This extends the use of Domain Specific Languages in this area more than has been done in the past.

that formal analysis of these transformations can be undertaken.

The work described in this thesis contributes to the state of the art of temporal logic based program transformations. By blending the theory and science of dataflow

analysis, model checking and theorem proving with the practise of program implementation and empirical analysis this thesis furthers the broader goal of advancing the understanding, and use, of program transformations.

Appendix A

Isabelle Source

```
theory Replacement
imports SmallStep LocalEquivalence
begin

fun replace :: "expr \<Rightarrow> expr \<Rightarrow>
   expr \<Rightarrow> expr"
where
"replace init from to = (if (from = init) then to else
   init)" |
"replace (Cast cls e) from to = (Cast cls (replace e from
    to))" |
"replace (l\<guillemotleft>bop\<guillemotright>r) from to
    = ((replace l from to)\<guillemotleft>bop\<
   guillemotright>(replace r from to))" |
"replace (V:=e) from to = (V := (replace e from to))" |
```

```
"replace (e\<bullet>V{C}) from to = ((replace e from to)
   \<bullet>V{C})" |
"replace (x\<bullet>V{C} := e) from to = ((replace x from
    to)\<bullet>V{C} := (replace e from to))" |
"replace (e\<bullet>meth'(args')) from to = ((replace e
   from to)\<bullet>meth'(args'))" |
"replace {a:t;e} from to = {a:t;(replace e from to)}" |
"replace (e1;;e2) from to = ((replace e1 from to);;(
   replace e2 from to))" |
"replace (Cond c y n) from to = (Cond (replace c from to)
    (replace y from to) (replace n from to))" |
"replace (while (c) b) from to = (while (replace c from
   to) (replace b from to))" |
"replace (throw e) from to = (throw (replace e from to))"
    |
"replace (TryCatch e c n ce) from to = (TryCatch (replace
    e from to) c n (replace ce from to))"

definition evals :: "J_prog => expr => expr => bool"
   where
"evals prog e1 e2 == ALL s. EX e' s'. prog \<turnstile>
   \<langle>e1, s\<rangle> \<rightarrow>* \<langle>e', s
   '\<rangle> --> prog \<turnstile> \<langle>e2, s\<
   rangle> \<rightarrow>* \<langle>e', s'\<rangle>"
```

```
lemma EvalsIdentity: "prog \<turnstile> \<langle>e1, s\<
   rangle> \<rightarrow>* \<langle>e', s'\<rangle> -->
   prog \<turnstile> \<langle>e2, s\<rangle> \<rightarrow
   >* \<langle>e', s'\<rangle>"

lemma SoundSub: "!!s. [|prog \<turnstile> \<langle>init,
   s\<rangle> \<rightarrow>* \<langle>e', s'\<rangle>;
   prog \<turnstile> \<langle>e, s\<rangle> \<rightarrow
   >* \<langle>e', s'\<rangle>|] ==> prog \<turnstile> \<
   langle> (if (from = init) then e else init), s\<rangle
   > \<rightarrow>* \<langle>e', s'\<rangle>" by simp

lemma SoundReplacement: "!! s. [|prog \<turnstile> \<
   langle>init, s\<rangle> \<rightarrow>* \<langle>e', s
   '\<rangle>;prog \<turnstile> \<langle>e, s\<rangle> \<
   rightarrow>* \<langle>e', s'\<rangle>|]
  ==> prog \<turnstile> \<langle> (replace init from e),
     s\<rangle> \<rightarrow>* \<langle>e', s'\<rangle>"
by simp

end
```

```
theory Predicates
imports SmallStep LocalEquivalence
begin

abbreviation
notdef :: "vname \<Rightarrow> J_prog \<Rightarrow> expr
   \<Rightarrow> heap \<Rightarrow> locals \<Rightarrow>
   expr \<Rightarrow> heap \<Rightarrow> locals \<
   Rightarrow> bool" where
"notdef var prog e h l e' h' l' \<equiv> prog \<turnstile
   > \<langle>e,(h,l)\<rangle> \<rightarrow> \<langle>e
   ',(h',l')\<rangle> & l var = l' var"

definition notuse :: "vname => J_prog => expr => expr =>
   heap=>locals => heap=>locals => heap=>locals => heap=>
   locals => bool" where
"notuse invar prog e e' hb lb hb' lb' ha la ha' la' ==
   (((res lb invar) = (res la invar) --> (res lb' invar)
   = (res la' invar)) & (hb = ha --> hb' = ha'))"

end
```

```
theory TransEquivalence
imports SmallStep BigStep
begin

definition Equiv :: "J_prog => expr => expr => bool"
   where
"Equiv prog e1 e2 == ALL s. EX e' s'. (final e')
  & prog \<turnstile> \<langle>e1, s\<rangle> \<
     rightarrow>* \<langle>e', s'\<rangle>
  & prog \<turnstile> \<langle>e2, s\<rangle> \<
