UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

Automatic Synthesis of Weakest Preconditions for Compiler Optimizations

Nuno Claudino Pereira Lopes

Supervisor: Doctor José Carlos Alves Pereira Monteiro

Thesis approved in public session to obtain the PhD degree in Information Systems and Computer Engineering

Jury final classification: Pass with Merit

Jury

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor Luís Manuel Marques da Costa Caires
Doctor Vasco Manuel Thudichum de Serpa Vasconcelos
Doctor José Carlos Alves Pereira Monteiro
Doctor Greta Yorsh
Doctor David Manuel Martins de Matos
Doctor Alexandre Paulo Lourenço Francisco

2014

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Automatic Synthesis of Weakest Preconditions for Compiler Optimizations

Nuno Claudino Pereira Lopes

Supervisor: Doctor José Carlos Alves Pereira Monteiro

Thesis approved in public session to obtain the PhD degree in Information Systems and Computer Engineering

Jury final classification: Pass with Merit

Jury

Chairperson: Chairman of the IST Scientific Board
Members of the Committee:

Doctor Luís Manuel Marques da Costa Caires, Professor Catedrático, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
Doctor Vasco Manuel Thudichum de Serpa Vasconcelos, Professor Catedrático, Faculdade de Ciências, Universidade de Lisboa
Doctor José Carlos Alves Pereira Monteiro, Professor Associado com Agregação, Instituto Superior Técnico, Universidade de Lisboa
Doctor Greta Yorsh, Lecturer, School of Electronic Engineering and Computer Science, Queen Mary University of London, UK
Doctor David Manuel Martins de Matos, Professor Auxiliar, Instituto Superior Técnico, Universidade de Lisboa
Doctor Alexandre Paulo Lourenço Francisco, Professor Auxiliar, Instituto Superior Técnico, Universidade de Lisboa

Funding Institutions

Fundação para a Ciência e a Tecnologia

2014

Resumo

As optimizações de compiladores são cada vez mais importantes e complexas. Os programadores dependem delas para melhorar o desempenho, reduzir o tamanho do código e reduzir o consumo de energia dos seus programas. A generalização do uso de equipamentos móveis com recursos limitados e a necessidade de diminuir os custos operacionais dos centros de dados coloca ainda uma maior pressão na qualidade dos resultados das optimizações dos compiladores. Assim, as equipas de desenvolvimento de compiladores têm sido forçadas a desenvolver e a disponibilizar rapidamente novas optimizações cada vez mais complexas, o que compromete a sua correcção.

As optimizações de compiladores ainda são desenhadas e implementadas manualmente e tipicamente sem garantir a sua correcção formal. Por outro lado, além de ter que desenhar a transformação de código, o programador tem ainda que desenvolver uma análise que determina em que casos é que a optimização pode ser aplicada. Por outras palavras, quem desenvolve uma optimização tem que especificar a pré-condição que garante que a optimização preserva a semântica do código, bem como implementar um algoritmo que verifica se um dado fragmento de código satisfaz a pré-condição.

Determinar pré-condições para optimizações manualmente é uma tarefa não trivial. É fácil esquecer-se de um caso, tornando assim a optimização incorrecta. É igualmente fácil especificar uma pré-condição que, embora correcta, é demasiado restritiva e que, portanto, faz perder algumas oportunidades para optimização.

Nesta tese proponho um algoritmo para a síntese automática das pré-condições mais fracas para optimizações de compiladores. Trata-se do primeiro algoritmo conhecido especialmente concebido para esta tarefa. O funcionamento do algoritmo é guiado por contra-exemplos e é completo, desde que a ferramenta de verificação utilizada também o seja.

Proponho também um novo algoritmo para provar equivalência de programas, com aplicação na verificação de optimizações de compiladores. Este algoritmo consegue verificar automaticamente a correcção de mais optimizações que qualquer uma das técnicas propostas no passado. O algoritmo primeiro transforma as optimizações em programas numéricos e depois gera um sumário preciso do seu funcionamento.

Abstract

Compiler optimizations are increasingly important and complex. Developers rely on them to improve performance, reduce code size, and reduce power consumption of their programs. The advent of mobile devices with limited resources and the need to minimize the operating costs of data centers puts even more pressure on the quality of the results of compiler optimizations. Therefore, compiler developers are being forced to devise and quickly deploy new and more complex optimizations, which compromises correctness.

Compiler optimizations are still designed and implemented by hand, and usually without providing any formal guarantee of correctness. Moreover, in addition to devising the code transformations, developers have to come up with an analysis that determines in which cases the optimization can be applied. In other words, the optimization designer has to specify a precondition that ensures that the optimization is semantics-preserving, as well as implement an algorithm to check if a given code fragment fulfills the precondition.

Devising preconditions for optimizations by hand is a non-trivial task. It is easy to miss a corner case, making the optimization unsound. It is also easy to specify a precondition that, although correct, is too restrictive, and therefore misses some optimization opportunities.

In this thesis, I propose an algorithm for the automatic synthesis of weakest preconditions for compiler optimizations. It is, to the best of my knowledge, the first known algorithm for this task. The algorithm works in a counterexample-driven way and is complete, modulo the employed verification tool.

I also propose a new algorithm to prove program equivalence, with an application to proving correctness of compiler optimizations. This algorithm is able to automatically verify more optimizations than any previously proposed technique. The algorithm works by transforming optimizations into numeric programs and then computing a precise summary of their behavior.

Palavras-Chave
Keywords

Palavras-Chave

Compiladores
Optimizações de compiladores
Funções de transformação
Transformação de código
Verificação de software
Síntese de pré-condições mais fracas
Equivalência de programas
Recorrências
Interpolação polinomial
Resolução de restrições

Keywords

Compilers
Compiler optimizations
Transformation functions
Code transformation
Software verification
Weakest precondition synthesis
Program equivalence
Recurrences
Polynomial interpolation
Constraint solving

Acknowledgments

The work presented in this document wouldn't have been possible without the help, collaboration, guidance, and friendship of many people.

First and foremost, I would like to thank my adviser, José Monteiro. He was always very supportive and encouraging during these five years, whether I was throwing crazy ideas or about to give up.

Secondly, I would like to thank all the jury members for their time, as well as for the insightful comments given.

I also would like to thank to (in alphabetical order): João Pedro Afonso, Carlos Caleiro, Francisco Miguel Dionísio, Henrique Amaral Fernandes, Ruslán Ledesma-Garza, Ken McMillan, João Pimentel Nunes, Andrey Rybalchenko, and Philippe Suter. They have all contributed to this work in the form of insightful discussions, reviews of previous drafts, etc. I'm indebted to them.

I would like to thank the LLVM community. They embraced me as a compiler developer and shared a lot of their knowledge in developing and maintaining production-quality compilers. I'm particularly grateful to Evan Cheng and Chris Lattner for hosting me as an intern at Apple's compiler team. That was a great experience, from which I learned a lot. Many of the insights I learned there were very helpful for better shaping this work towards having real applicability in a not-so-distant future.

I also would like to thank Nikolaj Bjørner and George Varghese for hosting me as an intern at Microsoft Research. That was a wonderful experience, where I had the chance to learn more about computer networks, software verification, and Z3.

This work was financially supported in part by FCT (Fundação para a Ciência e a Tecnologia) under grant SFRH/BD/63609/2009, INESC-ID, and FLAD (Fundação Luso-Americana para o Desenvolvimento). I'm very grateful and indebted for their support.

Finally, a big thank you to my family and friends for their support throughout these five years. A big thank you also to my conference buddies, for making the experience of attending conferences so enjoyable and productive, technically and socially.

Contents

List of Figures

List of Tables

List of Abbreviations

ASM	Assembly
BMC	Bounded model checker
BNF	Backus-Naur form
CEGAR	Counterexample-guided abstraction refinement
DNF	Disjunctive normal form
DSL	Domain specific language
IA	Integer arithmetic
IR	Intermediate representation
ISA	Instruction set architecture
LIA	Linear integer arithmetic
SIMD	Single instruction, multiple data
SMT	Satisfiability modulo theories
UF	Uninterpreted function
VC Gen	Verification condition generator
WLP	Weakest liberal precondition
WP	Weakest precondition

Chapter 1

Introduction

Compilers are widely used for software development since they allow programmers to describe programs at increasingly higher levels of abstraction, while achieving as good or better performance than with low-level languages. Additionally, high-level languages are known to enable more succinct implementations of programs, which in turn leads to shorter development cycles.

Current compilers are usually divided in three major components (Figure 1.1): the front-end, which is responsible for parsing the input program and transforming it into an efficient representation for later passes; the middle-end, that is responsible for performing high-level code optimizations; and the back-end, which performs low-level machine dependent code optimizations and emits the final assembly (ASM) or binary code.

Dividing compilers in three components is beneficial for several reasons. First, by decoupling the front-end and the back-end from the middle part and from each other allows a single compiler to process multiple source-code languages and to generate code for multiple instruction sets architectures (ISAs). This decoupling enables a significant reduction in implementation effort, since supporting n source-code languages and m ISAs in a compiler requires effort proportional to $n+m$, instead of $n \times m$ if no decoupling was done.

Second, transforming the source-code into an intermediate representation (IR) allows the compiler developers to choose the most efficient internal representation for the class of transformations and analyses that are supported by the compiler. Moreover, a decoupling of the source-code language(s) and the IR enables the compiler developers to evolve the languages separately as needed.

Third, the decoupling of the back-end from the rest of the compiler, allows a separation of concerns between ISA (mostly) independent and ISA specific code. Therefore, compiler developers working on the back-end do not need to be knowledgeable in the other parts of the compiler, and the remaining developers do not need to be experts in any particular ISA. The same is true for front-end developers, who are the experts in source-code language semantics, but do not necessarily need in-depth knowledge about specific ISAs.

Recently, there has been a trend towards the development of standard IRs, such as SSA (static single assignment form [CFR+91]) for compilers targeting imperative languages, and CPS (continuation-passing style [Plo75, App92]) for compilers targeting functional languages. These IRs have been adopted by most major compilers. The standardization is important since it enables efficient algorithms that leverage the specifics of the IR to be studied independently of the compiler, which can then be shared across compilers.

The largest and most complex part of a compiler is usually the middle-end optimizer. This optimizer is responsible for performing high-level code optimizations, such as removal or insertion of loops, change of memory access patterns, modification of the control-flow graph, merging or splitting functions, vectorization and/or parallelization of portions of the code, and so on. Less commonly, some optimizers can

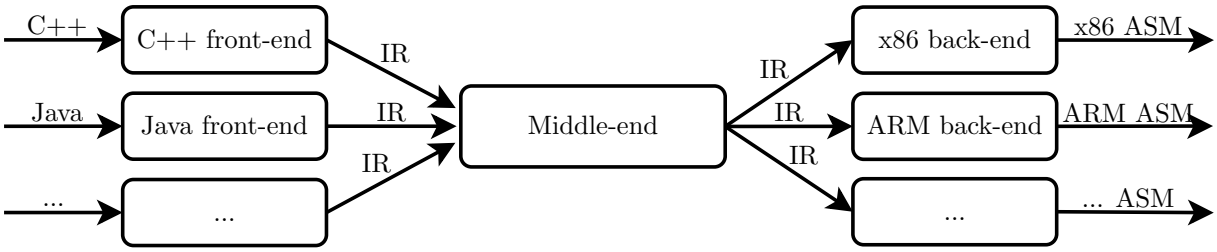

Figure 1.1: Traditional compiler architecture with three components.

also change the layout of data-structures in memory or even commute data-structures altogether to more efficient versions.

Compiler optimizations can speedup a program by orders of magnitude, considerably reduce its size, and/or reduce power consumption (with results heavily dependent on the type of program). For example, LLVM 3.2, released in December of 2012, introduced a new optimization to perform automatic loop vectorization for SIMD CPU architectures. The performance improvement in benchmarks was in the range of 10–300%. [1]

1.1 Motivation

Nowadays, the advent of mobile devices with limited resources and the need to reduce the operating costs of data centers puts a significant pressure on the quality of the results of compiler optimizations. Therefore, compiler developers are being forced to devise and quickly deploy new and more complex optimizations, potentially compromising correctness.

Currently available compilers, although mature (with many being in development for over 20 years), still exhibit many bugs as recent studies show [ER08, YCER11, WHTY13, LAS14]. The bugs found by these studies range from non-critical compiler crashes to miscompilations (i.e., the compiler accidentally and silently changes the semantics of the program under compilation). Most of these bugs were attributed to an optimization pass.

There is no sign that bugs in compilers will vanish any time soon since there is significant churn in the code of compilers. For example, LLVM/clang added a net of 0.5 million lines of new code last year alone (2013), spread over about 20,000 commits. Therefore, new bugs are likely being introduced in compilers every day.

As of May 2014, there are 411 open bug reports in the GCC database[2] marked as wrong code generation. In LLVM's bug database[3] there are currently 13 open bug reports for wrong code generation (note that this is a lower bound, since the LLVM developers often do not categorize bug reports).

Given this scenario, it is easy to understand why compiler vendors take a very conservative approach when considering new optimizations (see, e.g., a report by an Intel compiler's developer [Rob01]). New optimizations are usually only enabled by default several years after being developed. Moreover, more ambitious code optimizations are not even considered, since they pose a serious risk of miscompilation and because of the high development and maintenance cost.

Compilers are the weakest link in a formal development process, since the correctness of the final product crucially depends on them. Even if a program has been formally verified at the source-code level

[1] The improvements in real programs are usually smaller, but it is often the case that upgrading the compiler yields a performance improvement of a few percentage points.

[2] `http://gcc.gnu.org/bugzilla/`

[3] `http://llvm.org/bugs/`

(the most common case), no guarantee is carried over to the binary program generated by the compiler, since the compiler may have inadvertently introduced bugs during the compilation process if it is itself buggy.

Buggy compilers may even introduce exploitable security bugs in applications. For example, GCC 4.1/4.2 had a bug that made it remove valid pointer comparisons that could lead to security bugs in safe applications (CVE-2006-1902 [CVE06]). In particular, GCC would remove certain bounds checks from programs since it would wrongly prove that they could never be triggered.

There have been several attempts to automatically verify the correctness of compiler optimizations. However, these have either failed to handle optimizations that manipulate loops and/or change the control flow significantly, or they have relied on heuristics designed by hand. These heuristics had to be tuned for each family of optimizations, and therefore they imposed a significant burden on the (supposedly automatic) verification process.

1.2 Preconditions of Compiler Optimizations

A precondition of a compiler optimization is a sufficient (and necessary, if weakest) condition that the code under transformation must satisfy such that a given optimization can be safely performed.

Preconditions of optimizations can be stated in a variety of forms. For low-level optimizations, it may consist of the value of an input variable being required to be even/odd, or respecting a certain bit pattern. Higher-level optimizations usually have preconditions to restrict memory access patterns, such as the memory positions (e.g., array indexes) written by a certain statement must not overlap with the positions read by another statement.

These preconditions are often derived by hand by compiler developers from textbooks and/or published papers. In the process, it is unlikely that the developer will be able to generate the weakest precondition (which is not critical; the compiler will just miss some optimization opportunities), but the developer might also come up with a condition that is not strong enough to ensure that the transformation is sound for all allowed inputs.

As an example, we analyze LLVM's bug #17827. [4] In this bug report, LLVM was shown to be generating wrong code when optimizing certain integer comparisons involving shifts. The reason was that the precondition for one of the transformations of InstCombine was incorrect. InstCombine is LLVM's middle-end peephole optimizer that folds instructions within basic blocks with the goal of eliminating redundant instructions, replacing instructions with cheaper alternatives whenever possible, and putting the IR in a canonical form.

The compiler developer that fixed the aforementioned bug wrote two very interesting comments in the source-code (in the file `lib/Transforms/InstCombine/InstCombineCompares.cpp`), with the first being:

```
// For a left shift, we can fold if the comparison is not signed.
// We can also fold a signed comparison if the mask value and
// comparison value are not negative. These constraints may not be
// obvious, but we can prove that they are correct using an SMT
// solver such as Z3:
// http://rise4fun.com/Z3/DyMp
if (!ICI.isSigned() || (!AndCst->isNegative() && !RHS->isNegative()))
  CanFold = true;
```

Here the developer is basically saying that reasoning about the soundness of the precondition used is non-trivial to do by hand, and therefore an SMT solver was used for the proof instead. Although

[4] `http://llvm.org/PR17827`

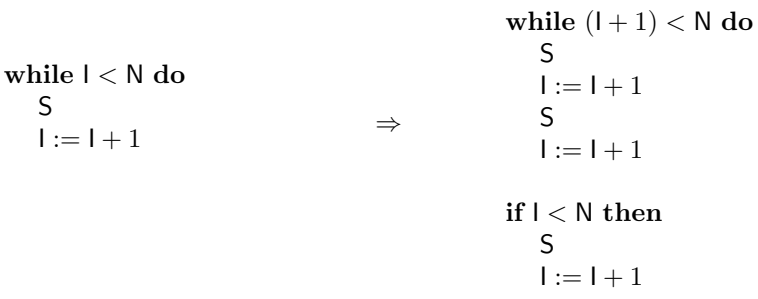

Figure 1.2: Loop unrolling: the source template is on the left, and the transformed template on the right. Template statement S cannot modify template variables I and N.

the precondition is small and may even look simple, it should be noted that reasoning about bitwise properties by hand is not trivial. Moreover, the precondition had been wrong before (hence the bug report).

The same developer wrote a second interesting comment a couple of lines later for another type of shift (arithmetic right shift):

```
if (ShiftOpcode == Instruction::AShr) {
  // There may be some constraints that make this possible,
  // but nothing simple has been discovered yet.
  CanFold = false;
}
```

In summary, the developer is saying that he does not know how a reasonable sufficient precondition looks like, let alone the weakest. Therefore, this last optimization is currently disabled in LLVM.

Although this is just one example, it clearly shows that compiler developers can benefit from the help of tools to reason about the correctness of compiler optimizations, as well as tools for the automatic synthesis of weak(est) preconditions. Without the help of auxiliary tools, compilers will continue to have unsound preconditions (and therefore miscompile code) and will continue to miss optimization opportunities that would otherwise not have to.

1.3 Thesis Overview

In this work, we propose a new technique for the automatic verification of the correctness of compiler optimizations, specified as transformation functions over code templates. Additionally, we propose the first known technique to generate weakest preconditions of optimizations, that are provably correct by construction.

Transformation functions specified over code templates enable succinct descriptions of compiler optimizations. Moreover, formally verified specifications of optimizations can be used to automatically synthesize an implementation of the optimizations for a specific compiler. The analysis used to determine whether a transformation's precondition holds in the code being compiled can also be generated with varying degrees of precision. In this way, implementations of optimizations are guaranteed to be correct by construction.

For example, Figure 1.2 shows a specification of loop unrolling in the high-level language that we consider. This optimization transforms a loop into a new loop that performs only half of the iterations of the original loop, but where each iteration of the new loop performs twice the work of an iteration of the original. Loop unrolling is a common optimization that is usually employed to expose opportunities for other optimizations, such as loop vectorization and software pipelining.

The weakest precondition for loop unrolling in the language of read and write sets (the language considered in this work) is: $\mathsf{N} \notin \mathsf{W(S)} \wedge (\mathsf{I} \notin \mathsf{W(S)} \vee \mathsf{R(S)} \cap \mathsf{W(S)} = \emptyset)$. This means that the placeholder statement S cannot write to variable N and either S does not write to I or S is idempotent (i.e., it cannot write to the memory locations it reads from).

We consider the language of read and write sets to specify preconditions since it concisely captures the informal language used by both compiler books and compiler developers, and since it is a reasonable abstraction for preconditions for most high-level loop-manipulating optimizations.

To the best of my knowledge, automatic precondition synthesis for compiler optimizations has never been proposed before. It is a novelty of this work, both conceptually and technically. Automatically deriving and verifying preconditions of compiler optimizations is of extreme importance, since it helps establishing an end-to-end correctness relation between the source-code of a program and its generated binary, carrying all the correctness guarantees of proofs done at the source-code level.

The proposed algorithms for precondition synthesis and optimization verification are also articulated as part of a broader goal of improving the way compiler optimizations are developed, as well as improving the overall reliability of compilers. We therefore present a modern architecture for implementing compiler optimizations, where these algorithms play a key role, which further motivates our work.

Parts of this thesis were previously published by the author in several venues:

- Weakest Precondition Synthesis for Compiler Optimizations, VMCAI'14 [LM14];
- Automatic Equivalence Checking of UF+IA Programs, SPIN'13 [LM13], best paper award.

1.4 Organization

This document is organized as follows. Chapter 2 presents a modern architecture for implementing compiler optimizations. Chapter 3 presents the state-of-the-art in correctness verification of compiler optimizations, as well as related techniques to improve the reliability of compilers in general. Chapter 4 describes a specification language for compiler optimizations and respective preconditions. Chapter 5 presents a new algorithm for the automatic synthesis of weakest preconditions for compiler optimizations. Chapter 6 presents a new algorithm for the automatic verification of equivalence of programs specified in the theory of integer arithmetic and uninterpreted function symbols, as well as an application of this algorithm to the verification of compiler optimizations. Finally, Chapter 7 presents a discussion on the novelty, strengths and weaknesses of the techniques proposed in this document and Chapter 8 concludes this thesis.

Chapter 2

A Modern Architecture for Compiler Optimizations

The architecture of compiler optimizations has remained mostly the same in the last 50 years. Developing new optimizations, or even performing maintenance work on older ones, is very expensive and imposes significant risks of introducing subtle correctness bugs (see, e.g., a report by an Intel compiler developer [Rob01] and a report by a GCC developer [Wei03]).

In this chapter, we propose a new architecture for compiler optimizations. The goals of this architecture are reducing the costs of development and maintenance of optimizations through the usage of high-level specification languages, as well as improving the reliability of optimizations through the consistent usage of software verification techniques. At the same time, we aim to deliver optimizers that are as efficient to execute as the current generation, while possibly generating better code.

The architecture here described motivates the work presented in subsequent chapters.

2.1 Traditional Architecture

Traditionally, a compiler optimizer has been composed by a linear sequence of individual optimization steps (Figure 2.1). Each optimization step performs a specific code transformation, such as propagating constants, simplifying the control-flow graph, or removing invariant expressions from loop bodies. Optimizations use the compiler's IR as input and output media.

Industrial-strength compilers have a number of optimizations in the order of several dozens. Some optimizations are run more than once, such as code cleanups and IR canonicalization passes, since some optimizations assume that the IR is cleaned up afterwards, and/or assume that the input IR is in a canonical form, respectively. Both assumptions usually allow simpler implementations of optimizations at the small cost of executing certain (cheap) optimizations several times.

Optimizations are usually run in a pre-defined order chosen by compiler developers, which is tuned over the years against some benchmark. There is no sequence of optimizations that is optimal for every single program because an optimization may modify the IR in a way that hinders the execution of a second optimization, and reversing their execution can have the opposite effect.

The typical optimization pipeline starts with a set of cheap intra-procedural optimizations that cleanup the code, performing tasks such as propagate constants, simplify the control-flow graph, remove dead code, and replace redundant or complex instructions with simpler variants. Then, higher-level optimizations are performed, including transformations to loops and inter-procedural optimizations. Finally, lower-level optimizations, possibly dependent on the specifics of the target CPU, are performed. This order

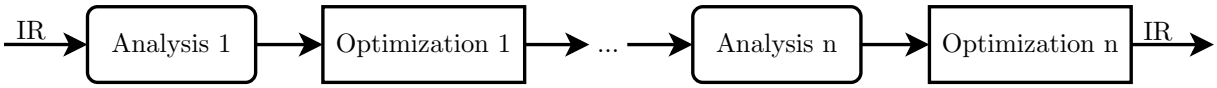

Figure 2.1: Traditional architecture for compiler optimizations.

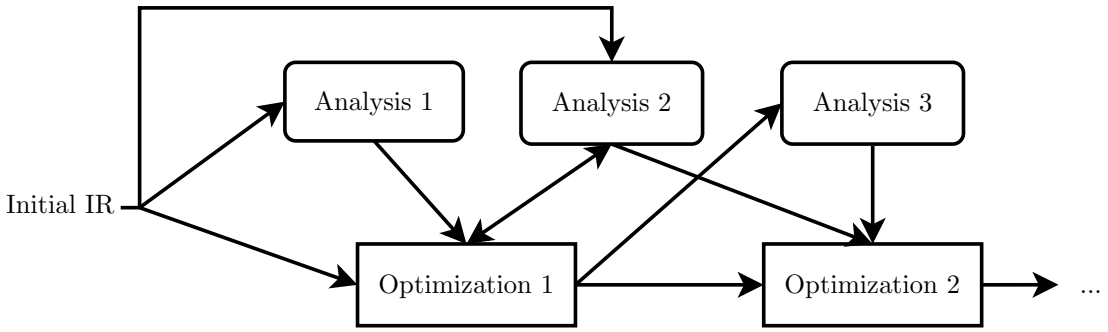

Figure 2.2: LLVM compiler's optimization pipeline.

is, however, merely indicative, and industrial-strength compilers may have different orders for different optimization goals, or even interleave and run optimizations multiple times.

Software developers are most often allowed to do only limited changes to the optimization pipeline and heuristics through compiler configuration options (e.g., `-O1`, `-O2`, `-Os`, etc). These options commonly inform the compiler that the developer is willing to wait longer for the results in the hope of obtaining a program with better performance (e.g., `-O1` vs `-O2`). Alternatively, these compiler options may specify different optimization goals, e.g., performance (`-O2`) vs code size (`-Os`).

Before each optimization step is run, usually one or more analyses are performed. These are usually data-flow analyses, using very simple lattices for performance reasons, that produce information required by the optimization to determine whether its transformation can be applied in a sound way. Typical abstractions include (non-)nullness of pointers, points-to information for variable and function pointers, ranges for integer variables, etc.

More recent compilers, such as LLVM, already attempt to share both analyses' code and results across optimizations, making the optimization pipeline less linear (Figure 2.2). This sharing is, however, usually limited to a few analyses (such as alias analysis or dominators). Moreover, there is a significant implementation burden and correctness risk for optimizations that attempt to preserve the results of an analysis, since the compiler developer has to explicitly program how each specific code transformation changes an analysis' result.

There has been some work in performing analysis work lazily and query-driven. For example, the alias analysis of LLVM is driven by queries, i.e., instead of running the whole analysis before optimizations (and therefore potentially computing unneeded information), the analysis internally decides which parts of the code should be analyzed in order to answer a particular query from an optimization (e.g., *can these two pointers ever point to the same memory location?*). The analysis can stop as soon as it has a correct answer for the query.

