RANGE ANALYSIS OF BINARIES WITH DECISION PROCEDURES

a thesis submitted to
The University of Kent
in the subject of computer science
for the degree
of doctor of philosophy.

By
Edward Barrett
September 2014

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

In the past few years, there has been increased interest in automating the reverse engineering and verification of binary executable code. The importance of this subject has been highlighted by the growing relevance of security, of reliability and of legacy code. Since dynamic analysis is of limited use for whole-program analyses, there has been a renewed enthusiasm for the development of automated static analyses, which can prove a property holds over all paths of the program. The abstract interpretation framework serves this purpose and has been widely adopted in both academic and industrial circles. Yet, since its introduction in 1977, standard abstract interpretation has been formulated as the least solution of a set of fixpoint equations.

The work in this thesis deviates from the standard approach to static analysis, proposing that recent advances in decision procedures could be leveraged to tackle the problem. The thesis can be considered to be a survey of the application of Boolean satisfiability (SAT) and linear optimisation to the problem of static analysis, specifically range analysis of binary executable code. It is shown (with experimental results) that SAT and linear optimisation can be used to infer ranges of register values which, amongst others, are useful for control flow recovery and for detecting binary vulnerabilities, such as buffer and heap overflows.

Acknowledgements

Copyright

- The work presented in Chapter 3 is derived from work by Edd Barrett and Andy King, published by Elsevier [9].
- The work presented in Chapter 5 is derived from work by Edd Barrett and Andy King, published by Springer [10].

Personal Acknowledgements

First and foremost, I would like to thank my supervisor Andy King, for giving me the opportunity to study for my Ph.D. I have learned a lot from Andy and on numerous occasions he has risen above and beyond the expectations of a supervisor.

A big thanks goes to my wonderful family, who have always believed in me and supported me through thick and thin. A special thanks goes to my Grandma and Grandpa, who helped me through the final months of my studies when my monetary situation was difficult. For this I am most grateful. I regret that my Father, Glen, will not see my achievement, as sadly he passed away during the course of my Ph.D. A further thanks goes to everyone that helped me during this difficult time.

Thanks to all of my friends in Canterbury and back home in Basingstoke, who have offered endless amounts of encouragement, food and beer. A special thanks goes out to Debora Claros for being nothing less than *awesome*. During the long months of thesis writing, Debora fed me, proof read my work and offered a great deal of moral support. I can't emphasise enough how much I appreciate this.

I would like to thank my fellow postgraduate students who are currently residing in, or have resided in SW104 (a.k.a. the zoo). In particular I thank Thomas Schilling, Martin Ellis, Edward Robbins and Jael Kriener for the insightful discussions we had and for their willingness to freely share their knowledge. I also thank Edward Robbins for letting me stay in his spare room for a few months.

Thanks to the anonymous reviewers of my papers, and to those who proof read this thesis. A big thanks to Fred Barnes and Sebastian Hunt for examining this thesis. Thanks to Jörg Brauer, who offered some valuable feedback on a research idea that later become the published work presented in Chapter 5. Thanks to Laurence Tratt for allowing me to take time off work to apply corrections to this thesis.

Finally, I would like to take the opportunity to thank the hundreds of software authors whose code I used under an open-source license. I strongly believe that software should be free and open, especially for educational purposes, so thank you all.

Typesetting Conventions

The following typesetting conventions are used in this thesis:

- Vectors are set in bold, e.g. $\boldsymbol{v} = \langle 1, 2, 3 \rangle$.
- Source code and program variables are set in a monospace typewriter font. For example, `eax` refers to a CPU register.
- Assembler instructions and sequences of assembler instructions in the body of the main text are set in vector brackets using semi-colons to separate instructions. For example, ⟨`mov eax, 666; shl eax, 2`⟩.
- Assmbler code is shown in Intel syntax (as opposed to AT&T syntax).
- When discussing program semantics, abstract domain operations are squared, whereas concrete domain operations are not. For example $\sqsubseteq$ vs. $\subseteq$.

Chapter 1

Introduction

Because programming a computer in its native binary language is highly impractical, tooling has been developed which eases the software development process. Instead of working directly with native binary code, typically programmers input program code in a high-level dialect (a programming language) which is then translated into the native language of the computer by a compiler. The output of a high-level language compiler is a collection of object files, which are then linked to give a binary file, ready for execution at the user's convenience[1]. There are numerous advantages to programming a computer in this way. Development time is greatly reduced because the low-level intricacies of the underlying architecture need not be relevant to the programmer. There is also a huge number of programming languages available from which to choose. Each language targets a different kind of problem and in its own unique way. Instead of being restricted to a single programming paradigm, as with coding at the binary-level, the programmer is free to choose the right tool for the job on a per-project basis. Further, high-level source code can often be compiled for different platforms and architectures with minimal alterations. Native binary code, however, is restricted to a single architecture, meaning that a binary code implementation would need to be written for each individual architecture that the program is required to execute upon. For these reasons (and others), programming using a high-level language is by now, standard software engineering practice.

Compiler construction and programming language design are topics which have

[1]Scripting languages and virtual machines are not considered in this thesis.

been studied in great detail; at least to the point where definitive texts are available from mainstream publishers [3, 4]. A lesser explored topic however, is the process of reversing compilation. Sometimes there is the need to extract meaning from binary code. Traditionally this was a task reserved only for so-called black-hat crackers, who inspect and modify commercial software packages at the binary level. Typically the aim is to circumvent copy protection mechanisms to illegally re-distribute the software. Since commercial software houses do not publish the source code of their products, crackers are forced to analyse and modify the programs at the binary level. Recently however, many legitimate applications for the (automated) analysis of binary code have surfaced.

1.1 Applications for Binary Analysis

The applications for binary analysis fall under one of two broad classifications: reverse engineering and software verification. In the context of software, reverse engineering is the art of developing an understanding of a program from the compiled code alone. Usually the need to reverse engineer binary code stems from the fact that there is no access to the high-level source code from which the binary is built. There are a number of reasons for binary reversing, for example:

Malware detection and classification. Malware is the umbrella term used to describe software with malicious intent or adverse behaviour. Malware is developed illegally and then transmitted, traditionally via removable media, but more recently via the Internet. The function of malware can range from annoying messages to catastrophic data loss or even damage to hardware. It is the role of anti-virus firms to distribute software which can identify and remove unwanted malware from infected systems. Typically anti-virus software will contain a database of virus signatures which serves as a basis for malware identification. The information in the database is collected by security engineers whose job it is to reverse engineer suspect binaries. Malware writers take advantage of the fact that binary code is difficult to reverse engineer and do not disclose the source code of their malware. In more recent years, malware authors have gone as far as to apply obfuscation, anti-debugging and packing techniques to make reversing harder still. The

war on malware has become an arms race, with malware writers deploying successively more intricate attacks, and with anti-virus companies allocating more time and money into ways of counteracting the latest malware developments.

Loss of code. It is not uncommon for companies to develop, or outsource the development of, bespoke software for internal use. If, after deployment, the source code of such a project is destroyed, or if an outsourced company ceases to trade, then it is not easy to modify the software since the source code is now unavailable. It may be possible to consult to the original specifications of the software to develop a new implementation, but in the absence of such artefacts, companies must resort to reverse engineering. By analysing the binary code it may be possible to develop a sufficient understanding of the inner workings of the program to implement a new source code reimplementation. Alternatively, if the required modifications are small, the binary may be modified in-place.

Penetration testing. The process of testing the resilience of a software product to malicious parties is referred to as penetration testing (or just pen testing). Typically, pen testing is outsourced to external companies whose job it is to test the product against (amongst others) denial of service, unauthorised access, privilege escalation and data breach. Since pen testing is conducted without prior knowledge of the system (without access to source code or documentation, so as to emulate a real attacker), pen testing may benefit from reverse engineering. For example, reverse engineering may be applied to gain an insight into security mechanisms. A thorough study may unveil exploitable vulnerabilities in the system. Issues of this nature are reported back to the software developers so that corrective measures (patching) may be performed.

Interfacing and interoperating. Sometimes the need arises to interoperate with existing software or hardware for which there is no readily available documentation or high-level code specifying the interface. In such cases it may be possible to discover the workings of the interface via reverse engineering. A rather famous case of this occurred in 1992 when Accolade Inc. were found to have reverse engineered several software titles from Sega Enterprises. This

effort was expended for the sole purpose of developing unlicensed software for use on Sega hardware [101].

The list is not exhaustive, but it covers many of the applications of reverse engineering in industry today. To reiterate, the above applications of binary analysis exist because access to high-level source code is not always possible. This is by contrast to software verification, where the source code is usually available, but analysis may occur at the binary level for different reasons.

Software verification is a quality assurance measure usually integrated into the software development life-cycle and typically amounts to proving that a software product conforms to a set of safety, correctness and reliability criteria. By satisfying these criteria, the developers can be confident in the robustness and safety of the system as a whole. Several aspects of software can be verified in this way, including:

Memory Accesses There is often the need to decide whether a piece of software is correct with regards to the handling of memory buffers. For example, writing past the end of an array is almost always incorrect. In languages like C, where array writes are not checked, writing past the end of an array usually invokes so-called undefined behaviour. When undefined behaviour is invoked, the program continues to execute, but in an unpredictable and most likely unintentional manner, usually ending with a crash. Worse, an attacker who has studied the memory errors of a program that is deficient in this way, may be able to write into other areas of memory, thus altering the behaviour of the program for malicious purposes. We are of course referring to the classic buffer overflow exploit, which in some cases can allow unauthorised access or privilege escalation, etc. By deploying verification, engineers can check that no out of bounds memory accesses may occur. By doing so the engineer gains confidence that the program is free from the reliability and security concerns associated with memory errors.

Error state reachability Sometimes programmers identify error states in their programs which should not be reachable. In such situations it is common to use an assertion to mark the error state and terminate the program should it be reached. Adding assertions to programs is good software development

practice and can help to find bugs early in the software development life-cycle. But how can a programmer be certain that an error state is never reachable? Software verification can be deployed in order to prove that the error state is not reached in any possible execution path. If indeed this is the case, then the programmer gains confidence in the correctness of the program. If the reachability of an error state cannot be disproved however, then a bug may exist and further investigation is warranted.

Memory consumption Embedded systems are purpose built for a single task, and thus to minimise costs, are designed with the bare minimum of hardware required. Software targeting such architectures is expected to execute within tight memory constraints, for example within a very small stack size. Since in embedded architectures there is often no support for protected memory pages, a stack overflow may corrupt other aspects of the system leading to undefined behaviour, including crashes [24]. In order to avoid these cases, it is possible to apply software verification to determine if the software in question has a bounded stack size and that the maximum possible stack consumption fits within the reserved memory allowance. Software satisfying these criteria can be guaranteed not to crash as a consequence of stack overflows.

Worst case execution time Some software systems include strict timing requirements which, if violated, could yield severe consequences. In the automotive industry, in-car embedded computers must respond quickly, as unexpected latency may endanger human lives [81]. Similarly, in the aerospace industry, software product assurance standards force on-board spacecraft software to be checked for unacceptable latency [48]. This kind of analysis is referred to as worst case execution time analysis (WCET). The aim of WCET is to compute the maximum possible time a given software task can take to complete, which is then checked to be within predefined allowances. A system which does not meet these requirements is deemed not fit for purpose and must be revised.

Notice that in many of the above verification scenarios, the source code is most likely available, yet in many cases it is beneficial to apply verification at the binary level. This stems from the fact that verification at the source-level is

somewhat inopportune. After all it is not the source code that is executed, but rather the binary counterpart created by the compiler and linker. Balakrishnan et al. [7] show that the compilation step itself can transform program behaviour such that the semantics of the end binary can be quite different from that of the source code. Assumptions made by optimising compilers can even introduce vulnerabilities; the code shown in Listing 1.1 shows one such case:

```
// password no longer used
memset(password, '\0', len);  // line optimised away
free(password);
```

Listing 1.1: A compiler-induced vulnerability [7].