     rightarrow>* \<langle>e', s'\<rangle>"

definition Sound :: "(J_prog => expr => expr) => bool"
   where
"Sound f == ALL prog x y. (y = f prog x) --> Equiv prog x
    y"

end
```

```
theory LocalEquivalence
imports SmallStep
begin

definition equivSt :: "J_prog => expr => state => expr =>
    state => bool" where
"equivSt prog e1 s1 e2 s2 ==
  EX e' s'. prog \<turnstile> \<langle>e1, s1\<rangle> \<
     rightarrow>* \<langle>e', s'\<rangle> = prog \<
     turnstile> \<langle>e2, s2\<rangle> \<rightarrow>*
     \<langle>e', s'\<rangle>"

lemma EquivalentIfReduceable: "prog \<turnstile> \<langle
   >e,(h,l)\<rangle> \<rightarrow>* \<langle>e',(h',l')\<
   rangle> ==>
  equivSt prog e (h,l) e' (h',l')"
  by (auto simp: equivSt_def)

lemma CommutativityOfEquivalence: "equivSt prog e s e' s'
    = equivSt prog e' s' e s" (is "?A = ?B")
proof
  assume ?A thus ?B by (auto simp: equivSt_def)
next
  assume ?B thus ?A by (auto simp: equivSt_def)
qed
```

```
definition mutation :: "expr => J_prog => state => expr
   => (state => state) => bool" where
"mutation e prog s e' f = prog \<turnstile> \<langle>e, s
   \<rangle> \<rightarrow>* \<langle>e', f s\<rangle>"

definition res :: "('a ~=> 'b) => 'a => 'a ~=> 'b" where
"res f v x = (if (x=v) then None else (f x))"

lemma Res_Split_Simp: "ALL x. (f d = g d ) & res f d x =
   res g d x --> f x = g x"
  apply (simp only: res_def)
  apply (split split_if)
  apply (simp)
done

lemma "(res f d) d = None" by (simp add: res_def)
lemma "(d1 ~= d2) ==> ((f (d1|->r)) d2) = (f d2)" by simp
lemma Res_Map: "res (f(d\<mapsto>r)) d = res f d" by (
   simp add: res_def ext)

lemma Res_Split: "(f d = g d) & res f d = res g d --> f =
    g"
  apply (clarsimp simp add:expand_fun_eq res_def)
  apply (case_tac "x=d")
```

```
  by simp_all

end
```

```
theory ConstantPropagation
imports SmallStep Predicates
begin

abbreviation init :: "vname \<Rightarrow> val \<
   Rightarrow> J_prog \<Rightarrow> expr \<Rightarrow>
   state \<Rightarrow> expr \<Rightarrow> state \<
   Rightarrow> bool" where
"init sub_var const prog e s e' s' \<equiv> (e = sub_var
   := (Val const)) & prog \<turnstile> \<langle>e,s\<
   rangle> \<rightarrow> \<langle>e',s'\<rangle>"

lemma SideConditionInit: "init sub_var const prog e (h,l)
    e' (h', l') ==>
  l' sub_var = (Some const)"
proof -
  assume "init sub_var const prog e (h,l) e' (h',l')"
  hence NewState: "l' = l(sub_var\<mapsto>const)" by auto
  show " l' sub_var = (Some const)" by (simp add:
     NewState)
qed

fun side_cond :: "vname \<Rightarrow> val \<Rightarrow> J
   _prog \<Rightarrow> expr \<Rightarrow> state \<
   Rightarrow>  (expr * state) list \<Rightarrow> bool"
```

```
   where
"side_cond sub_var const prog e s [(e',h',l')] = init sub
   _var const prog e s e' (h',l')" |
"side_cond sub_var const prog pe ps ((e,h,l)#(e',h',l')#
   es) =
  (notdef sub_var prog e h l e' h' l' & side_cond sub_var
      const prog pe ps ((e',(h',l'))#es))"

lemma SideConditionPropagation: "[| side_cond sub_var
   const prog e s es; es \<noteq> [] ;
  (e,h,l) = (last es) |] ==> l sub_var = (Some const)"
  by (induct sub_var const prog e s es rule: side_cond.
     induct, auto)

lemma EvalPre1: "lcl s sub_var = Some const ==>
  (prog \<turnstile> \<langle>ass_var:=(Var sub_var), s\<
     rangle> \<rightarrow> \<langle>ass_var:=(Val const),
      s\<rangle>)" by (simp only: LAssRed RedVar)

lemma Into_rtc2: "[| (x,y) \<in> r ; (y,z) \<in> r |] ==>
    (x,z) \<in> r\<^sup>*"
  by (simp only: rtrancl_trans)

lemma "[|lcl s sub_var = Some const ; s = (h,l)|] ==>
  (P \<turnstile> \<langle>ass_var:=(Var sub_var), s\<
```

```
     rangle> \<rightarrow>* \<langle>unit, (h,l(ass_var\<
     mapsto>const))\<rangle>)"
proof -
  assume "lcl s sub_var = Some const"
  assume A: "s = (h,l)"
  show "P \<turnstile> \<langle>ass_var:=(Var sub_var), s
     \<rangle> \<rightarrow>* \<langle>unit, (h,l(ass_var
     \<mapsto>const))\<rangle>"
    apply (rule rtrancl_trans)
     apply (rule r_into_rtrancl)
     apply (rule EvalPre1)
     apply(assumption)