Similarly, LLVM's range analysis (LVI – lazy value info) is run lazily, although it is not query-driven. While the analysis is not run eagerly for the whole program, when an optimization asks for the range of a given variable, the whole analysis is run for that variable even if a wider range that was sufficient to prove the optimizer's query had been discovered already.

In order to implement a new optimization, a compiler developer has not only to implement the specific

code transformation, but he also has to derive a correct precondition that states when the optimization is safe to apply. Moreover, it is desirable that the precondition be the weakest in order to avoid missing optimization opportunities.

Even when a compiler developer has a correct precondition for an optimization, he still has to design an analysis that implements the verification of such condition. This is usually accomplished by implementing some sort of data-flow analysis, which is easy to get wrong. Moreover, the design of an analysis already has a built-in trade-off between precision and execution cost. This trade-off is usually not configurable once the analysis is developed, and so the developer has to make this complicated, and possibly uninformed, decision very early.

2.2 A New Architecture

As noted in the previous section, current compiler optimization architectures have several development inefficiencies and drawbacks. This includes, in particular, high complexity of sharing analyses' results between optimizations and keeping them in sync along code transformations, of developing new analyses (especially the more efficient query-driven), of choosing the right analysis for a given optimization, of the manual implementation of code transformations, and so on.

In this section, we propose a new architecture for compiler optimizers. The goal is to reduce the development cost of compilers, by improving compiler developers' productivity and by reducing the probability of introducing miscompilation bugs, while at the same time achieve compiler execution efficiency levels comparable (if not better) with today's architectures.

The architecture that we propose is divided in four stages, with each being specialized in a single task. This separation of concerns improves maintainability and allows compiler developers to focus on the module of their expertise. The benefits arising from the splitting have already been largely tried and confirmed by developers with the common division of compilers in three major components (front-end, middle-end, and back-end).

The four stages that we propose for a new architecture for compiler optimizers are the following (pictured in Figure 2.3):

1. Pattern matching

2. Precondition verification

3. Profitability heuristic

4. Code transformation

The idea is that the effort to develop and maintain such an architecture is amortized across all optimizations, since the four components are to be used by all optimizations. In this way, an optimization is an input to the four stages (along with the code to be optimized).

In contrast, in traditional architectures, each optimization is responsible for performing all the four stages (modulo stage 2 where, as explained in the previous section, there is already limited sharing of analyses between optimizations). This traditional architecture leads, therefore, to significant duplication of implementation effort.

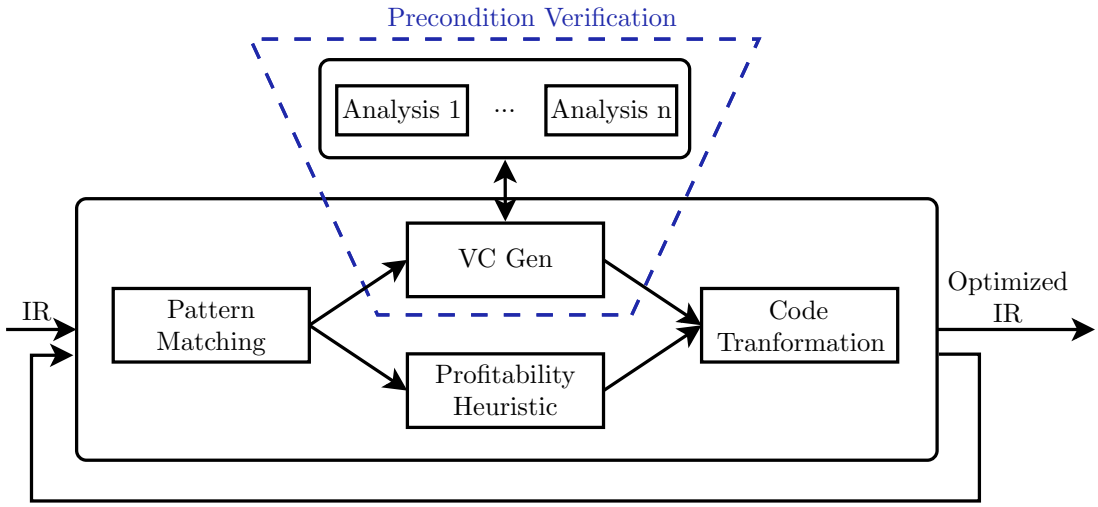

Figure 2.3: Proposed architecture for compiler optimizations.

We now describe each stage individually with an example. Consider the following program in C:

```
if (x >= 0) {
    v = 2 * x + y;
    f(v);
}
v = 2 * x + y;
```

We will show how to apply the following example transformation, which removes computations of redundant expressions within "**if**" statements:

if B **then**		V := E
V := E	⇒	**if** B **then**
S		S
V := E		

On the left of the arrow, we have the source code template, and on the right the optimized or target code template. S is a template statement, which a placeholder for arbitrary statements, including loops, function calls, assignments, etc. Similarly, B and E are template boolean and integer expressions, respectively. This transformation function states that a program fragment that matches the source template should be transformed to the target template with the proper instantiation. A full description of the specification language for transformation functions is given in Section 4.2.

Applying the given transformation function to the example program fragment yields the following code:

```
v = 2 * x + y;
if (x >= 0) {
    f(v);
}
```

2.2.1 Pattern Matching

The first stage is pattern matching, which identifies potential optimization opportunities in the code. It takes a set of patterns and tries to match them against the code under optimization, and then passes the patterns and respective instantiations to the following stage.

This stage can be implemented in multiple ways. It can be implemented, for example, as a strictly "dumb" syntactic matcher, whose goal is solely to find all possible instantiations for the template statements/expressions. This has the potential disadvantage of possibly finding too many matches that trivially do not fulfill the precondition of the optimization. Similarly, many matches may not satisfy the profitability heuristic.

Another alternative for implementing this first stage is to incorporate some preliminary and purely syntactic checking of optimizations' preconditions. We believe many instantiations can be avoided in this way, which should be beneficial in terms of performance, at the cost of a more complex implementation of the pattern matching engine.

This stage does not attempt to solve the optimization ordering problem. The order in which patterns are tried should be defined by the compiler developer or using techniques that heuristically try to decide on the best optimization sequence for a given program (or function).

Recently, there has been substantial work on using machine learning algorithms to automate the optimization order learning process (e.g., [CO06, FKM+11]). This line of work is, however, orthogonal to ours, and can be incorporated in this stage.

For the running example, it is easy to see that the transformation pattern does indeed match against the code, and we obtain the following instantiation: $\mathsf{B} \mapsto \texttt{x >= 0}$, $\mathsf{V} \mapsto \texttt{v}$, $\mathsf{E} \mapsto \texttt{2 * x + y}$, $\mathsf{S} \mapsto \texttt{f(v)}$. This instantiation is unique, but that is not the case in general, since there can be multiple possible instantiations for a given template and program fragment.

2.2.2 Precondition Verification

The second stage is precondition verification, which ensures that the candidate that the pattern matching engine identified can be transformed with no change to the program semantics.

Each optimization has a sufficient (and desirably weakest) precondition that states when the optimization is semantics preserving. This precondition is stated over the template symbols, including template statements and expressions.

For the running example, the weakest precondition stated in the language of read and write sets (the one that is used in the present document) is: $\mathsf{W}(\mathsf{S}) \cap \mathsf{R}(\mathsf{E}) = \emptyset \wedge \mathsf{V} \notin \mathsf{R}(\mathsf{B}) \wedge \mathsf{V} \notin \mathsf{R}(\mathsf{E}) \wedge \mathsf{V} \notin \mathsf{W}(\mathsf{S})$. In other words, the precondition is that statement S cannot write to any variable that is read by expression E, that V is not read by either B or E, and finally that statement S does not write to variable V.

For the instantiation we obtained in the previous step, we have the following read and write sets (assuming that the function call `f(v)` does not modify any global variable): $\mathsf{R}(\mathsf{E}) = \{\texttt{x}, \texttt{y}\}$, $\mathsf{R}(\mathsf{B}) = \{\texttt{x}\}$, $\mathsf{R}(\mathsf{S}) = \{\texttt{v}\}$, $\mathsf{W}(\mathsf{S}) = \emptyset$. In this case, it is trivial to conclude that the matched code satisfies the precondition of the example transformation, and therefore the transformation can be safely applied. More complicated cases involving, e.g., pointer arithmetic, arrays, or nested loops would likely not be as obvious to prove.

This second stage, precondition verification, is split itself in two steps. First, we do verification condition generation (VC Gen), which takes the precondition of the transformation and the instantiation computed by the previous pattern matching step, and produces a set of proof obligations that must be satisfied in order for the transformation to be marked as safe. Second, a verification pipeline discharges the proof obligations.

Discharging proof obligations can be done by using current compiler technology, which is already implemented in compilers and is very efficient. Analyses like alias analysis, range analysis, scalar evolution, and so on, can all be used together to discharge the generated proof obligations.

This clear separation of the verification part is highly beneficial, since then all analyses become available to all optimizations, and the compiler is free to choose which to use for a particular proof obligation. Moreover, faster analyses can be used to discharge easy cases quickly, while leaving more complex obligations to more precise analyses. The amount of precision should be specifiable by the user as, e.g., an optimization level (e.g., `-O1` vs `-O2`).

In addition to regular compiler data-flow analyses, it becomes possible to use more heavyweight verification tools for the cases where the compiler analyses fail. Tools like HSF [GLPR12], SLAyer [BCI11], Terminator [CPR06], Z3 [dMB08], etc, can all be part of this verification pipeline. Researchers have also shown how to perform data-flow analysis using model checkers [Ste91, SS98].

2.2.3 Profitability Heuristic

Semantics preserving transformations are not always performance improving or even performance preserving (for some performance metric). Therefore, the third stage is responsible for deciding which of the correct transformations should be applied in order to optimize the code for a given metric (or metrics). Popular metrics include running time, code size, energy consumption, and a combination of these.

This step may be fully target independent, or it may use information from the specified target CPU architecture, such as cache line size, overall cache size, latency and throughput of CPU instructions, available ISA extensions, and so on.

For the running example, the transformation may seem obviously and universally beneficial, since it eliminates a redundant computation. However, it may increase register pressure, for example, since `V` will be live for the whole execution of `S`. If `S` is long and requires many registers, it may be cheaper to recompute the expression `E` than to keep its value around. So, even in this simple example, it is not clear in which cases it is beneficial to apply the transformation.

Some compilers, such as LLVM, prefer to optimize regardless of register pressure constraints, leaving the job of splitting long live ranges to a specialized algorithm (the register allocation pass, performed in the back-end). Some register allocation algorithms (such as LLVM's) can even introduce recomputations of expressions whenever recomputing them is cheaper than saving their value (usually to the stack). Therefore, a profitability heuristic designed for LLVM for the running example could ignore register pressure altogether, since `E` would be recomputed if necessary (effectively undoing the transformation if later the compiler realized it was not profitable).

Similar to the optimization ordering problem, specifying heuristic functions is an art, and we do not attempt to solve this problem either. Likewise, machine learning techniques have been proposed to automate the discovery of heuristic functions, e.g., [MBQ02, SCWK13].

The work presented in this document is focused solely on the correctness of optimizations. Therefore, profitability heuristics, which cannot influence correctness, will be ignored for the rest of this document. They are, of course, very important in practice.

2.2.4 Code Transformation

The final step is performing the code transformation itself. This step is simple, since we have already proved before that the transformation is both safe and performance improving.

This step consists, conceptually, in replacing the pattern-matched code with an instantiation of the optimized code template. In practice, however, it is highly desirable to replace and/or move only the instructions that differ in the source and optimized code templates.

This step should also be responsible for updating debug information, to ensure that the newly created instructions remain associated with the respective source-code that originated them (even if indirectly). This is very important for usability concerns, since developers need to be able to debug compiler-optimized code, and the quality of debug information plays a significant role in developers' productivity in investigating bugs.

Another important aspect is the preservation of cached analysis results. It is desired that any analysis already performed is not repeated, and so any code transformation should try to preserve as much cached results as possible.

2.3 Summary

In this chapter, we proposed a new architecture for the implementation of compiler optimizations. The main goals of this new architecture are to improve the productivity of compiler developers, as well as to increase the reliability of compilers.

This modern architecture motivates the work presented in the next chapters, including the algorithms for automatic precondition synthesis and automatic verification of compiler optimizations. These algorithms fit particularly well within this architecture, since the logical next step after discovering the weakest precondition of an optimization is to use it in a compiler. However, with a traditional architecture, the amount of work and the risk involved in the implementation of a precondition checker is non-trivial. Therefore, it naturally gives rise to the desire of automating the usage of the computed preconditions.

Our algorithm for the verification of compiler optimizations works over high-level specifications. Thus, it is highly desirable that the respective implementations are generated automatically from these specifications, so that the implementations are guaranteed to be correct by construction. Again, this view can be materialized with the proposed architecture.

Chapter 3

Related Work

In this chapter, a survey of the techniques related to the work described in this document is presented.

In this survey, both closely related and alternative techniques to the work proposed in this document are presented. Closely related techniques include: translation validation; manual, semi-automatic, and automatic optimization verification techniques; compiler verification techniques; and precondition synthesis. Alternative techniques to improve the reliability of compilers and optimizations include: superoptimization; automatic generation of optimizations; and automatic bug finding.

3.1 Overview

There is a wide range of techniques aimed at improving the reliability of compilers and optimizations in particular. In this section, a summary of the techniques is presented, leaving further details to the following sections.

Compiler Verification In terms of verification, there are manual, semi-automated, and automated techniques. We believe that non-fully automatic techniques are too hard and too time consuming for compiler developers to grasp. Therefore, attention should be focused on fully automatic techniques.

Most automatic techniques that support loop-manipulating optimizations require the synthesis of bisimulation relations (e.g., [KTL09]). We believe this is a major caveat since bisimulation relations for non-structurally equivalent programs are usually complex and non trivial to synthesize (either automatically or by hand).

Verification of whole compilers is still very difficult in practice. The experience of CompCert [Ler09a] shows that developing a fully verified and industrial-strength compiler is a highly complex and costly project. Moreover, many proofs have to be done by hand, and their size is often larger than the code itself.

Software verification tools State-of-the-art software verification tools are unable to verify the correctness of compiler optimizations through sequential program composition. Sequentially composing the source and target templates into a single program is a simple and direct solution to the problem of verifying optimizations. However, verifying the resulting composed programs requires complex reasoning and intricate loop invariants, which is out of reach for state-of-the-art tools and techniques.

Other approaches Translation validation is a direct competitor to optimization verification. Such tools validate *a posteriori* that a given optimization preserved the semantics of the code. While it is a potentially easier problem, and therefore better solutions can be derived, it has two important drawbacks.

First, compilers can still be shipped with bugs, which will only be found later by a regular user. Second, translation validation imposes a non-negligible overhead to the compilation time.

Random testing is a cheap technique, but can only provide limited guarantees. Still, Csmith [YCER11] found bugs in CompCert [Ler09a], showing that verification and testing are complementary.

An alternative approach to increase the reliability of compilers is to generate optimizations automatically, hence guaranteeing correctness by construction. Automatic optimization generation is, however, still in its infancy, and it is not expected that the techniques will be able to reason about complex loop optimizations in the near future.

3.2 Bug Finding

Automatic bug finding in compilers is a topic that has been under research for about 50 years. The proposed methods consist in generating many random programs and check if any of those is miscompiled. A survey on the topic can be found in [BS97].

Quest [Lin05] is a tool that generates test cases to detect bugs in the calling conventions of C programs. The generated tests are self-testing, i.e., each test contains a set of assertions that automatically check if all parameters were correctly passed.

Zhao et al. [ZXT+09] presented a test case generator that is guided by a formula in computation tree logic (CTL) that describes the optimization to be tested. The expected output of each test is given by a reference compiler, which is assumed to be correct.

Csmith [YCER11] is a test case generator that only generates C test programs with well-specified behavior, i.e., it does not generate any program that has unspecified or undefined behavior according to the C99 standard. The expected output of each test is assigned by the consensus of the set of compilers under testing. Csmith has also been applied for testing static analyzers [CMP+12].

NeonGoby [WHTY13] is a tool that can be used to find bugs in implementations of alias analyses of compilers. NeonGoby instruments compiled code, so that the generated binary detects when the aliasing information produced by the compiler's alias analysis is violated at run time. NeonGoby successfully found several bugs in different alias analyses of LLVM.

Morisset et al. [MPZN13] presented a technique to detect miscompilations regarding the C11/C++11 memory model by analyzing the dynamic memory trace of compiled programs. The goal was to detect miscompilations that could affect the correctness of multi-threaded programs due to illegal memory accesses introduced by the compiler.

Orion [LAS14] generates test cases by stochastically performing changes to an existing test suite. Orion has found hundreds of bugs in mainstream compilers.

3.3 Verified Compilers

There has been interest in formally verifying the whole compilation process (and not only optimizations) since, at least, 1967, when McCarthy and Painter published a paper on the verification of a compiler for arithmetic expressions [MP67].

Here, only the most recent efforts to develop verified compilers are reviewed. The reader is kindly referred to [Dav03] for a more complete bibliography on the subject.

CompCert CompCert [Ler09a] is a compiler for a subset of C that was developed from scratch with formal verification in mind. CompCert was verified using the interactive (i.e., semi-automatic) theorem prover Coq [BC04]. Proofs of correctness are an order of magnitude larger than the implementation itself, which significantly increases the development cost. Moreover, no powerful verified middle-end

optimizations are provided. The correctness of most optimizations is guaranteed by specialized and formally verified translation validators (one per optimization). The middle-end and the back-end are described in [Ler09b].

The C front-end of CompCert is partially verified [BDL06]. Recent studies found a few bugs in unproved parts of the front-end [YCER11, LAS14]. The results of these studies suggest that building good front-ends is still not a solved problem, contrary to popular belief.

Barthe et al. [BDP14] presented an SSA-based middle-end for CompCert. This middle-end first transforms CompCert's RTL-based IR into SSA, then runs SSA-based analyses and optimizations, and then converts the code in SSA back to RTL (register transfer language). Optimizations are not verified, but the correctness of their result is checked *a posteriori* (using translation validation). The translations to SSA and back to RTL are verified in Coq.

Others Leinenbach et al. [LPP05] presented a compiler for the C0 language (a subset of C), which was developed and verified in Isabelle/HOL [NWP02].

Jinja [KN06] is a verified virtual machine for a Java-like language that was developed in Isabelle/HOL.

Benton, Dreyer, and Hur [BH09, HD11] applied step-indexing of Kripke logical relations to verify the correctness of a simple compiler.

Chlipala [Chl10] presented a verified compiler for an untyped Mini-ML like language. Although the size of the proofs is within the same order of magnitude of the code, the presented technique was only applied to a simple compiler prototype.

3.4 Translation Validation

Translation validation is a special case of program equivalence checking. The objective is to prove the equivalence between the input and output of a code transformer, such as an optimizer, a code generator, or a compiler.

The roots of translation validation can be traced back to Samet's PhD thesis [Sam75]. The translation validation term was, however, only coined later in 1998 by Pnueli et al. [PSS98].

Program equivalence is known to be undecidable. Therefore, a general, sound, and complete translation validator cannot be built. However, translation validators specialized for, e.g., a specific compiler optimization can be made both sound and complete. Some validators (such as [TL09]) are believed to be complete, although so far there is no formal proof of such an assertion (due to the complexity of producing one).

Code transformers are inherently mechanical, and so the set of possible transformations they can perform on the input code is limited (although not necessarily small). Therefore, since translation validators are usually targeted for a specific transformer (e.g., a specific compiler, or a specific set of optimizations), the problem of translation validation is significantly simpler (and tractable) than general program equivalence checking.

Translation validators can be classified according to three orthogonal categories. The first is related with the applicability: whether a validator is general (such as [PSS98, RM99, Nec00, PHG05, ZPG+05, ZP08, KSK09, TGM11, STL11]), or specific to a particular set of translations (such as instruction selection [TL08], lazy code motion [TL09], software pipelining [LP02, TL10b], or register allocation [HCS06, RL10]). General validators can potentially validate a broad range of translations, while specific ones can only validate one particular translation (or a class of them). However, specific validators can be made more efficient and potentially complete, as they can take advantage of the limited possible code transformations that a given, say, optimization may perform.

The second category is related with the technology used to prove the equivalence: symbolic execution [Kin76] and/or static analysis (such as [Nec00, HCS06, TL08, TL09, TL10b, RL10]), or generation and automatic discharge of verification conditions (such as [PSS98, RM99, LP02, PHG05, ZPG+05, ZP08, KSK09, TGM11, STL11]). Most symbolic evaluation engines of these tools are quite efficient as they only manipulate syntactic terms, at the expense of a reduction in precision. Generating verification conditions and discharge them automatically using, e.g., off-the-shelf theorem provers, SAT or SMT solvers, or model checkers, is a much more powerful technique, but more costly. Relying on off-the-shelf tools has the benefit that advances in these tools will automatically benefit a translation validator built with them.

Finally, some validators require translators to be instrumented to provide extra information to conduce the equivalence proof (e.g., [RM99, PHG05, KSK09, TL09, NZ13]), while others do not. The information provided by the translator does not need to be correct since it is checked for correctness. The information is only used as a guide for the correctness proofs.

3.5 Automatic Optimization Verification

Formally verifying a compiler optimization can be reduced to proving the equivalence of the source and optimized program templates under a given precondition. While program equivalence is, in general, undecidable, some techniques to automatically verify optimizations have been proposed in the past few years.

Cobalt and Rhodium Cobalt [LMC03] and its successor Rhodium [LMRC05] are languages to specify compiler optimizations and data-flow analyses. Their accompanying system is able to automatically verify the correctness of optimizations written in the proposed DSLs using the theorem prover Simplify [DNS05]. However, these systems are not capable of reasoning about optimizations that manipulate loops. Moreover, only optimizations that can be specified with one-to-one rewrite rules are supported. Cobalt and Rhodium were inspired in the work of Lacey et al. [LJVWF04], with the major difference that preconditions are not specified using CTL formulas like in Lacey's work.

PEC Parameterized equivalence checking (PEC [KTL09, TL10a]) is a technique to prove compiler optimizations correct automatically. It works by automatically finding a bisimulation relation [San09] between the original and the optimized template programs.

In principle, PEC can verify the correctness of all optimizations that do not perform significant changes to the structure of a program. Optimizations that change the number of loops, the number of branches, or the number of iterations per loop, are more difficult for a method based on bisimulation relation synthesis.

PEC includes a permutation module (a set of heuristics inspired in [GZB05, ZPG+05]) that reshuffles the code of loops around to make it easier to find a bisimulation relation automatically. Otherwise, PEC would not be able to find a bisimulation relation for most optimizations that are not structurally preserving. However, in the degenerate case, this may imply having one heuristic per optimization. Moreover, these permutation heuristics must be verified (possibly by hand).

Others Dold et al. [DHPR97] presented a technique to verify peephole optimizations in the PVS system [ORR+96].

The technique proposed by Guo and Palsberg [GP11] also requires the discovery of a bisimulation relation, thus presenting similar disadvantages as PEC. Additionally, it only supports optimizations over traces (as used by, e.g., JIT compilers), hence it does not need to reason about loops explicitly. Dissegna

et al. [DLR14] present a more general framework to reason about the correctness of trace optimizations based on abstract interpretation.

3.6 Manual Optimization Verification

The verification of optimizations by hand or in semi-automated ways has been proposed over the years.

For example, researchers have used the theorem prover Isabelle/HOL [NWP02] to perform semi-automated correctness proofs [BGG05, GGS08, MG10]. Burckhardt et al. [BMS10] proposed a semi-automated technique to verify program transformations for concurrent programs under different (weak) memory models. Kozen and Patron [KP00] proposed a technique to verify (by hand) the correctness of optimizations specified using regular expressions.

Relational Hoare logic [Ben04] is a proof system that enables the verification of equivalence between two programs. However, the system only supports the verification of structurally equivalent programs (while many optimizations do not obey this constraint). Barthe et al. [BCK11] lift some of the restrictions of this work through the usage of product programs. The set of possible verifiable transformations is still dependent on the set of built-in proof rules.

Liang et al. [LFF12] adapted relational Hoare logic to the setting of concurrent programs. Proofs were mechanized in the interactive theorem prover Coq [BC04].

Vellvm [ZNMZ12, ZNMZ13] is a framework to reason about program transformations expressed in LLVM's intermediate representation [LA04]. Proofs are done in Coq, and a concrete implementation of a transformation can be extracted automatically so that it can be run from within LLVM.

3.7 Alternative Techniques to Verification

3.7.1 Superoptimization

Massalin [Mas87] proposed a new approach for program optimization named Superoptimization. The idea is to generate an optimal instruction sequence for a given function by performing an exhaustive search over all possible combinations of instructions and their parameters.

The advantage of the superoptimization techniques is that they are capable of producing a correct and optimal sequence of instructions for a given function and performance metric. However, since the search space grows exponentially with the size of the function to optimize, this technique does not scale beyond small functions. Moreover, most superoptimizers have limited support for loops.

TOAST [BCDVF06] is a superoptimizer that uses an answer set programming (ASP) solver to perform the search space exploration.

Denali [JNZ06] is a superoptimizer that uses automated theorem proving techniques, such as E-graphs, to perform exhaustive search. Tate et al. [TSTL09] presented a similar system, but with support for some loop transformations.

STOKE [SSA13] is a superoptimizer that trades optimality with scalability. STOKE uses stochastic search to find better instruction sequences, and is limited to loop-free code.

3.7.2 Automatic Optimization Synthesis

Instead of verifying the correctness of compiler optimizations, another possibility is to automatically synthesize them. Provided the optimization generator is itself correct, the optimizations generated are automatically correct by construction.

Davidson and Fraser [DF84] presented a simple peephole pattern generator. Their generator receives as input a processor description (such as the ones usually built for instruction selector generators) and outputs a set of patterns for peephole optimization.

Granlund and Kenner [GK92] applied superoptimization techniques to generate peephole optimizations for the GCC compiler. Bansal and Aiken [BA06] extended the technique to include a sound equivalence check using a SAT solver.

Tate et al. [TSL10] extended their superoptimizer to generate optimizations by extracting common sequences of axiom applications. The set of possible optimizations that can be synthesized is, however, limited by the set of axioms used by the superoptimizer, since the resulting optimizations are a combination of small known transformation functions (the axioms).

Scherpelz et al. [SLC07] propose an algorithm to automatically synthesize flow functions from preconditions of compiler optimizations.