The code is supposed to overwrite the in-memory password buffer with zeroes before it is freed. The developer's intent is commendable. By minimising the time in which the password is stored in memory, it is much harder for an attacker to capture the password. Yet this sense of security is merely an illusion, since in the eyes of the compiler, to zero the buffer immediately prior to freeing it is a waste of CPU cycles. As a consequence, the call to `memset` is removed by the optimiser. When the resulting binary is executed, the password remains in memory for longer than intended, i.e. until the buffer is reallocated and overwritten elsewhere. Note that most source-level analyses would fail to detect this behaviour. Similarly, deficiencies in the development tool-chain itself are hard to detect at the source-level. Sadly, occurrences of tool-chain bugs are common and the problem is significant. The LLVM (low level virtual machine) project documents broken versions of the GNU tool-chain (compiler, linker and binutils) that generate incorrect code[2].

In summary, there are many real-world applications for binary analysis in today's computing industry. But, given the motivation to analyse binary code, how does one begin to tackle the problem? The approaches generally fall under one of two broad categories: dynamic analysis and static analysis.

1.2 Static or Dynamic Analysis?

When planning a binary analysis, the first decision that must be made is whether to approach the problem statically or dynamically:

[2] `http://llvm.org/docs/GettingStarted.html`

Dynamic Analysis Refers to the class of analyses which involve actually running code within the program's execution environment.

Static Analysis By contrast to dynamic analyses, static analyses do not execute any code concretely. Instead, execution is symbolic and may consider whole classes of execution traces at once.

To help put these two approaches into context, consider the verification of a binary program to check that no memory errors occur. To dynamically determine this, a tool such as ValGrind [108] could be used. Such tooling supervises the execution of the binary, scrutinising each memory access as it is encountered. When execution terminates, a report is emitted detailing any invalid memory reads or writes that were observed. Alternatively, a static analysis tool such as the Clang analyser could be used[3]. Such a tool would attempt to verify the memory safety of the program as a whole and without executing the binary at all. To achieve this, a simplified model of the execution environment (abstraction) is used to gather the possible states that may arise in concrete executions. From this information it may be possible to infer whether the program is safe with regards to memory handling.

The dynamic approach has the advantage that, because analysis is based upon actual execution (concrete traces), little extra tooling is required and any successful deduction comes with a proof of correctness in the form of an execution trace. Unfortunately, dynamic analysis is often infeasible where, as in the example above, there is the need to prove a property holds over all paths through the program. To achieve this goal dynamically means exhaustively executing all possible paths through the program; a task generally considered impossible for realistically-sized programs due to the sheer number of execution paths existing. In his book [89], Myers shows that there are an astonishing number of unique execution traces through even seemly small programs. Consider the control flow graph (CFG) shown in Figure 1. The numbered nodes represent program points, whereas the edges depict transitions between program points. Program points 2 through 11 constitute the body of a loop. There are five paths through a single execution of this loop alone, so if the loop were to execute 20 times, this gives a total of $5^{20} + 5^{19} + \ldots + 5^1 = 119209289550780$ possible concrete traces. To scrutinise all

[3] `http://clang-analyzer.llvm.org`

of these executions would take a lifetime and more. This illustrates the need to merge paths to prove that a property consistently holds. Often a weaker coverage metric, such as function coverage or statement coverage can be used to exercise a subset of the execution cases, but such an approach still fails to prove that a property always holds for all paths.

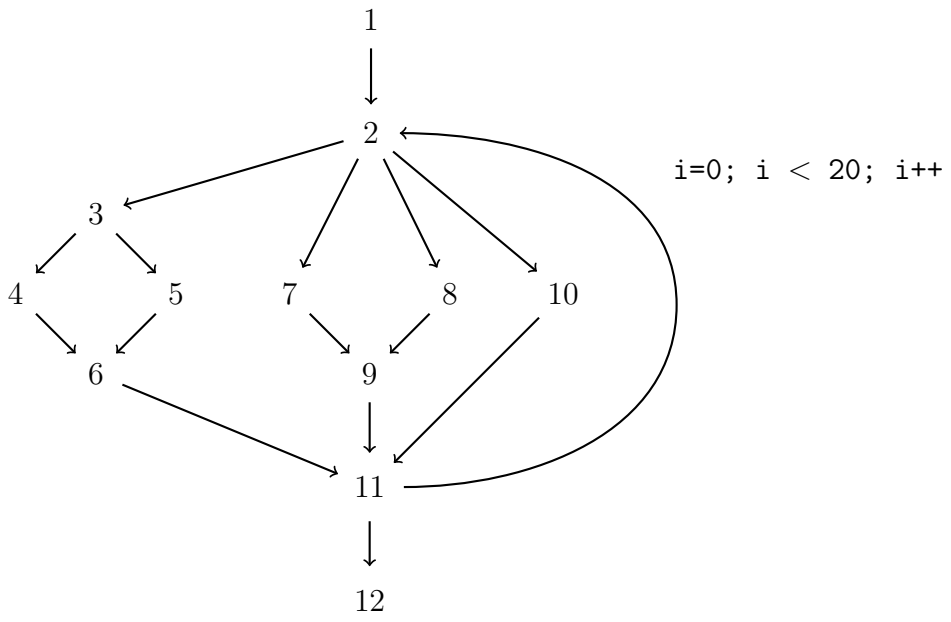


Figure 1: There are a large number of traces through even small CFGs.

It is the path coverage problem that motivates the development of static analyses. Static analyses do not suffer from this problem because, as previously mentioned, no code is actually executed, so many paths may be considered all at once. Static analysis however, is not without its own drawbacks. First and foremost, because no code is executed for real, the CFG of the program must be recovered a priori. This poses a problem in itself and will serve as a starting point for the contributions of this thesis.

1.3 Static CFG Recovery and Indirect Jumps

Recovering the CFG of a binary program requires significant effort. The raw bit-stream of a binary is cumbersome to work with. This is hardly surprising, as raw binary code was not designed to be decoded by anything other than the

CPU itself. To make a binary more comprehensible, typically the instruction stream is translated into assembler mnemonics. This process is called disassembly. For example, suppose the hex byte sequence `0x4883ec08e8f70100004883c408c3` appears at the address `0x400250` in an x86-64 binary. The disassembly for this sequence of bytes is shown in Listing 1.2.

```
400250:        48 83 ec 08              sub    rsp,0x8
400254:        e8 f7 01 00 00           call   0x400450
400259:        48 83 c4 08              add    rsp,0x8
40025d:        c3                       ret
```

Listing 1.2: Disassembly of the binary code using GNU objdump.

From the disassembled code it is easier to see the control flow of the program. The listing above shows a function call and a return from the current function. Unfortunately though, the process of disassembly is notoriously difficult. To appreciate this, the two standard static disassembly approaches should be discussed.

Linear Sweep Disassembly Given a start address and an end address, linear sweep will disassemble instructions in a straight line between the two addresses. When linear sweep encounters a control flow despatch (`jmp`, `call`, etc.), no attempt is made to follow the despatch. Instead, disassembly continues at the address immediately after the despatch.

Recursive Traversal Disassembly A recursive traversal disassembler will disassemble instructions linearly until a control flow despatch is met. Disassembly then resumes (if possible) at the destination address(es) of the despatch. Using this strategy, disassembly becomes an exercise in control flow exploration.

Linear sweep disassembly is the simplest to implement of the two algorithms and is usually sufficient if the code conforms to a strict and well-known compilation model. For example, the GNU compilers generate code which can be disassembled by the GNU `objdump` utility. Unfortunately, if the binary does not conform to the correct compilation model, or if it has been obfuscated, then linear sweep can inadvertently misinterpret the instruction stream [80]. In this case, the recovered CFG will be incorrect and, in turn, any analysis underpinned by this

CFG will be unsound. Recursive traversal disassembly is less likely to fail in this manner since it is driven from the control flow. However, a major hindrance with recursive traversal is that often the target of a control flow despatch is computed using register values; i.e. via an indirect jump. For example, the instruction ⟨`jmp [rax]`⟩ transfers execution to the address held in `rax`. Because recursive traversal does not collect register values, when an indirect jump is encountered, the only safe course of action is to assume that every address is a potential target of the jump. The control flow graph of such a disassembly is inevitably a gross over-approximation of the actual control flow graph, meaning that the precision of an analysis underpinned by such a CFG will be poor. One may wonder why recursive traversal could not be extended to collect register values, as this would mean that the possible jump targets could be known and a more precise CFG could be found. Yet this is not easy either because the register values themselves can only be collected with the aid of a control flow graph. This cyclic problem is often referred to as the "chicken and egg" problem [38].

It would seem that if a correct disassembly, and thus an accurate CFG, is required, register tracking must be inherent in the disassembly and control flow recovery process itself. Indeed this is the approach of Kinder et al. [71], whose analysis grows the control flow graph incrementally based upon facts such as register values. First, facts are propagated to trivially known control flow successors, before a **resolve** operator is called. The **resolve** operator consults the currently known facts to add new control flow edges. Upon the discovery of new edges, facts may then be propagated further. This process continues until a fixpoint is achieved, whereby an accurate CFG has been obtained.

Kinder's analysis works on a toy assembler language, JUMP. For real-world assembler languages such as x86 or SPARC64, there are extra obstacles that must be taken into consideration. Conditional branching in JUMP, for example, has been greatly simplified. A conditional branch in JUMP takes the form ⟨`jmp` e_1, e_2⟩, where e_1 and e_2 are expressions. If e_1 is true, then transfer control flow to e_2. By contrast, x86 conditional branches typically consist of two instructions: one instruction to assign some status flags, and a further instruction to conditionally despatch control flow based upon the assignment of the status flags. This gives rise to the problem of relating these Boolean status flags to control flow, both statically and in an automated fashion.

Ideally, ranges or sets of register values could be statically inferred from binary code. From this information it would be easy to gain an insight as to the possible targets of indirect jumps. Furthermore, ranges and sets could help to decide if a conditional jump is ever despatched (reachability analysis). Yet to infer ranges and sets, low-level bitwise details such as the status flags must be accounted for. In fact, assembler instructions can easily be modelled at the bit-level as Boolean formulae [16]. After all, a computer system is merely a collection of Boolean circuitry. Through the composition of smaller Boolean formulae, larger formulae can be derived to model the basic blocks and functions from which a binary program is built. This would suggest that it may be possible to leverage Boolean decision procedures to help infer ranges and sets of register values. In turn this information could be used to refine a control flow graph by, for example, inferring a range of indirect jump targets. The first body of work in this thesis explores exactly this. The work is separated into two parts (Chapters 3 and 4), beginning with the following contributions presented in Chapter 3:

- A method is shown that automatically abstracts a set of Boolean satisfiability (SAT) models as a range. To achieve this, the method uses repeated calls to a SAT solver. This is referred to as range abstraction.
- Building upon range abstraction, it is shown that a range of models can be iteratively refined into a successively more precise set of models. Eventually the set converges upon a precise set of models. This is referred to as set abstraction. The algorithm need not be run to termination and can be halted in an anytime fashion, so as to terminate upon an over-approximation (or under-approximation, if desired) of the set of models.
- It is shown that the techniques can be structured so as to exploit incremental SAT, thereby eliminating duplication of solver work and in turn improving solving times.
- Experimental results are shown which indicate that the method can infer ranges and sets for Boolean formulae whose models are representative of a set of indirect jump addresses.

To summarise, the above contributions represent the first step towards an automated binary analysis which could automatically infer ranges (and sets) of

register values. In turn this information could be used to, for example, infer control flow edges at indirect branching sites in binary code.

To apply range and set abstraction in this way, one would first encode CPU instructions as Boolean formulae. To illustrate, consider the instruction sequence $\langle$`shl eax, 2; jmp [eax]`$\rangle$. Following the approach of [16], each register at each program point would be encoded as a bit-vector and a Boolean formula would relate the vectors. The $\langle$`shl eax, 2`$\rangle$ operation would be encoded as follows:

$$\neg eax'_0 \wedge \neg eax'_1 \wedge (eax'_2 \Leftrightarrow eax_0) \wedge \ldots \wedge (eax'_{31} \Leftrightarrow eax_{29})$$

where the vector $\boldsymbol{eax} = \langle eax_0, \ldots, eax_{31} \rangle$ represents `eax` prior to the shift, and the vector $\boldsymbol{eax'} = \langle eax'_0, \ldots, eax'_{31} \rangle$ represents the mutated value of `eax` after the shift. It was hoped that, to infer the possible targets of the indirect jump, set abstraction could be applied to infer the possible values that $\boldsymbol{eax'}$ could assume.