    by (simp add: r_into_rtrancl A RedLAss)
qed

end
```

```
theory LoopInvariantCodeMotion
imports SmallStep Predicates LocalEquivalence
begin

definition mutate :: "vname => val => state => state"
   where
"mutate invar v s = (let (h,l) = s in (h,l(invar\<mapsto>
   v)))"

lemma  SoundStatementRemoval: "[|(h,l) = s; prog \<
   turnstile> \<langle>exp,s\<rangle> \<rightarrow> \<
   langle>Val v,s\<rangle>;
  (s' = mutate invar v s)|] ==> (equivSt prog (invar:=exp
     ) s unit s')"
   apply (auto simp: mutate_def)
   apply (rule EquivalentIfReduceable)
  apply (rule rtrancl_trans)
  apply (rule r_into_rtrancl)
  by (auto elim: LAssRed simp: RedLAss)

definition mutation :: "expr => J_prog => state => expr
   => (state => state) => bool" where
"mutation e prog s e' f = prog \<turnstile> \<langle>e, s
   \<rangle> \<rightarrow>* \<langle>e', f s\<rangle>"
```

```
definition ambivalent :: "expr => J_prog => state =>
   state => bool" where
"ambivalent e prog s1 s2 == EX f. ALL e'. mutation e prog
    s1 e' f & mutation e prog s2 e' f"

abbreviation trans :: "expr => J_prog => expr => state =>
    expr => state => bool" where
"trans transp prog e s e' s' == (prog \<turnstile> \<
   langle>e, s\<rangle> \<rightarrow> \<langle>e', s'\<
   rangle>) --> (ambivalent transp prog s s')"

lemma "prog \<turnstile> \<langle>exp,(h, l)\<rangle> \<
   rightarrow> \<langle>Val v,(h, l)\<rangle> ==>
  mutation (invar:=exp) prog (h,l) unit (mutate invar v)"
  apply (simp add: mutation_def)
  apply (rule rtrancl_trans)
  apply (rule r_into_rtrancl)
  by (auto elim: LAssRed simp: RedLAss mutate_def Let_def
     )

lemma InvarEquiv: "[|notdef invar p ea ha la ea' ha' la';
    la invar = v|] ==> la' invar = v" by simp

lemma HeapEquiv: "notuse invar prog e e' hb lb hb' lb' ha
    la ha' la' & hb = ha ==> hb' = ha'" by (simp add:
```

```
   notuse_def)

lemma NonInvarLocalEquiv: "[|notuse invar prog e e' hb lb
    hb' lb' ha la ha' la' ;  la = lb(invar|->v)|] ==> (
   res lb' invar) = (res la' invar)"
  apply (auto simp: notuse_def)
  apply (erule conjE)
  by (auto)

lemma ResEquiv: assumes a: "notuse invar prog e e' hb lb
   hb' lb' ha la ha' la' & hb = ha" shows "hb' = ha'"
proof -
  assume "" have "(hb = ha --> hb' = ha') &  (hb = ha)"
    apply (unfold notuse_def)
    apply auto
  by auto

lemma "[|p \<turnstile> \<langle>eb,(hb,lb)\<rangle> \<
   rightarrow> \<langle>eb',(hb',lb')\<rangle> ; p \<
   turnstile> \<langle>eb,(ha,la)\<rangle> \<rightarrow>
   \<langle>eb',(ha',la')\<rangle>; notdef invar p ea ha
   la ea' ha' la'; notuse invar p ea ha la ea' ha' la'; (
   ha,la) = mutate invar v (hb,lb)|] ==> (ha',la') =
   mutate invar v (hb',lb')"
  apply (simp add: mutate_def)
```

```
  apply auto
  apply (auto simp: mutate_def)
  by auto

fun sc :: "J_prog => vname => expr => val => (expr*state*
   expr*state) list => bool" where
"sc p invar exp v [(e,s,e',s')] = (s' = mutate invar v s
   & e = e')" |
"sc p invar exp v ((eb,(hb,lb),ea,(ha,la))#(eb',(hb',lb')
   ,ea',(ha',la'))#ess) =
     ((notdef invar p eb hb lb eb' hb' lb')  &
      (notuse invar p eb eb' hb lb hb' lb' ha la ha' la')
           &
      trans exp p eb (hb,lb) eb' (hb',lb') &
      p \<turnstile> \<langle>eb,(hb,lb)\<rangle> \<
         rightarrow> \<langle>eb',(hb',lb')\<rangle> &
      p \<turnstile> \<langle>ea,(ha,la)\<rangle> \<
         rightarrow> \<langle>eb',(ha',la')\<rangle>&
      sc p invar exp v ((eb',(hb',lb'),ea',(ha',la'))#ess
         ))"

lemma trans_def: "trans expr prog e s e' s' & prog \<
   turnstile> \<langle>expr,s\<rangle> \<rightarrow> \<
   langle>expr',s_expr'\<rangle> & mutation expr prog s
   expr' f ==>
```

```
prog \<turnstile> \<langle>expr,s'\<rangle> \<rightarrow>
    \<langle>expr',f s'\<rangle>"
    apply (auto simp: mutation_def)
    by auto

lemma LICMSideCondition: "[|sc p invar exp v es;(eaf,saf,
   ebf,sbf) = last es;es \<noteq> []|] ==> (saf = mutate
   invar v sbf)"
   apply (induct p invar exp v es rule: sc.induct)
   apply (auto simp: mutate_def)
   by (auto simp: trans_def)

end
```