3.8 Software Verification Techniques

State-of-the-art software verification tools, such as Blast [HJMS02, HJMM04], CPAchecker [BK11], Duality [MR13], HSF [GLPR12], Slam [BR02], and UFO [ALGC12], are unable to prove equivalence of most programs containing loops, since they are usually unable to automatically derive sufficiently strong loop invariants to complete the proof, even when limited to the theory of integer arithmetic, let alone the combined theory of uninterpreted function symbols and linear integer arithmetic (UF+LIA). Similar problems have been faced by techniques doing information flow proofs through program self-composition [TA05].

Beyer et al. [BHMR07] present an algorithm to synthesize loop invariants over the UF+LIA theory, and Rybalchenko and Sofronie-Stokkermans [RSS07] present an algorithm to synthesize interpolants over the same theory. McMillan [McM11] introduced an algorithm to generate interpolants from the unsatisfiability proofs of the SMT solver Z3 [dMB08]. Gulwani et al. [GSV09] present a technique to synthesize invariants based on constraint solving. However, the language of interpolants/invariants supported by these algorithms is not able to express an unbounded number of UF applications, which is often required to prove equivalence of programs that have UF applications inside loops.

Polynomial loop invariant generation techniques (e.g., [SSM04, MOS04, RCK07, CJJK12, DXZ13]) can only generate non-linear invariants with bounded exponents. However, this is not sufficient for the verification of the non-linear integer programs generated by the algorithm proposed in this document (after removing the UF applications), since these programs often require loop invariants with unbounded exponents (arising from, e.g., UF applications with self-feedback). Other invariant synthesis techniques, such as the ones based on abstract interpretation (e.g., [CH78]), usually only support linear arithmetic.

Acceleration (e.g., [CFLZ08, BIK10, HIK+12, GS13]) is a set of techniques to summarize periodic relations (arising from, e.g., loops) in a precise way. The resulting relation has usually to be expressible either in Presburger arithmetic or in an appropriate abstract domain. However, transitive closures of loops arising from the verification of compiler optimizations are usually not expressible in Presburger arithmetic.

Gupta et al. [GPR11] present an algorithm to solve recursion-free Horn clauses in the theory of UF+LIA. Grebenshchikov et al. [GLPR12] extend this work to recursive Horn clauses in order to support the verification of programs with loops and recursive functions. The interpolation algorithm used by the corresponding tool suffers from the same limitations as the others previously described.

Gulwani and Tiwari [GT06] present an algorithm for the verification of programs over the UF+LIA theory. However, only equalities over UF applications are supported, and conditional branches are abstracted non-deterministically, which is too weak for the application of equivalence checking.

Blanc et al. [BHHK10] and Gulwani et al. [GMC09] present algorithms to compute symbolic bounds of loop trip counts. However, the computed trip counts may not be sufficiently precise for equivalence checking proofs.

3.9 Program Equivalence

Proving correctness of a compiler optimization can be reduced to proving that its original and optimized code templates are equivalent. Moreover, program equivalence checking has several other important applications, including, for example, algorithm recognition [AB03], regression checking [GS09, CGS12, LHKR12], and information flow proofs [BDR04, TA05]. In this section, we summarize the several approaches to program equivalence checking that have not been covered in previous sections, as well as some of the most important applications.

Applications The objective of algorithm recognition is to identify known algorithms (such as a sorting algorithm, or even a specific algorithm like quicksort) out of large and complex programs. This can be useful, for example, to improve code comprehension and for automatic documentation generation. Algorithm recognition can be accomplished by searching for an equivalent algorithm in a database.

Regression verification aims at tracking the functional differences in a program in each code change. The idea is that a tool that performs regression verification can pinpoint the parts of the program where the semantics were changed since the previous code revision, so that the developer can manually confirm if those were the intended changes. Additionally, these tools can help the developer confirm if some code refactoring or manual optimization preserved the semantics or not.

In the domain of information flow, proofs for the non-existence of information leaks can be accomplished by establishing the equivalence of the program with itself (self-composition). Since the programs have some associated non-determinism (the private information), a program will not be equivalent to itself if some of the non-determinism may be observable (meaning that it may leak secure information).

Recurrence Equivalence It is sometimes possible to use recurrence relations to precisely summarize the effects of loops. Therefore, algorithms to prove equivalence between (systems of) recurrences can be used as a key building block for program equivalence checking.

Barthou et al. [BFR02] and Shashidhar et al. [SBCJ05] present different algorithms to prove the equivalence of systems of affine recurrence equations that are structurally similar.

Verdoolaege et al. [VJB09] propose an algorithm to prove the equivalence of integer affine programs where loops are described as recurrences. The algorithm does not compute the closed-form solution for the recurrences, but instead uses widening to reach a fixed point. The algorithm handles commutative operators by trying all possible permutations.

Symbolic Execution Symbolic execution is a technique to analyze programs which consists in executing programs with a specialized interpreter where inputs are treated and propagated symbolically.

Matsumoto et al. [MSF06] and Person et al. [PDEP08] present different techniques to detect differences between two programs that are mostly equal.

Ramos and Engler [RE11] present an algorithm to check for program equivalence automatically. The implemented tool is based on KLEE [CDE08] and can only prove equivalence up to a bounded number of loop unrollings.

Mutual Summaries Godlin and Strichman [GS08] propose a set of proof rules to prove equivalence of programs and to prove mutual termination using UFs to abstract recursive function calls. Loops are

encoded are recursive functions.

The technique of Godlin and Strichman is later extended with the introduction of mutual summaries [HKLR13], which consists in logical relations between the input/output relation of the two implementations of each function present in both programs under equivalence checking.

Bisimulation Synthesis Multiple approaches for the automatic synthesis of bisimulation relations have already been described in Section 3.4 and Section 3.5.

DDEC [SSCA13] is a tool that can prove equivalence of loops written in x86 assembly automatically. DDEC uses randomly generated tests to guess potential sets of cut points and equality constraints between cut points, which are then confirmed using an SMT solver. DDEC can only prove equivalence between structurally similar programs.

3.10 Precondition Synthesis

The concepts of weakest preconditions (WPs) and weakest liberal preconditions (WLPs) have long been introduced by Dijkstra [Dij75]. WLP is a weaker variant of WP in that WLP is a condition that an initial state of a program must verify in order to establish a given property in all terminating paths. A WP has the additional constraint that all paths must terminate (and likewise establish the given property).

Since the seminal work of Dijkstra, several algorithms have been published to accomplish the automatic generation of WPs and WLPs. However, there is no known previous work on automatic synthesis of preconditions for compiler optimizations. Compiler optimizations are distinct from other applications in the sense that preconditions have to be generated for transformation functions over program templates rather than for concrete programs. Techniques to generate preconditions for other applications are presented instead.

There are several competing approaches for WLP synthesis. These include, for example, precondition templates and constraint solving (e.g., [GSV09]), quantifier elimination (e.g., [Moy08]), abstract interpretation (e.g., [CCFL13]), and CEGAR [CGJ$^+$00], predicate abstraction, and interpolation for predicate generation (e.g., [SK13]). Some algorithms combine multiple techniques to achieve better performance.

The UNSAT core minimization algorithm presented in this document, that biases the result towards certain literals, is similar to the one presented by Seghir and Kroening [SK13].

Leino [Lei05] describes a compact encoding for verification conditions generated from the weakest precondition calculus.

Cook et al. [CGLA$^+$08] propose a counterexample-driven algorithm to synthesize sufficient (but not necessarily weakest) preconditions that ensure that a given program terminates. The algorithm works by iteratively strengthening potential ranking functions. Bozga et al. [BIK12] propose an algorithm based on abstract interpretation for the same task, but with guaranteed completeness results for certain abstract domains (such as octagons).

Calcagno et al. [CDOY07, CDOY09] present a technique to automatically infer WLPs of heap manipulating programs using shape analysis with separation logic, with the purpose of doing bottom-up program analysis.

Gulwani et al. [GJTV11] present an algorithm to synthesize loop-free programs that implement a given specification. While the goal of the algorithm is not to synthesize preconditions, there is a similarity in the encoding of program equivalence and in the usage of an SMT solver to find assignments to variables that represent the synthesized artifact.

3.11 Specification Languages

Many specification languages for compiler optimizations have been proposed, including ones based on graph rewriting systems [Aßm96], regular expressions [KP00], rewriting rules [WS97, LMC03, LMRC05], CTL logic [RW98, LJVWF04], type systems [SU08], and C-like imperative language rewrite patterns [KTL09]. We believe that specification languages based on rewrite patterns with syntax close to mainstream programming languages are the best option, since compiler developers usually think in optimizations as "I would like to transform this code fragment into that one". Therefore, rewrite patterns are the more natural and easier way to develop new compiler optimizations.

Similarly, many specification languages for data-flow analyses have been proposed. Usually the accompanying systems provide a common fixed-point engine to compute data-flow facts in programs. Thus, the analysis developer only has to specify the transfer and meet/join operators. Most frameworks can only handle boolean facts.

Sharlit [TH92] is based on C++ and generates code for the SUIF compiler, while AG [ZME06] generates code for the Phoenix compiler.

In TVLA [LAS00], analyses are specified as three-valued logic formulas. TVLA was used to implement a shape analysis to prove safety properties.

Silver [VWK06] is a framework where data-flow analyses are specified using attribute grammars extended with constructs for handling control flow graphs and CTL formulas. Analyses are performed using the NuSMV model checker.

Hoopl [RDPJ10] is a framework for implementing data-flow analyses and transformations in Haskell. Data-flow analyses are composed using the technique of Lerner et al. [LGC02]. Hoopl is used by the GHC compiler.

TSL [LR08] is a framework for implementing data-flow analyses for assembly code. TSL has two specification languages: one for specifying ISAs, and another for specifying data-flow analyses independently of a specific ISA. TSL can then generate data-flow analyses for a specific ISA from the specifications.

Dincklage and Diwan [vDD08] present a logic-based language to specify data-flow analyses. They then build a system that can explain to the developer why a certain optimization was not performed and even suggest assumptions that the developer can provide so that the analysis succeeds and the optimization is applied [vDD09].

The framework presented by Sittampalam et al. [SdML04] can perform incremental updates to the results of analyses specified in a Prolog-like language after a transformation is performed to the analyzed code.

3.12 Optimization Validation

Another line of work is verifying if an optimization is in fact an optimization, i.e., check if a given transformation function increases the performance of the transformed program under some metric. The focus of the presented work is only on the correctness of transformations, leaving the reasoning about code improvements to the compiler developer.

Sands [San98] proposed a logic where the classical observability correctness is extended with intensional information about program's performance.

Aspinall et al. [ABM07] proposed a technique to manually verify performance improvements of transformation functions in Isabelle/HOL [NWP02].

Zhao et al. [ZCS06] presented a technique to predict the performance impact of an optimization, taking into account the context of the instantiation of the transformation function.

3.13 Summary

There is a broad range of techniques that have been proposed over the past decades to improve the reliability of compilers. We gave an overview of those techniques in this chapter.

The closest work to ours is PEC [KTL09, TL10a], which consists in a DSL to specify compiler optimizations, on which we base ours, as well as a system for the automatic verification of correctness of optimizations. Nonetheless, our equivalence checker is not based on bisimulation relation synthesis like PEC.

In terms of precondition synthesis, the closest work to ours is that of Seghir and Kroening [SK13], whose algorithm also works in a counterexample-driven way and uses predicate abstraction.

Chapter 4

Specifying Compiler Optimizations

This chapter presents how compiler optimizations are specified throughout the document. First, a definition of compiler optimizations is given. Then, a domain specific language for the specification of transformation functions is presented (first by example, and then formally). Finally, a language for the specification of preconditions used to state in which cases an optimization is provably correct is presented.

Several specifications of optimizations and their preconditions are given as an example in Table 5.1. This is to demonstrate the expressiveness and succinctness of the proposed framework.

4.1 Compiler Optimizations

Compiler optimizations are represented by a triple (τ, ψ, h).

A transformation function $\tau \equiv Src \Rightarrow Tgt$ is a function that takes an instantiation of the source template program Src and returns the target template program Tgt properly instantiated. An instantiation of a template program is a mapping from all the template statements/expressions to concrete (without templates) statements/expressions, respectively. Section 4.2 presents the language used to write template programs.

The precondition ψ is a sufficient condition that makes τ semantics-preserving. Ideally, ψ should also be a necessary condition, i.e., ψ should be the weakest precondition. Section 4.3 describes the language of preconditions that is considered in this work.

Finally, h is a profitability heuristic, which states under which conditions the compiler should apply τ, since τ may not always be performance improving (or even performance preserving). Profitability heuristics will be ignored for the rest of this document because they do not interfere with the correctness of optimizations.

4.2 Transformation Functions

The language proposed in this section is targeted for the specification of transformation functions, and in particular compiler optimizations. The language is heavily inspired in PEC [KTL09, TL10a].

Each transformation function is described by a source and target program templates, representing the original and the transformed program fragments, respectively.

4.2.1 The Language by Example

An example of a transformation function is given below. On the left of the arrow, we write the source template, and on the right we write the target template, meaning that a program fragment matching the

source template should be replaced with the appropriate instantiation of the target template.

$\mathsf{X} := \mathsf{E}$ $\mathsf{X} := \mathsf{X} + 1$	$\Rightarrow$	$\mathsf{X} := \mathsf{E} + 1$

The function specified above performs the following transformation: when matching a sequence of one assignment plus an increment by one to a given variable, transform the code so that the assignment and increment are done in a single statement.

X is a template variable and can match any program variable, and E is a template expression. Template expressions can only be instantiated with program expressions without side-effects (meaning that their computation cannot modify the program state — a valuation of the program variables). When applying a transformation function, template variables and expressions must be instantiated with concrete program variables and expressions, respectively.

For example, it is possible to apply the transformation function given above to the following program fragment (in C), provided that the template variable X is instantiated with the program's variable "`v`", and the template expression E is instantiated with the program's expression "`i + 2`".

```
...
v = i + 2;
v = v + 1;
...
```

The resulting program after applying the transformation function is the following:

```
...
v = i + 2 + 1;
...
```

The specification language also supports more complex constructors, such as template boolean expressions (B_i), template statements (S_i), conditionals, and loops. For example, the following transformation function rewrites a loop with a branch in the body into a loop with no branching:

while $\mathsf{I} < \mathsf{N}$ **do**
  if B **then**
  S
  else
  S
  $\mathsf{I} := \mathsf{I} + 1$
	$\Rightarrow$	**while** $\mathsf{I} < \mathsf{N}$ **do**
		  S
		  $\mathsf{I} := \mathsf{I} + 1$

The template statement S is a placeholder for an arbitrary statement, and can be instantiated, for example, with an assignment, a compound statement, a loop, a function call, and so on. B is a template boolean expression, and can be instantiated with any program (side-effect free) expression that evaluates to a boolean value.

4.2.2 Language Grammar

The grammar in Backus-Naur form (BNF) for the proposed specification language is shown in Figure 4.1.

$TFunc$ describes a transformation function, going from a source statement to a target statement.

A statement ($Stmt$) can be either a compound statement, a branching statement, a looping statement, an assignment statement, a template statement ($TStmt$), or a do-nothing statement (**skip**).

A template statement ($TStmt$) is a placeholder for an arbitrary (and possibly non-deterministic) concrete statement. A template statement can be instantiated with an arbitrary program statement, including an assignment, a function call, a loop, a compound statement, etc. Template statements are represented by literals S, S_1, S_2, and so on.

Expressions are divided in two categories: integer expressions (Exp), and boolean expressions ($BExp$). Integer expressions can be either an application of a binary operator ($BinOp$), a template expression ($TExp$), a read of a left-hand side expression (LHS), or a number. Boolean expressions can be any of the following: an application of a boolean binary operator ($BBinOp$), a negation of a boolean expression, a comparison between integer expressions, a template boolean expression ($TBExp$), or a constant. Template expressions are placeholders for arbitrary program expressions, and whose value is unknown.

Left-hand side expressions (LHS) can only be represented by a template variable (Var) at the moment. Support for pointers and arrays is left for future work. A template variable can be instantiated with any program variable, register or memory location.

4.2.3 Language Semantics

The semantics of the specification language just presented is mostly as expected. A transformation function states how each template statement/expression is transformed (e.g., moved, duplicated, eliminated) to produce the optimized program. Only a few non-obvious details are described next.

First, all variables are assumed to be signed integers (i.e., in $\mathbb{Z}$). Therefore, there is no need to reason about overflows (although the language can be extended in the future to support different types of variables and expressions).

A template expression can be instantiated with a constant, or with an arbitrary algebraic expression that depends on several variables, for example. Template expressions are side-effect free, and therefore they cannot write to any memory location, nor raise exceptions nor trap. Such erroneous behavior must be explicitly modeled in the control flow and/or with assignments to control variables.

In terms of template code, a given template object may appear several times in a source template. For example, a template expression E may get assigned to two different variables. The concrete code used to instantiate each of the two expression appearances must be semantically equivalent, although it may differ syntactically. If there are multiple occurrences of a template object in the source template, the transformation function is free to choose any of them to produce the target code (since they are semantically equivalent), or even to synthesize a new semantically equivalent object.

$$
\begin{aligned}
\mathit{TFunc} &::= \mathit{Stmt} \;\Rightarrow\; \mathit{Stmt} \\
\\
\mathit{Stmt} &::= \mathit{Stmt} \;;\; \mathit{Stmt} \\
&\quad \mid \textbf{if } \mathit{BExp} \textbf{ do } \mathit{Stmt} \textbf{ else } \mathit{Stmt} \\
&\quad \mid \textbf{while } \mathit{BExp} \textbf{ do } \mathit{Stmt} \\
&\quad \mid \mathit{LHS} \;:=\; \mathit{Exp} \\
&\quad \mid \mathit{TStmt} \\
&\quad \mid \textbf{skip} \\
\\
\mathit{TStmt} &::= \mathsf{S} \mid \mathsf{S}_1 \mid \mathsf{S}_2 \mid \ldots \\
\\
\mathit{LHS} &::= \mathit{Var} \\
\\
\mathit{Var} &::= \mathsf{I} \mid \mathsf{J} \mid \mathsf{N} \mid \mathsf{X} \mid \mathsf{V}_1 \mid \mathsf{V}_2 \mid \ldots \\
\\
\mathit{Exp} &::= \mathit{Exp}\ \mathit{BinOp}\ \mathit{Exp} \\
&\quad \mid \mathit{TExp} \\
&\quad \mid \mathit{LHS} \\
&\quad \mid (\ \mathit{Exp}\ ) \\
&\quad \mid \mathit{integer} \\
\\
\mathit{TExp} &::= \mathsf{E} \mid \mathsf{E}_1 \mid \mathsf{E}_2 \mid \ldots \\
\\
\mathit{BinOp} &::= + \mid - \mid \times \mid \div \mid \ldots \\
\\
\mathit{BExp} &::= \mathit{BExp}\ \mathit{BBinOp}\ \mathit{BExp} \\
&\quad \mid \neg\ \mathit{BExp} \\
&\quad \mid \mathit{Exp}\ \mathit{CmpOp}\ \mathit{Exp} \\
&\quad \mid \mathit{TBExp} \\
&\quad \mid (\ \mathit{BExp}\ ) \\
&\quad \mid \mathsf{true} \\
&\quad \mid \mathsf{false} \\
\\
\mathit{BBinOp} &::= \wedge \mid \vee \mid \ldots \\
\\
\mathit{CmpOp} &::= < \mid \leq \mid > \mid \geq \mid = \mid \neq \mid \ldots \\
\\
\mathit{TBExp} &::= \mathsf{B} \mid \mathsf{B}_1 \mid \mathsf{B}_2 \mid \ldots
\end{aligned}
$$

Figure 4.1: BNF grammar of the transformation function DSL.

4.3 Preconditions

Transformation functions may have preconditions restricting the possible instantiations of template statements, template expressions, and template variables. In this section, we present the language that is considered for specifying preconditions for transformation functions.

4.3.1 Example

Consider the following transformation function τ_1:

$$\frac{\begin{array}{l}\mathsf{S} \\ \mathsf{V} := \mathsf{E}\end{array}}{\Rightarrow} \quad \begin{array}{l}\mathsf{V} := \mathsf{E} \\ \mathsf{S}\end{array}$$

As an example, we apply the transformation function τ_1 to the following program fragment:

$x := 0$
$v := x + 1$

The output of the transformation function is (with the instantiation $\mathsf{S} \mapsto x := 0,\ \mathsf{E} \mapsto x + 1,\ \mathsf{V} \mapsto v$):

$v := x + 1$
$x := 0$

The transformed program is not equivalent to the original one, since if the initial value of x is not zero, then the programs will yield different values for v. Therefore, a necessary (but not sufficient) precondition to ensure that the transformation function is always semantics-preserving is that S cannot write to a variable that is read by E.

Preconditions of transformation functions are specified as constraints over the read and write sets of template statements/expressions, which contain the variables that the template statements/expressions *may* read and write, respectively.

For the previous example, we could use the constraint $\mathsf{W}(\mathsf{S}) \cap \mathsf{R}(\mathsf{E}) = \emptyset \wedge \mathsf{V} \notin \mathsf{R}(\mathsf{S}) \wedge \mathsf{V} \notin \mathsf{W}(\mathsf{S})$ as the precondition. In other words, this precondition states that statement S cannot write to any variable read by expression E and that S cannot read nor write to variable v. This precondition is sufficient to ensure that the transformation function τ_1 is always semantics-preserving, and it would therefore rule out the instantiation above. Moreover, this precondition is the *weakest* in the language of read and write sets (as confirmed by the tool described in Chapter 5).

4.3.2 Language

Preconditions of transformation functions restrict the input/output relation of template statements and expressions. This in turns limits the possible instantiations on which the transformation function can operate.

In this work, preconditions of transformation functions are modeled as quantifier-free constraints to the read and write sets of each template statement/expression. Preconditions over template variables may restrict their initial value, and/or possible usage after the source code fragment.

Let $\mathsf{Vars}(\tau)$ be the set of variables of the transformation function τ. All variables in $\mathsf{Vars}(\tau)$ are integer valued (in $\mathbb{Z}$). If V is a template variable appearing in either the source or target template of τ, then $\mathsf{V} \in \mathsf{Vars}(\tau)$.

For each template statement S (resp. expression E), its read set, $\mathsf{R}(\mathsf{S}) \subseteq \mathsf{Vars}(\tau)$ (resp. $\mathsf{R}(\mathsf{E}) \subseteq \mathsf{Vars}(\tau)$), is defined as the set of variables that S (resp. E) *may* read.

Similarly, let $\mathsf{W}(\mathsf{S}) \subseteq \mathsf{Vars}(\tau)$ be the write set of S. Since template expressions are side-effect free, their write set is empty, i.e., for every template expressions E_i and B_i, we have $\mathsf{W}(\mathsf{E}_i) = \emptyset$ and $\mathsf{W}(\mathsf{B}_i) = \emptyset$.

Constraints over read and write sets can only use certain operations, such as union, intersection, membership, containment, and so on. Quantification is not supported.

4.3.3 Common Idioms

Examples of preconditions for real optimizations are given in Table 5.1. Here, a few common idioms of preconditions are exemplified.

For instance, to specify that template statements S_1 and S_2 may commute, it is necessary and sufficient that $\mathsf{R}(\mathsf{S}_1) \cap \mathsf{W}(\mathsf{S}_2) = \mathsf{W}(\mathsf{S}_1) \cap \mathsf{R}(\mathsf{S}_2) = \mathsf{W}(\mathsf{S}_1) \cap \mathsf{W}(\mathsf{S}_2) = \emptyset$.

To specify that a statement S does not write to any variable which an expression E depends, we write $\mathsf{W}(\mathsf{S}) \cap \mathsf{R}(\mathsf{E}) = \emptyset$.

To state that an expression E is loop invariant, we need to have $\mathsf{W}(\mathsf{S}_i) \cap \mathsf{R}(\mathsf{E}) = \emptyset$ for every statement S_i in the loop body.

Finally, a statement S is idempotent if when executed multiple times produces the same result as if executed only once. In terms of read and write sets, we have that S is idempotent if $\mathsf{W}(\mathsf{S}) \cap \mathsf{R}(\mathsf{S}) = \emptyset$.

4.3.4 Discussion

We believe the specification language for preconditions just presented is suitable for the task for several reasons. First, both compiler developers and compiler books already informally talk about read and write sets (even if only implicitly) when arguing about correctness of an optimization. Therefore, the language of read and write sets is a good means to communicate with compiler developers.

Second, we believe that proof obligations arising from the verification of such preconditions by compilers can be efficiently discharged using standard compiler technology, which has been tuned over the last decades. In particular, analyses like pointer analysis, scalar evolution, range analysis, an so on, can be used to discharge such proof obligations. More heavyweight software verification technology can also be employed if additional reasoning is required.

Being able to quickly verify whether a given code instantiation satisfies a precondition is of extreme importance, since compilation time must be reasonable. Although different developers will have different definitions for acceptable compilation time, compiling a project should not take years.

Previous work in the area of optimization verification, most notably PEC [KTL09, TL10a], also used a similar specification language with success.

An important question is whether it is always possible to specify the weakest precondition of an optimization in the proposed precondition language. The answer is no. For example, the constraint to specify commutativity between statements given in the previous section is not the weakest. Two statements may also be commutative if they perform algebraic operations that are commutative. For example, the instantiation $\mathsf{S}_1 \mapsto x := x + a,\ \mathsf{S}_2 \mapsto x := x + b$ does not satisfy the precondition for commutativity given previously, but statements S_1 and S_2 may in fact execute in any order.

Another case where the proposed precondition language is not sufficiently expressive is for peephole optimizations. These optimizations often require reasoning about bitwise operations, which are not supported by the proposed language. Work needs to be done in order to identify a reasonable set of predicates that can cover this type of optimizations.

4.4 Definitions

A few additional definitions used throughout the document are now given.

Let $\mathsf{Tmpl}(\tau)$ be the set of template statements/expressions of the transformation function τ. Let $\mathsf{Stmts}(\tau) \subseteq \mathsf{Tmpl}(\tau)$ be the set of template statements of the transformation function τ. In the example of the previous section, we have $\mathsf{Tmpl}(\tau_1) = \{\mathsf{S}, \mathsf{E}\}$ and $\mathsf{Stmts}(\tau_1) = \{\mathsf{S}\}$.

Program Paths A program path π is a sequence of straight-line statements (no loops nor **if** statements) and boolean expressions. For example, the path $i := 0 \ ; \ i < n \ ; \ i := i + 1 \ ; \ i \geq n$ could be a 1-step unrolling of a simple counting loop. We naturally extend $\mathsf{Stmts}(\pi)$ and $\mathsf{Tmpl}(\pi)$ to program paths.