Unfortunatley, it was found that the range and set abstraction algorithms could not be applied directly to such a problem due to issues with termination. This unexpected outcome relates to the fact that alone, range and set abstraction are only able to infer ranges and sets for SAT models and not for subvectors over which a model is defined. In the context of the above example, notice that each possible $\boldsymbol{eax'}$ value corresponds to an assignement to a sub-vector of the variables over which an entire SAT model is defined. When set abstraction was applied to infer ranges for this sub-vector, termination did not occur.

The work shown in Chapter 4 aims to remedy this by extending the range and set abstraction methods presented in Chapter 3, allowing them to work over sub-vectors of models:

- An explanation is offered as to why the plain range and set abstraction algorithms are insufficient when applied to sub-vectors of SAT models.
- It is shown that ranges and sets can be inferred for sub-vectors of SAT models using a quantified Boolean formula of the form $\forall I.\ \exists T.\ f$.
- A novel method is presented by which to eliminate the existential and universal quantifiers from the formula, thereby allowing it to be passed to the range and set abstraction algorithms proposed in Chapter 3. The method

works by computing the prime implicates through mixed-integer linear programming, namely through the generation of Chvátal cuts. Once the prime implicates have been found, quantifier elimination is trivial.

- It is shown that, when computing the prime implicates, an objective function and blocking constraints can be deployed to find short implicates so as to avoid duplication of work.
- Experimental results are presented which suggest that the prime implicates are found in much fewer operations than by traditional methods.

1.4 Range Analysis as an Optimisation Problem

So far it has been established that SAT solving and mathematical optimisation can be used to infer the values of individual registers at select program points, thus easing the recovery of the CFG of a binary program. The next body of work assumes that the CFG is known and explores the possibility of conducting a more general binary range analysis with decision procedures. In fact, there is a significant advantage to this approach; fixpoint acceleration, as commonly used in static analyses, is not required. To appreciate the force of this, the abstract interpretation (AI) framework must first be discussed[4].

Abstract interpretation is the de facto framework for static program analysis and was devised in 1977 by Cousot and Cousot [31]. Since then, AI has been widely adopted in both the industrial and academic program analysis communities. The idea behind abstract interpretation is that through a systematic loss of precision, an over-approximation of the states that arise in concrete executions can be computed in a tractable manner. The basic components of an abstract interpretation are: a concrete domain, a collecting semantics, an abstract domain, an abstract semantics and a solving strategy.

The concrete domain serves as a data structure to hold concrete state, i.e. the actual states that can arise at each program point in concrete executions of the program. The concrete domain should possess the ability to describe precisely the program property of interest. This property could be the numeric values of

[4]An overview of AI is offered here, enough to explain the next set of contributions. The underlying formalities are postponed until Chapter 2, where AI is discussed in detail.

the program variables, memory allocations and deallocations, timing constraints, data types, etc. A range analysis is typically concerned with the values of program variables, which at the binary level correspond to register values[5]. A set of n-tuples could be used to represent the possible numeric values of n registers. One such set would be held for each program point.

Accompanying the concrete domain is a collecting semantics, so-called because it collects sets of possible concrete values. This is an inductive system of semantic equations that describe how the concrete state is transformed and propagated across the control flow of the program. Typically one semantic equation is defined for each program statement. The solution to the collecting semantics describes the possible concrete states that can arise at each point in the program. Solving the collecting semantics directly is usually impractical and may not terminate. Instead, AI proposes that the problem is transformed into a simplified, more tractable form. This is achieved by selectively discarding information through the process of abstraction.

The first step to applying abstraction is to select an abstract domain. The role of the abstract domain is to provide a data structure for representing abstract program states. The abstract domain should be able to over-approximate any given concrete state efficiently. Many such domains exist, each targeting a different purpose; to mention a few, the interval [31], the congruence [59], the octagon [86] and the polyhedron [35]. In the context of range analysis, intervals are used to abstract sets of concrete states. An interval describes a set of consecutive numeric values as a lower and upper bound, denoted $[l, u]$. Using an interval, large sets can be described at the cost of storing only the two bounding values, thus storage requirements are very modest. Since a binary range analysis would typically be required to abstract n registers at each program point, a single n-vector of intervals would suffice to represent the abstract state at each program point.

Once the concrete and abstract domains have been selected, the correspondence between them should be identified. The correspondence is characterised by a pair of mappings: α (abstraction) and γ (concretisation). The former maps elements of the concrete domain to elements of the abstract domain. Conversely, the latter maps elements of the abstract domain to elements of the concrete domain. Suppose an analysis uses a set of 2-vectors to track the concrete register values

[5]For now, memory values are not considered.

that can arise at each program point. The interval abstraction of the concrete state $\{\langle 0,6\rangle, \langle 1,6\rangle, \langle 1,8\rangle\}$ is:

$$\alpha(\{\langle 0,6\rangle, \langle 1,6\rangle, \langle 1,8\rangle\}) = \langle [0,1], [6,8]\rangle$$

Conversely, the concretisation of $\langle [0,1], [6,8]\rangle$ is:

$$\gamma(\langle [0,1], [6,8]\rangle) = \{\langle 0,6\rangle, \langle 0,7\rangle, \langle 0,8\rangle, \langle 1,6\rangle, \langle 1,7\rangle, \langle 1,8\rangle\}$$

Notice that $\gamma(\alpha(\{\langle 0,6\rangle, \langle 1,6\rangle, \langle 1,8\rangle\})) \supseteq \{\langle 0,6\rangle, \langle 1,6\rangle, \langle 1,8\rangle\}$. The imprecision incurred through over-approximation is the price paid for computational tractability. An over-approximation is accepted as a sound abstraction, since all possible states are captured.

Next, the abstract semantics is defined. The role of the abstract semantics is to describe the effect of program statements upon the abstracted state. The abstract semantics, like the collecting semantics, is a system of semantic fixpoint equations. Again, for each program statement, one semantic equation is defined. Unlike the collecting semantics, however, the abstract semantic equations operate over elements of the abstract domain. It is this system of equations which is solved. The least solution, although approximate, it is often accurate enough to prove that certain properties hold across all paths of the program. For example, range information can be used to prove that all memory writes are within range and this can help to show that a program is not susceptible to buffer overflow vulnerabilities.

The solving of the abstract semantics itself typically involves repeatedly applying the fixpoint equations of the abstract semantics until a fixpoint is met (Kleene iteration). This solving strategy is effective for code without loops or for code with short-running loops, however, long-running loops may arise. Long running loops induce long ascending chains of abstract states, meaning that to achieve fixpoint convergence, the semantic equations must be applied many times. In turn this can lead to suboptimal solving times. To illustrate, consider the simple loop shown in Listing 1.3.

Suppose an analysis uses intervals drawn from $D : \{[l,u] \mid l,u \in \{0,\ldots,65535\} \wedge l \leq u\}$ as the abstract domain. The analysis would first compute $i = [0,0]$ for the

entry to the loop. Subsequent iterations would compute:

$$\langle [0,1], [0,2], \ldots, [0,9999], [0,10000], [0,10000] \rangle$$

Since the abstract state did not change between the last two iterations, a fixpoint has been reached, but convergence takes 10001 iterative solving steps. Evaluating the abstract equations this many times is likely to be time consuming. Furthermore, under certain circumstances, it is possible that the fixpoint computation never terminates.

```
for (uint16_t i = 0; i < 10000; i++) {
        ...
}
```

Listing 1.3: A long-running loop.

To improve solving times, fixpoint convergence must be achieved in fewer solving iterations. To this end, fixpoint acceleration techniques, such as widening [31, 76, 109], have been proposed. For intervals, the standard widening works as follows. If after a predetermined number of iterations, an abstract state has not converged upon a fixpoint, then unstable bounds are proactively extrapolated so as to skip over a number of intermediate iterations in one leap. Suppose standard widening were to be applied to the previous example if a fixpoint is not reached after three iterations. The analysis would compute $\langle [0,0], [0,1], [0,2], [0,65535] \rangle$. This is both safe and ensures fast termination, but has impact upon the precision of the analysis. Notice that the best fixpoint (or least-fixpoint) is $[0, 10000]$ but widening finds a less precise fixpoint $[0, 65535]$. The weaker solution is referred to as a post-fixpoint.

Obviously this further loss of precision is undesired. Ideally, the fixpoint that is found is the best possible characterisation of the abstract semantics. This motivates the development of new solving techniques which altogether dispose of Kleene iteration and fixpoint acceleration techniques such as widening. The next body of work of the thesis explores the possibility of replacing Kleene iteration with mathematical optimisation. The work is presented in two parts (Chapters 5 and 6). In Chapter 5:

- It is shown that the abstract semantics of a binary interval analysis can be

re-formulated as a system of *min* and *max* constraints.

- A method is shown for solving the reformulated abstract semantics. The solving technique computes the least-fixpoint as a series of linear programming problems, thus without the need for Kleene iteration or fixpoint acceleration techniques.

- It is shown that the number of linear programs that must be solved can be minimised through the use of heuristics. Experimental results are shown to support this claim.

Finally, Chapter 6 proposes an extension to the work shown in Chapter 5, which allows the method to account for integer overflow scenarios.

At the fundamental level, a computer integer is a collection of bits stored in a register. An unsigned 32-bit integer is a vector $\boldsymbol{x} = \langle x_0, \ldots, x_{31} \rangle$ whose interpretation is $\sum_{i=0}^{31} 2^i x_i$. Such an interpretation allows the representation of integers between 0 and $2^{32} - 1$. Signed values are stored using two's complement encoding. A signed 32-bit integer is the same bit-vector, just interpreted differently, namely as $-2^{31}x_{31} + \sum_{i=0}^{30} 2^i x_i$. The signed interpretation of the vector allows the encoding of integers between -2^{31} and $2^{31} - 1$. Notice that since there are a finite number of bits in which to encode numeric values, the range of values which a register may assume is bounded. Now consider the following snippet of x86 assembler code:

```
mov eax, 0xffffffff
add eax, 0x5
```

Listing 1.4: Integer overflow scenario.

The first instruction moves a constant hex value into the 32-bit `eax` register. After the execution of this instruction `eax` may either be interpreted as $2^{32} - 1$ in an unsigned context, or as -1 in a signed context. The second instruction adds five to the value of the `eax` register. Notice that if `eax` is interpreted unsigned, then $(2^{32} - 1) + 5$ is outside of the numeric range that the register can express, yet this code is completely valid. The exact outcome varies between hardware platforms, but in most general purpose CPU architectures, an integer overflow occurs. In this setting, when the result of an arithmetic operation is outside of the integer

range of the destination register, the result wraps around like a modulo system and a flag in the status register is set. At the end of the code snippet shown in Listing 1.4, the unsigned value of `eax` is equal to 4, since $(2^{32}-1)+5 \bmod 2^{32} = 4$.

Wrapping is one intricacy which must be taken into consideration, but wrapping itself depends upon the intended interpretation of a register. Consider that `eax` is interpreted signed in the above snippet. In this case `0xffffffff` corresponds to the signed value -1 and $-1 + 5 = 4$. Notice how no integer overflow occurs for the signed interpretation of `eax`, whereas an overflow does occur for an unsigned interpretation of `eax`. In fact, when the `add` instruction computes the value of the destination register, it does so with no regard for signedness. The two's complement encoding ensures that the result is correct for both signed and unsigned interpretations of `eax`. This is shown diagrammatically in Figure 2.