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools*, 2nd ed. Pearson Education, 2007.

[2] Assmann, U. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Compiler Construction 1996* (1996), P. Fritzson, Ed., vol. 1060 of *Lecture Notes in Computer Science*, Springer.

[3] Assmann, U. OPTIMIX, A Tool for Rewriting and Optimizing Programs. In *Graph Grammar Handbook, Vol. II*. Chapman-Hall, 1999.

[4] Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. Mechanized metatheory for the masses: The poplmark challenge. Springer-Verlag, pp. 50–65.

[5] Bacon, D., Bloch, J., Bogda, J., Click, C., Haahr, P., Lea, D., May, T., Maessen, J.-W., Manson, J., Mitchell, J. D., Nilsen, K., Pugh, B., and Sirer, E. G. The "double-checked locking is broken" declaration, 2010.

[6] Berndl, M., Lhoták, O., Qian, F., Hendren, L., and Umanee, N. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Con-*

ference on Programming Language Design and Implementation (2003), ACM Press, pp. 103–114.

[7] Bollig, B., and Wegener, I. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Comput. 45*, 9 (1996), 993–1002.

[8] Boyle, J. M. A transformational component for programming languages grammar. Technical Report ANL-7690, Argonne National Laboratory, IL, 1970.

[9] Boyle, J. M. Abstract programming and program transformation. In *Software Reusability Volume 1* (1989), Addison-Wesley, pp. 361–413.