Context Variables Let $c_i \in C$ be a context variable. These variables c_i represent the variables that are possibly in scope where a program template may be instantiated (possibly none) and that do not appear in the transformation function. Variables c_i must be distinct from every template variable.

For every template statement S_i, we have that the context variable $c_i = \mathsf{CtxVar}(\mathsf{S}_i)$ is always in its write set, i.e., $c_i \in \mathsf{W}(\mathsf{S}_i)$. Therefore, each template statement *may* write to at least one distinct context variable, and hence we have that $|C| \geq |\mathsf{Stmts}(\tau)| + 1$. The additional context variable represents the variables potentially present in an instantiation but not in the transformation function (i.e., they are in the context), and that are not written to within the instantiation code fragment.

For the example of Section 4.3.1 we need two context variables and therefore we have $C = \{c_1, c_2\}$. Variable c_1 represents, e.g., the effects of S on x. While variable x does not appear explicitly in the transformation function, S does indeed modify x in the example instantiation. Variable c_2 represents all the other variables of a program that may be used to instantiate the template that are in scope (possibly none) that can be read by E and that cannot be written by S.

Let $\mathsf{Vars}(\tau)$ and $\mathsf{Vars}(\pi)$ be the set of variables in a transformation function τ or in a path π, respectively. Moreover, context variables are contained in these sets, i.e., $C \subseteq \mathsf{Vars}(\tau)$. In our example, we have $\mathsf{Vars}(\tau_1) = \{\mathsf{V}, c_1, c_2\}$.

Miscellaneous Throughout this document, the constraint $x = \text{ite}(a, b, c)$ is used as the usual shorthand for $(a \rightarrow x = b) \wedge (\neg a \rightarrow x = c)$.

4.5 Summary

In this chapter, we presented a declarative language for the specification of code transformations. In this language, transformations are specified as a rewrite between two code templates.

Secondly, we presented a language for the specification of preconditions of code transformations, which is based on read and write sets of template statements and expressions.

Together, these languages enable succinct specifications of compiler optimizations, and enable the usage of software verification algorithms to ensure correctness, as shown in the following chapters.

Chapter 5

Synthesizing Weakest Preconditions for Compiler Optimizations

In this chapter, a new algorithm for the automatic synthesis of provably *weakest* preconditions for compiler optimizations is given. It is, to the best of my knowledge, the first known algorithm specifically designed for this task.

Preconditions are specified in the language of read and write sets of template statements and expressions, which we believe to be adequate to express the most used conditions in the domain of compiler optimizations.

The algorithm works in a counterexample-driven way. It requires a black box that can prove the correctness of a compiler optimization, or produce a counterexample otherwise (in the form of two paths going through the source and target templates). The algorithm starts with the precondition true and processes a counterexample to produce the weakest precondition guaranteed to prevent that counterexample. The algorithm then strengthens the precondition and repeats this process until no more counterexamples can be found, i.e., until the precondition is strong enough (which is guaranteed to occur, since the set of possible preconditions is finite).

The presented algorithm is more generally applicable than the domain of compiler optimizations. The algorithm can synthesize weakest preconditions for any problem where the set of preconditions is finite, although possibly too large to test each case individually, as long as there exists an oracle that can prove correctness or produce counterexamples if not correct.

We evaluate the proposed algorithm on a set of optimizations, and we present the synthesized preconditions. As an example, we also show a precondition that we synthesized automatically that is weaker than what had been published previously by experts.

5.1 Illustrative Example

We illustrate our algorithm to synthesize weakest preconditions for compiler optimizations on a simple example.

Figure 5.1 shows an optimization known as loop unswitching. The goal of this optimization is to remove a potentially expensive expression evaluation from a loop body (where it could be performed many times) and perform the evaluation just once before the loop.

Loop unswitching may improve the performance of a program, since expression B will be evaluated potentially fewer times (if the loop is executed). On the other hand, loop unswitching may increase the overall code size since it duplicates the remaining code in the loop (although not necessarily, since it may

Figure 5.1: Loop unswitching: the source template is on the left, and the target template is on the right.

expose more optimization opportunities). Increasing the code size may be highly undesirable for certain applications (e.g., embedded systems). The decision of whether to apply a given optimization, and loop unswitching in particular, is usually the responsibility of a profitability heuristic.

Intuitively, loop unswitching as described in Figure 5.1 looks correct iff B evaluates to the same boolean value in every iteration (i.e., B must be loop invariant).

Since it is not known what the template statements and expressions exactly do (this is only defined when the optimization is applied to a specific piece of code), we need to derive a generic precondition to restrict their operation to guarantee that the transformation will be always semantics-preserving.

The language of preconditions we use for compiler optimizations is that of read and write sets of template statements/expressions. For example, to state that the template expression B cannot read variable I we use the notation $\mathsf{I} \notin \mathsf{R}(\mathsf{B})$. This condition is actually necessary (but not sufficient) for the precondition of loop unswitching.

Our algorithm works in a counterexample-driven way. We require the existence of a black box that can prove the correctness of compiler optimizations, or produce a counterexample if the optimization is not correct. One such algorithm is given in Chapter 6.

We consider counterexamples in the form of two program paths, with one being a path through the source template, and the other through the target template. Counterexample paths can be generated by, e.g., doing a breadth-first search (BFS), a depth-first search (DFS), or a hybrid of these, to discover paths that are not equivalent. The algorithm is naturally oblivious to the technique employed for counterexample discovery.

For our example, starting with the precondition $P = \mathsf{true}$, we could get the following counterexample paths π_1 and π_2 (respectively, for the source and target templates):

$$\mathsf{I} < \mathsf{N} \ ; \ \mathsf{B} \ ; \ \mathsf{S}_1 \ ; \ \mathsf{I} := \mathsf{I} + 1 \ ; \ \mathsf{I} < \mathsf{N} \ ; \ \neg\mathsf{B} \ ; \ \mathsf{S}_2 \ ; \ \mathsf{I} := \mathsf{I} + 1 \ ; \ \mathsf{I} \geq \mathsf{N}$$

and:

$$\mathsf{B} \ ; \ \mathsf{I} < \mathsf{N} \ ; \ \mathsf{S}_1 \ ; \ \mathsf{I} := \mathsf{I} + 1 \ ; \ \mathsf{I} < \mathsf{N} \ ; \ \mathsf{S}_1 \ ; \ \mathsf{I} := \mathsf{I} + 1 \ ; \ \mathsf{I} \geq \mathsf{N}$$

Without any knowledge about S_1 and S_2, these paths are clearly a counterexample since S_1 and S_2 execute a different number of times in the source and target templates. For example, the instantiation $\mathsf{I} \mapsto i,\ \mathsf{S}_1 \mapsto i := i + 1,\ \mathsf{S}_2 \mapsto i := i + 2,\ \mathsf{B} \mapsto i \leq 0$ makes the paths of the source and target programs terminate with different values for variable i. Therefore, we need to constrain the set of possible instantiations of S_1 and S_2 with a suitable precondition.

To generate a precondition for a given counterexample, we first encode the counterexample paths into logic in the usual way (process commonly known as VC Gen), with the variables of the target template being renamed in order to be different from the variables used in the source template. We explain only how template statements and expressions are encoded.

Each template statement is encoded as a conditional assignment of a fresh variable to each variable

that may potentially be in the write set of that statement. Then, for each pair of the same template symbol, we assert that their corresponding fresh variables are equal iff the values of their corresponding input variables that are in the read set are equal. Similarly, template expressions are replaced by fresh variables.

For our counterexample, we obtain the following constraint ϕ_1 for the source path (with $\mathsf{Vars}(\pi_1) = \{I, N, c_1, c_2, c_3\}$):

$$
\begin{aligned}
& I_0 < N_0 \wedge \\
& B_0 \wedge \\
& I_1 = \text{ite}(w^I_{S1}, S1^I_0, I_0) \wedge N_1 = \text{ite}(w^N_{S1}, S1^N_0, N_0) \wedge \\
& c1_1 = \text{ite}(w^{c1}_{S1}, S1^{c1}_0, c1_0) \wedge c2_1 = \text{ite}(w^{c2}_{S1}, S1^{c2}_0, c2_0) \wedge c3_1 = \text{ite}(w^{c3}_{S1}, S1^{c3}_0, c3_0) \\
& \wedge I_2 = I_1 + 1 \wedge \\
& I_2 < N_1 \wedge \\
& \neg B_1 \wedge \\
& I_3 = \text{ite}(w^I_{S2}, S2^I_0, I_2) \wedge N_2 = \text{ite}(w^N_{S2}, S2^N_0, N_1) \wedge \\
& c1_2 = \text{ite}(w^{c1}_{S2}, S2^{c1}_0, c1_1) \wedge c2_2 = \text{ite}(w^{c2}_{S2}, S2^{c2}_0, c2_1) \wedge c3_2 = \text{ite}(w^{c3}_{S2}, S2^{c3}_0, c3_1) \\
& \wedge I_4 = I_3 + 1 \wedge \\
& I_4 \geq N_2
\end{aligned}
$$

The encoding (ϕ_2) of the target path is similar.

The boolean variables w^v_s mean that statement s writes to variable v. This is required because $v \in \mathsf{W}(s)$ states that s *may* write to variable v, but it is not mandatory to do so. We define ϕ_w to be the conjunction of the following set of constraints (for each pair of template statement s and variable v):

$$w^v_s \rightarrow v \in \mathsf{W}(s)$$

We now generate the constraints ϕ_u that assert when the fresh values generated from the template statements/expressions are equal. These constraints are akin to Ackermann's reduction for uninterpreted function symbols [Ack54]. For example, to state when B_0 and B_1 are equal, we use the following constraint:

$$
\begin{aligned}
& \big((I \in \mathsf{R(B)} \rightarrow I_0 = I_2) \wedge (N \in \mathsf{R(B)} \rightarrow N_0 = N_1) \wedge (c_1 \in \mathsf{R(B)} \rightarrow c1_0 = c1_1) \wedge \\
& (c_2 \in \mathsf{R(B)} \rightarrow c2_0 = c2_1) \wedge (c_3 \in \mathsf{R(B)} \rightarrow c3_0 = c3_1)\big) \rightarrow B_0 = B_1
\end{aligned}
$$

Set membership constraints ($x \in y$) are encoded as boolean variables.

Now that we have all the necessary constraints, we construct the following formula ϕ and give it to an SMT solver:

$$\forall V : (\phi_1 \wedge \phi_2 \wedge \phi_w \wedge \phi_u \rightarrow I_4 = I_8 \wedge N_2 = N_4 \wedge c1_2 = c1_4 \wedge c2_2 = c2_4 \wedge c3_2 = c3_4)$$

where I_4/I_8, N_2/N_4, $c1_2/c1_4$, $c2_2/c2_4$, and $c3_2/c3_4$ are the final values of the $I/N/c_1/c_2/c_3$ variables of the source and target templates, respectively. V is the set of variables that are universally quantified. These include the variables w^v_s, and all the fresh variables created by the path encoding process (e.g., $S1^I_0$, $S1^N_0$, etc).

Giving this formula to an SMT solver will yield assignments to the boolean variables corresponding to the read and write sets' membership (the only existentially quantified variables). Each set of these

assignments (a model of the formula) is a sufficient precondition that makes the two paths equivalent (or infeasible).

For this formula, we may obtain the model $\mathsf{R}(\mathsf{B}) = \emptyset \wedge \mathsf{W}(\mathsf{S}_1) = \{c_1\} \wedge \mathsf{W}(\mathsf{S}_2) = \{c_2\}$. Although this condition is certainly sufficient to make the first path infeasible (because it implies that B is a constant, and therefore it cannot evaluate to two different values), this condition is not the weakest.

As we stated before, our algorithm is iterative and so it keeps weakening the counterexample's precondition until it gets the weakest precondition. We do so by negating each model, adding it to formula ϕ, and then retrieving another model from the SMT solver. We stop when there are no more models (i.e., the conjunction of ϕ and the negation of all the previously discovered models is unsatisfiable). The weakest precondition for the counterexample is the disjunction of all models.

After processing one counterexample, we strengthen the transformation function's weakest precondition with the counterexample's weakest precondition. We iterate until there are no more counterexamples, i.e., until the transformation function is correct.

The algorithm always terminates because the precondition is strengthened when each counterexample is processed and because the language of preconditions we consider is finite.

Finally, the precondition we obtain for our example (loop unswitching) after processing all the counterexamples is the following:

$$P = \mathsf{I} \notin \mathsf{R}(\mathsf{B}) \wedge \mathsf{W}(\mathsf{S}_1) \cap \mathsf{R}(\mathsf{B}) = \emptyset \wedge \mathsf{W}(\mathsf{S}_2) \cap \mathsf{R}(\mathsf{B}) = \emptyset$$

This precondition is the weakest in the language of read and write sets.

Optimizations The algorithm we just presented informally will usually take significant time to terminate, since it will usually enumerate many models. We present two optimizations that improve the speed of convergence significantly, as well as improve the compactness of the generated preconditions.

The first optimization we perform is model weakening, meaning that given a model generated by an SMT solver, we try to make it weaker (more general) by dropping literals from it (which are therefore "don't cares"). For a model μ of ϕ, we know that $\neg\phi \wedge (\bigwedge l \in \mu)$ is unsatisfiable. Moreover, if for some literal l', $\neg\phi \wedge (\bigwedge l \in \mu')$ is still unsatisfiable, where $\mu' = \mu \setminus \{l'\}$, then we know that both $\mu' \cup \{l'\}$ and $\mu' \cup \{\neg l'\}$ are models of ϕ. Therefore, μ' is a weaker (partial) model of ϕ. We leverage this knowledge to iterate over each of the literals in a model to check which ones can be removed.

The second optimization we perform is to add additional constraints to ϕ that represent common precondition patterns. In particular, we noticed that stating that the intersection of read/write sets must be empty (e.g., $\mathsf{W}(\mathsf{S}_1) \cap \mathsf{R}(\mathsf{B}) = \emptyset$) is a common pattern. We therefore associate a boolean variable to each of such constraints, and try to bias the weakening of the models (as previously described) towards these variables. This optimization not only produces more compact preconditions, but also reduces the number of models considerably (since it avoids enumerating all the models that correspond to the constraint they succinctly imply).

5.2 The Algorithm

Our precondition synthesis algorithm, named PSyCO, is counterexample-guided. The algorithm relies on a verification tool as a black box that can prove the correctness of optimizations or return a counterexample otherwise (such as the one given in Chapter 6).

In order not to restrict the generated preconditions, we require the algorithm to be run with at least one more context variable than template statements, i.e., for transformation function τ we must have $|C| \geq |\mathsf{Stmts}(\tau)| + 1$. This lower bound on the size of C is sufficient to express all combinations of

```
     function PSyCO
     input
        τ – transformation function
     vars
        ψ – generated precondition
     begin
1       ψ := true
2       repeat
3           match CheckTF(τ, ψ) with
4            | correct ->
5               return ψ
6            | counterexample (π1, π2) ->
7               ψ := ψ ∧ SynthWP(π1, π2)
     end.
```

Figure 5.2: PSyCO algorithm.

constraints of the form $\mathsf{R}(t) \cap \mathsf{W}(s_1) = \emptyset$ and $\mathsf{W}(s_1) \cap \mathsf{W}(s_2) = \emptyset$ (and their respective negations) for any template t and statements s_1 and s_2. Constraints of the form $\mathsf{R}(t_1) \cap \mathsf{R}(t_2) = \emptyset$ are not considered, since we are not aware of any optimization requiring such kind of preconditions.

5.2.1 PSyCO

The pseudo-code for the PSyCO algorithm is shown in Figure 5.2. The algorithm takes as input a transformation function τ and returns the corresponding weakest precondition ψ. Starting with the precondition true, the algorithm iteratively calls the CheckTF function that checks whether τ is correct under the given precondition or returns a counterexample otherwise (given as two paths, π_1 and π_2, of the source and target programs, respectively). The CheckTF function is a black box given as input.

At each step, PSyCO strengthens the precondition with a condition that is sufficient (and necessary) to discharge the counterexample. Therefore, a given counterexample is never seen more than once. Since the number of possible combinations of preconditions in the considered language is finite and we keep strengthening the precondition at each step, we have that PSyCO terminates (assuming that CheckTF always terminates).

5.2.2 SynthWP

Figure 5.3 shows the function SynthWP that takes two paths, π_1 and π_2, as input, and returns the weakest precondition that makes the two paths equivalent.

The idea is to construct a universally quantified formula such that a model for it guarantees that the two paths are equivalent (or either one becomes infeasible) for all possible program inputs. The union of all such models is the weakest precondition. The models can be generated with an off-the-shelf SMT solver.

SynthWP starts by generating a formula that corresponds to each of the counterexample paths (lines 3 and 4). This is done using standard techniques, that we do not describe here. VCGen takes as input a path π, a map σ_0 with the initial (symbolic) values of the program variables, a set of variables V containing the variables of the source path, and a map w containing a fresh boolean variable for each pair of statements and variables. If a certain statement writes to a given program variable, its corresponding boolean variable in w will be true.

VCGen replaces each template statement s with the following constraint (a conditional assignment

function SynthWP
input
π_1, π_2 – counterexample paths
vars
ψ – generated precondition
begin

1 $\sigma_0 := \{v \mapsto \text{fresh integer var} \mid v \in \mathsf{Vars}(\pi_1; \pi_2)\}$

2 $w := \{(s, v) \mapsto \text{fresh boolean var} \mid s \in \mathsf{Stmts}(\pi_1; \pi_2) \wedge v \in \mathsf{Vars}(\pi_1; \pi_2)\}$

3 $\phi_1, \sigma_1, u_1 := \text{VCGen}(\pi_1, \sigma_0, \mathsf{Vars}(\pi_1), w)$

4 $\phi_2, \sigma_2, u_2 := \text{VCGen}(\pi_2, \sigma_0, \mathsf{Vars}(\pi_1), w)$

5 $\phi_u := \bigwedge_{(\sigma,v,t),(\sigma',v',t') \in (u_1 \cup u_2) \wedge t = t'} \Big(\Big(\bigwedge_{v'' \in \mathsf{Vars}(\pi_1)} (\mathcal{B}(v'' \in \mathsf{R}(t)) \rightarrow \sigma(v'') = \sigma'(v'')) \Big) \rightarrow v = v' \Big)$

6 $\phi_w := \bigwedge_{((s,v) \mapsto l) \in w} (l \rightarrow \mathcal{B}(v \in \mathsf{W}(s)))$

7 $\phi_c := \bigwedge_{s \in \mathsf{Stmts}(\pi_1; \pi_2)} (\mathcal{B}(\mathsf{CtxVar}(s) \in \mathsf{W}(s)))$

8 $\phi_d, d := \text{MkDisj}(\pi_1; \pi_2)$

9 $V := \{\sigma_0(v) \mid v \in \mathsf{Vars}(\pi_1; \pi_2)\} \cup \{l \mid ((s, v) \mapsto l) \in w\} \cup \{v \mid (\sigma, v, t) \in (u_1 \cup u_2)\} \cup d$

10 $\phi := \forall V \ \Big(\phi_u \wedge \phi_w \wedge \phi_c \wedge \phi_d \wedge \phi_1 \wedge \phi_2 \rightarrow \bigwedge_{v \in \mathsf{Vars}(\pi_1)} (\sigma_1(v) = \sigma_2(v)) \Big)$

11 $\mu_f := \{\neg\mathcal{B}(v \in \mathsf{R}(t)) \mid v \in \mathsf{Vars}(\pi_1; \pi_2) \wedge t \in \mathsf{Tmpl}(\pi_1; \pi_2)\} \cup \{\neg\mathcal{B}(v \in \mathsf{W}(s)) \mid v \in \mathsf{Vars}(\pi_1; \pi_2) \wedge s \in \mathsf{Stmts}(\pi_1; \pi_2)\} \cup d$

12 $\psi := \mathsf{false}$

13 **while** $\phi \wedge \neg\psi$ is satisfiable **do**

14 $\psi := \psi \vee \text{GeneralizeWP}(\phi, \text{GetModel}(\phi \wedge \neg\psi) \cap \mu_f, d)$

15 **return** ψ

end.

Figure 5.3: SynthWP algorithm.

to all variables that may potentially be in the write set of s):

$$\bigwedge_{v \in V} (v' = \text{ite}(w(s, v), fv, v))$$

where v' is the new value of v, $w(s, v)$ is the value associated with (s, v) in map w (the so called must-write variables), fv is a fresh variable (one per variable v), and $V \subseteq \mathsf{Vars}(\pi_1)$ is the set of all variables that can potentially be in the write set of s. Template expressions are replaced with a fresh variable.

VCGen returns a formula corresponding to the input path, a map σ with the final value of each of the program variables and a set u. The set u contains triples (σ, v, t), one per each fresh variable v created for template statement/expression t, with σ being a map with the value of the variables at the point where the template statement/expression was evaluated.

In line 5, we generate a formula akin to Ackermann's reduction for uninterpreted function symbols [Ack54]. We call it conditional Ackermannization, since that if we fix the read set of t, $\mathsf{R}(t)$, the resulting constraint is equal to Ackermann's reduction. In this case, the read set $\mathsf{R}(t)$ is not known and that is precisely what we are looking for to synthesize. As an optimization, variables v'' that are not modified by either path (π_1 or π_2) do not need to be considered, since then $\sigma(v'') = \sigma'(v'')$ becomes a tautology.

Conditional Ackermannization (line 5) works as follows. For each pair of triples (σ, v, t) and (σ', v', t') in w coming from the same template (i.e., $t = t'$), we assert that the fresh variables v and v' must be equal if each of the variables in the read set of t has the same value in σ and σ'. We use the notation $\mathcal{B}(x \in y)$ to introduce a boolean variable that represents that $x \in y$.

In line 6, we generate a constraint that asserts that a template statement can only write to a variable

v if v is in its write set. The constraint generated in line 7 asserts that each template statement s_i has at least one distinct context variable c_i in its write set. This is an important optimization, since it avoids the generation of multiple equivalent models that are equal modulo a renaming of the context variables.

In line 8, we introduce a set of boolean variables to represent intersections of read and write sets. In particular, we consider constraints of the form $\mathsf{W}(s_1) \cap \mathsf{W}(s_2) = \emptyset$, $\mathsf{R}(t) \cap \mathsf{W}(s_1) = \emptyset$, and $\mathsf{W}(s_1) \cap \mathsf{W}(s_2) \cap \mathsf{R}(t) = \emptyset$ for every tuple of template statements s_1, s_2, and s_3 and template statements/expressions t. This is an optimization that enables us to more succinctly express preconditions of these forms without having to enumerate all the possible combinations of read and write sets that satisfy the corresponding constraint.

For a path π, MkDisj generates constraints of the form:

$$\bigwedge\nolimits_{s,s' \in \mathsf{Stmts}(\pi)} \left(\left(\neg \bigvee\nolimits_{v \in \mathsf{Vars}(\pi)} (\mathcal{B}(v \in \mathsf{W}(s)) \wedge \mathcal{B}(v \in \mathsf{W}(s'))) \right) \leftrightarrow fv \right)$$

with fv being a fresh variable (one per each pair (s, s')). MkDisj generates similar constraints for every tuple of read and write sets as mentioned previously. In addition to the generated constraint, MkDisj returns the set of fresh variables used.

In line 9, we collect the set of variables that will be universally quantified, namely the set of initial values of the variables and the set of fresh variables used in previous steps. The remaining variables (the booleans representing set membership and the variables in d) are implicitly existentially quantified. Finally, in line 10, we assemble the final constraint. It states that either one of the paths is infeasible or the final value of the variables of the two paths must be equal.

In line 11, we compute a model filter, since we are only interested in negative membership constraints and empty intersection constraints (d). We do not need to consider positive membership constraints, since e.g., $v \in \mathsf{R}(t)$ means that t *may* read v (but not necessarily). Therefore, a model μ including a positive membership constraint l, e.g., $\mu = \mu' \cup \{l\}$, implies that $\mu' \cup \{\neg l\}$ is also a model.

Lines 12–14 implement the main synthesis loop. We iterate over the models of the formula ϕ (filtered by μ_f) and generalize each one in the hope that we will produce more succinct preconditions and converge faster. We pass the set of variables d to GeneralizeWP, so that it can bias the result and express it over more literals of d whenever possible. Function GetModel is given by the SMT solver and returns a model for the formula given as input.

Encoding Size In the worst case, the size of formula ϕ is dominated by either ϕ_u or ϕ_d, since these are the only constraints that grow quadratically or cubically with the size of the input. Given a counterexample (π_1, π_2), the worst-case size of ϕ_u is $O\left(\left(|\pi_1 \ ; \ \pi_2| \cdot |\mathsf{Vars}(\pi_1)|\right)^2\right)$. This case corresponds to paths π_1, π_2 being equal to a sequence of a single template statement (e.g., $\mathsf{S} \ ; \ \mathsf{S} \ ; \ \ldots \ ; \ \mathsf{S}$). The quadratic growth with the number of variables is due to how template statements are encoded, which is with conditional assignments to all possible variables in their write set (the whole $\mathsf{Vars}(\pi_1)$ set in the worst case) and because each of said assignments adds a tuple to u_1/u_2.

The worst-case size of ϕ_d is $O\left(|\pi_1 \ ; \ \pi_2|^3\right)$, which corresponds to the case where paths π_1, π_2 are a sequence of distinct template statements (e.g., $\mathsf{S}_1 \ ; \ \mathsf{S}_2 \ ; \ \ldots \ ; \ \mathsf{S}_n$), originating a cubic number of constraints of the form $\mathsf{W}(s_1) \cap \mathsf{W}(s_2) \cap \mathsf{R}(s_3) = \emptyset$.

The overall worst-case size of ϕ is therefore $O\left(\left(|\pi_1 \ ; \ \pi_2| \cdot |\mathsf{Vars}(\pi_1)|\right)^2 + |\pi_1 \ ; \ \pi_2|^3\right)$. In practice, we have observed that the size of the encoding usually grows linearly with the size of the counterexamples. The number of template statements and expressions is usually small, and thus the quadratic and cubic effects are not relevant to the overall size of the encoding.

```
    function GeneralizeWP
    input
        φ – a formula
        μ – a model of formula φ (set of literals)
        ψ – set of preferred literals to bias the solution
    begin
1       if ¬φ ∧ (⋀ l ∈ μ ∩ ψ) is unsatisfiable then
2           return MinimizeCore(¬φ, GetUnsatCore(¬φ, μ ∩ ψ))
3       else
4           return MinimizeCore(¬φ, GetUnsatCore(¬φ, μ) ∪ (μ ∩ ψ))
    end.
```

Figure 5.4: GeneralizeWP algorithm.

```
    function MinimizeCore
    input
        φ – a formula
        ζ – an unsat core of formula φ (set of literals)
    vars
        Ψ – minimized core
    begin
1       Ψ := ∅
2       while ζ ≠ ∅ do
3           κ := take one from ζ
4           if φ ∧ (⋀ l ∈ Ψ ∪ ζ) is satisfiable then
5               Ψ := Ψ ∪ {κ}
6       return Ψ
    end.
```

Figure 5.5: MinimizeCore algorithm.