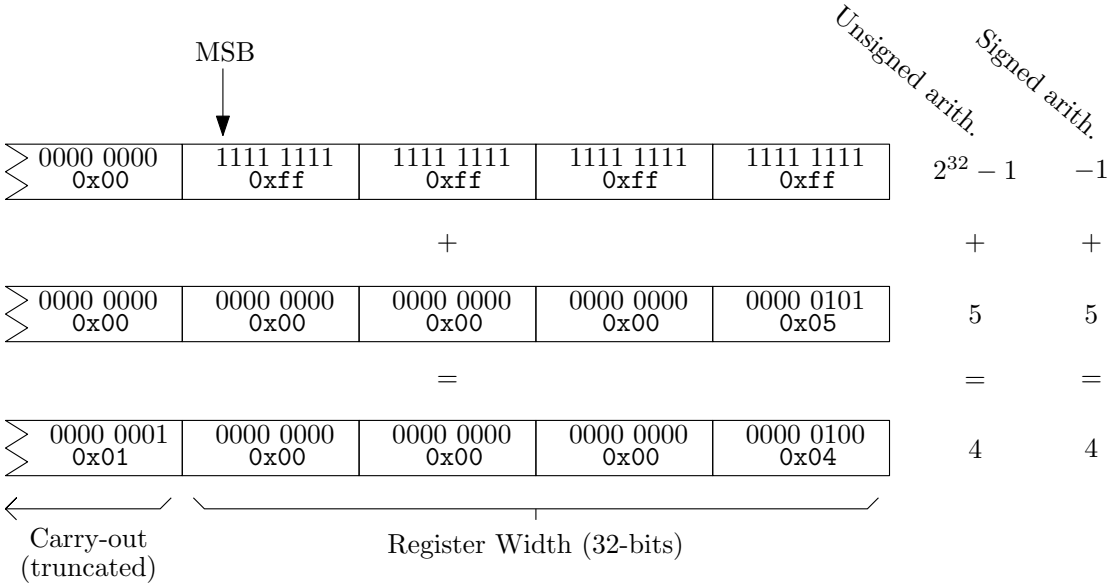


Figure 2: Signed and unsigned interpretations of ⟨`add eax, 5`⟩.

Whilst it may be tempting to ignore the awkward details of integer overflows, to do so is unsound. Besides, some of the most subtle software bugs stem from unforeseen integer overflow scenarios. The infamous heap overflow vulnerability (discussed later in Chapter 6) usually manifests itself as an overlooked integer overflow. Further, sometimes integer overflow behaviours are a part of the intended functionality of a program. For these reasons, the range analysis described in Chapter 5 should be extended to cater for integer overflows. However, the faithful

modelling of these scenarios represents a problem. Recall that the range analysis proposed in Chapter 5 is underpinned by linear optimisation. It is not clear how non-linear modular arithmetic should be incorporated into linear programs. Furthermore, modular arithmetic cannot be encoded as *min/max* constraints, so the binary search method shown in Chapter 5 does not help. One possible course of action is to make a conservative over-approximation for every operation that may cause an overflow, but since most arithmetic operations fall into this category, such an approach is likely to produce very weak results. The problem is further complicated by the differing overflow behaviours of signed and unsigned arithmetic.

Motivated by the need to integrate integer overflow (and thus modular arithmetic) into the range analysis described in Chapter 5, the following contributions are presented in Chapter 6:

- It is shown that modulo arithmetic operations are merely piecewise linear functions.
- A framework is presented by which piecewise linear functions can be encoded within a mixed-integer linear program. This allows integer overflow scenarios to be encoded within the range analysis.
- It is shown that by inferring the intended interpretations of the registers of a binary program a priori, the number of optimisation variables required can be reduced, thereby improving the performance of the analysis. Experimental results are shown to support this.

1.5 Roadmap

The remainder of this thesis is structured as follows. Since the work presented is closely related to abstract interpretation, Chapter 2 discusses AI in detail, giving examples. Chapter 3 describes how to abstract SAT models as ranges (and sets). Chapter 4 then shows how the algorithms from Chapter 3 can, when complimented with quantified Boolean formulae, be harnessed to incrementally infer control flow from binary code that includes indirect jumps. The focus of the thesis then shifts slightly, assuming that the control graph has been acquired. Chapter 5 introduces

a new (and more general) range analysis underpinned by linear optimisation, thus sidestepping the need for fixpoint acceleration techniques. Chapter 6 extends the method shown in Chapter 5, allowing integer overflow scenarios to be modelled. Chapter 7 discusses related work and finally, Chapter 8 draws this thesis to a close with some concluding remarks and the discussion of some possible future work.

Chapter 2

Abstract Interpretation

Chapter 1 offered a brief overview of the abstract interpretation framework. This chapter describes the framework in more detail through the development of several simple abstract interpretations.

2.1 A Toy Programming Language

The example interpretations presented in this chapter will analyse programs written in a simple grammar which shall be referred to as $\mathcal{G}$. The language reflects most of the features of a typical programming language: variables, assignment, arithmetic, conditional branches and loops. The grammar of $\mathcal{G}$ is shown in Backus-Naur form in Figure 3. To simplify the presentation, first, integer variables may be arbitrarily large or small and the same applies for integer constants (fixed-width integers are studied in Chapter 6). Secondly, assume that the three variables available in $\mathcal{G}$ programs (`x`, `y` and `z`) are always in scope, even if they are not used. This means that the example interpretations need not be parameterised by the set of variables in scope. Finally, assume all operator associativity and precedence is as one might expect.

In the following sections, a discussion is offered regarding the design of Galois-connection-based abstract interpretations [32] for $\mathcal{G}$ programs. The theme of the examples will be related to inferring variable signedness at each program point. First, an analysis is presented which abstracts numeric values as a set of possible signs. After identifying the shortcomings of such a coarse domain, the interpretation is revised to intervals. Issues relating to slow fixpoint convergence are then

$$
\begin{array}{lll}
\textsc{Var} & ::= & \texttt{x} \mid \texttt{y} \mid \texttt{z} \\
\textsc{RelOp} & ::= & > \mid \geq \mid < \mid \leq \mid = \mid \neq \\
\textsc{ArithOp} & ::= & + \mid - \mid \cdot \mid \div \\
\textsc{Term} & ::= & \mathbb{Z} \mid \textsc{Var} \\
\textsc{Expr} & ::= & \textsc{Term} \\
 & & \mid \textsc{Term ArithOp Term} \\
\textsc{Cond} & ::= & (\textsc{Term RelOp Term}) \\
\textsc{Stmt} & ::= & \mid \textsc{Var} \leftarrow \textsc{Expr}; \\
 & & \mid \texttt{if } \textsc{Cond} \texttt{ then } \textsc{StmtList} \texttt{ fi} \\
 & & \mid \texttt{if } \textsc{Cond} \texttt{ then } \textsc{StmtList} \texttt{ else } \textsc{StmtList} \texttt{ fi} \\
 & & \mid \texttt{while } \textsc{Cond} \texttt{ do } \textsc{StmtList} \texttt{ done} \\
\textsc{StmtList} & ::= & \textsc{Stmt StmtList} \mid \epsilon \\
\textsc{Prog} & ::= & \textsc{StmtList}
\end{array}
$$

Figure 3: The grammar of the toy language, $\mathcal{G}$.

studied, before a fixpoint acceleration tactic is proposed. It is shown that whilst fixpoint acceleration guarantees fast termination, the precision of the analysis may be compromised.

2.2 Abstraction with Signs

Consider the small $\mathcal{G}$ program `example1.g` and the corresponding control flow graph (CFG) shown in Figure 4. Each statement is numbered, thus dictating a set of program points. Each program point refers to the program state immediately prior to the execution of the corresponding statement. Program points are henceforth referred to as P_i, where in this case $1 \leq i \leq 11$. The program consists of a simple loop with a conditional nested inside. In each arm of the conditional, `x` is modified; in one arm by addition and in the other, by subtraction. The program terminates when `x` exceeds 9. Now suppose that an abstract interpretation is required to confirm that the value of `x` is never negative after its initialisation at P_3. It should be clear from a cursory inspection of the program that this is indeed the case. The remainder of this section will discuss how an abstract interpretation can be formulated to formally prove this claim.

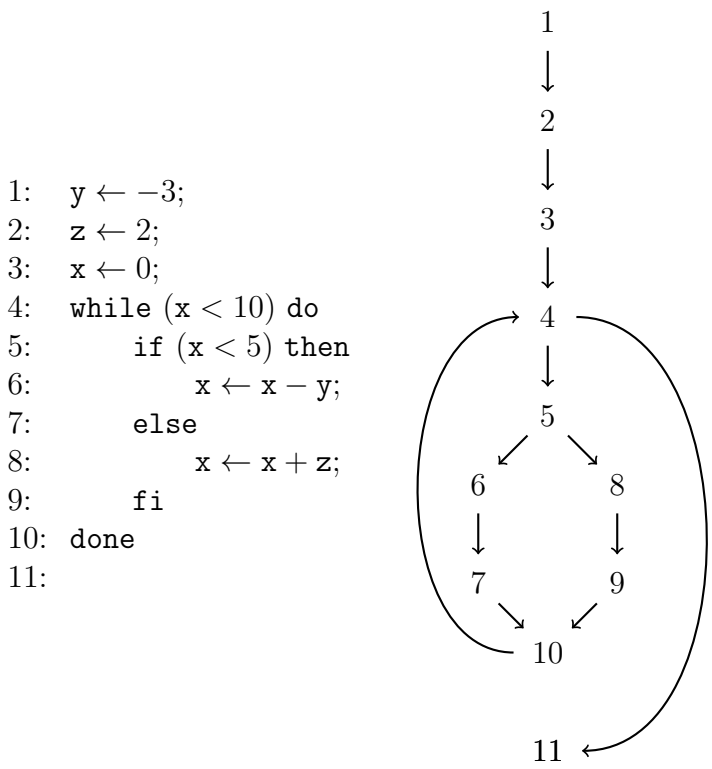


Figure 4: `example1.g`.

2.2.1 Concrete Domain and Collecting Semantics

Recall from Chapter 1 that the first step in designing an abstract interpretation is to define a concrete domain. For a signedness analysis the concrete domain needs the ability to describe (precisely) the possible signs of each of the three program variables at a single point in the program. Let signedness be defined as follows:

Definition 1 (Signedness)**.** *A value less than zero is negative. A value greater than zero is positive. Zero is a special case which is neither positive nor negative.*

The possible signs of a variable at a single program point is both naturally and precisely described by a set of possible numeric values that the variable may assume. Such a set is referred to as a value-set. Formally, the domain of value-sets, V, is the set of all possible sets of integers, i.e. $\wp(\mathbb{Z})$. From a value-set it is possible to say whether a variable may be positive, negative, zero, or any combination of these. Because $\mathcal{G}$ programs always deploy three variables, `x, y` and `z`, the domain of value-sets is lifted so as to accommodate the values of the

three variables. This is achieved by using a set of 3-tuples. Formally, the set of all possible sets of 3-tuples, L, is defined as $\wp(\mathbb{Z}^3)$; this shall serve as the concrete domain. An ordering is placed upon L, thus forming a complete (but infinite) lattice $\langle L, \subseteq_L, \cup_L, \cap_L \rangle$.

Definition 2 (The ordering and domain operations of L)**.**

$$\begin{array}{lcll} \subseteq_L : L \times L & & a \subseteq_L b & \iff & a \subseteq b \\ \cup_L : L \times L & \to \quad L & a \cup_L b & \triangleq & a \cup b \\ \cap_L : L \times L & \to \quad L & a \cap_L b & \triangleq & a \cap b \end{array}$$

The bottom element of the lattice, $\bot_L = \emptyset$, indicates the absence of values, whereas the top element, $\top_L = \mathbb{Z}^3$, indicates that the variables could take any value.

Next, a collecting semantics is defined. The role of the collecting semantics is to enumerate the values that arise at each program point. This amounts to, for each program point P_i, specifying a semantic equation that characterises operationally the concrete program state. Each state S_i is an element drawn from the concrete domain and is defined in terms of predecessor program points as described by the control flow of the program. For example, the values that could arise at P_3 are described by the set $S_3 \in L$, which is defined in terms of $S_2 \in L$. If a program point has multiple predecessors, as is the case for P_4, then the semantic equation describing the state must include a join ($\cup_L$) to merge the possible concrete states from both predecessor states.