[10] Brunel, J., Doligez, D., Hansen, R. R., Lawall, J. L., and Muller, G. A foundation for flow-based program matching: using temporal logic and model checking. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2009), ACM, pp. 114–126.

[11] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput. 35*, 8 (1986), 677–691.

[12] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop* (London, UK, 1982), Springer-Verlag, pp. 52–71.

[13] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8* (1986), 244–263.

[14] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems 8* (1996), 244–263.

[15] Cordy, J. R., Carmichael, I. H., and Halliday, R. The TXL programming language, version 8. Legasys Corporation, April 1995.

[16] Cornes, C., Courant, J., Fillitre, J.-C., Huet, G., Murthy, C., Muoz, C., Parent, C., Werner, A. S. B., Symbolique, C., Manoury, P., Manoury, P., Saibi, A., Werner, B., and Coq, P. The coq proof assistant - reference manual v 5.10, 1995.

[17] Curzon, P. A structured approach to the verification of low level microcode. Tech. Rep. 215, University of Cambridge, Computer Laboratory, Feb. 1991. PhD Thesis.

[18] Dawson, S., Ramakrishnan, C. R., and Warren, D. S. Practical program analysis using general purpose logic programming systems — A case study. *ACM SIGPLAN Notices 31*, 5 (May 1996), 117–126.

[19] de Moor, O., and Sittampalam, G. Higher-order matching for program transformation. *Theoretical Computer Science 269*, 1-2 (October 2001), 135–162.

[20] Eclipse Foundation. Eclipse website, 2009. `http://www.eclipse.org`.

[21] 'Eric Bruneton, Lenglet, R., and Coupaye, T. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journ'ees Composants 2002 : Syst'emes 'a composants adaptables et extensibles (Adaptable and extensible component systems)* (2002).

[22] Ernst, M. D. Type Annotations Specification (JSR 308). http://types.cs.washington.edu/jsr308/, October 5, 2009.

[23] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming 69*, 1–3 (Dec. 2007), 35–45.

[24] Faith, R. E., Nyland, L. S., and Prins, J. F. KHEPERA: A system for rapid implementation of domain-specific languages. In *Proceedings USENIX Conference on Domain-Specific Languages* (1997), pp. 243–255.

[25] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1995.

[26] Gerwin Klein, Tobias Nipkow, L. P. The archive of formal proofs.

[27] Gordon, M., and Melham, T. Introduction to hol: a theorem proving environment for higher order logic. *Journal of Functional Programming 4* (1993), 557–559.

[28] Gordon, M., and Wadsworth. *LCF: A Mechanised Logic of Computation, volume 78 of Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[29] Griesmayer, A., Bloem, R., and Cook, B. Repair of boolean programs with an application to c. In *In CAV 2006* (2006), pp. 358–371.

[30] Harrison, J. HOL Light: An overview. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009* (Munich, Germany, 2009), S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., vol. 5674 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 60–66.

[31] Heckmann, R. A functional language for the specification of complex tree transformations. In *ESOP '88* (1988), Lecture Notes in Computer Science, Springer-Verlag, pp. 175–190.

[32] Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12*, 10 (1969), 576–580.

[33] Hovemeyer, D., and Pugh, W. Finding bugs is easy. *ACM SIGPLAN Notices 39*, 12 (2004), 92–106.

[34] Huth, M., and Ryan, M. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, New York, NY, USA, 2004.

[35] Jones, S. P., Tolmach, A., and Hoare, T. Playing by the rules: Rewriting as a practical optimisation technique in ghc. In *Proceedings of the 2001 Haskell Workshop* (September 2001), pp. 203–233.

[36] Kalvala, S., Warburton, R., and Lacey, D. Program transformations using temporal logic side conditions. Tech. Rep. 439, Department of Computer Science, University of Warwick, 2008.

[37] Kanade, A., Sanyal, A., and Khedker, U. A PVS based framework for validating compiler optimizations. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 108–117.

[38] Kanade, A., Sanyal, A., and Khedker, U. P. Structuring optimizing transformations and proving them sound. *Electr. Notes Theor. Comput. Sci. 176*, 3 (2007), 79–95.

[39] Klein, G., and Nipkow, T. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Program. Lang. Syst. 28*, 4 (2006), 619–695.