5.2.3 GeneralizeWP

Figure 5.4 shows the function GeneralizeWP. As input, it takes a formula ϕ, a model μ of ϕ given as a set of literals, and a set of preferred literals ψ. The purpose of this function is to compute a new model for ϕ, hopefully smaller than μ (and obviously not bigger), while maximizing the set of literals of ψ that will be part of the result.

From the definition of model of a formula, we know that $\neg\phi \wedge (\bigwedge l \in \mu)$ is unsatisfiable. If we a drop a literal, say l', from μ and if $\neg\phi \wedge (\bigwedge l \in \mu \setminus \{l'\})$ is still unsatisfiable, then $\mu' = \mu \setminus \{l'\}$ is a model of ϕ as well. However, μ' contains fewer literals than μ, and is therefore more generic (since we now know that both $\mu' \cup \{l\}$ and $\mu' \cup \{\neg l\}$ are models of ϕ).

Function GeneralizeWP works as follows. First, it checks whether restricting the model to the set of preferred literals is sufficient to make ϕ unsatisfiable. If so, it calls MinimizeCore to further reduce the size of the solution. We use the function GetUnsatCore, which is usually provided by SMT solvers, as an optimization. GetUnsatCore(x, y) returns a set $y' \subseteq y$ such that $x \wedge (\bigwedge l \in y')$ is unsatisfiable.

If the set of preferred literals is not enough, then we call MinimizeCore with the whole model. Since we are not able to bias the result of GetUnsatCore, we need to ensure that the set of preferred literals is passed to MinimizeCore.

```
    function SynthWP2
    vars
        ψ – generated precondition
        β –blocked models
    begin
        ...
12      ψ := false
13      β := false
14      while φ ∧ ¬β is satisfiable do
15          (ψ′, add) := GeneralizeWP2(φ, GetModel(φ ∧ ¬β), μ_f, d)
16          β := β ∨ ψ′
17          if add then
18              ψ := ψ ∨ ψ′
19      return ψ
    end.
```

Figure 5.6: SynthWP2 algorithm.

5.2.4 MinimizeCore

Figure 5.5 shows the function MinimizeCore. Given a formula ϕ and a set of literals ζ such that $\phi \wedge (\bigwedge l \in \zeta)$ is unsatisfiable, the objective of this function is to find a possibly smaller set $\Psi \subseteq \zeta$ such that $\phi \wedge (\bigwedge l \in \Psi)$ is still unsatisfiable.

MinimizeCore works by checking if each literal $l \in \zeta$ is necessary for the formula to be unsatisfiable. If so, l is added to the result set Ψ. We employ a linear search, as opposed to potentially better search strategies such as QuickXplain [Jun04] or Progression [MSJB13], since linear search proved to perform well in our benchmarks.

In our implementation, ζ is a list and we perform a linear search from the beginning to the end of the list. This strategy enables us to bias the search to give priority for removal of certain literals. In particular, in the function GeneralizeWP, we put all the preferred literals ψ at the end of the list, which biases the solution towards having a higher number of literals of ψ.

5.2.5 Alternative Encoding

In this section, we discuss an alternative encoding to the one used in SynthWP. It is slightly more succinct and uses fewer SMT variables than the previously given encoding, but it may require more calls to the SMT solver. This encoding requires only a few changes to the SynthWP and GeneralizeWP procedures.

SynthWP creates a constraint ϕ_w to state that if a template statement s writes to a given variable v, then v must be in the write set of s (line 6). We call this the must-write constraint, as opposed to the may-write meaning of a variable being in the write set of a template statement.

Alternatively, it is possible to avoid the must-write variables altogether and use the set membership variables directly when encoding template statements, i.e., a template statement s can be encoded as follows:

$$\bigwedge\nolimits_{v \in V} (v' = \text{ite}(\mathcal{B}(v \in \mathsf{W}(s)), fv, v))$$

This encoding has the obvious advantage that it has fewer variables. However, it requires a change to the SynthWP procedure, since we want to keep write set membership with may-write semantics. The reason is that just because a variable is in the write set of a template statement, it does not mean that that statement has to necessarily write to that variable. Therefore, we need to ignore preconditions that state that a template statement *must* write to a certain variable.

```
    function GeneralizeWP2
    input
        φ – a formula
        μ – a model of formula φ (set of literals)
        μ_f – a model filter
        ψ – set of preferred literals to bias the solution
    begin
1       if ¬φ ∧ (⋀ l ∈ μ ∩ ψ) is unsatisfiable then
2           return (MinimizeCore(¬φ, GetUnsatCore(¬φ, μ ∩ ψ)), true)
3       else if ¬φ ∧ (⋀ l ∈ μ ∩ μ_f) is unsatisfiable then
4           return (MinimizeCore(¬φ, GetUnsatCore(¬φ, μ ∩ μ_f) ∪ (μ ∩ ψ)), true)
5       else
6           return (MinimizeCore(¬φ, GetUnsatCore(¬φ, μ) ∪ (μ ∩ ψ)), false)
    end.
```

Figure 5.7: GeneralizeWP2 algorithm.

The change that we need to do is to separate the formula that is used to block models previously seen from the models that constitute the precondition. In this way, we can still make progress by blocking a model that was found, while filtering the models that make up interesting preconditions. The alternative procedure is shown in Figure 5.6. Lines 1–11 are omitted, since they remain equal.

The main change is that we used a new GeneralizeWP2 procedure that returns two parameters, with the first being the generalized model and the second a boolean indicating whether the model was expressed solely in terms of literals passed in the third parameter or if it required additional literals. In our case, we want to discard models that are not expressible solely with the literals in μ_f.

Figure 5.7 shows the GeneralizeWP2 procedure. The main difference lies in the fact that it is not assumed anymore that $\neg\phi \wedge (\bigwedge l \in \mu \cap \mu_f)$ is unsatisfiable. As with GeneralizeWP, it tries to bias the result towards the set of preferred literals (ψ). Then, it tries to compute a solution that depends only on the model filtered by μ_f (in our case, negated membership literals). If all fails, it computes a generalized model using all literals, but still minimizes the UNSAT core with bias towards the literals in ψ.

An experimental evaluation of both encodings for run time and practical completeness is given in Section 5.3.

5.2.6 Discussion

The proposed algorithm, although agnostic to the verification algorithm used, assumes that only counterexamples for partial functional correctness proofs are generated. This means that the algorithm as presented will produce weakest *liberal* preconditions. To produce weakest preconditions, the algorithm has to be extended so that it can handle counterexamples for relative termination mismatches (based upon, e.g., [CGLA+08, BIK12, HKLR13]).

The proposed specification language does not include instructions to access heap locations or arrays. This means that the current algorithm does not handle optimizations that perform explicit transformations to memory access instructions. It does, however, support instantiation of templates with memory accessing instructions (such as instantiating a template expression with a load from a memory location), provided that the instantiation meets the precondition (which can be verified using, e.g., a data dependency analysis).

In this work, we only consider preconditions in the language of read and write sets. However, arithmetic preconditions may be needed for some optimizations. For example, a specialization of the loop

unrolling optimization requires the number of iterations of the source loop to be even. [1] Synthesizing such preconditions could be done by, for example, adapting the counterexample-driven algorithm of Seghir and Kroening [SK13].

In terms of read sets, we only consider one read set per template statement. However, it is possible to improve precision by considering one read set per each pair of template statement and written variable. For example, for template statement s with write set $\mathsf{W}(s) = \{x, y, z\}$, we would have three read sets: $\mathsf{R}(s_x)$, $\mathsf{R}(s_y)$, and $\mathsf{R}(s_z)$. This change can be trivially incorporated in the proposed algorithm, at the expense of producing more complicated preconditions. The effects in practice of this change are, however, unknown.

5.3 Evaluation

5.3.1 Implementation

We implemented a prototype named PSyCO[2], which stands for Precondition Synthesizer for Compiler Optimizations. PSyCO is implemented in Python (in about 1,500 lines of code), and uses Z3 4.3.2 [dMB08] (and Z3Py) for constraint solving.

In principle, PSyCO can be used with any compiler optimization verification tool that can produce counterexamples (to implement CheckTF). However, we chose to implement a simple bounded model checker (BMC) within PSyCO for convenience. This BMC only checks optimizations for partial correctness, and therefore the results presented in this section are weakest *liberal* preconditions.

We did not use our own verification tool, CORK (described in Chapter 6), since it is several orders of magnitude slower at producing counterexamples than our simple BMC, since it is optimized towards proving correctness. Furthermore, CORK does not support disjunctive preconditions natively, and therefore it has to first convert such preconditions to DNF and test each of the conjuncts separately.

The BMC that was implemented is fairly simple. It enumerates paths using a breadth-first search (BFS) across the source and target templates and then checks for inconsistencies. The BMC leverages Z3's incremental solving capabilities for improved performance.

The BMC supports two ways of encoding template statements and expressions. The first approach is by using conditional Ackermannization as done in the SynthWP function. The second approach is by using uninterpreted function symbols (UFs), one per template statement/expression and written variables. For example, S would get encoded as a sequence of assignments as follows:

$$v' = S_v\left(\text{ite}(\mathcal{B}(v_1 \in \mathsf{R}(\mathsf{S})), \sigma(v_1), 0), \ldots, \text{ite}(\mathcal{B}(v_n \in \mathsf{R}(\mathsf{S})), \sigma(v_n), 0)\right)$$

for every variable $v \in \mathsf{W}(\mathsf{S})$, $\mathsf{R}(\mathsf{S}) = \{v_1, \ldots, v_n\}$, and σ a map with the value of the variables before the evaluation of S. Similarly, template expressions are encoded as a single UF application.

The second encoding uses UF applications where the input is conditioned to the read set. That way, if a certain variable v is not in the read set of a template statement/expression, all its corresponding UF applications get the same value as input in the parameter corresponding to v. With this technique, we effectively reduce the set of inputs of UF symbols to the variables in the read set of their corresponding template statements/expressions.

The advantage of the first encoding is that it does not require support for UFs from the SMT solver, which can sometimes be suboptimal or nonexistent. On the other hand, it requires the quadratic condi-

[1] A more general version of loop unrolling (that accounts for an even and odd number of loop iterations) was used in the experiments.

[2] Prototype and benchmarks available from `http://web.ist.utl.pt/nuno.lopes/psyco/`.

tional Ackermannization procedure to be run upfront. The second encoding leaves the expensive Ackermannization to the SMT solver, which can be smarter and do it lazily as needed.

5.3.2 Examples of Generated Preconditions

Table 5.1 shows a few examples of compiler optimizations and the corresponding preconditions generated by PSyCO. Details about the presented optimizations can be found in modern compiler textbooks (e.g., [Muc97, KA02, ALSU06]).

This list of Table 5.1 is not supposed to be exhaustive, since there are many optimizations and each one of them may be specified in slightly different ways. We show these examples so that the reader can truly appreciate the simplicity of the generated preconditions and realize how surprising the weakest preconditions can be (from what you would expect at first thought).

There are very few published formally stated preconditions for compiler optimizations. However, the PEC paper [KTL09] does include a precondition for software pipelining (in a slightly different language than the one we used), that was written down by hand and then verified correct by PEC. The precondition synthesized by PSyCO (as shown in Table 5.4) is, however, weaker than that in PEC's paper. Their precondition requires that $\mathsf{V}_1 \notin \mathsf{W}(\mathsf{S}_1)$ and $\mathsf{V}_2 \notin \mathsf{W}(\mathsf{S}_1)$, while the precondition generated by PSyCO does not. Therefore, the precondition generated by PSyCO is weaker than that published by experts in formal methods and compiler optimizations, showing that automatic precondition synthesis does indeed help to make optimizations more widely applicable.

An interesting discussion is whether the language used to specify WPs is rich enough and, in particular, whether the generated preconditions are in practice weaker that those used in compilers. We did not performed this study, however, and leave that for future work.

Another thing that can be speculated about is whether some parts of the preconditions that are synthesized may be irrelevant in practice. For example, the WP that is synthesized for loop fusion is overly complicated. It includes cases in which although the transformation is sound, they are unlikely to appear in practice. In particular, the precondition considers cases where the first loop could be eliminated altogether. In practice, compilers run optimizations in a pipeline, and the compiler would probably already have an optimization to remove useless loops. We did not perform a throughout study on the usefulness in practice of having the weakest precondition, if some parts of the WPs could be dropped, and if the performance of the compiler is impacted if the WP is more complicated.

5.3.3 Quantitative Experiments

We ran PSyCO over a set of optimizations (mostly loop manipulating). The experiments were run on a machine running Linux 3.14.2 with an Intel Core 2 Duo 3.00 GHz CPU, and 4 GB of RAM. Memory was never a bottleneck. The results are shown in Table 5.5 and in Table 5.6.

Since we are not aware of any other algorithm for precondition synthesis for compiler optimization, we cannot compare PSyCO against other tools. In Table 5.5, we show the number of counterexamples required for each optimization to reach convergence. We notice that in general only a few counterexamples are required, with certain optimizations with a trivial precondition (true) requiring none.

Then, we present the total number of models produced by Z3 for preconditions when the algorithm is run with and without the must-write constraints. The ratio of the number of models per number of counterexamples is small because of the employed optimizations described before (model weakening and inference of common precondition patterns).

The number of models when not using the must-write constraints is significantly higher than the number of models when using those constraints for many loop-manipulating optimizations. The reason for the few cases where it is smaller is because the precondition generalization procedure is sensitive to

the ordering of variables in the UNSAT cores computed by the SMT solver. Since different constraints may yield different variable orderings in the UNSAT cores, it happens that sometimes the algorithm can produce more general preconditions than with other orderings, meaning fewer models are required to converge.

In the fourth column, we show the percentage of models that were generalized by our algorithm. This percentage is pretty high overall, meaning that Z3 consistently fails to produce a minimal model and/or UNSAT core, which is to expect since Z3 does not employ any specific technique to minimize models nor UNSAT cores (other than the lazy decision procedure). Therefore, the number of models would be exponentially larger if PSyCO did not use a procedure to generalize models.

In the last column of Table 5.5, we show the number of models that required positive set membership constraints when the algorithm is run without the must-write encoding. These models are discarded, and therefore a low number is better for performance reasons. In the loop fission test, Z3 gave up when solving one of the constraints, and therefore the resulting precondition is not guaranteed to be the weakest.

Table 5.6 shows the running times of PSyCO when run with different configurations.

First, we show the time taken by the precondition synthesis algorithm with and without the must-write constraints, as well as the difference (in percentage) between the two alternatives. When not using the must-write encoding, there are significant slowdowns. Therefore, although the formula given to the SMT is smaller, it does not pay off the increase in the number of models.

Second, we show the overall time taken by the tool with the must-write encoding (the default of the tool). The overall time includes not only the synthesis algorithm, but also the BMC time as well as minor initializations performed by the tool. We show the running time for two alternative implementations of the BMC that differ in the encoding of template statements/expressions: one using conditional Ackermannization, and another that uses UFs directly.

We argue that the time taken by the precondition synthesis algorithm is low. Overall, PSyCO is usually fast, with a few exceptions due to high inefficiencies in the Z3Py module exposed by our BMC. However, the running time for this kind of tool is not critical, since it is supposed to be run off-line, and only once per optimization specification.

The difference in running time between the two encodings of the BMC is not conclusive. On one hand, there is one case (loop skewing) where the running time is reduced significantly (by 62%) when using UFs. On the other hand, there are a number of cases where the encoding with UFs is slower by over 20%.

The encoding with UFs is generally smaller than when using conditional Ackermannization (since it produces a constraint whose size is quadratic in the number of template symbols). However, apparently Z3 is not very efficient when solving constraints with UF symbols, and therefore sometimes it is faster to do Ackermannization upfront manually.

5.4 Summary

In this chapter, we presented the first known algorithm for the automatic synthesis of weakest preconditions for compiler optimizations. The generated preconditions are specified in the language of read and write sets over template statements/expressions.

The presented algorithm is able to handle all classic optimizations we have tried. Moreover, one of the preconditions that we synthesized automatically is weaker than the one that had been previously published by experts, who derived it by hand, confirming the need in practice for our algorithm and tool.

Optimization	Weakest Liberal Precondition

Code hoisting
if B **then**
  S_1
  S_2
else
  S_1
  S_3
$\Rightarrow$
S_1
if B **then**
  S_2
else
  S_3
	$R(B) \cap W(S_1) = \emptyset$

Constant propagation $V_1 := E$ S $V_2 := E$ $\Rightarrow$ $V_1 := E$ S $V_2 := V_1$	$R(E) \cap W(S) = \emptyset \wedge$ $V_1 \notin R(E) \wedge$ $V_1 \notin W(S)$
Copy propagation $V_1 := V_2$ $V_3 := V_1$ $\Rightarrow$ $V_1 := V_2$ $V_3 := V_2$	true

If-conversion
if B **then**
  $V := E$
$\Rightarrow$
if B **then**
  $V := E$
else
  $V := V$
	true

Partial redundancy elimination (PRE)
if B **then**
  S_1
  $V_1 := E$
  S_2
else
  S_3
$V_2 := E$
$\Rightarrow$
if B **then**
  S_1
  $V_1 := E$
  S_2
  $V_2 := V_1$
else
  S_3
  $V_2 := E$
	$V_1 \notin R(E) \wedge V_1 \notin W(S_2) \wedge$
	$R(E) \cap W(S_2) = \emptyset$

Loop fission
$V_1 := E$
while $V_1 < V_2$ **do**
  S_1
  S_2
  $V_1 := V_1 + 1$
$\Rightarrow$
$V_1 := E$
while $V_1 < V_2$ **do**
  S_1
  $V_1 := V_1 + 1$
$V_1 := E$
while $V_1 < V_2$ **do**
  S_2
  $V_1 := V_1 + 1$
	$V_1 \notin R(E) \wedge V_1 \notin W(S_1) \wedge$
	$V_2 \notin W(S_1) \wedge$
	$W(S_1) \cap R(E) = \emptyset \wedge$
	$W(S_2) \cap R(S_1) = \emptyset \wedge$
	$W(S_1) \cap W(S_2) \cap R(S_2) = \emptyset \wedge$
	$\Big(\big((W(S_2) \cap R(S_2) = \emptyset \vee$
	$W(S_1) \cap R(S_2) = \emptyset) \wedge$
	$V_1 \notin W(S_2) \wedge V_2 \notin W(S_2)\big) \vee$
	$\big(W(S_1) \cap R(S_1) = \emptyset \wedge$
	$V_1 \notin R(S_1)\big)\Big)$

Table 5.1: Weakest preconditions synthesized by PSyCO.

Optimization	Weakest Liberal Precondition

Loop flattening
$V_1 := 0$
while $V_1 < V_2$ **do**
  $V_3 := 0$
  while $V_3 < V_2$ **do**
  S
  $V_3 := V_3 + 1$
  $V_1 := V_1 + 1$
$\Rightarrow$
$V_1 := 0$
$V_4 := 0$
if $V_1 < V_2$ **then**
  while $V_4 < (V_2 \times V_2)$ **do**
  $V_1 := V_4 \div V2$
  $V_3 := (V_4 - V_2) \times V_1$
  S
  $V_4 := V_4 + 1$
  $V_1 := V_2$
  $V_3 := V_2$
	$V_1 \notin W(S) \wedge V_2 \notin W(S) \wedge$
	$V_3 \notin W(S)$

Loop fusion
$V_1 := E$
while $V_1 < V_2$ **do**
  S_1
  $V_1 := V_1 + 1$
$V_1 := E$
while $V_1 < V_2$ **do**
  S_2
  $V_1 := V_1 + 1$
$\Rightarrow$
$V_1 := E$
while $V_1 < V_2$ **do**
  S_1
  S_2
  $V_1 := V_1 + 1$
	$V_1 \notin R(E) \wedge V_1 \notin W(S_1) \wedge$
	$V_2 \notin W(S_1) \wedge$
	$W(S_1) \cap R(E) = \emptyset \wedge$
	$W(S_2) \cap R(S_1) = \emptyset \wedge$
	$W(S_1) \cap W(S_2) \cap R(S_2) = \emptyset \wedge$
	$\Big(\big((W(S_2) \cap R(S_2) = \emptyset \vee$
	$W(S_1) \cap R(S_2) = \emptyset) \wedge$
	$V_1 \notin W(S_2) \wedge V_2 \notin W(S_2)\big) \vee$
	$\big(W(S_1) \cap R(S_1) = \emptyset \wedge$
	$V_1 \notin R(S_1)\big)\Big)$

Loop interchange
$V_1 := E_1$
$V_3 := E_2$
if $V_3 < V_4$ **then**
  while $V_1 < V_2$ **do**
  $V_3 := E_2$
  while $V_3 < V_4$ **do**
  S
  $V_3 := V_3 + 1$
  $V_1 := V_1 + 1$
$\Rightarrow$
$V_1 := E_1$
$V_3 := E_2$
if $V_1 < V_2$ **then**
  while $V_3 < V_4$ **do**
  $V_1 := E_1$
  while $V_1 < V_2$ **do**
  S
  $V_1 := V_1 + 1$
  $V_3 := V_3 + 1$
	$W(S) \cap R(E_1) = \emptyset \wedge$
	$W(S) \cap R(E_2) = \emptyset \wedge$
	$V_1 \notin R(E_1) \wedge V_3 \notin R(E_1) \wedge$
	$V_1 \notin R(E_2) \wedge V_3 \notin R(E_2) \wedge$
	$V_2 \notin W(S) \wedge V_4 \notin W(S) \wedge$
	$\big(W(S) \cap R(S) = \emptyset \vee$
	$(V_1 \notin W(S) \wedge V_3 \notin W(S) \wedge$
	$V_1 \notin R(S) \wedge V_3 \notin R(S))\big)$

Loop invariant code motion (LICM)
while $V_1 < V_2$ **do**
  S_1
  S_2
  $V_1 := V_1 + 1$
$\Rightarrow$
if $V_1 < V_2$ **then**
  S_2
  while $V_1 < V_2$ **do**
  S_1
  $V_1 := V_1 + 1$
	$W(S_1) \cap R(S_2) = \emptyset \wedge$
	$W(S_1) \cap W(S_2) = \emptyset \wedge$
	$W(S_2) \cap R(S_1) = \emptyset \wedge$
	$W(S_2) \cap R(S_2) = \emptyset \wedge$
	$V_1 \notin R(S_2) \wedge V_1 \notin W(S_2) \wedge$
	$V_2 \notin W(S_2)$

Loop peeling
while $V_1 < V_2$ **do**
  S
  $V_1 := V_1 + 1$
$\Rightarrow$
if $V_1 < V_2$ **then**
  S
  $V_1 := V_1 + 1$
  while $V_1 < V_2$ **do**
  S
  $V_1 := V_1 + 1$
	true

Table 5.2: Weakest preconditions synthesized by PSyCO (continued).

Optimization	Weakest Liberal Precondition

Loop reversal
$\mathsf{V}_1 := \mathsf{E}$
while $\mathsf{V}_1 < \mathsf{V}_2$ **do**
  S
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
$\Rightarrow$
if $\mathsf{E} < \mathsf{V}_2$ **then**
  $\mathsf{V}_1 := \mathsf{V}_2 - 1$
  while $\mathsf{V}_1 \geq \mathsf{E}$ **do**
  S
  $\mathsf{V}_1 := \mathsf{V}_1 - 1$
  $\mathsf{V}_1 := \mathsf{V}_2$
else
  $\mathsf{V}_1 := \mathsf{E}$
	$\mathsf{V}_1 \notin \mathsf{R(E)} \wedge \mathsf{V}_1 \notin \mathsf{R(S)} \wedge$
	$\mathsf{V}_2 \notin \mathsf{W(S)} \wedge$
	$(\mathsf{W(S)} \cap \mathsf{R(E)} = \emptyset \vee$
	$\mathsf{V}_1 \notin \mathsf{W(S)}) \wedge$
	$\big(\mathsf{W(S)} \cap \mathsf{R(S)} = \emptyset \vee$
	$(\mathsf{V}_1 \notin \mathsf{W(S)} \wedge$
	$\mathsf{W(S)} \cap \mathsf{R(E)} = \emptyset)\big)$

Loop skewing
while $\mathsf{V}_1 < \mathsf{V}_2$ **do**
  $\mathsf{V}_3 := \mathsf{E}$
  while $\mathsf{V}_3 < \mathsf{V}_4$ **do**
  S
  $\mathsf{V}_3 := \mathsf{V}_3 + 1$
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
$\Rightarrow$
while $\mathsf{V}_1 < \mathsf{V}_2$ **do**
  $\mathsf{V}_5 := \mathsf{E} + \mathsf{V}_6$
  $\mathsf{V}_3 := \mathsf{V}_5 - \mathsf{V}_6$
  if $\mathsf{V}_5 < (\mathsf{V}_4 + \mathsf{V}_6)$ **then**
  while $\mathsf{V}_5 < (\mathsf{V}_4 + \mathsf{V}_6)$ **do**
  $\mathsf{V}_3 := \mathsf{V}_5 - \mathsf{V}_6$
  S
  $\mathsf{V}_5 := \mathsf{V}_5 + 1$
  $\mathsf{V}_3 := \mathsf{V}_4$
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
	$\mathsf{V}_3 \notin \mathsf{W(S)} \wedge \mathsf{V}_4 \notin \mathsf{W(S)}$

Loop strength reduction
while $\mathsf{V}_1 < \mathsf{V}_2$ **do**
  $V_3 := \mathsf{V}_1 * \mathsf{E}$
  S
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
$\Rightarrow$
$V_4 := \mathsf{V}_1 * \mathsf{E}$
while $\mathsf{V}_1 < \mathsf{V}_2$ **do**
  $V_3 := V_4$
  $V_4 := V_4 + \mathsf{E}$
  S
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
	$\mathsf{V}_1 \notin \mathsf{R(E)} \wedge V_3 \notin \mathsf{R(E)} \wedge$
	$\mathsf{R(E)} \cap \mathsf{W(S)} = \emptyset \wedge$
	$\big(\mathsf{V}_1 \notin \mathsf{W(S)} \vee$
	$(V_3 \notin \mathsf{R(S)} \wedge$
	$\mathsf{R(S)} \cap \mathsf{W(S)} = \emptyset)\big)$

Loop tiling
while $\mathsf{V}_1 < \mathsf{V}_2$ **do**
  S
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
$\Rightarrow$
$\mathsf{V}_3 := \mathsf{V}_1$
while $\mathsf{V}_3 < \mathsf{V}_2$ **do**
  $\mathsf{V}_1 := \mathsf{V}_3$
  while $\mathsf{V}_1 < \min(\mathsf{V}_2, \mathsf{V}_3 + \mathsf{V4})$ **do**
  S
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
  $\mathsf{V}_3 := \mathsf{V}_3 + \mathsf{V}_4$
	$\mathsf{V}_1 \notin \mathsf{W(S)} \vee$
	$\mathsf{R(S)} \cap \mathsf{W(S)} = \emptyset$

Loop unrolling
while $\mathsf{V}_1 < \mathsf{V}_2$ **do**
  S
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
$\Rightarrow$
while $(\mathsf{V}_1 + 1) < \mathsf{V}_2$ **do**
  S
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
  S
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
 	
if $\mathsf{V}_1 < \mathsf{V}_2$ **then**
  S
  $\mathsf{V}_1 := \mathsf{V}_1 + 1$
	$\mathsf{V}_2 \notin \mathsf{W(S)} \wedge$
	$(\mathsf{V}_1 \notin \mathsf{W(S)} \vee$
	$\mathsf{R(S)} \cap \mathsf{W(S)} = \emptyset)$

Table 5.3: Weakest preconditions synthesized by PSyCO (continued).