The collecting semantics of `example1.g` is shown below. For each program point the semantic equation is shown along with a brief description of its construction:

- P_1**:** Prior to the start of program execution, the values of the variables `x`, `y` and `z` are uninitialised and thus could assume any value:

$$S_1 = \top_L$$

- P_2**:** `y` is initialised to -3. The other variables, `x` and `z`, remain the same:

$$S_2 = \{\langle x, -3, z \rangle \mid \langle x, y, z \rangle \in S_1\}$$

- P_3: `z` is initialised to 2. Other variables remain the same:

$$S_3 = \{\langle x, y, 2\rangle \mid \langle x, y, z\rangle \in S_2\}$$

- P_4: A control flow join occurs. One incoming edge initialises `x` to 0:

$$S_4 = \{\langle 0, y, z\rangle \mid \langle x, y, z\rangle \in S_3\} \cup_L S_{10}$$

- P_5: Execution enters the loop, so the loop condition $\texttt{x} < 10$ is true. To reflect this the values of `x` are restricted:

$$S_5 = \{\langle x, y, z\rangle \mid \langle x, y, z\rangle \in S_4 \wedge x < 10\}$$

- P_6: Execution enters the true branch of the conditional, therefore $\texttt{x} < 5$ holds. Again, `x` is restricted:

$$S_6 = \{\langle x, y, z\rangle \mid \langle x, y, z\rangle \in S_5 \wedge x < 5\}$$

- P_7: The value of `y` is subtracted from `x`. It is assumed that integer underflow may not occur:

$$S_7 = \{\langle x - y, y, z\rangle \mid \langle x, y, z\rangle \in S_6\}$$

- P_8: Execution enters the false branch of the conditional, $\texttt{x} < 5$, therefore the converse ($\texttt{x} \geq 5$) holds. `x` is restricted to reflect this:

$$S_8 = \{\langle x, y, z\rangle \mid \langle x, y, z\rangle \in S_5 \wedge x \geq 5\}$$

- P_9: The value of `z` is added to `x`:

$$S_9 = \{\langle x + z, y, z\rangle \mid \langle x, y, z\rangle \in S_8\}$$

- P_{10}: Control flow joins from P_7 and P_9:

$$S_{10} = S_7 \cup_L S_9$$

- P_{11}**:** Execution leaves the loop, therefore the converse of the loop condition holds:

$$S_{11} = \{\langle x, y, z\rangle \mid \langle x, y, z\rangle \in S_4 \wedge x \geq 10\}$$

The least solution of the collecting semantics describes the values that can arise at each point in the program. From this information it is possible, at least theoretically, to decide the signage of `x` at each point in the program. In reality though, the least solution of the collecting semantics is rarely computable due to resource limits. In principle, the fixpoint equations could be solved by Kleene iteration, i.e. repeated evaluation until a fixpoint is found. However, notice that the height of the L lattice is not finite. This means that the S_i sets could grow arbitrarily, meaning that Kleene iteration may not terminate. From a practical standpoint, a non-terminating analysis is useless since no information is inferred. Even supposing that the concrete domain were finite, there is still the possibility that the sets could become large and unmanageable. Abstraction aims to overcome these limitations. By applying abstraction, it is possible to approximate the least solution of the collecting semantics efficiently, whilst also guaranteeing termination. The remainder of this section will demonstrate by applying abstraction to the collecting semantics of `example1.g`.

2.2.2 Abstract Domain and Domain Correspondence

The first step of abstraction requires the definition of an abstract domain. The role of the abstract domain is to approximate elements of the concrete domain. Recall that the aim of the analysis is to determine the possible signs of each variable at each program point and specifically, if `x` can ever be negative after its initialisation. To this end, the abstract domain needs to express that a variable at any given program point could be positive, negative, zero, or any combination of these. Therefore, it seems natural to abstract the state of a single variable at any given program point as an element drawn from $\wp(\{-, 0, +\})$. Let this set be denoted W. The signs of the three variables at any given program point is then an element drawn from W^3. Let this set be denoted M. This shall serve as the abstract domain. Note that the largest possible element of M in terms of storage is $\langle\{-, 0, +\}, \{-, 0, +\}, \{-, 0, +\}\rangle$, which can easily be encoded in a few bytes. The domain is also finite, which contributes to a termination argument (discussed

later in Section 2.2.4, Page 32).

Next, an ordering is placed upon W, therefore inducing a complete and finite lattice $\langle W, \sqsubseteq_W, \sqcup_W, \sqcap_W \rangle$:

Definition 3 (Ordering and domain operations of W)**.**

$$\begin{array}{rcl} s \sqsubseteq_W s' & \iff & s \subseteq s' \\ s \sqcup_W s' & \triangleq & s \cup s' \\ s \sqcap_W s' & \triangleq & s \cap s' \end{array}$$

The bottom element $\bot_W = \emptyset$ signifies the absence of sign information and the top element $\top_W = \{-, 0, +\}$ represents any possible sign. Note the distinction between $\bot_W$ and $\top_W$: the former indicates that nothing is described, whereas the latter indicates that nothing is known.

The ordering and domain operations of W are then lifted pointwise to accommodate M, thus also forming a complete and finite lattice $\langle M, \sqsubseteq_M, \sqcup_M, \sqcap_M \rangle$:

Definition 4 (Ordering and domain operations of M)**.**

$$\begin{array}{rcl} \langle x, y, z \rangle \sqsubseteq_M \langle x', y', z' \rangle & \iff & x \sqsubseteq_W x' \wedge y \sqsubseteq_W y' \wedge z \sqsubseteq_W z' \\ \langle x, y, z \rangle \sqcup_M \langle x', y', z' \rangle & \triangleq & \langle x \sqcup_W x', y \sqcup_W y', z \sqcup_W z' \rangle \\ \langle x, y, z \rangle \sqcap_M \langle x', y', z' \rangle & \triangleq & \langle x \sqcap_W x', y \sqcap_W y', z \sqcap_W z' \rangle \end{array}$$

The bottom element $\bot_M = \langle \emptyset, \emptyset, \emptyset \rangle$ indicates that no variable can take any sign. The top element $\top_M = \langle \{-, 0, +\}, \{-, 0, +\}, \{-, 0, +\} \rangle$ indicates that the variables may be of any sign.

Domain Correspondence

With the domains in place, a domain correspondence is next defined. Specifically, the abstraction mapping $\alpha_L : L \to M$ over-approximates an element of the concrete domain as an element of the abstract domain.

Definition 5 (Abstraction mapping)**.**

$$\begin{aligned} \alpha_L(l) &= \langle x, y, z \rangle \\ \textit{where } x &= \{\textit{sign}(x') \mid \langle x', y', z' \rangle \in l\} \quad \wedge \\ y &= \{\textit{sign}(y') \mid \langle x', y', z' \rangle \in l\} \quad \wedge \\ z &= \{\textit{sign}(z') \mid \langle x', y', z' \rangle \in l\} \end{aligned}$$

where the function $\textit{sign} : \mathbb{Z} \to \{-, 0, +\}$ *maps an arbitrary integer to its sign.*

$$\textit{sign}(n) = \begin{cases} - & \textit{if } n < 0 \\ 0 & \textit{if } n = 0 \\ + & \textit{if } n > 0 \end{cases}$$

The concretisation mapping $\gamma_M : M \to L$ then describes the concrete states expressed by an element of the abstract domain.

Definition 6 (Concretisation mapping)**.**

$$\gamma_M(\langle x, y, z \rangle) = \{\langle x', y', z' \rangle \mid x' \in \textit{from\_sign}(x) \wedge y' \in \textit{from\_sign}(y) \wedge z' \in \textit{from\_sign}(z)\ \}$$

where the function $\textit{from\_sign} : W \to V$ *maps a variable's abstract sign information to possible integer values:*

$$\begin{aligned} \textit{from\_sign}(s) &= N \cup Z \cup P \\ \textit{where}: \ N &= \begin{cases} \{-\infty, \ldots, -1\} & \textit{if } (-) \in s \\ \emptyset & \textit{otherwise} \end{cases} \quad \bigwedge \\ Z &= \begin{cases} \{0\} & \textit{if } 0 \in s \\ \emptyset & \textit{otherwise} \end{cases} \quad \bigwedge \\ P &= \begin{cases} \{1, \ldots, +\infty\} & \textit{if } (+) \in s \\ \emptyset & \textit{otherwise} \end{cases} \end{aligned}$$

Note the concretisation mapping can be defined directly in terms of the abstraction mapping, i.e. $\gamma_M(m) = \cup_L\{l \in L \mid \alpha_L(l) \sqsubseteq_M m\}$, however, to aid reader comprehension, a more explicit definition is used.

The α and γ mappings can be shown to form a Galois connection, meaning that every concrete state has a unique best abstraction. This is a useful algebraic property for an abstract interpretation, since it means that any element of the concrete domain can be abstracted and furthermore, solving may not cycle between equivalent abstractions in a fixpoint computation.

Definition 7 (Galois Connection [32])**.** *Consider two partially ordered domains* $\langle C, \subseteq_C \rangle$ *and* $\langle A, \sqsubseteq_A \rangle$*. The correspondence between* C *and* A *is described by abstraction and concretisation mappings,* $\alpha_C : C \to A$ *and* $\gamma_A : A \to C$*. The domains* A *and* C *form a Galois connection, written* $A \underset{\gamma_A}{\overset{\alpha_C}{\leftrightarrows}} C$*, when:*

$$\forall c \in C.\ \forall a \in A.\ \alpha_C(c) \sqsubseteq_A a \iff c \subseteq_C \gamma_A(a)$$

Showing that the correspondence between L and M forms a Galois connection amounts to showing that $\forall l \in L.\ \forall m \in M.\ \alpha_L(l) \sqsubseteq_M m \iff l \subseteq_L \gamma_M(m)$. A proof of this is shown in Corollary 1 on Page 166.

2.2.3 Abstract Semantics

With the abstract domain and the domain correspondence defined, the abstract semantics is now specified. The abstract semantics, like the collecting semantics, describe the effect that each program statement has upon the analysis. The abstract semantics, however, works over elements of the abstract domain. As with the concrete semantics, one semantic equation is defined for each program point, thus for each P_i, an equation S'_i defines the abstract state:

- P_1**:** Prior to the start of program execution, the program variables are uninitialised and could take any value:

$$S'_1 = \top_M$$

- P_2**:** `y` is assigned the value -3, therefore `y` is negative:

$$S'_2 = \langle x, \{-\}, z \rangle \text{ where } \langle x, y, z \rangle = S'_1$$

- P_3**:** `z` is assigned the value 2, therefore `z` is positive:

$$S'_3 = \langle x, y, \{+\}\rangle \text{ where } \langle x, y, z\rangle = S'_2$$

- P_4**:** Control flow converges from P_3 and P_{10}. The update of `x` to zero and the merge of the state from P_{10} must be captured:

$$S'_4 = \langle \{0\}, y, z\rangle \sqcup_M S'_{10} \text{ where } \langle x, y, z\rangle = S'_3$$

- P_5**:** Control flow enters the loop body, so $\mathtt{x} < 10$ holds. To reflect this, the possible signs of `x` are restricted. However, because $\forall s \in \{-, 0, +\}.\ \exists x \in \mathbb{Z}.\ x < 10 \wedge sign(\mathtt{x}) = s$, no suitable restriction exists. This is a symptom of the coarseness of the abstract domain:

$$S'_5 = S'_4$$

- P_6**:** Control flow enters the true branch of the conditional $\mathtt{x} < 5$. Again, no suitable restriction exists:

$$S'_6 = S'_5$$

- P_7**:** The value of `y` is subtracted from `x`. The following definitions are used to capture this update:

  Definition 8 (Subtraction of single signs)**.** *The infix function* $-'_W : \{-, 0, +\} \times \{-, 0, +\} \to W$ *maps a single sign,* x*, and a single sign,* y*, to a set of signs that can result from subtraction:*

$$x -'_W y = \begin{cases} \{-\} & \text{if } (x = (-) \wedge y \in \{0, +\}) \vee (x = 0 \wedge y = (+)) \\ \{0\} & \text{if } x = y = 0 \\ \{+\} & \text{if } (x = (+) \wedge y \in \{-, 0\}) \vee (x = 0 \wedge y = (-)) \\ \top_W & \text{otherwise} \end{cases}$$

  The above function is then lifted to work over sets of signs.