[40] Klein, M., Knoop, D., Koschutzki, D., and Steffen, B. DFA & OPT-METAFrame: A toolkit for program analysis and optimization. In *Procs. of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96)* (1996), vol. 1055 of *Lecture Notes in Computer Science*, Springer, pp. 422–426.

[41] Knoop, J., Rüthing, O., and Steffen, B. Lazy code motion. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)* (1992), pp. 224–234.

[42] Knoop, J., Rüthing, O., and Steffen, B. Lazy strength reduction. *Journal of Programming Languages 1*, 1 (1993), 71–91.

[43] Kozen, D. Results on the proposition mu-calculus. *Theoretical Computer Science 27* (1983).

[44] Lacey, D. *Program Transformation using Temporal Logic Specifications*. PhD thesis, Oxford University Computing Laboratory, 2003.

[45] Lacey, D., and de Moor, O. Imperative program transformation by rewriting. In *Compiler Construction* (2001), R. Wilhelm, Ed., vol. 2027 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 52–68.

[46] Lacey, D., Jones, N., Wyk, E. V., and Frederikson, C. C. Proving the correctness of classical compiler optimisation by temporal logic. In *Priniciples of Programming Languages* (2002).

[47] Lacey, D., Jones, N., Wyk, E. V., and Frederikson, C. C. Proving correctness of compiler optimizations by temporal logic. *Higher-Order and Symbolic Computation 17*, 2 (2004).

[48] Lerner, S., Millstein, T., and Chambers, C. Automatically proving the correctness of compiler optimizations. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (2003), ACM Press. citeseer.ist.psu.edu/lerner03automatically.html.

[49] Lerner, S., Millstein, T., Rice, E., and Chambers, C. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2005), ACM Press, pp. 364–377.

[50] Lhoták, O., and Hendren, L. Jedd: A BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (2004), ACM Press.

[51] Lhoták, O., and Hendren, L. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction, 15th International Conference* (Vienna, Mar. 2006), A. Mycroft and A. Zeller, Eds., vol. 3923 of *LNCS*, Springer, pp. 47–64.

[52] Lipps, P., Mönke, U., and Wilhelm, R. OPTRAN – a language/system for the specification of program transformations: system overview and experiences. In *Proceedings 2nd Workshop on Compiler Compilers and High Speed Compilation* (1988), vol. 371 of *Lecture Notes in Computer Science*, pp. 52–65.

[53] Mansky, W., and Gunter, E. A framework for formal verification of compiler optimizations. In *ITP 2010: Proceedings of Interactive Theorem Proving 2010* (2010), Springer-Verlag, p. 16.

[54] Milner, R., Tofte, M., and Macqueen, D. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[55] Muchnick, S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[56] Müller-Olm, M., and Rüthing, O. On the complexity of constant propagation. In *ESOP* (2001), D. Sands, Ed., vol. 2028 of *LNCS*, Springer-Verlag.

[57] Nielson, F., and Nielson, H. R. Interprocedural control flow analysis. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems* (London, UK, 1999), Springer-Verlag, pp. 20–39.

[58] Nipkow, T. Jinja: Towards a comprehensive formal semantics for a Java-like language. In *Proof Technology and Computation* (2006), H. Schwichtenberg and K. Spies, Eds., IOS Press, pp. 247–277.

[59] Norrish, M. *Formalising C in HOL*. PhD thesis, Computer Lab., University of Cambridge, December 1998.

[60] Padioleau, Y., Hansen, R. R., Lawall, J. L., and Muller, G. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *PLOS '06: Proceedings of the 3rd workshop on Programming languages and operating systems* (New York, NY, USA, 2006), ACM, p. 10.

[61] Padua, D. A., and Wolfe, M. J. Advanced compiler optimizations for supercomputers. *Communications of the ACM 29*, 12 (1986), 1184–1201.

[62] Paige, R. Viewing a program transformation system at work. In *Proceedings Programming Language Implementation and Logic Programming (PLILP), and Algebraic and Logic Programming (ALP)* (1994), vol. 844 of *Lecture Notes in Computer Science*, Springer, pp. 5–24.

[63] Park, D. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science* (London, UK, 1981), Springer-Verlag, pp. 167–183.

[64] Paulson, L. C. The foundation of a generic theorem prover. *Journal of Automated Reasoning 5* (1989).