Optimization	Weakest Liberal Precondition

Loop unswitching
while $V_1 < V_2$ **do**
  if B **then**
  S_1
  else
  S_2
  $V_1 := V_1 + 1$
$\Rightarrow$
if B **then**
  while $V_1 < V_2$ **do**
  S_1
  $V_1 := V_1 + 1$
else
  while $V_1 < V_2$ **do**
  S_2
  $V_1 := V_1 + 1$
	$V_1 \notin R(B) \wedge$
	$W(S_1) \cap R(B) = \emptyset \wedge$
	$W(S_2) \cap R(B) = \emptyset$

Software pipelining
while $V_1 < V_2$ **do**
  S_1
  S_2
  $V_1 := V_1 + 1$
$\Rightarrow$
if $V_1 < V_2$ **then**
  S_1
  while $V_1 < (V_2 - 1)$ **do**
  S_2
  $V_1 := V_1 + 1$
  S_1
  S_2
  $V_1 := V_1 + 1$
	$V_2 \notin W(S_2) \wedge$
	$\big((R(S_1) \cap W(S_2) = \emptyset \wedge$
	$R(S_1) \cap W(S_1) = \emptyset \wedge$
	$R(S_2) \cap W(S_2) = \emptyset) \vee$
	$V_1 \notin W(S_2)\big)$

Table 5.4: Weakest preconditions synthesized by PSyCO (continued).

Optimization	# CExs.	With Must-Write		Without Must-Write	
		# Models	Non-min. Cores	# Models	# Pos. Models
Code hoisting	1	2	100%	1	0
Constant propagation	1	2	100%	1	0
Copy propagation	0	—	—	—	—
If-conversion	0	—	—	—	—
Partial redundancy elimin.	1	16	100%	7	0
Loop fission	6	48	96%	39	2 (Inc.)
Loop flattening	1	1	100%	1	0
Loop fusion	6	50	92%	83	17
Loop interchange	11	42	83%	64	7
Loop invariant code motion	3	7	86%	7	1
Loop peeling	0	—	—	—	—
Loop reversal	4	12	92%	15	0
Loop skewing	1	1	100%	1	0
Loop strength reduction	1	3	100%	7	4
Loop tiling	1	2	50%	2	0
Loop unrolling	2	5	80%	6	1
Loop unswitching	2	7	86%	22	4
Software pipelining	1	4	100%	62	38

Table 5.5: List of compiler optimizations tested with PSyCO, including the number of counterexamples processed, the number of models generated by Z3 for preconditions with and without must-write constraints, percentage of non-minimal UNSAT cores computed by Z3, and the number of models with mandatory positive set membership constraints when not using the must-write encoding. Z3 without must-write constraints reports incomplete results with loop fission.

Optimization	WP Time			Total Time		
	w/ MW	wo/ MW		cond. Ackermann.	UFs	
Code hoisting	0.28s	0.26s	(-7.1%)	0.62s	0.63s	(+1.6%)
Constant propagation	0.09s	0.29s	(+222%)	0.15s	0.13s	(-13%)
Copy propagation	0s	0s	(—)	0.02s	0.02s	(0%)
If-conversion	0s	0s	(—)	0.03s	0.03s	(0%)
Partial redundancy elimin.	2.27s	3.83s	(+69%)	2.98s	2.99s	(+0.3%)
Loop fission	4.03s	4.88s	(+21%)	4.93s	5.93s	(+20%)
Loop flattening	0.08s	0.08s	(0%)	3.14s	3.82s	(+22%)
Loop fusion	3.76s	4.48s	(+19%)	4.70s	4.93s	(+4.9%)
Loop interchange	3.42s	3.67s	(+7.3%)	24.2s	23.8s	(-1.7%)
Loop invariant code motion	0.63s	0.44s	(-30%)	0.92s	0.95s	(+3.3%)
Loop peeling	0s	0s	(—)	0.18s	0.15s	(-17%)
Loop reversal	0.50s	0.50s	(0%)	0.70s	0.73s	(+4.3%)
Loop skewing	0.11s	0.11s	(0%)	147s	55.2s	(-62%)
Loop strength reduction	0.78s	1.92s	(+146%)	0.99s	0.93s	(-6.1%)
Loop tiling	0.09s	0.09s	(0%)	4.37s	4.40s	(+0.7%)
Loop unrolling	0.17s	0.18s	(-5.3%)	0.45s	0.47s	(+4.4%)
Loop unswitching	0.53s	3.06s	(+477%)	1.18s	1.25s	(+5.9%)
Software pipelining	0.53s	2.69s	(+408%)	0.98s	0.78s	(-20%)

Table 5.6: Time taken by the precondition generation algorithm, with and without must-write constraints, and the overall time taken by the tool (including the BMC) when using conditional Ackermannization or UFs directly in the BMC.

Chapter 6

Automatic Equivalence Checking of UF+IA Programs

In this chapter, we present a new algorithm for the automatic partial equivalence checking of programs under the combined theory of uninterpreted function symbols and integer arithmetic (UF+IA). The proposed algorithm is applicable, in particular, to programs containing nested loops.

Program equivalence checking, despite being inherently complex, has several interesting and important applications, such as algorithm recognition, regression checking, compiler verification and validation, and information flow checking.

The algorithm works as follows. Applications of UFs are first rewritten to integer arithmetic expressions (polynomials over the inputs of the applications), and then our equivalence checking algorithm works on purely integer manipulating loop-free programs. Loops are summarized using recurrences, for which we compute the closed-form solution. The provably correct conversion of UF applications to integer expressions makes possible the representation of loops with UF applications using recurrences. The algorithm then composes the two programs sequentially and checks the resulting program for safety.

In this chapter, we also show how to automatically verify the correctness of compiler optimizations using the proposed equivalence checking algorithm. The procedure works by transforming each of the two template programs of a transformation function into a UF+IA program, which are then checked for equivalence.

Finally, we present CORK, a tool for the automatic verification of compiler optimizations using the techniques presented in this chapter. Additionally, we show experimentally that CORK can prove more optimizations correct than previously proposed techniques.

6.1 Illustrative Example

We illustrate our algorithm for program equivalence checking on a simple example. Figure 6.1 shows two equivalent example programs, where f is a UF symbol. [1] Our objective is to prove that these two programs are indeed equivalent.

The first step of the algorithm is to do sequential composition of the two programs, where the second program is renamed to operate over a distinct set of variables from the first. We then add an assertion at the end of the composed program to verify that the value of the corresponding variables of the two programs are equal when the programs terminate. Similarly, we assume that the corresponding variables

[1] As an anecdote, when developing this example, we forgot the **if** command in the program on the right. Fortunately, our prototype quickly pointed out our mistake (of different values of i at the end of the programs when the loops do not execute).

```
i := 0
while i < N do
    k := f(k, i)
    i := i + 1
```

```
i := N
while i ≥ 1 do
    k := f(k, N − i)
    i := i − 1

if N ≤ 0 then
    i := 0
else
    i := N
```

Figure 6.1: Example of two equivalent programs.

```
assume i = ī ∧ k = k̄ ∧ N = N̄

i := 0
if i < N then
while i < N do
    k := f(k, i)
    i := i + 1

ī := N̄
while ī ≥ 1 do
    k̄ := f(k̄, N̄ − ī)
    ī := ī − 1

if N̄ ≤ 0 then
    ī := 0
else
    ī := N̄

assert i = ī ∧ k = k̄ ∧ N = N̄
```

Figure 6.2: Sequential composition of the programs of Figure 6.1. The program on the right was renamed, so that each variable v becomes $\bar{v}$.

of the two programs have the same value at the beginning of the composed program. The resulting composed program can be seen in Figure 6.2.

Now if we prove that the composed program is safe, i.e., that the condition of the **assert** command is true for all inputs, then we have proved that the two input programs are equivalent. However, state-of-the-art software verification tools are not able to verify the correctness of the program of Figure 6.2, since it requires complex invariants to be synthesized.

We now show how our algorithm proceeds.

The second step of the algorithm is to replace applications of uninterpreted functions (UFs) with expressions over integers. In the left program, we replace the UF application with the following expression (a polynomial of degree one on k and i):

$$a \times k + b \times i + c$$

where a, b, and c are fresh variables not occurring in the input programs, and are associated with this specific UF symbol. Other UF symbols occurring in the program would have different fresh variables associated with each input parameter. Similarly, for the UF application of the right program we obtain:

$$a \times \bar{k} + b \times (\bar{N} - \bar{i}) + c$$

Intuitively, these expressions (polynomials) have a unique value for each set of UF symbol and input parameters (since variables a, b, and c are fresh). Therefore, no other sequence of commands can always produce the same value without doing the same UF application with the same inputs, meaning that with this abstraction we do not lose information necessary for the safety proof. This is because there always exists an assignment to fresh variables a, b, and c that leads to different results for different UF applications, which would therefore violate the assertion.

This transformation is closely related to *polynomial interpolation*, which consists in determining a polynomial of a certain degree that passes through a given set of points.

As we shall see later, the degree of the polynomials that replace UF applications is not always one. We give a lower bound for this degree in Section 6.4.2.

The third step that the algorithm performs is summarizing and subsequently removing the loops. This is accomplished by replacing each loop with a set of assignments to the variables modified in the loop. The expressions assigned to each variable are expressed over the closed-form solution of a system of recurrences that summarizes the loop precisely.

For the left program, we obtain the following system of recurrences:

$$\begin{aligned} R_i(n) &= R_i(n-1) + 1 \\ R_i(0) &= 0 \\ R_k(n) &= a \times R_k(n-1) + b \times R_i(n-1) + c \\ R_k(0) &= k_0 \end{aligned}$$

where n represents the loop iteration number, and k_0 is the (arbitrary) value of k when the program starts (required since k is not initialized before its first usage). A recurrence for N is not needed, since it is not modified in the loop.

The recurrence $R_x(y)$ represents the value of variable x at iteration number y. For example, the recurrence $R_i(n)$ defined previously means that the value of i in any given iteration is equal to the value of i in the previous iteration plus one. Moreover, before the loop starts, i has the value zero.

Similarly, for the right program we obtain the following system of recurrences:

$$\begin{aligned} R_{\bar{i}}(n) &= R_{\bar{i}}(n-1) - 1 \\ R_{\bar{i}}(0) &= \bar{N} \\ R_{\bar{k}}(n) &= a \times R_{\bar{k}}(n-1) + b \times (\bar{N} - R_{\bar{i}}(n-1)) + c \\ R_{\bar{k}}(0) &= k_0 \end{aligned}$$

Figure 6.3 shows the programs of Figure 6.2 after both transformations (elimination of loops and UF applications) have been applied. The references to recurrences were not replaced with their closed-form solutions to avoid cluttering the example.

The **assume** command ensures that its input boolean expression is satisfiable, or the program execution is blocked otherwise. We use this command to implicitly compute the trip count of loops.

Intuitively, if m is the number of iterations performed by a loop, in the iterations numbered $0 \ldots (m-1)$ the loop guard is true, and it is false in the following iteration (m). Therefore, m is the first iteration when the loop guard becomes false.

After the **assume** command in the example is evaluated, the value of n is the number of times that the corresponding loop would have been executed and therefore $R_x(n)$ represents the value of the variable x after the loop terminates.

The expression used in this example to compute the trip count is correct only for linear loop conditions,

assume $i = \bar{i} \wedge k = \bar{k} \wedge N = \bar{N}$

$i := 0$
if $i < N$ **then**
  assume $R_i(n-1) < N \wedge R_i(n) \geq N$
  $k := R_k(n)$
  $i := R_i(n)$

$\bar{i} := \bar{N}$
if $\bar{i} \geq 1$ **then**
  assume $R_{\bar{i}}(\bar{n}-1) \geq 1 \wedge R_{\bar{i}}(\bar{n}) < 1$
  $\bar{k} := R_{\bar{k}}(\bar{n})$
  $\bar{i} := R_{\bar{i}}(\bar{n})$

if $\bar{N} \leq 0$ **then**
  $\bar{i} := 0$
else
  $\bar{i} := \bar{N}$

assert $i = \bar{i} \wedge k = \bar{k} \wedge N = \bar{N}$

Figure 6.3: Program of Figure 6.2 after removing loops and UF applications.

since in that case there is only one solution for the specified expression. For non-linear loop conditions, we need to compute the minimum positive n that satisfies the expression, which can be accomplished for instance using optimizing solvers or with multiple calls to regular constraint solvers.

We can now compute the closed-form solution of the previously given systems of recurrences. For the left program we obtain the following solution (computed by Wolfram Mathematica 8):

$$R_i(n) = n$$
$$R_k(n) = \frac{b\,(a^n - an + n - 1)}{(a-1)^2} + \frac{a^n((a-1)k_0 + c) - c}{a-1}$$

For the right program, the solution for $R_{\bar{k}}(n)$ is equal to $R_k(n)$ of the left program, and for $R_{\bar{i}}$ is:

$$R_{\bar{i}}(n) = \bar{N} - n$$

The fourth and final step of the algorithm is to prove that the composed program after the described transformations (which is now only over integer arithmetic and loop-free) is correct.

To prove program safety, we can use standard software verification techniques (e.g., software model checking [JM09]). Since the number of control-flow paths of the composed programs is always finite (as we remove the loops), we can use a simple algorithm that enumerates all paths and checks if the assertion is violated in any of them.

6.2 Program Model

We assume that programs are specified in the WHILE language, whose syntax is given in Figure 6.4. Expressions are side-effect free and are over the combined theory of uninterpreted function symbols and integer arithmetic (UF+IA). The evaluation of expressions is parameterized on an interpretation for each UF symbol.

For the sake of ease of reading, in the examples given throughout this chapter, we relax the syntax

$$e ::= n \mid v \mid e_1 \oplus e_2 \mid f(e_1, \ldots, e_n)$$
$$b ::= e \leq 0 \mid b_1 \otimes b_2$$
$$c ::= \textbf{skip} \mid v := e \mid c_1 \; ; \; c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \textbf{while } b \textbf{ do } c_1 \mid \textbf{assume } b \mid \textbf{assert } b \mid \textbf{abort}$$

Figure 6.4: WHILE language syntax. n is an integer number, v is a variable name, f is an uninterpreted function symbol, $\oplus$ is a binary operator over integer expressions (e.g., $+$, $-$), and $\otimes$ is a binary operator over boolean expressions (e.g., $\wedge$, $\vee$).

$$\frac{}{\langle \textbf{skip}, \; \sigma \rangle \to \sigma} \qquad \frac{}{\langle v := e, \; \sigma \rangle \to \sigma[v \mapsto \sigma(e)]}$$

$$\frac{\langle c_1, \; \sigma \rangle \to \langle c_1', \; \sigma' \rangle}{\langle c_1 \; ; \; c_2, \; \sigma \rangle \to \langle c_1' \; ; \; c_2, \; \sigma' \rangle} \qquad \frac{\langle c_1, \; \sigma \rangle \to \sigma'}{\langle c_1 \; ; \; c_2, \; \sigma \rangle \to \langle c_2, \; \sigma' \rangle}$$

$$\frac{}{\langle \textbf{abort} \; ; \; c, \; \sigma \rangle \to \langle \textbf{abort}, \; \sigma \rangle}$$

$$\frac{\sigma(b) = \mathsf{true}}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \; \sigma \rangle \to \langle c_1, \; \sigma \rangle} \qquad \frac{\sigma(b) = \mathsf{false}}{\langle \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2, \; \sigma \rangle \to \langle c_2, \; \sigma \rangle}$$

$$\frac{\sigma(b) = \mathsf{true}}{\langle \textbf{while } b \textbf{ do } c, \; \sigma \rangle \to \langle c \; ; \; \textbf{while } b \textbf{ do } c, \; \sigma \rangle} \qquad \frac{\sigma(b) = \mathsf{false}}{\langle \textbf{while } b \textbf{ do } c, \; \sigma \rangle \to \sigma}$$

Figure 6.5: Operational semantics of the WHILE language.

of expressions (e.g., to accept more operators than $\leq$), but those examples can be trivially converted to the WHILE language we present. Additionally, we use two additional commands, **assume** and **assert**, as syntactic sugar, which are defined as follows:

$$\textbf{assume } b \equiv \textbf{if } b \textbf{ then skip else } (\textbf{while } 0 \leq 0 \textbf{ do skip})$$

$$\textbf{assert } b \equiv \textbf{if } b \textbf{ then skip else abort}$$

Let σ be a program state, which is a map from program variables to integers and from UF symbols to maps from tuples of integers (of the same arity as the function) to integers (an interpretation of the UFs). Let $\sigma(v)$ be the value of variable v in program state σ. Let $\sigma(f)(v_1, \ldots, v_n)$ be the value of the interpretation of the UF symbol f in σ applied to $v_1, \ldots, v_n$. This notation is extended for expressions, such that $\sigma(e)$ is the value of expression e with each variable evaluated in σ. Let $\sigma[v \mapsto n]$ be a program state that is identical to state σ, except for the value of variable v, which is n. Let σ_0 be the initial state of an execution of a program. We have that $\sigma_0(v) = v_0$ and $\sigma_0(f) = f_0$ for each variable v and UF symbol f used in the program, with fresh variables v_0 and arbitrary maps f_0.

A configuration $\langle c, \; \sigma \rangle$ is a pair where c is a command and σ is a state. Let $\langle c, \; \sigma \rangle \to \langle c', \; \sigma' \rangle$ be the reduction of the configuration $\langle c, \; \sigma \rangle$ to the configuration $\langle c', \; \sigma' \rangle$ in one step. Let $\langle c, \; \sigma \rangle \to \sigma'$ be the reduction in one step of the configuration $\langle c, \; \sigma \rangle$ to the state σ' when there are no further commands left to execute. Finally, let $\langle c, \; \sigma \rangle \to^* \sigma'$ be the reduction in one or more steps of the configuration $\langle c, \; \sigma \rangle$ to the state σ'.

The operational semantics of the commands of the WHILE language is shown in Figure 6.5. The command **abort** is irreducible, and therefore there is no corresponding reduction rule for it.

A program P (a command) is said to be safe iff there is no initial state σ_0 such that P terminates in

an irreducible command, i.e., P is safe iff

$$\neg\exists\sigma_0 : \langle P,\ \sigma_0\rangle \rightarrow^* \langle \mathbf{abort},\ \sigma'\rangle$$

Let $\mathsf{Vars}(P)$ be the set of variables of program P. A variable v is fresh in program P if $v \notin \mathsf{Vars}(P)$. Let $\mathsf{Out}(P) \subseteq \mathsf{Vars}(P)$ be the set of output observable variables of a program P (defined by the user). Let $\sigma \downarrow V$ be the projection of state σ over the set of variables V and let $\sigma \downarrow \mathsf{Out}(P)$ be the observable state of σ of program P.

Two programs are considered partially equivalent iff starting in the same arbitrary state, they terminate in the same observable state for all possible UF interpretations, i.e., P_1 and P_2 are partially equivalent iff the following holds:

$$\langle P_1,\ \sigma_0\rangle \rightarrow^* \sigma_1 \wedge \langle P_2,\ \sigma_0\rangle \rightarrow^* \sigma_2 \implies \sigma_1 \downarrow V = \sigma_2 \downarrow V$$

with $V = \mathsf{Out}(P_1) = \mathsf{Out}(P_2)$.

6.3 Restrictions

We impose the following restrictions on the programs that our equivalence checking algorithm can handle:

1. UFs must have exactly one output parameter.
2. There can be no branching (i.e., **if** statements) inside loops. Nested loops, however, are allowed.
3. The trip count of inner loops may not depend on the outer loops, i.e., the number of times that inner loops iterate is constant relative to outer loops.
4. Loop conditions cannot include UF applications nor depend on variables whose value may depend directly or indirectly on the evaluation of an UF application.

Restriction 1 can be lifted by splitting UFs with more than one output into newly created UFs (one per output).

Restriction 2 can be relaxed by allowing branching conditions that always evaluate to the same value in all loop iterations. In that case, the program can be rewritten to move the branches out of the loop (transformation commonly known as loop unswitching [ALSU06]). Similarly, phase-change loops can be rewritten as multiple loops, using, e.g., splitter predicates [SDDA11].

We speculate that Restriction 4 could be lifted, and give a brief discussion in Section 6.7.

6.4 The Algorithm

The algorithm has four steps:

1. Sequential composition of the two programs.
2. Eliminate UF applications.
3. Replace loops with recurrences.
4. Check the correctness of the resulting program.

$$\mathsf{T}(e) = \begin{cases} n & \text{if } e = n \\ v & \text{if } e = v \\ \mathsf{T}(e_1) \oplus \mathsf{T}(e_2) & \text{if } e = e_1 \oplus e_2 \\ \mathsf{p}\,(f, \mathsf{T}(e_1), \ldots, \mathsf{T}(e_n)) & \text{if } e = f(e_1, \ldots, e_n) \end{cases}$$

$$\mathsf{T}(b) = \begin{cases} \mathsf{T}(e) \leq 0 & \text{if } b = e \leq 0 \\ \mathsf{T}(b_1) \otimes \mathsf{T}(b_2) & \text{if } b = b_1 \otimes b_2 \end{cases}$$

$$\mathsf{T}(c) = \begin{cases} \textbf{skip} & \text{if } c = \textbf{skip} \\ v \ := \ \mathsf{T}(e) & \text{if } c = v \ := \ e \\ \mathsf{T}(c_1) \ ; \ \mathsf{T}(c_2) & \text{if } c = c_1 \ ; \ c_2 \\ \textbf{if } \mathsf{T}(b) \textbf{ then } \mathsf{T}(c_1) \textbf{ else } \mathsf{T}(c_2) & \text{if } c = \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \\ \textbf{while } \mathsf{T}(b) \textbf{ do } \mathsf{T}(c_1) & \text{if } c = \textbf{while } b \textbf{ do } c_1 \\ \textbf{abort} & \text{if } c = \textbf{abort} \end{cases}$$

Figure 6.6: Definition of the program transformation T.

Applications of UFs are abstracted using polynomials in order to obtain programs with integer operations only. This allows us to compute the closed-form of loops using recurrences.

Although our algorithm is sound and relatively complete (under the stated restrictions and for certain variable domains), computing the closed-form solution of recurrences is undecidable, and therefore the overall method is incomplete. A thorough discussion on the completeness of the algorithm is given in Section 6.7.

In the following sections, we describe each step of the algorithm separately.

6.4.1 Sequential Composition

The first step of the algorithm is to do the sequential composition of the two input programs that we would like to check for equivalence. The second program is renamed so that it operates over a different set of variables from the first.

Let P_1 and P_2 be the two input programs. The composed program is as follows.

assume $\forall v \in \mathsf{Vars}(P_1) \cap \mathsf{Vars}(P_2) : v = \bar{v}$
P_1
$\bar{P_2}$
assert $\forall v \in \mathsf{Out}(P_1) : v = \bar{v}$

Program $\bar{P_2}$ is the same as the program P_2, but where each variable v was renamed to $\bar{v}$. Moreover, we assume that $\mathsf{Out}(P_1) = \mathsf{Out}(P_2)$.

6.4.2 Eliminate UF applications

The second step of the algorithm is to eliminate UF applications. This is accomplished by replacing each UF application with a polynomial over its inputs. This rewriting must only preserve program safety (i.e., the program is safe iff the rewritten program is safe). The program transformation T implements such a replacement and is shown in Figure 6.6.

The polynomial $\mathsf{p}(f, e_1, \ldots, e_n)$ can be defined in multiple ways in order to accomplish our goal of preserving safety. For the domain of rationals, reals or complex numbers, we can use a standard polynomial usually used to interpolate functions with multiple inputs [GS00, Olv06], which is as follows:

$$\sum_{\alpha \cdot \mathbf{1} \leq d} C_\alpha \, X^\alpha$$

where $C = \begin{pmatrix} f_1 & \cdots & f_m \end{pmatrix}$ is an m-tuple containing variables f_i associated with the given UF symbol f, $X = \begin{pmatrix} e_1 & \cdots & e_n \end{pmatrix}$ is an n-tuple with the input values of the given UF application, the exponent vector $\alpha = \begin{pmatrix} \alpha_1 & \cdots & \alpha_n \end{pmatrix}$ is an ordered partition with nonnegative entries, and $\alpha \cdot \mathbf{1} = \sum_{i=0}^{n} \alpha_i$ is the usual vector dot product. $X^\alpha = \prod_{i=1}^{n} X_i^{\alpha_i}$ is a monomial of degree $\sum_{i=0}^{n} \alpha_i$, with $X_i = e_i$, and C_α being the element of C corresponding to α.

This summation produces a polynomial where each term (a monomial) has a degree up to d. The degree of a monomial is the sum of the exponents of its variables. For example, x^3 and $x\,y^2$ both have degree three. Polynomial p is, therefore, a summation of all combinations of monomials of degree up to d with n variables (the number of inputs to the UF application). For example, if we have $d = 3$, $\mathsf{p}(f, x, y)$ would be equal to:

$$f_1\,x^3 + f_2\,y^3 + f_3\,x^2\,y + f_4\,x\,y^2 + f_5\,x^2 + f_6\,y^2 + f_7\,x\,y + f_8\,x + f_9\,y + f_{10}$$

The maximum degree d of the monomials is the smallest nonnegative integer that satisfies the following constraint:

$$\mathsf{u}(f) \leq \binom{n+d}{n}$$

where $\binom{n}{k} = \dfrac{n!}{k!\,(n-k)!}$ is the binomial coefficient. We use $m = \binom{n+d}{n}$ to denote the number of monomials of p.