Definition 9 (Subtraction of elements of W (sets of signs))**.** *The infix function* $-_W : W \times W \to W$ *maps a set of signs* x *and a set of signs* y *to a set of signs that can result from subtraction:*

$$x -_W y = \bigcup \{m -'_W n \mid m \in x \wedge n \in y\}$$

The abstract equation for P_7 is then:

$$S'_7 = \langle x -_W y, y, z\rangle \text{ where } \langle x, y, z\rangle = S'_6$$

- P_8**:** Execution enters the false branch of the conditional $\texttt{x} < 5$, therefore the converse ($\texttt{x} \geq 5$) holds. Since all integers greater than or equal to 5 are positive, `x` is restricted to positive signs only:

$$S'_8 = \langle x \sqcap_W \{+\}, y, z\rangle \text{ where } \langle x, y, z\rangle = S'_5$$

- P_9**:** The value of `z` is added to `x`. The semantics of addition of signs works analogously to the semantics of subtraction of signs:

  Definition 10 (Addition of single signs)**.** *The infix function,* $+'_W : \{-, 0, +\} \times \{-, 0, +\} \to W$*, computes the outcome of the addition of two signs:*

$$x +'_W y = \begin{cases} \{-\} & \textit{if } (x = (-) \wedge y \in \{-, 0\}) \vee (x \in \{-, 0\} \wedge y = (-)) \\ \{0\} & \textit{if } x = y = 0 \\ \{+\} & \textit{if } (x = (+) \wedge y \in \{0, +\}) \vee (x \in \{0, +\} \wedge y = (+)) \\ \top_W & \textit{otherwise} \end{cases}$$

  Definition 11 (Addition of elements of W)**.** *The infix function* $+_W : W \times W \to W$ *computes the result of the addition of two sets of signs:*

$$x +_W y = \bigcup \{m +'_W n \mid m \in x \wedge n \in y\}$$

The abstract equation for P_9 is then:

$$S'_9 = \langle x +_W z, y, z\rangle \text{ where } \langle x, y, z\rangle = S'_8$$

- P_{10}**:** Control flow joins from P_7 and P_9:

$$S'_{10} = S'_7 \sqcup_M S'_9$$

- P_{11}**:** Execution leaves the while loop, so the loop condition `x` < 10 is false. Since all integers greater than or equal to 10 are positive, the sign of `x` is restricted to positive:

$$S'_{11} = \langle x \sqcap_W \{+\}, y, z \rangle \text{ where } \langle x, y, z \rangle = S'_4$$

2.2.4 Termination and Monotonicity

The abstract semantics are almost ready to be solved. However, first an argument for termination needs to be constructed. The sequence of abstract states that arise at a given program point over the course of solving is referred to as a chain. Each element in the chain is the result of one solving iteration. If the domains form a Galois connection, the abstract lattice is of finite height and the abstract semantics is monotonic, then termination is guaranteed under the ascending chain condition. Since $L \underset{\alpha_L}{\overset{\gamma_M}{\leftrightarrows}} M$ is a Galois connection and the abstract lattice is of finite height, all that remains is to show that the abstract semantics is monotonic.

Definition 12 (Monotonic Function)**.** *Consider a function $f : D \to D$ where D is an ordered set; f is a monotonic function if it preserves the ordering $\leq_D$ of the elements of D:*

$$\forall a \in D.\ \forall b \in D.\ a \leq_D b \implies f(a) \leq_D f(b)$$

Each abstract semantic equation S'_i can be interpreted as a transfer function F'_i which maps direct predecessor states to a new state. For example, S'_2 can be interpreted as a function $F'_2 : M \to M$ such that $F'_2(\langle x, y, z \rangle) = \langle x, \{-\}, z \rangle$. Then F'_2 is monotonic if $\forall a \in M.\ \forall b \in M.\ a \sqsubseteq_M b \implies F'_2(a) \sqsubseteq_M F'_2(b)$. The proofs of monotonicity are given on Page 167 (Theorem 3).

2.2.5 Solving

Finally, the abstract semantics are solved to find an over-approximation of the collecting semantics. From this approximation a deduction can be made as to

P_i	$S'_i \in M$	
	Iteration 1	Iteration 2
P_1	$\top_M$	$\top_M$
P_2	$\langle\{-,0,+\},\{-\},\{-,0,+\}\rangle$	$\langle\{-,0,+\},\{-\},\{-,0,+\}\rangle$
P_3	$\langle\{-,0,+\},\{-\},\{+\}\rangle$	$\langle\{-,0,+\},\{-\},\{+\}\rangle$
P_4	$\langle\{0\},\{-\},\{+\}\rangle$	$\langle\{0,+\},\{-\},\{+\}\rangle$
P_5	$\langle\{0\},\{-\},\{+\}\rangle$	$\langle\{0,+\},\{-\},\{+\}\rangle$
P_6	$\langle\{0\},\{-\},\{+\}\rangle$	$\langle\{0,+\},\{-\},\{+\}\rangle$
P_7	$\langle\{+\},\{-\},\{+\}\rangle$	$\langle\{+\},\{-\},\{+\}\rangle$
P_8	$\langle\{\},\{-\},\{+\}\rangle$	$\langle\{+\},\{-\},\{+\}\rangle$
P_9	$\langle\{\},\{-\},\{+\}\rangle$	$\langle\{+\},\{-\},\{+\}\rangle$
P_{10}	$\langle\{+\},\{-\},\{+\}\rangle$	$\langle\{+\},\{-\},\{+\}\rangle$
P_{11}	$\langle\{+\},\{-\},\{+\}\rangle$	$\langle\{+\},\{-\},\{+\}\rangle$

Table 1: Fixpoint computation (via Kleene iteration) applied to the abstract semantics for `example1.g`.

whether `x` is indeed always positive after its initialisation. Typically, the abstract semantics are solved via Kleene iteration, which works as follows. Prior to solving, each S'_i is initialised to $\bot_M$. Then each abstract semantic equation is evaluated one after the other. Further iterations are performed until no S'_i set changes between consecutive iterations; in other words, until a fixpoint has been met. When a fixpoint is met, a solution to the abstract semantics has been found.

The first two iterations of the solving process are shown in Table 1. Iteration 3 is not shown, since the outcome is the same as for iteration 2, meaning that solving has converged. From the finalised abstract states, it is easy to see that `x` cannot be negative after program point 3, i.e. $\forall\langle x,y,z\rangle \in \{S'_4,\ldots,S'_{11}\}.\ (-) \notin x$.

2.3 Range Analysis with Intervals

In the last section, an interpretation was designed which abstracted numeric values as signs to determine whether a variable could turn negative. The abstraction was sufficient in proving such a property for `example1.g`, but unfortunately in many cases the use of such a coarse abstract domain will introduce false positives. In the case of the previous example, a false positive would manifest as the possibility of a negative value for `x` after P_4. In concrete executions it is impossible for `x` to be negative between P_4 and P_{11}.

```
1:  x ← 5;
2:  while x > 0 do
3:      x ← x −1
4:  done
5:
```

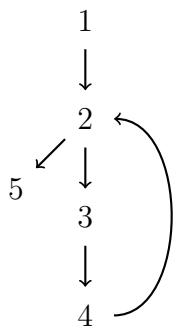


Figure 5: `example2.g`.

To illustrate how a coarse abstraction can introduce false positives, consider the simple program `example2.g` (shown in Figure 5). From a cursory inspection of the program, it should be clear that after `x` has been initialised it can only assume positive or zero values. Suppose that the sign abstraction from the last section is applied to this program. The abstract semantics is:

$$
\begin{array}{lll}
S'_1 & = \top_M & \\
S'_2 & = \langle \{+\}, y, z\rangle \sqcup_M S'_4 & \text{where } \langle x, y, z\rangle = S'_1 \\
S'_3 & = \langle x \sqcap_W \{+\}, y, z\rangle & \text{where } \langle x, y, z\rangle = S'_2 \\
S'_4 & = \langle x -_w \{+\}, y, z\} & \text{where } \langle x, y, z\rangle = S'_3 \\
S'_5 & = \langle x \sqcap_W \{-, 0\}, y, z\rangle & \text{where } \langle x, y, z\rangle = S'_2
\end{array}
$$

The abstract equations can be shown to be monotonic analogously to before. Solving via Kleene iteration computes the information shown in Table 2. The least-fixpoint is found after two iterations. Notice that the analysis was unable to determine that after P_1, the variable `x` may not be negative. This stems from the fact that a positive number minus a positive number could yield either a positive, zero, or negative result. Consequently, after iteration 1, the abstraction of `x` at P_4 must be approximated as $\top_W$. Because $(-) \in \top_W$ it cannot be deduced that `x` is not negative at P_4. This is a false positive since in concrete executions of the program, `x` is never less than one at P_3, so the smallest possible value of `x` at P_4 is zero. Whilst this is not a problem from a soundness point of view (the inferred information safely over-approximates the possible states), false positives are undesirable because manual intervention is required to determine if

P_i	Iteration	
	1	2
1	$\langle \top_W, \top_W, \top_W \rangle$	$\langle \top_W, \top_W, \top_W \rangle$
2	$\langle \{+\}, \top_W, \top_W \rangle$	$\langle \top_W, \top_W, \top_W \rangle$
3	$\langle \{+\}, \top_W, \top_W \rangle$	$\langle \{+\}, \top_W, \top_W \rangle$
4	$\langle \top_W, \top_W, \top_W \rangle$	$\langle \top_W, \top_W, \top_W \rangle$
5	$\langle \bot_W, \top_W, \top_W \rangle$	$\langle \{-, 0\}, \top_W, \top_W \rangle$

Table 2: False positives introduced via a coarse abstract domain.

they are faults. False positives are, in most cases, unavoidable due to the over-approximation introduced by abstraction. Nevertheless, the fewer false positives inferred the better, as this means fewer cases must be checked by hand.

To eliminate excessive false positives, a more expressive abstract domain, such as the interval domain, could be deployed. By tracking ranges of values, the analysis is able to collect more precise information about the possible values of the program variables, whilst keeping the cost of the analysis low. In the remainder of this section, a revised abstraction is developed which is based upon the interval domain. Using this revised abstraction it is shown that it is possible to infer more precise sign information for `example2.g`.

2.3.1 Revised Abstraction

Recall from the introductory chapter, that an interval is a compact way of approximating a set of numeric values. Let the domain of intervals $I = \{\emptyset\} \cup \{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, +\infty\} \wedge l \leq u\}$. Note that the extremal bounds, $-\infty$ and $+\infty$, are symbolic values that bound any integer value below and above respectively. The domain forms a complete lattice $\langle I, \sqsubseteq_I, \sqcup_I, \sqcap_I \rangle$:

Definition 13 (Ordering and domain operations for I)**.**

$$
\begin{array}{rcl}
\emptyset \sqsubseteq_I x & \triangleq & \textsc{True} \\
[a,b] \sqsubseteq_I [c,d] & \iff & c \leq a \wedge b \leq d \\
\emptyset \sqcup_I x & \triangleq & x \\
[a,b] \sqcup_I [c,d] & \triangleq & [min(a,c), max(b,d)] \\
\emptyset \sqcap_I x & \triangleq & \emptyset \\
[a,b] \sqcap_I [c,d] & \triangleq & \begin{cases} [max(a,c), min(b,d)] & \textit{if } max(a,c) \leq min(b,d) \\ \emptyset & \textit{otherwise} \end{cases}
\end{array}
$$

The bottom element, $\bot_I = \emptyset$, expresses the absence of values, whereas the top element, $\top_I = [-\infty, \infty]$, expresses that the interval contains all values. To model the numeric values of all three variables, `x`, `y` and `z`, the domain is lifted to interval triples, I^3. Let this revised abstract domain be denoted K. The lifted domain then forms a complete lattice of infinite height $\langle K, \sqsubseteq_K, \sqcup_K, \sqcap_K \rangle$, where the domain ordering and operations are pointwise liftings:

Definition 14 (Ordering and domain operations for K)**.**

$$
\begin{array}{rcl}
\langle x,y,z \rangle \sqsubseteq_K \langle x',y',z' \rangle & \iff & (x \sqsubseteq_I x') \wedge (y \sqsubseteq_I y') \wedge (z \sqsubseteq z') \\
\langle x,y,z \rangle \sqcup_K \langle x',y',z' \rangle & \triangleq & \langle x \sqcup_I x', y \sqcup_I y', z \sqcup_I z' \rangle \\
\langle x,y,z \rangle \sqcap_K \langle x',y',z' \rangle & \triangleq & \langle x \sqcap_I x', y \sqcap_I y', z \sqcap_I z' \rangle
\end{array}
$$

Next, the correspondence between the concrete domain, L, and the revised abstract domain, K, is specified. Since the abstract domain is a pointwise lifting, it is natural to first define the mappings between a single value-set and a single interval. The mapping $\alpha_V : V \to I$ abstracts a value set as the smallest interval that includes the values. Conversely, the mapping $\gamma_I : I \to V$ computes the numeric values that an interval encloses.