[65] Paulson, L. C. Isabelle: The next 700 theorem provers. *CoRR cs.LO/9301106* (1993).

[66] Plotkin, G. D. A structural approach to operational semantics. *J. Log. Algebr. Program. 60-61* (2004), 17–139.

[67] Pnueli, A., and Zaks, G. Translation validation of interprocedural optimizations. In *Proceedings of the 4th International Workshop on Software Verification and Validation (SVV 2006)* (Aug. 2006), Computing Research Repository (CoRR).

[68] Pozo, R., and Miller, B. Java Scimark 2.0. National Institute of Standard and Technology, 2004. Available at `http://math.nist.gov/scimark2/`.

[69] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. *Logic in Computer Science, Symposium on 0* (2002), 55.

[70] Rinard, M. Credible compilers. Tech. Rep. MIT/LCS/TR-776, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999.

[71] Samanta, R., Deshmukh, J. V., and Emerson, E. A. Automatic generation of local repairs for boolean programs. In *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design* (Piscataway, NJ, USA, 2008), IEEE Press, pp. 1–10.

[72] Scherpelz, E. R., Lerner, S., and Chambers, C. Automatic inference of optimizer flow functions from semantic meanings. In *PLDI* (2007), pp. 135–145.

[73] Schmidt, D., and Steffen, B. Data-flow analysis as model checking of abstract interpretations. In *5th Static Analysis Symposium* (September 1998), G. Levi, Ed., vol. 1503 of *LNCS*.

[74] Schmidt, D. C., and Harrison, T. Double-checked locking - an optimization pattern for efficiently initializing and accessing thread-safe objects, 1997.

[75] Steffen, B. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming 21* (1993), 115–139.

[76] Stoy, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[77] Stroustrup, B. *The C++ Programming Language: Special Edition*, 3 ed. Addison-Wesley Professional, February 2000.

[78] Tarjan, R. Testing flow graph reducibility. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing* (New York, NY, USA, 1973), ACM, pp. 96–107.

[79] Thakur, A., and Govindarajan, R. Comprehensive path-sensitive data-flow analysis. In *CGO '08: Proceedings of the 6th annual IEEE/ACM international*

symposium on Code generation and optimization (New York, NY, USA, 2008), ACM, pp. 55–63.

[80] Vallée-Rai, R., Gagnon, E., Hendren, L. J., Lam, P., Pominville, P., and Sundaresan, V. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)* (2000), pp. 18–34.

[81] Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P. Soot - a Java optimization framework. In *Proceedings of CASCON 1999* (1999), pp. 125–135.

[82] van Engelen, R. *Ctadel: A Generator of Efficient Codes*. PhD thesis, Leiden University, 1998.

[83] van Engelen, R. A. Efficient symbolic analysis for optimizing compilers. In *Compiler Construction* (2001), R. Wilhelm, Ed., vol. 2027 of *Lecture Notes in Computer Science*, Springer Verlag.

[84] Vardi, M. Y., and Wolper, P. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science* (1986), pp. 322–311.

[85] Visser, E., Benaissa, Z., and Tolmach, A. Building program optimizers with rewriting strategies. In *International Conference on Functional Programming '98* (1998), ACM SigPlan, ACM Press, pp. 13–26.

[86] Wegman, M. N., and Zadeck, F. K. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS) 13*, 2 (April 1991), 181–210.

[87] Wenzel, M., and Mnchen, T. Using axiomatic type classes in isabelle, 2000.

[88] Whitfield, D. L., and Soffa, M. L. The design and implementation of genesis. *Software — Practice and Experience 24*, 3 (1994), 307–325.

[89] Whitfield, D. L., and Soffa, M. L. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems 19*, 6 (1997), 1053–1084.

[90] Wildmoser, M., and Nipkow, T. Certifying machine code safety: Shallow versus deep embedding. In *Theorem Proving in Higher Order Logics (TPHOLs 2004)* (2004), K. Slind, A. Bunker, and G. Gopalakrishnan, Eds., vol. 3223, Springer-Verlag, pp. 305–320.

[91] Zhao, W., Cai, B., Whalley, D., W.Bailey, M., Engelen, R. V., Yuan, X., Hiser, J. D., Davidson, J. W., Gallivan, K., and Jones, D. L. Vista: A system for interactive code improvement. In *LCTES'02-SCOPES'02* (June 2002), ACM.