A discussion of the presented polynomial and of alternatives for $\mathsf{p}(f, e_1, \ldots, e_n)$ for other domains is given in Section 6.8.

The value of $\mathsf{u}(f)$ is the maximum number of times that the given uninterpreted function f is possibly applied with a set of distinct values in each and every *static* program path. Only function applications whose value is possibly used in a boolean expression need to be considered. Function applications appearing in a loop body are only counted once.

Two UF applications are equivalent iff they are of the same UF symbol and have the same input values, for all possible program input. Transformation T captures this information precisely by replacing each UF application with a polynomial over the inputs of the application. Each UF symbol is assigned a set of fresh variables f_i that is used only by applications of that symbol. Therefore, and together with results from the domain of polynomial interpolation (shown in Section 6.7), we can guarantee that the value of an UF application cannot be reproduced by any sequence of commands for all inputs.

For example, the following boolean expression

$$f(x, y) = 0 \wedge f(x, z) = 3 \wedge f(w, z) = 1 \wedge f(2, 3) = 0 \wedge$$
$$g(x) \leq 0$$

is translated to (assuming no more applications of f nor g in the rest of the program):

$$
\begin{aligned}
& f_1\, x^2 + f_2\, y^2 + f_3\, x\, y + f_4\, x + f_5\, y + f_6 = 0 \wedge \\
& f_1\, x^2 + f_2\, z^2 + f_3\, x\, z + f_4\, x + f_5\, z + f_6 = 3 \wedge \\
& f_1\, w^2 + f_2\, z^2 + f_3\, w\, z + f_4\, w + f_5\, z + f_6 = 1 \wedge \\
& 2^2\, f_1 + 3^2\, f_2 + 6\, f_3 + 2\, f_4 + 3\, f_5 + f_6 = 0 \wedge \\
& g_1 \leq 0 \wedge
\end{aligned}
$$

where all f_i and g_1 are fresh variables. These variables are never written by the program, and are only read by transformed expressions that originally contained the same UF symbols (f and/or g).

In this expression we have four applications of f with (possibly) different input parameters. Therefore, we have $\mathsf{u}(f) = 4$, and also $n = 2$ (since f has two input parameters). The smallest d such that $4 \leq \binom{2+d}{2}$ is $d = 2$, and so each polynomial has six terms ($m = \binom{4}{2} = 6$). The applications of the uninterpreted function f were, consequently, transformed into summations of all the six monomials of two variables of degree up to two.

Computing the value of $\mathsf{u}(f)$ as defined is hard (and is in fact undecidable in general), and thus may require prior static analysis. This value can, however, be safely over-approximated by the number of syntactic occurrences of f in the whole program, at the expense of generating more complex expressions.

For example, the optimal value for u in the following program excerpt is $\mathsf{u}(f) = 3$ (assuming no other UF applications in the rest of the program). Although there are four applications of f with possibly distinct input values, only up to three applications are ever statically encountered and used in a boolean expression in a single path. Moreover, the application of f in the loop body is counted only once in $\mathsf{u}(f)$, despite that that application may appear multiple times in a path in which the loop is traversed more than once.

if ... **then**
  $j := f(x)$
else
  $k := f(y)$

while ... **do**
  $l := f(l)$

if $f(z) \leq 0 \wedge j \leq 0 \wedge k \leq 0 \wedge l \leq 0$ **then**
  ...

The proof of soundness and completeness of transformation T, i.e., that program P is safe iff program $\mathsf{T}(P)$ is safe is given in Section 6.7.

6.4.3 Replace loops with recurrences

The third step of the algorithm is to eliminate loops, by replacing each loop with a system of recurrences. The transformation is carried out as follows. Each variable that is assigned in the loop gets a recurrence over a newly introduced variable that represents the loop trip count. For nested loops, the initial value of a recurrence in an inner loop is the value of the recurrence for the previous iteration of the outer loop.

An example program and its system of recurrences is shown in Figure 6.7. The recurrences $R_v(n)$ and $V_v(n)$ represent the value of variable v at iteration n of the inner loop and the outer loop, respectively.

```
while i < n do
    k := 2 × k
    j := 0
    while j < m do
        k := k + j
        j := j + 1
    i := i + 1
```

$$
\begin{aligned}
R_j(x) &= R_j(x-1) + 1 \\
R_j(0) &= 0 \\
R_k(x) &= R_k(x-1) + R_j(x-1) \\
R_k(0) &= 2\,V_k(y-1) \\
V_i(y) &= V_i(y-1) + 1 \\
V_i(0) &= i_0 \\
V_k(y) &= R_k(x) \\
V_k(0) &= k_0
\end{aligned}
$$

Figure 6.7: An example program and the corresponding system of recurrences that summarizes the two loops, where R_j and R_k represent the behavior of the inner loop on the variables j and k, respectively, and V_i and V_k represent the outer loop.

For example, the value of variable k in the iteration x of the inner loop, $R_k(x)$, is equal to the sum of the values of variables k and j of the previous (inner loop) iteration. The value of k in the beginning of the first inner loop iteration, $R_k(0)$, is equal to twice the value of k in the previous outer loop iteration.

The closed-form solution for the system of recurrences is the following:

$$
\begin{aligned}
R_j(x) &= x & R_k(x) &= \frac{4\,V_k(y-1) + x^2 - x}{2} \\
V_i(y) &= i_0 + y & V_k(y) &= k_0\,2^y + \frac{\left(x^2 - x\right)\,\left(2^y - 1\right)}{2}
\end{aligned}
$$

We note that while the solution of $R_k(x)$ still includes a reference to a recurrence — $V_k(y-1)$ — it is only used to compute the solution of $V_k(y)$ and it is never used directly by the next steps of the algorithm. We only need the value of k after the outer loop terminates, which is represented by $V_k(y)$.

After computing the closed-form solution for the system of recurrences, each loop of the form "**while** b **do** c" is replaced with the following code:

```
if b then
    assume σ_{n-1}(b) ∧ σ_n(¬b)
    v := σ_n(v)
else
    assume n = 0
```

The fresh variable n represents the number of iterations performed by the loop. State σ_n maps each variable to the closed-form solution of its corresponding recurrence at iteration n, or to itself if the variable is not modified in the loop body c. Variable v ranges over all variables that are possibly modified in the loop body. For the previous example, we have for the inner loop that, e.g., $\sigma_x(j) = R_j(x) = x$ and $\sigma_x(n) = n$.

Intuitively, a loop executes n times if the loop guard is true for the first n iterations (iterations $0 \ldots (n-1)$) and false in the following iteration (iteration n). The number of iterations is implicitly computed when the **assume** command of the true branch is evaluated. Its expression states that the loop guard of iteration $n-1$ should be true, and that at iteration n the guard should be false instead.

We note that there can be multiple solutions for the expression given to the **assume** command if the loop guard is non-linear. In this case, the number of loop iterations is the smallest positive n that makes the formula satisfiable. Computing the smallest n can be achieved, for example, by using an optimizing solver (e.g., [LAK+14]) or by doing multiple calls to an SMT solver. The condition for non-linear guards

```
if i < n then
   assume V_i(y − 1) < n ∧ V_i(y) ≥ n
   j := 0
   if j < m then
      assume R_j(x − 1) < m ∧ R_j(x) ≥ m
      j := R_j(x)
   else
      assume x = 0
   k := V_k(y)
   i := V_i(y)
else
   assume y = 0
```

Figure 6.8: Program of Figure 6.7 after replacing the loops with a set of assignments over the system of recurrences including $V_i(n)$, $V_k(n)$, and $R_j(n)$.

can also be expressed directly using a quantified formula (with a single quantifier alternation).

For the example in Figure 6.7, the program after removing the loops is shown in Figure 6.8. The command "**assume** $y = 0$" at the end can be removed as an optimization, since there are no further uses of y afterward.

6.4.4 Safety Checking

The fourth and final step of the algorithm is to prove the resulting composed program correct. If the composed program is safe, i.e., if the condition of the **assert** command is true for all inputs, then the two original programs are partially equivalent.

To prove program safety, we can use standard software verification techniques (e.g., model checking [JM09]). Since the number of paths is finite, we can also use an algorithm that enumerates all paths and tests if any of those makes the condition of the **assert** command falsifiable.

6.5 Compiler Optimization Verification as Program Equivalence

We specify a compiler optimization as a transformation function from a source template program to a target template program. These template programs can be modeled as UF+IA programs, where UFs represent arbitrary statements, expressions, or conditions that should be matched within a program under optimization.

To verify a compiler optimization correct, we split the transformation function into two programs (the source and target templates), and then we convert the template programs into UF+IA programs. Finally, we use the proposed equivalence checking algorithm to prove that the source and target templates are equivalent, which implies that the optimization is correct.

The conversion of a template program to an UF+IA program is done by replacing each template statement S with a set of assignments of the following form:

$$v \;:=\; S_i(r_1, \ldots, r_n)$$

where $v \in \mathsf{W}(\mathsf{S})$ and $\mathsf{R}(\mathsf{S}) = \{r_1, \ldots, r_n\}$. Template expressions are replaced with a single UF application over their read set.

6.6 Evaluation

We implemented a prototype named CORK[2], which stands for Compiler Optimization coRrectness checKer. CORK is implemented in OCaml (with approximately 1,100 lines of code), and uses Wolfram Mathematica 8.0.4 for both constraint and recurrence solving.

CORK takes as input a transformation function in the format of the example in Figure 1.2. CORK then derives two programs over the UF+IA theory as described in the previous section, and subsequently checks if they are equivalent. The equivalence check is done by enumerating each path of the composed program, since the number of paths is finite and small, and then using Mathematica to check validity of the equivalence assertion. If the equivalence check fails, CORK prints a counterexample path.

CORK performs three optimizations to improve the performance. First, CORK reduces the number of satisfiability queries issued to Mathematica by discharging itself equality tests of syntactically equal expressions. Second, CORK performs equality propagation on the satisfiability queries sent to Mathematica. Finally, CORK checks the equality of program variables (arising from the **assert** command at the end of the composed program) one-by-one, instead of just one satisfiability query per path. CORK then uses the established equalities in the following queries. Moreover, variable equality checks are ordered so that first are checked the induction variables, and the remaining variables are ordered by the length of their value expressions. Establishing first the equality of expressions involving induction variables improves the performance significantly.

We ran CORK over a set of optimizations (mostly loop-manipulating). The experiments were run on a machine running Linux 3.6.2 with an Intel Core 2 Duo 3.00 GHz CPU, and 4 GB of RAM. The results are shown in Table 6.1.

We first note that the number of recurrence solving queries is higher than expected (more than one per loop), since we compute the recurrences per path and we do not cache any information across paths. Optimizations that do not manipulate loops explicitly do not generate any recurrence.

We compare the results of CORK with the state-of-the-art tool PEC [KTL09]. Since PEC is not publicly available, we compare only with the published results.

The table is divided in four sets of optimizations (described in, e.g., [ALSU06, Muc97]). The first part is a set of optimizations that do not manipulate loops explicitly. These optimizations are trivially proven correct by both CORK and PEC. The second part is a set of optimizations that PEC can prove correct without the help of heuristics. The third part is a set of optimizations that PEC can only prove correct by using the permute heuristic [GZB05, ZPG+05], since otherwise it could not find a bisimulation relation automatically. The fourth and last part of the table contains a set of optimizations that PEC cannot prove correct, since it cannot find a bisimulation automatically, even with the permute heuristic. CORK, on the other hand, is able to prove correct the loop strength reduction and loop tiling optimizations. CORK fails to prove correct the loop flattening optimization, since Mathematica could not finish the satisfiability check of a constraint within the timeout of 15 minutes.

The execution times of PEC and CORK are within the same order of magnitude, but CORK advances the state-of-the-art by being able to prove correct more optimizations than PEC.

6.7 Proof of Soundness and Completeness

Let $\mathsf{S}^P(\sigma)$ be a copy of state σ where the interpretation of every uninterpreted function (UF) symbol is replaced with values for variables f_i used in Section 6.4.2. Moreover, the values for variables f_i and the initial variables values v_0 are chosen such that for every boolean expression b appearing in program P, it

[2]Prototype and benchmarks available from `http://web.ist.utl.pt/nuno.lopes/cork/`.

Optimization	PEC	Queries	Recurrences	Time
Code hoisting	✓	2	0	0.32s
Constant propagation	✓	0	0	0.33s
Copy propagation	✓	0	0	0.33s
If-conversion	✓	2	0	0.34s
Partial redundancy elim.	✓	2	0	0.34s
Loop inv. code motion	✓	7	5	3.48s
Loop peeling	✓	9	5	3.26s
Loop unrolling	✓	13	8	12.17s
Loop unswitching	✓	14	14	8.19s
Software pipelining	✓	9	5	8.02s
Loop fission	$\checkmark_p$	10	12	23.45s
Loop fusion	$\checkmark_p$	10	12	23.34s
Loop interchange	$\checkmark_p$	15	24	29.30s
Loop reversal	$\checkmark_p$	7	5	8.41s
Loop skewing	$\checkmark_p$	16	24	8.50s
Loop flattening	×	—	—	T/O
Loop strength reduction	×	6	4	5.63s
Loop tiling	×	7	9	10.94s

Table 6.1: List of compiler optimizations [ALSU06, Muc97], how PEC performs ($\checkmark_p$ means PEC needs the permute heuristic), the number of satisfiability and recurrence solving queries issued to Mathematica, and the time that CORK took to prove each optimization correct.

is guaranteed that $\sigma(b) = \mathsf{S}^P(\sigma)(\mathsf{T}(b))$. The justification of the existence of such an assignment is given in Theorem 1.

In the remainder of this section, we use the term free variable to denote logic or program variables (depending on the context) that are not constrained and therefore can take any value. In particular, a free program variable is never assigned to and cannot be constrained in any program path.

Let $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{C}$ be, respectively, the set of rational, real, and complex numbers.

Lemma 1 (Solution for nested $f(x)$)**.** *For a function $f(x) = a_n\, x^n + \ldots + a_1\, x + a_0$, an arbitrary number of nested applications of f to x can take any value in the codomain ($\mathbb{R}$ or $\mathbb{C}$) if x and a_n are free, i.e., $f(f(\ldots f(x)\ldots)) = b$ always has a solution for fixed $a_{n-1}, \ldots, a_0, b$ and free a_n and x.*

Proof. The maximum degree of the polynomial given by $p = f(f(\ldots f(x)\ldots)) - b$ in x is n^k, where $k > 0$ is the number of applications of f. If n (the degree of x in $f(x)$) is odd, then n^k will be odd as well. Therefore, if n is odd, it follows from the intermediate value theorem [Str80] that there always exists a value for x for arbitrary $a_n, \ldots, a_0, b$ such that $p = 0$.

The maximum degree of a_n in p is given by the following recurrence: $d(k) = n\, d(k-1) + 1$ and $d(0) = 0$. The closed-form solution for this recurrence is $d(k) = \dfrac{n^k - 1}{n - 1}$. Now assume that n is even, since we already proved the lemma for n odd. We can then conclude that $d(k)$ is odd for any nonnegative k, and therefore there exists a_n for arbitrary $a_{n-1}, \ldots, a_0, b, x$ such that $p = 0$. □

Lemma 2 (Solution for conjunction of nested $f(x)$)**.** *For a function $f(x) = a_n\, x^n + \ldots + a_1\, x + a_0$, a conjunction of nested applications of f of the form $f(f(\ldots f(x_1)\ldots)) = b_1 \wedge \ldots \wedge f(f(\ldots f(x_q)\ldots)) = b_q$ is satisfiable if any of the following statements holds:*

1. *Coefficients a_i range over $\mathbb{C}$ and at least $q \leq n + 1$ of those are free.*
2. *Variables x_i range over $\mathbb{C}$ and are free.*
3. *n is odd and variables x_i range over $\mathbb{R}$ and are free.*

4. $q = 1$ *and* a_n *and* x_1 *range over* $\mathbb{R}$ *and are free.*

Proof. For condition 1, we note that there is at least one free coefficient a_i for each polynomial in the conjunction. Then it follows from the fundamental theorem of algebra [Str80] that it is always possible to find values for the free a_i that satisfy the equalities. Similar reasoning apply for condition 2, where each equality can be solved in order of its respective x_i.

Conditions 3 and 4 follow directly from Lemma 1. □

We now state under which conditions the transformation T as given in Section 6.4.2 is sound and complete, which we will use later to prove Theorems 1 and 2.

Definition 1. T *is sound and complete if one of the following statements holds:*

1. *There are no nested applications of UFs in loops and program variables range over* $\mathbb{Q}$, $\mathbb{R}$, *or* $\mathbb{C}$.
2. *Variables* f_i *range over* $\mathbb{C}$.
3. *There is only one nested UF application produced by a loop, say* $f(f(\ldots f(x)\ldots))$, *with* x *being free, and variables* f_i *and* x *ranging over* $\mathbb{R}$.
4. $\mathsf{u}(f)$ *is odd for all* f *appearing in nested applications in loops, and the input to these applications are variables that are free and range over* $\mathbb{R}$.

We note that $\mathsf{u}(f)$ can always be arbitrarily increased (to, e.g., become odd) if need be. Also, in order to guarantee soundness, program variables and polynomial coefficients can be changed to take values in larger domains (say, convert from $\mathbb{Z}$ to $\mathbb{R}$), by giving up on completeness. With such a change, the algorithm remains sound, i.e., if it proves that two programs are equivalent then they are. However, losing completeness means that the algorithm may fail to prove equivalence of two equivalent programs because a larger variable domain may increase the set of possible behaviors/outcomes of a program, which can lead to the loss of equivalence.

Definition 2. *We define* statically implied equalities *of UF symbols as the set of all equalities involving applications of UFs that are implied by any* static *path in a given program (e.g.,* $f(x) = 3$*). Nested applications of UFs arising from loops are not unfolded. For example, for a program "****while*** *...* ***do*** $x := f(x)$*" and a path that traverses the loop three times, we only consider the equality* $x = f(f(f(x_0)))$.

Theorem 1 (Existence of $\mathsf{S}^P(\sigma_0)$)**.** *For every program* P *respecting Definition 1,* σ_0 *is a possible initial state of* P *iff* $\mathsf{S}^P(\sigma_0)$ *also is.*

Proof. If P does not contain any application of UF symbols, then the statement is trivially correct, since $P = \mathsf{T}(P)$ and therefore $\sigma_0 = \mathsf{S}^P(\sigma_0)$.

Otherwise, we consider the set of statically implied equalities of UF symbols. Let c be the conjunction of the elements of said set that refer only to non-nested UF applications, and r the conjunction of the remaining elements (arbitrarily nested applications from loops). Moreover, we trivially have that $\sigma_0(b) = \sigma_0(c \wedge r)$.

We now assume that all UFs have only one input parameter and that there is only one UF symbol f. Therefore, $\mathsf{T}(c)$ can be seen as a linear system $Ax = b$, where A is a square matrix of size $n \times n$ with the powers 0 to $(n-1)$ of the input parameters of the UF applications, and x is a vector with fresh variables f_i. Moreover, A is a Vandermonde matrix [Str80].

For example, for $c = f(x_1) = b_1 \wedge \cdots \wedge f(x_n) = b_n$, $\mathsf{T}(c)$ results in the following linear system:

$$\begin{bmatrix} 1 & x_1^1 & \cdots & x_1^{n-1} \\ 1 & \vdots & \ddots & \vdots \\ 1 & x_n^1 & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

If $x_i \neq x_j$ for all $i \neq j$, then c is satisfiable. Moreover, the lines and the columns of the coefficient matrix A are linearly independent, which guarantees that the system has a solution (by the unisolvence theorem [Str80]). Therefore, $\mathsf{T}(c)$ is also satisfiable.

If there are i, j with $i \neq j$ such that $x_i = x_j$, and c is satisfiable, then $b_i = b_j$. In this case, the corresponding system of $\mathsf{T}(c)$ has infinitely many solutions, and therefore $\mathsf{T}(c)$ is satisfiable as well.

Finally, if c is unsatisfiable, then there are i, j with $i \neq j$ such that $x_i = x_j$ and $b_i \neq b_j$. The linear system of $\mathsf{T}(c)$ has no solution, and therefore $\mathsf{T}(c)$ is unsatisfiable as well.

If c is unsatisfiable, then there is no interpretation for the UF symbols that makes c be true, and therefore we have $\sigma_0(c) = \mathsf{S}^P(\sigma_0)(\mathsf{T}(c)) = \mathsf{false}$. If c is satisfiable, then $\sigma(c)$ may or may not be true depending on the interpretation of the UFs in σ_0, but we are always guaranteed to be able to find coefficients for $\mathsf{S}^P(\sigma_0)$ either way such that $\sigma_0(c) = \mathsf{S}^P(\sigma_0)(\mathsf{T}(c))$.

If c contains UFs symbols with more than one parameter, then the resulting polynomials in $\mathsf{T}(c)$ are more complex. Similar reasoning can be done by using generalized versions of the unisolvence theorem for multivariate polynomial interpolation (c.f., [GS00, Olv06]).

If c contains multiple UF symbols, the evaluation of c can be split in multiple linear systems, one per symbol.

If r is empty, then the proof is completed.

Otherwise, let $\#c$ and $\#r$ be, respectively, the number of equalities in c and r. By definition of u, we have for any f that $\#c + \#r \leq \mathsf{u}(f)$. Moreover, only the first $f_1, \ldots, f_{\#c}$ coefficients are defined by c, and the remaining $f_{\#c+1}, \ldots, f_{\mathsf{u}(f)}$ remain free. Therefore, the proof follows immediately from Lemma 2.

For UFs with more than one input parameter, Lemma 2 also applies by observing that the degree of the polynomial obtained by nested applications is dominated by the nested input variable and the coefficient of that parameter. □

From Theorem 1, it follows that if $\mathsf{u}(f)$ is odd, then $\mathsf{u}(f)$ does not need to count with applications with free variables as input. This fact can be used as an optimization to reduce the degree of polynomials to the smallest odd number that is greater than or equal to the number of applications with non-free inputs.

Theorem 2 (Soundness and completeness of T)**.** *Transformation* T *preserves safety of programs, i.e., for any state* σ_0 *and program* P *respecting Definition 1, the following holds:*

$$\langle P,\ \sigma_0\rangle \to^* \sigma \iff \langle \mathsf{T}(P),\ \mathsf{S}^P(\sigma_0)\rangle \to^* \sigma'$$

Proof. The proof goes by structural induction on the syntax of P.

The base cases are: $P = \mathbf{skip}$, $P = v := e$, and $P = \mathbf{abort}$, which are all trivially correct.

For the induction step, we need to consider three cases. As the induction hypothesis, assume that the theorem holds for commands c_1 and c_2.

For $P = \mathbf{if}\ b\ \mathbf{do}\ c_1\ \mathbf{else}\ c_2$, we have $\mathsf{T}(P) = \mathbf{if}\ \mathsf{T}(b)\ \mathbf{do}\ \mathsf{T}(c_1)\ \mathbf{else}\ \mathsf{T}(c_2)$. By definition of $\mathsf{S}^P(.)$, we know that $\sigma(b) = \mathsf{S}^P(\sigma)(\mathsf{T}(b))$ and therefore P reduces to c_1 (resp. c_2) iff $\mathsf{T}(P)$ reduces to $\mathsf{T}(c_1)$ (resp. $\mathsf{T}(c_2)$).

For $P = \mathbf{while}\ b\ \mathbf{do}\ c_1$, we note that since b cannot include nor depend on UF applications (per restriction 4 in Section 6.3), then $\mathsf{T}(b) = b$, and therefore $\mathsf{T}(P) = \mathbf{while}\ b\ \mathbf{do}\ \mathsf{T}(c_1)$. Moreover, we have that $\sigma_1(b) = \sigma_1'(b)$ for every states σ_1 and σ_1' resulting from the reduction of c_1 and $\mathsf{T}(c_1)$, respectively, since b cannot depend on the result of any UF symbol. Therefore we are left to prove that $\langle c_1\ ;\ \ldots\ ;\ c_1,\ \sigma_0\rangle \to^* \sigma$ iff $\langle \mathsf{T}(c_1)\ ;\ \ldots\ ;\ \mathsf{T}(c_1),\ \mathsf{S}^P(\sigma_0)\rangle \to^* \sigma'$, which is covered in the following case.

For $P = c_1\ ;\ c_2$, assume that $\langle c_1,\ \sigma_0\rangle \to^* \sigma_1$ and $\langle \mathsf{T}(c_1),\ \mathsf{S}^P(\sigma_0)\rangle \to^* \sigma_1'$. If $\mathsf{S}^P(\sigma_1) = \sigma_1'$, then the theorem is trivially correct. Otherwise, and without loss of generality, consider that $\mathsf{S}^P(\sigma_1)$ and

σ_1' differ only in the value of variable v because c_1 contained an assignment of the form $v := f(x)$. Let c_2' be a copy of c_2 where references to v were replaced with $f(x)$. Therefore, our proof goal of $\langle c_2,\ \sigma_1\rangle \to^* \sigma_2 \iff \langle \mathsf{T}(c_2),\ \sigma_1'\rangle \to^* \sigma_2'$ is equivalent to $\langle c_2',\ \sigma_1\rangle \to^* \sigma_2'' \iff \langle \mathsf{T}(c_2'),\ \mathsf{S}^{c_2'}(\sigma_1)\rangle \to^* \sigma_2'''$, which holds per the induction hypothesis. □

We speculate, but leave the proof for future work, that restriction 4 in Section 6.3 could be lifted altogether, i.e., it may be possible to allow UFs in loop guards. We believe this could be done by counting UF symbols in loop guards twice when computing $\mathsf{u}(f)$ for any symbol f. Intuitively, we may only need to interpolate the values of an UF symbol when the loop guard flips (i.e., when in one iteration it was true and in the following it became false).

6.8 Discussion on Polynomial Interpolation

The polynomial for $\mathsf{p}(f, e_1, \ldots, e_n)$ given in Section 6.4.2 requires coefficients to range over the set of rational ($\mathbb{Q}$), real ($\mathbb{R}$), or complex numbers ($\mathbb{C}$). Therefore, for the domain of integers ($\mathbb{Z}$), it is unsound to use the given polynomial, since in general there may not exist integer values for variables f_i such that Theorem 1 holds.