Definition 15 (Domain correspondence between I and V)**.**

$$
\begin{array}{lll}
\alpha_V(\emptyset) = \emptyset & \gamma_I(\emptyset) & = \emptyset \\
\alpha_V(l) = [min(l), max(l)] & \gamma_I([l,u]) & = \{x \in \mathbb{Z} \mid l \leq x \leq u\}
\end{array}
$$

The correspondence between L and K is then defined in terms of α_V and γ_I. The abstraction mapping $\alpha_L : L \to K$ boxes the component sets of a concrete state into

a 3-tuple of the smallest possible over-approximate intervals. The concretisation mapping $\gamma_K : K \to L$ describes the sets of possible concrete states represented by a 3-tuple of intervals.

Definition 16 (Domain correspondence between L and K)**.**

$$\begin{array}{rcll} \alpha_L(l) & = & \langle \alpha_I(x'), \alpha_I(y'), \alpha_I(z') \rangle & \\ \textit{where } x' & = & \{x \mid \langle x, y, z \rangle \in l\} & \wedge \\ y' & = & \{y \mid \langle x, y, z \rangle \in l\} & \wedge \\ z' & = & \{z \mid \langle x, y, z \rangle \in l\} & \end{array}$$

$$\gamma_K(\langle x, y, z \rangle) = \{\langle x', y', z' \rangle \mid x' \in \gamma_I(x) \wedge y' \in \gamma_I(y) \wedge z' \in \gamma_I(z)\}$$

The correspondence can be shown to form a Galois connection (see Corollary 2 on Page 171). To reiterate, this means that every value set has a canonical best abstraction. Note however, that different abstractions can share the same concretisation, e.g. $\gamma_K(\langle \emptyset, [-\infty, +\infty], [-\infty, +\infty] \rangle) = \gamma_K(\langle [-\infty, +\infty], \emptyset, [-\infty, +\infty] \rangle) = \emptyset$.

2.3.2 Abstract Semantics

Following an analogous process to before, the abstract semantics is now defined, although this time over 3-tuples of intervals. For each program point P_i, a semantic equation $S'_i \in K$ characterises an over-approximation of the concrete state $S_i \in L$. The equations are as follows:

- P_1**:** Prior to execution, the variables are considered uninitialised and thus could assume any value:

$$S'_1 = \top_K$$

- P_2**:** `x` is initialised to 5 and control flow converges from P_4:

$$S'_2 = \langle [5, 5], y, z \rangle \sqcup_K S'_4 \text{ where } \langle x, y, z \rangle = S'_1$$

- P_3**:** Execution enters the body of the loop, so the loop condition ($\texttt{x} > 0$) must be true. To reflect this, the values of `x` are restricted using the interval's meet operator:

$$S'_3 = \langle x \sqcap_I [1, +\infty], y, z \rangle \text{ where } \langle x, y, z \rangle = S'_2$$

- P_4: `x` is decremented. The infix function, $-_I : I \times I \to I$ computes the result of subtracting of one interval from another.

  Definition 17 (Interval Subtraction)**.** *Given two intervals, i_1 and i_2:*

  $$i_1 -_I i_2 = \begin{cases} \emptyset & \text{if } i_1 = \emptyset \vee i_2 = \emptyset \\ [l - u', u - l'] & \text{if } i_1 = [l, u] \wedge i_2 = [l', u'] \end{cases}$$

  The abstract equation for P_4 is then:

  $$S'_4 = \langle x -_I [1, 1], y, z \rangle \text{ where } \langle x, y, z \rangle = S'_3$$

- P_5: Execution leaves the loop, therefore the loop condition is false, i.e. $\mathtt{x} \leq 0$. Again, this is expressed using the interval meet:

  $$S'_5 = \langle x \sqcap_I [-\infty, 0], y, z \rangle \text{ where } \langle x, y, z \rangle = S'_2$$

The abstract semantics can be shown to be monotonic (see Theorem 6 on Page 173).

Recall that before the abstract semantics are solved, a termination argument should be constructed. In the previous interpretation (with signs), termination was guaranteed by the ascending chain condition. Since the revised abstraction does not satisfy the ascending chain condition (the abstract lattice is infinite in height), a different argument must be used to guarantee termination. Actually, `example2.g` was engineered so that the abstract semantics are free of infinite chains; this fact alone suffices as a termination argument. In practice however, it is usually not known whether the solving of an abstract semantics will encounter infinite chains. In such cases, fixpoint acceleration techniques such as widening [32] can used to guarantee termination. An example interpretation that uses widening is detailed later in Section 2.4.

P_i	Iteration				
	1	2	...	6	7
1	$\top_K$	$\top_K$		$\top_K$	$\top_K$
2	$\langle [5,5], \top_I, \top_I \rangle$	$\langle [4,5], \top_I, \top_I \rangle$	...	$\langle [0,5], \top_I, \top_I \rangle$	$\langle [0,5], \top_I, \top_I \rangle$
3	$\langle [5,5], \top_I, \top_I \rangle$	$\langle [4,5], \top_I, \top_I \rangle$	...	$\langle [1,5], \top_I, \top_I \rangle$	$\langle [1,5], \top_I, \top_I \rangle$
4	$\langle [4,4], \top_I, \top_I \rangle$	$\langle [3,4], \top_I, \top_I \rangle$	...	$\langle [0,4], \top_I, \top_I \rangle$	$\langle [0,4], \top_I, \top_I \rangle$
5	$\langle \bot_I, \top_I, \top_I \rangle$	$\langle \bot_I, \top_I, \top_I \rangle$	...	$\langle [0,0], \top_I, \top_I \rangle$	$\langle [0,0], \top_I, \top_I \rangle$

Table 3: Fixpoint computation for `example2.g`.

2.3.3 Solving

The abstract semantics are now solved to compute over-approximate intervals of the numeric values that can arise at each program point. The Kleene solving process is shown in Table 3. Fixpoint convergence is achieved after seven iterations. The analysis correctly infers that `x` can never turn negative after its initialisation; i.e. no interval corresponding to `x` contains a value less than 0 after P_2.

To summarise the message of this section, the abstract domain of signs, M, could only infer weak information about `example2.g` and as a consequence gave false positives. The more expressive domain K, built upon intervals, inferred more precise information and without a false positive. This demonstrates how the choice of abstract domain can affect the precision of the analysis.

2.4 Ensuring Fast Termination with Widening

The first interpretation (in Section 2.2) used domains that formed a Galois connection, a small finite abstract lattice and monotonic abstract semantics, meaning that long or infinite chains were impossible. As a consequence, termination was both fast and assured. The second interpretation (in Section 2.3) also used domains which form a Galois connection and monotonic abstract semantics. Although the abstract lattice was infinite, solving did not encounter long or infinite chains of abstract states. But what about other cases, where for example, the abstract domain is not finite and it is not known whether long or infinite ascending chains exist? In such cases, fixpoint acceleration can be deployed to ensure that both termination occurs and quickly. In this section, widening is demonstrated as a fixpoint acceleration technique.

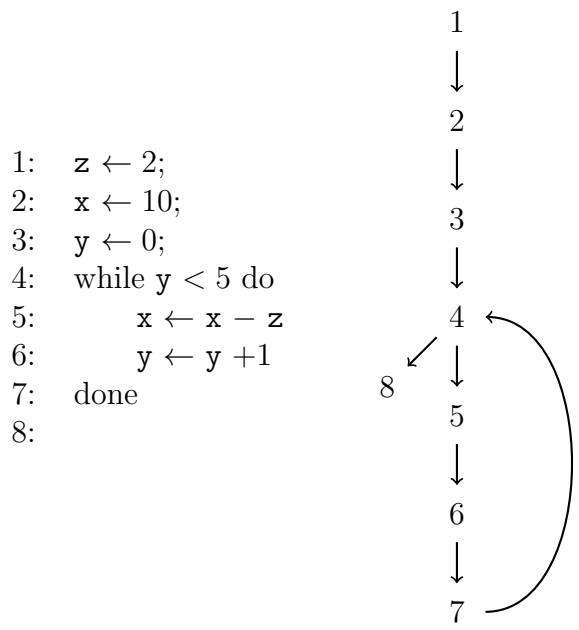


Figure 6: A third $\mathcal{G}$ program, `example3.g`, and the corresponding CFG.

Widening was originally proposed [32] as a complete alternative to the Galois-connection-based approach to abstract interpretation. However, widening can be used to supplement an existing Galois-connection-based interpretation to ensure fast termination. To illustrate the need for widening as a fixpoint acceleration technique, consider the third $\mathcal{G}$ program listed in Figure 6. The program is a simple loop, the body of which will be executed a total of five times. In each loop iteration, two is subtracted from `x` (indirectly via the variable `z`). At the end of program execution (P_8), the value of `x` is zero. The abstract semantics formulated over elements of K are as follows:

$$
\begin{array}{lll}
S'_1 & = \top_K & \\
S'_2 & = \langle x, y, [2,2] \rangle & \text{where } \langle x, y, z \rangle = S'_1 \\
S'_3 & = \langle [10,10], y, z \rangle & \text{where } \langle x, y, z \rangle = S'_2 \\
S'_4 & = \langle x, [0,0], z \rangle \sqcup_K S'_7 & \text{where } \langle x, y, z \rangle = S'_3 \\
S'_5 & = \langle x, y \sqcap_I [-\infty, 4], z \rangle & \text{where } \langle x, y, z \rangle = S'_4 \\
S'_6 & = \langle x -_I z, y, z \rangle & \text{where } \langle x, y, z \rangle = S'_5 \\
S'_7 & = \langle x, y +_I [1,1], z \rangle & \text{where } \langle x, y, z \rangle = S'_6 \\
S'_8 & = \langle x, y \sqcap_I [5, +\infty], z \rangle & \text{where } \langle x, y, z \rangle = S'_4
\end{array}
$$

P_i	Iteration		
	1	2	...
1	$\top_K$	$\top_K$	...
2	$\langle \top_I, \top_I, [2,2] \rangle$	$\langle \top_I, \top_I, [2,2] \rangle$	...
3	$\langle [10,10], \top_I, [2,2] \rangle$	$\langle [10,10], \top_I, [2,2] \rangle$	...
4	$\langle [10,10], [0,0], [2,2] \rangle$	$\langle [8,10], [0,1], [2,2] \rangle$	...
5	$\langle [10,10], [0,0], [2,2] \rangle$	$\langle [8,10], [0,1], [2,2] \rangle$	...
6	$\langle [8,8], [0,0], [2,2] \rangle$	$\langle [6,8], [0,1], [2,2] \rangle$	...
7	$\langle [8,8], [1,1], [2,2] \rangle$	$\langle [6,8], [1,2], [2,2] \rangle$	...
8	$\langle [10,10], \bot_I, [2,2] \rangle$	$\langle [8,10], \bot_I, [2,2] \rangle$	...
P_i	Iteration		
	5	6	7
1	$\top_K$	$\top_K$	$\top_K$
2	$\langle \top_I, \top_I, [2,2] \rangle$	$\langle \top_I, \top_I, [2,2] \rangle$	$\langle \top_I, \top_I, [2,2] \rangle$
3	$\langle [10,10], \top_I, [2,2] \rangle$	$\langle [10,10], \top_I, [2,2] \rangle$	$\langle [10,10], \top_I, [2,2] \rangle$
4	$\langle [2,10], [0,4], [2,2] \rangle$	$\langle [0,10], [0,5], [2,2] \rangle$	$\langle [-2,10], [0,5], [2,2] \rangle$
5	$\langle [2,10], [0,4], [2,2] \rangle$	$\langle [0,10], [0,4], [2,2] \rangle$	$\langle [-2,10], [0,4], [2,2] \rangle$
6	$\langle [0,8], [0,4], [2,2] \rangle$	$\langle [-2,8], [0,4], [2,2] \rangle$	$\langle [-4,8], [0,4], [2,2] \rangle$
7	$\langle [0,8], [1,5], [2,2] \rangle$	$\langle [-2,8], [1,5], [2,2] \rangle$	$\langle [-4,8], [1,5], [2,2] \rangle$
8	$\langle [2,10], \bot_I, [2,2] \rangle$	$\langle [0,10], [5,5], [2,2] \rangle$	$\langle [-2,10], [5,5], [2,2] \rangle$

Table 4: Non-terminating solving.