Integer-valued polynomials are polynomials with coefficients in some domain, whose value for every point (or for every interpolating point) is an integer [CC97]. In particular, it is possible to interpolate a set of points using integer-valued polynomials with rational coefficients [Fri99, CCF00]. However, these polynomials can only be used if the verification tool used in the algorithm supports rational numbers and their combined operation with integer variables from the remainder of the program.

There is still ongoing research on interpolation by integer-valued polynomials that may yield interesting results that could be of use for our algorithm. We leave as a conjecture that the following polynomial can interpolate any set of $n+1$ integer points:

$$f(x) = \sum_{i=0}^{n} \left\lfloor \frac{a_i\, x^i}{b_i} \right\rfloor$$

where a_i and b_i are integer coefficients, and $\left\lfloor \frac{x}{y} \right\rfloor$ is the integer division. A drawback of this polynomial is that solving recurrences with integer division is harder than with, say, division in $\mathbb{Q}$, because the function may become discontinuous. Moreover, it is unclear whether it would be possible to amend Theorem 1 for such a polynomial.

6.9 Summary

In this chapter, we presented a new algorithm for the automatic verification of partial equivalence of programs under the UF+IA theory. We also showed how this algorithm can be applied for the verification of correctness of compiler optimizations. Finally, equipped with this technique, we implemented a tool that can prove more optimizations correct than previously known approaches.

Chapter 7

Discussion and Future Work

The work presented in this document is a small step towards a much needed evolution in how compilers function internally and how compilers are developed today. In this chapter, the shortcomings of the presented techniques are presented, as well as some of the remaining work left for the future.

7.1 Specification Languages

We argued that the presented language for specifying transformation functions and the language to specify preconditions for said functions are adequate. Moreover, they are similar to languages used in previous work (e.g., [KTL09, TL10a]).

Transformation Functions Regarding the specification language for transformation functions, we presented several classic optimizations specified in this language. However, the language lacks, most noticeably, support for function calls, pointers, arrays, heap allocation, and so on. Support for these language features is required in order to support another set of optimizations, such as call devirtualization, function inlining, bounds-check elimination, conversion of heap to stack allocations, etc.

Another class of optimizations that we currently miss are automatic vectorization and automatic parallelization. Both require extensions to the language in order to be able to represent SIMD instructions and to reason about concurrency.

We did not include support for floating-point numbers, since that would require entirely different algorithms to perform automatic reasoning and equivalence checking. Likewise, overflows in arithmetic operations are ignored, since all variables are assumed to be of infinite precision.

The lack of features in the specification language does not, however, limit the set of possible instantiations for template statements/expressions. For example, a template expression may be instantiated with a load from an array, despite the specification language not supporting such a feature, as long as the instantiation fulfills the precondition (which can be verified using, e.g., a data dependency analysis).

Preconditions Regarding the language used to specify preconditions of transformation functions, we argued that the language of read and write sets is a good means to communicate with compiler developers and that it can be used to generate verification conditions that can be discharged efficiently.

The language is not, however, sufficiently expressible to specify preconditions of all optimizations nor to express the *weakest* precondition of some. For example, we do not have a predicate to express that two template statements are commutative, which would include reasoning about commutativity of arithmetic operations. There is no support for points-to predicates as well (nor for other predicates to specify aliasing information), for example.

The language of read and write sets is not precise enough for the specification of certain predicates computed by compilers' memory dependence analyses, such as distance vectors between memory accesses. For example, when performing automatic vectorization, it is often necessary to widen memory/array accesses. This operation is possible even if a following instruction writes to the same array in the same loop iteration, provided that the distance between the accesses is at least equal to the amount of widening (so that the accesses never overlap).

For example, widening both array operations in the following program (in C) such that each accesses four elements at a time is correct. However, widening those accesses to more than four elements would not be correct, as then the accessed memory ranges would overlap. Currently, we have no way to express a precondition that would state that memory accesses of the target program's statements would not overlap, since our preconditions can only constrain the original template symbols but not the target ones.

```
for (int i = 0; i < n; ++i) {
  ... = a[i];
  a[i+4] = ...;
}
```

Another example of an unsupported set of optimizations are low-level peephole optimizations that require reasoning about bitwise operations. For example, currently we cannot produce preconditions for most of LLVM's InstCombine pass.

Regarding the suitability of the language, there is no hard evidence whether it is a good solution or not. A major evaluation still needs to be done to compare implementations of analyses in current compilers with analyzers generated from the preconditions we synthesized to determine if our preconditions trigger as or more often than state-of-art compiler analyses do. Such an evaluation also needs to compare efficiency, to determine whether generated verification conditions can in fact be efficiently discharged.

Another interesting study would be to determine whether the generated preconditions that are weaker than the ones currently deployed in compilers can enable more transformations in practice.

7.2 Optimization Architecture

The proposed optimization architecture of Figure 2.3 is not entirely new, and several other researchers have proposed similar parts of it in the past. However, I believe it is the first attempt to organize a coherent proposal that gathers all the relevant parts from previous proposals.

I believe the proposed architecture for optimizers is a significant advance regarding how compilers are developed today and how they could be developed in the future.

First, this architecture promotes separation of concerns. Therefore, developers and researchers can more easily focus on their area of interest and expertise, let it be pattern matching, profitability heuristics, or VC Gen, for example.

Second, this architecture can provide significant productivity gains for compiler developers. The cost of developing new optimizations should be reduced considerably. Moreover, the effort of performing certain labor-intensive tasks, such as propagating debug information when doing code transformations can be amortized across optimizations, since these only need to be implemented once in the architecture, and then reused by all optimizations.

Third, this architecture enables the generation of implementations of compiler optimizations automatically, and more importantly, that are correct by construction. This in turn allows compiler developers to explore more complex and potentially more effective optimizations than what they would otherwise consider due to fear of miscompilation or due to the (high) cost of development.

Fourth, this architecture opens up very exciting research opportunities and allows the development of new features for compilers. For example, this architecture potentially enables more fine grained control over optimizations, precision of analyses, compilation time, etc.

We did not implement any part of the proposed architecture in a production compiler. That is a major challenge, but it is of course something we would like to accomplish in the future.

The many technical challenges include, for example, how to do efficient pattern matching of transformation functions in a compiler's IR, how to efficiently discharge parts of a precondition statically (possibly during the matching phase), how to cache the results of analyses across optimizations and how to preserve them after code transformations, how to synthesize pattern matching and VC Gen code automatically from transformation functions and their respective preconditions, and so on. There is no clear solution for any of the challenges just described.

The specification language we proposed to describe transformation functions is very high-level, while compiler's IRs are usually more low-level. For example, LLVM's IR has no loops nor any structured control flow; it is all represented using "gotos". It remains a challenge how to close this gap, first theoretically by formalizing the semantics of both languages, and secondly by designing algorithms to pattern match and transform unstructured control flow guided by specifications using structured control flow.

One of the phases of the proposed architecture is checking whether profitability heuristics hold. In this work, this was largely ignored, since this work is focused solely on correctness of optimizations. However, profitability heuristics are of extreme importance in practice. Therefore, to deploy such an architecture in a production compiler, some solution needs to be found to specify, check, and maybe even validate profitability heuristics (statically and/or dynamically).

7.3 Program Equivalence

The algorithm that we proposed in this document for checking program equivalence can still be improved and extended to a larger set of inputs and to more applications.

In the implementation of the algorithm, we use Wolfram Mathematica's recurrence solver. This solver is not able to compute closed-form solutions for discontinuous functions. Further research needs to be done here. Supporting functions with a bounded number of discontinuities seems straightforward (akin to phase-changing loops), but it is not clear how to handle functions with an unbounded number of discontinuities (arising from, e.g., a loop body with branching on the parity of a loop counter). New algorithms to compute closed-form solutions of recurrences will enable our algorithm to handle loops with more complex branching inside.

The algorithm for program equivalence that we propose is not limited to the domain of compiler optimizations and has potentially many applications. We would like to apply it in other domains, such as information flow, to assess whether this new algorithm improves the state-of-the-art for those domains, what challenges still prevail, and how they differ from those of compiler optimizations.

In this work, we have ignored termination and therefore all arguments present are for partial correctness, partial equivalence and weakest liberal preconditions. It is not clear whether for the domain of compiler optimizations it is necessary to introduce reasoning about termination, and mutual termination in particular. Further research is required on this front.

Chapter 8

Conclusion

In this document, I presented a new algorithm and tool for the automatic synthesis of weakest preconditions for compiler optimizations specified in a declarative and high-level language. Preconditions are expressed in the language of read and write sets over template statements and expressions.

One of the automatically synthesized preconditions is weaker than what has been known previously, which confirms the need for these kind of algorithms. I also presented a fairly complete set of specifications of classic compiler optimizations and their respective weakest preconditions. To the best of my knowledge, this is the first publicly available compilation of such specifications.

Secondly, I proposed a new algorithm for the automatic verification of equivalence of programs in the combined theory of integer arithmetic and uninterpreted function symbols. Then, it was shown how to apply this algorithm for the verification of correctness of compiler applications.

Finally, both of these contributions were articulated as a means towards a modern architecture for the implementation of compiler optimizations. This architecture will enable compiler developers to deploy optimizations in a more efficient way, provide better user experience through new features and increased reliability, and deliver implementations of optimizations that are correct by construction.

Bibliography

[AB03] Christophe Alias and Denis Barthou. On the recognition of algorithm templates. In *Proc. of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification*, 2003.

[ABM07] David Aspinall, Lennart Beringer, and Alberto Momigliano. Optimisation validation. *Electron. Notes Theor. Comput. Sci.*, 176:37–59, July 2007.

[Ack54] W. Ackermann. *Solvable cases of the decision problem.* North-Holland Publishing Company, 1954.

[ALGC12] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *Proc. of the 24th International Conference on Computer Aided Verification*, 2012.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley, 2006.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Aßm96] Uwe Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In *Proc. of the 6th International Conference on Compiler Construction*, 1996.

[BA06] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer-Verlag, 2004.

[BCDVF06] Martin Brain, Tom Crick, Marina De Vos, and John Fitch. TOAST: Applying answer set programming to superoptimisation. In *Proc of the 22nd International Conference on Logic Programming*, 2006.

[BCI11] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAyer: Memory safety for systems-level code. In *Proc. of the 23rd International Conference on Computer Aided Verification*, 2011.

[BCK11] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *Proc. of the 17th International Conference on Formal Methods*, 2011.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *Proc. of the 14th International Symposium on Formal Methods*, 2006.

[BDP14] Gilles Barthe, Delphine Demange, and David Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Trans. Program. Lang. Syst.*, 36(1):4:1–4:35, March 2014.

[BDR04] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proc. of the 17th IEEE Workshop on Computer Security Foundations*, 2004.

[Ben04] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[BFR02] Denis Barthou, Paul Feautrier, and Xavier Redon. On the equivalence of two systems of affine recurrence equations. In *Proc. of the 8th International Euro-Par Conference on Parallel Processing*, 2002.

[BGG05] Jan Olaf Blech, Lars Gesellensetter, and Sabine Glesner. Formal verification of dead code elimination in Isabelle/HOL. In *Proc. of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, 2005.

[BH09] Nick Benton and Chung-Kil Hur. Biorthogonality, step-indexing and compiler correctness. In *Proc. of the 14th ACM SIGPLAN International Conference on Functional Programming*, 2009.

[BHHK10] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. ABC: algebraic bound computation for loops. In *Proc. of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, 2010.

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In *Proc. of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2007.

[BIK10] Marius Bozga, Radu Iosif, and Filip Konečný. Fast acceleration of ultimately periodic relations. In *Proc. of the 22nd International Conference on Computer Aided Verification*, 2010.

[BIK12] Marius Bozga, Radu Iosif, and Filip Konečný. Deciding conditional termination. In *Proc. of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2012.

[BK11] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. of the 23rd International Conference on Computer Aided Verification*, 2011.

[BMS10] Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. Verifying local transformations on relaxed memory models. In *Proc. of the 19th International Conference on Compiler Construction*, 2010.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

[BS97] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and Software Technology*, 39(9):617–625, 1997.

[CC97] Paul-Jean Cahen and Jean-Luc Chabert. *Integer-Valued Polynomials*, volume 48 of *Mathematical Surveys and Monographs*. American Mathematical Society, 1997.

[CCF00] Paul-Jean Cahen, Jean-Luc Chabert, and Sophie Frisch. Interpolation domains. *Journal of Algebra*, 225(2):794–803, 2000.

[CCFL13] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *Proc. of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2013.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[CDOY07] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Footprint analysis: A shape analysis that discovers preconditions. In *Proc. of the 14th International Symposium on Static Analysis*, 2007.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proc. of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[CFLZ08] Nicolas Caniart, Emmanuel Fleury, Jérôme Leroux, and Marc Zeitoun. Accelerating interpolation-based model-checking. In *Proc. of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. of the 12th International Conference on Computer Aided Verification*, 2000.

[CGLA+08] Byron Cook, Sumit Gulwani, Tal Lev-Ami, Andrey Rybalchenko, and Mooly Sagiv. Proving conditional termination. In *Proc. of the 20th International Conference on Computer Aided Verification*, 2008.

[CGS12] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. Regression verification for multi-threaded programs. In *Proc. of the 13rd International Conference on Verification, Model Checking, and Abstract Interpretation*, 2012.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1978.

[Chl10] Adam Chlipala. A verified compiler for an impure functional language. In *Proc. of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[CJJK12] David Cachera, Thomas Jensen, Arnaud Jobin, and Florent Kirchner. Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. In *Proc. of the 19th International Conference on Static Analysis*, 2012.

[CMP+12] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *Proc. of the 4th NASA Formal Methods Symposium*, 2012.

[CO06] John Cavazos and Michael F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proc. of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, 2006.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proc. of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[CVE06] CVE-2006-1902: fold_binary in GCC 4.1 improperly handles pointer overflow. Common Vulnerabilities and Exposures, 2006. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-1902`.

[Dav03] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28, November 2003.

[DF84] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. In *Proc. of the 1984 SIGPLAN Symposium on Compiler Construction*, 1984.

[DHPR97] Axel Dold, Friedrich W. von Henke, Holger Pfeifer, and Harald Rueß. Formal verification of transformations for peephole optimization. In *Proc. of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, 1997.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[DLR14] Stefano Dissegna, Francesco Logozzo, and Francesco Ranzato. Tracing compilation by abstract interpretation. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proc. of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52:365–473, May 2005.

[DXZ13] Liyun Dai, Bican Xia, and Naijun Zhan. Generating non-linear interpolants by semidefinite programming. In *Proc. of the 25th International Conference on Computer Aided Verification*, 2013.

[ER08] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proc. of the 8th ACM International Conference on Embedded Software*, 2008.

[FKM+11] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. Milepost GCC: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327, 2011.

[Fri99] Sophie Frisch. Interpolation by integer-valued polynomials. *Journal of Algebra*, 211(2):562–577, 1999.

[GGS08] Lars Gesellensetter, Sabine Glesner, and Elke Salecker. Formal verification with Isabelle/HOL in practice: finding a bug in the GCC scheduler. In *Proc. of the 12th International Conference on Formal Methods for Industrial Critical Systems*, 2008.

[GJTV11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[GK92] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *Proc. of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, 1992.

[GLPR12] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Proc. of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[GMC09] Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Proc. of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[GP11] Shu-yu Guo and Jens Palsberg. The essence of compiling with traces. In *Proc. of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[GPR11] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *Proc. of the 9th Asian Conference on Programming Languages and Systems*, 2011.

[GS00] Mariano Gasca and Thomas Sauer. On the history of multivariate polynomial interpolation. *J. Comput. Appl. Math.*, 122(1-2):23–35, October 2000.

[GS08] Benny Godlin and Ofer Strichman. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica*, 45(6):403–439, 2008.

[GS09] Benny Godlin and Ofer Strichman. Regression verification. In *Proc. of the 46th Annual Design Automation Conference*, 2009.

[GS13] Laure Gonnord and Peter Schrammel. Abstract acceleration in linear relation analysis. *Science of Computer Programming*, 2013.

[GSV09] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Constraint-based invariant inference over predicate abstraction. In *Proc. of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2009.

[GT06] Sumit Gulwani and Ashish Tiwari. Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In *Proc. of the 15th European Conference on Programming Languages and Systems*, 2006.

[GZB05] Benjamin Goldberg, Lenore Zuck, and Clark Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. *Electron. Notes Theor. Comput. Sci.*, 132:53–71, May 2005.

[HCS06] Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In *Proc. of the 13th International Conference on Static Analysis*, 2006.

[HD11] Chung-Kil Hur and Derek Dreyer. A kripke logical relation between ML and assembly. In *Proc. of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011.

[HIK+12] Hossein Hojjat, Radu Iosif, Filip Konečný, Viktor Kuncak, and Philipp Rümmer. Accelerating interpolants. In *Proc. of the 10th International Conference on Automated Technology for Verification and Analysis*, 2012.

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[HKLR13] Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, and Henrique Rebêlo. Towards modularly comparing programs using automated theorem provers. In *Proc. of the 24th International Conference on Automated Deduction*, 2013.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, October 2009.

[JNZ06] Rajeev Joshi, Greg Nelson, and Yunhong Zhou. Denali: A practical algorithm for generating optimal code. *ACM Trans. Program. Lang. Syst.*, 28(6):967–989, November 2006.

[Jun04] Ulrich Junker. QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *Proc. of the 19th National Conference on Artifical Intelligence*, AAAI, 2004.

[KA02] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann, 2002.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28:619–695, July 2006.

[KP00] Dexter Kozen and Maria-Christina Patron. Certification of compiler optimizations using kleene algebra with tests. In *Proc. of the 1st International Conference on Computational Logic*, 2000.

[KSK09] Aditya Kanade, Amitabha Sanyal, and Uday P. Khedker. Validation of GCC optimizers through trace generation. *Softw. Pract. Exper.*, 39:611–639, April 2009.

[KTL09] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, 2004.

[LAK+14] Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. Symbolic optimization with SMT solvers. In *Proc. of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[LAS00] Tal Lev-Ami and Shmuel Sagiv. TVLA: A system for implementing static analyses. In *Proc. of the 7th International Symposium on Static Analysis*, 2000.

[LAS14] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[Lei05] K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, March 2005.

[Ler09a] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[Ler09b] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43:363–446, December 2009.

[LFF12] Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[LGC02] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[LHKR12] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Proc. of the 24th International Conference on Computer Aided Verification*, 2012.

[Lin05] Christian Lindig. Random testing of C calling conventions. In *Proc. of the 6th International Symposium on Automated Analysis-Driven Debugging*, 2005.

[LJVWF04] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. Compiler optimization correctness by temporal logic. *Higher Order Symbol. Comput.*, 17:173–206, September 2004.

[LM13] Nuno P. Lopes and José Monteiro. Automatic equivalence checking of UF+IA programs. In *Proc. of the 20th International SPIN Symposium on Model Checking of Software*, 2013.

[LM14] Nuno P. Lopes and José Monteiro. Weakest precondition synthesis for compiler optimizations. In *Proc. of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2014.

[LMC03] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, 2003.

[LMRC05] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[LP02] Raya Leviathan and Amir Pnueli. Validating software pipelining optimizations. In *Proc. of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: code generation and implementation correctness. In *Proc. of the 3rd IEEE International Conference on Software Engineering and Formal Methods*, 2005.

[LR08] Junghee Lim and Thomas Reps. A system for generating static analyzers for machine instructions. In *Proc. of the 17th International Conference on Compiler Construction*, 2008.

[Mas87] Henry Massalin. Superoptimizer: a look at the smallest program. In *Proc. of the 2nd International Conference on Architectual Support for Programming Languages and Operating Systems*, 1987.

[MBQ02] Antoine Monsifrot, François Bodin, and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proc. of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, 2002.

[McM11] Kenneth L. McMillan. Interpolants from Z3 proofs. In *Proc. of the International Conference on Formal Methods in Computer-Aided Design*, 2011.

[MG10] William Mansky and Elsa Gunter. A framework for formal verification of compiler optimizations. In *Proc. of the 1st International Conference on Interactive Theorem Proving*, 2010.

[MOS04] Markus Müller-Olm and Helmut Seidl. Computing polynomial program invariants. *Inf. Process. Lett.*, 91:233–244, September 2004.

[Moy08] Yannick Moy. Sufficient preconditions for modular assertion checking. In *Proc. of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2008.

[MP67] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In *Proc. of a Symposium in Applied Mathematics*, 1967.

[MPZN13] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[MR13] Kenneth L. McMillan and Andrey Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research, January 2013.

[MSF06] Takeshi Matsumoto, Hiroshi Saito, and Masahiro Fujita. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In *Proc. of the 7th International Symposium on Quality Electronic Design*, 2006.

[MSJB13] João Marques-Silva, Mikoláš Janota, and Anton Belov. Minimal sets over monotone predicates in boolean formulae. In *Proc. of the 25th International Conference on Computer Aided Verification*, 2013.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *Proc. of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, 2002.

[NZ13] Kedar S. Namjoshi and Lenore D. Zuck. Witnessing program transformations. In *Proc. of the 20th International Conference on Static Analysis*, 2013.

[Olv06] Peter J. Olver. On multivariate interpolation. *Studies in Applied Mathematics*, 116(2):201–240, 2006.

[ORR+96] Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and Mandayam K. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proc. of the 8th International Conference on Computer Aided Verification*, 1996.

[PDEP08] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

[PHG05] Arnd Poetzsch-Heffter and Marek Gawkowski. Towards proof generating compilers. *Electron. Notes Theor. Comput. Sci.*, 132:37–51, May 2005.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proc. of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 1998.

[RCK07] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42:443–476, April 2007.

[RDPJ10] Norman Ramsey, João Dias, and Simon Peyton Jones. Hoopl: A modular, reusable library for dataflow analysis and transformation. In *Proc. of the 3rd ACM Haskell Symposium on Haskell*, 2010.

[RE11] David A Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In *Proc. of the 23rd International Conference on Computer Aided Verification*, 2011.

[RL10] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In *Proc. of the 19th International Conference on Compiler Construction*, 2010.

[RM99] Martin C. Rinard and Darko Marinov. Credible compilation with pointers. In *Proc. of the FLoC Workshop on Run-Time Result Verification*, 1999.

[Rob01] Arch D. Robison. Impact of economics on compiler optimization. In *Proc. of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, 2001.

[RSS07] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. In *Proc. of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2007.

[RW98] Teodor Rus and Eric Van Wyk. Using model checking in a parallelizing compiler. *Parallel Processing Letters*, 8(4):459–471, 1998.

[Sam75] Hanan Samet. *Automatically Proving the Correctness of Translations Involving Optimized Code*. PhD thesis, Stanford University, May 1975.

[San98] David Sands. Improvement theory and its applications. In *Higher Order Operational Techniques in Semantics*, pages 275–306. Cambridge University Press, 1998.

[San09] Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, May 2009.

[SBCJ05] K. C. Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. Verification of source code transformations by program equivalence checking. In *Proc. of the 14th International Conference on Compiler Construction*, 2005.

[SCWK13] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. Automatic construction of inlining heuristics using machine learning. In *Proc. of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, 2013.

[SDDA11] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. Simplifying loop invariant generation using splitter predicates. In *Proc. of the 23rd International Conference on Computer Aided Verification*, 2011.

[SdML04] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[SK13] Mohamed Nassim Seghir and Daniel Kroening. Counterexample-guided precondition inference. In *Proc. of the 22nd European Conference on Programming Languages and Systems*, 2013.

[SLC07] Erika Rice Scherpelz, Sorin Lerner, and Craig Chambers. Automatic inference of optimizer flow functions from semantic meanings. In *Proc. of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[SS98] David Schmidt and Bernhard Steffen. Program analysis as model checking of abstract interpretations. In *Proc. of the 5th International Symposium on Static Analysis*, 1998.

[SSA13] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proc. of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[SSCA13] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. Data-driven equivalence checking. In *Proc. of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013.

[SSM04] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *Proc. of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[Ste91] Bernhard Steffen. Data flow analysis as model checking. In *Proc. of the International Conference on Theoretical Aspects of Computer Software*, 1991.

[STL11] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for LLVM. In *Proc. of the 23rd International Conference on Computer Aided Verification*, 2011.

[Str80] Gilbert Strang. *Linear Algebra and Its Applications (2nd Ed.)*. Academic Press, 1980.

[SU08] Ando Saabas and Tarmo Uustalu. Program and proof optimizations with type systems. *J. Logic Algebr. Program.*, 77(1–2):131–154, 2008.

[TA05] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *Proc. of the 12th International Conference on Static Analysis*, 2005.

[TGM11] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[TH92] Steven W. K. Tjiang and John L. Hennessy. Sharlit—a tool for building optimizers. In *Proc. of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, 1992.

[TL08] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[TL09] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[TL10a] Zachary Tatlock and Sorin Lerner. Bringing extensibility to verified compilers. In *Proc. of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[TL10b] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *Proc. of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[TSL10] Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. In *Proc. of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proc. of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[vDD08] Daniel von Dincklage and Amer Diwan. Explaining failures of program analyses. In *Proc. of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[vDD09] Daniel von Dincklage and Amer Diwan. Optimizing programs with intended semantics. In *Proc. of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009.

[VJB09] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *Proc. of the 21st International Conference on Computer Aided Verification*, 2009.

[VWK06] Eric Van Wyk and Lijesh Krishnan. Using verified data-flow analysis-based optimizations in attribute grammars. In *Proc. of the 5th International Workshop on Compiler Optimization meets Compiler Verification*, 2006.

[Wei03] Zachary Weinberg. A maintenance programmer's view of GCC. In *Proc. of the GCC Developer's Summit*, 2003.

[WHTY13] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. Effective dynamic detection of alias analysis errors. In *Proc. of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[WS97] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, November 1997.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[ZCS06] Min Zhao, Bruce R. Childers, and Mary Lou Soffa. An approach toward profit-driven optimization. *ACM Trans. Archit. Code Optim.*, 3(3):231–262, September 2006.

[ZME06] Jia Zeng, Chuck Mitchell, and Stephen A. Edwards. A domain-specific language for generating dataflow analyzers. In *Proc. of the 6th Workshop on Language Descriptions, Tools, and Applications*, 2006.

[ZNMZ12] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[ZNMZ13] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *Proc. of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[ZP08] Anna Zaks and Amir Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *Proc. of the 15th International Symposium on Formal Methods*, 2008.

[ZPG+05] Lenore Zuck, Amir Pnueli, Benjamin Goldberg, Clark Barrett, Yi Fang, and Ying Hu. Translation and run-time validation of loop transformations. *Form. Methods Syst. Des.*, 27:335–360, November 2005.

[ZXT+09] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated test program generation for an industrial optimizing compiler. In *Proc. of the ICSE Workshop on Automation of Software Test*, 2009.