When the abstract semantics are solved using Kleene iteration, termination never occurs. The first few iterations of the solving process are shown in Table 4. By the second iteration, S'_1, S'_2 and S'_3 have fixed. By iteration 6, all state relating to the variables `y` and `z` has reached a fixpoint, but the lower bounds of `x` from P_4 onward are still descending. It may be surprising that the analysis is unable to determine that `x` does not fall below zero in the loop body, as in concrete executions the final loop iteration starts with `x`=2. Since the interval domain is non-relational, the abstraction is unable to capture the fact that once `y` = 5, the value of `x` is fixed inside the loop. Instead, the lower bound of `x` continues to descend within the loop body. In fact, since the abstract lattice of the interval domain is in this case infinite, fixpoint convergence will never be achieved. In other words, this interpretation contains chains of abstract states which are infinite in length. This example serves as justification for why widening is required.

2.4.1 Widening for Intervals

The aim of widening is to accelerate the solving process by skipping over intermediate abstract states in a long ascending chain. This task is performed by the widening operator, $\triangledown$.

Definition 18 (Upward iteration sequence with widening [32])**.** *Consider a partially ordered abstract domain $\langle A, \sqsubseteq_A \rangle$ and a monotonic abstract transfer function $F_A : A \to A$. Then an increasing chain of elements drawn from A is $\boldsymbol{c} = \langle c_1, c_2, \ldots \rangle$, defined by a starting state c_1 and $c_{i+1} = F_A(c_i)$. The chain may be long or even infinite.*

A second chain $\boldsymbol{c'} = \langle c'_1, \ldots, c'_n \rangle$ is defined as:

$$c'_1 = c_1 \qquad c'_{i+1} = \begin{cases} c'_i & \text{if } F_A(c'_i) \sqsubseteq_A c'_i \\ c'_i \triangledown_A F_C(c'_i) & \text{otherwise} \end{cases}$$

where $\triangledown_A : A \times A \to A$ is a widening operator such that:

$$\forall s \in A.\ \forall t \in A.\ s \sqsubseteq_A (s \triangledown_A t) \quad \wedge \quad \forall s \in A.\ \forall t \in A.\ t \sqsubseteq_A (s \triangledown_A t)$$

The latter chain $\boldsymbol{c'}$ is finite, eventually meeting a least-fixpoint c'_n. Further c'_n is guaranteed to be a sound over-approximation of all of the elements of the possibly large or infinite chain $\boldsymbol{c}$, i.e. $\forall c_i \in \boldsymbol{c}.\ c_i \sqsubseteq_A c'_n$.

The classic widening operator for intervals, as suggested by Cousot and Cousot is as follows:

Definition 19 (Widening operator for intervals [32])**.**

$$\begin{aligned} \triangledown_I &: I \to I \\ \bot_I \triangledown_I [c, d] &= [c, d] \\ [a, b] \triangledown_I [c, d] &= [\text{if } c < a \text{ then } -\infty \text{ else } a, \\ &\qquad \text{if } d > b \text{ then } +\infty \text{ else } b] \end{aligned}$$

Given two consecutive abstract iterates in a chain, the interval widening operator returns a new widened interval. Namely, the lower bound is adjusted to $-\infty$ if the lower bound is descending and the upper bound is adjusted to $+\infty$ if the upper bound is ascending.

Example 1 (Widening a chain of intervals)**.** *Let* $\boldsymbol{i} = \langle i_1, i_2, \ldots \rangle$ *be a chain of intervals such that* $i_1 = [0,0]$ *and* $i_{j+1} = i_j \sqcup_I (i_j +_I [1,1])$*. The chain is indeed infinite, but may be over-approximated with widening. Let the chain* $\boldsymbol{i'} = \langle i'_1, i'_2, \ldots \rangle$ *be the chain obtained by widening. The first iterate* $i'_1 = i_1 = [0,0]$*, then the next iterate* $i'_2 = [0,0] \ \nabla_I \ [0,1] = [0,+\infty]$*. A further iteration computes* $i'_3 = i'_2$*, therefore a fixpoint is reached. Further* i'_2 *over-approximates all iterates of* $\boldsymbol{i}$*.*

This simple example shows that it is possible to over-approximate the elements of a long or infinite chain through widening alone. However, when widening is combined with a standard Galois-connection-style interpretation to accelerate solving, usually widening is not used immediately. Instead, only after the chain length exceeds a given length is widening applied. In the previous example, widening could be used after four iterations if a fixpoint had not yet been reached. In this case the abstract iterates would be: $\langle [0,0], [0,1], [0,2], [0,3], [0,+\infty], [0,+\infty] \rangle$. This approach allows short chains to converge before widening comes into effect, in some cases yielding more precise results.

2.4.2 A Widening for $\mathcal{G}$ Programs

Having discussed the widening for a chain of intervals, now a widening can be devised for abstractions of $\mathcal{G}$ programs. Recall that because $\mathcal{G}$ programs use three distinct variables, the abstract domain is a lifting of the interval domain, i.e. I^3. To define a widening for this domain, a widening operator $\nabla_K : K \to K$ is defined, which is a lifting of ∇_I:

Definition 20 (A widening operator for K)**.**

$$\langle x, y, z \rangle \ \nabla_K \ \langle x', y', z' \rangle \triangleq \langle x \ \nabla_I \ x', y \ \nabla_I \ y', z \ \nabla_I \ z' \rangle$$

The correctness of the operator should be apparent (for proof, see Theorem 7 on Page 175).

The solving process is revised to incorporate the widening operator. Widening shall be used if after a predetermined number of solving iterations, an abstract state has not yet converged. One possible widening strategy is shown in Algorithm 1. Each abstract state has an associated counter, which determines if and when widening should come into effect. All abstract states begin at the bottom of

the abstract lattice ($\perp_K$), then Kleene iteration begins as normal. The function APPLY: $K^{|S'|} \times \mathbb{Z} \to K$ performs the application of an abstract semantic equation and returns an updated abstract state. After each iteration of solving, if an abstract state changes (moves up the abstract lattice), then the associated counter is incremented. When the counter associated with an abstract state reaches a predetermined upper limit, w_k, subsequent solving steps for this abstract state are performed via widening.

The revised solving strategy can now be applied to the abstraction of the third example program `example3.g` (Figure 6), which, as discussed earlier, suffers from non-termination. The solving steps when $w_k = 6$ are shown in Table 5. To begin with, the abstract states are the same as for Kleene iteration. Like before, by the second iteration, S'_1, S'_2 and S'_3 have fixed and the remaining states continue to change. By iteration 6, all information alluding to `y` has fixed, but the intervals for `x` still have descending lower bounds in $S'_4, \ldots, S'_8$. However, after iteration 6, the widening counters for $S'_4, \ldots, S'_8$ reach w_k, so in the next iteration widening is used to compute subsequent states for $S'_4, \ldots, S'_8$. Specifically, the unstable lower bounds of the `x` intervals are extrapolated to $-\infty$. The outcome of iteration 8 (not shown) is the same as that of iteration 7, thus the analysis reaches a fixpoint and solving terminates. Notice that the final widened abstract states, $S'_4, \ldots, S'_8$, soundly over-approximate the infinite chains computed by Kleene iteration.

Precision vs. Speed: Choosing a Suitable w_k

In the above example, the application of widening led to the discovery of the least-fixpoint of the abstract semantics, i.e. the information inferred is the best possible abstraction (even in the presence of infinite chains). However, it is important to note that often widening can lead to weaker results. An often overlooked aspect of widening is the process of choosing a suitable number of iterations, w_k, after which widening comes into action. If w_k is too small, then chains are prematurely widened, often yielding a post-fixpoint. But on the other hand, if w_k is too large, then many unnecessary solving steps could ensue. To illustrate these problems, in the previous example suppose a value of w_k is selected which is less than 5, then the intervals corresponding to the variable `y` for S'_4 to S'_8 would have been widened to $[0, +\infty]$ in iteration $w_k + 1$. In this case the result of widening is a post-fixpoint; whilst the post-fixpoint is sound, the information inferred about `y`

Algorithm 1 One possible widening algorithm for $\mathcal{G}$ programs.

```
function HybridSolve(S', w_k)
    S' ← ⟨⊥_K, ..., ⊥_K⟩                    ▷ All abstract states start at bottom.
    counts ← ⟨0, ..., 0⟩                    ▷ One counter per abstract state.
    unstable ← True
    while (unstable) do                     ▷ Loop continues until a fixpoint is achieved.
        unstable ← False

        for (i ← 1; i ≤ |S'|; i ← i + 1) do          ▷ Loop over abstract states.
            old ← S'_i
            S'_i ← Apply(S', i)                      ▷ Apply the i^th abstract equation.
            if (counts_i < w_k) then                 ▷ Normal solving if true.
                if (S'_i ⋤_K old) then               ▷ Increment counter if state changed.
                    counts_i = counts_i + 1
                end if
            else                                     ▷ Accelerate solving for this abstract state.
                S'_i ← WidenIter(old, S'_i)
            end if
            if (S'_i ⋤_K old) then   ▷ A state changed, further iterations required.
                unstable ← True
            end if
        end for

    end while
    return S'
end function

function WidenIter(old, next)
    if (next ⊑_K old) then
        return old
    else
        return old ▽_K next
    end if
end function
```

P_i	$(S'_i, count)$		
	Iter. 1	Iter. 2	...
1	$(\top_K, 1)$	$(\top_K, 1)$	...
2	$(\langle \top_I, \top_I, [2,2] \rangle, 1)$	$(\langle \top_I, \top_I, [2,2] \rangle, 1)$	...
3	$(\langle [10,10], \top_I, [2,2] \rangle, 1)$	$(\langle [10,10], \top_I, [2,2] \rangle, 1)$	...
4	$(\langle [10,10], [0,0], [2,2] \rangle, 1)$	$(\langle [8,10], [0,1], [2,2] \rangle, 2)$	...
5	$(\langle [10,10], [0,0], [2,2] \rangle, 1)$	$(\langle [8,10], [0,1], [2,2] \rangle, 2)$	...
6	$(\langle [8,8], [0,0], [2,2] \rangle, 1)$	$(\langle [6,8], [0,1], [2,2] \rangle, 2)$	...
7	$(\langle [8,8], [1,1], [2,2] \rangle, 1)$	$(\langle [6,8], [1,2], [2,2] \rangle, 2)$	...
8	$(\langle [10,10], \bot_I, [2,2] \rangle, 1)$	$(\langle [8,10], \bot_I, [2,2] \rangle, 2)$	...
P_i	$(S'_i, count)$		
	Iter. 5	Iter. 6	Iter. 7
1	$(\top_K, 0)$	$(\top_K, 1)$	$(\top_K, 0)$
2	$(\langle \top_I, \top_I, [2,2] \rangle, 1)$	$(\langle \top_I, \top_I, [2,2] \rangle, 1)$	$(\langle \top_I, \top_I, [2,2] \rangle, 1)$
3	$(\langle [10,10], \top_I, [2,2] \rangle, 1)$	$(\langle [10,10], \top_I, [2,2] \rangle, 1)$	$(\langle [10,10], \top_I, [2,2] \rangle, 1)$
4	$(\langle [2,10], [0,4], [2,2] \rangle, 5)$	$(\langle [0,10], [0,5], [2,2] \rangle, 6)$	$(\langle [-\infty,10], [0,5], [2,2] \rangle, 6)$
5	$(\langle [2,10], [0,4], [2,2] \rangle, 5)$	$(\langle [0,10], [0,4], [2,2] \rangle, 6)$	$(\langle [-\infty,10], [0,4], [2,2] \rangle, 6)$
6	$(\langle [0,8], [0,4], [2,2] \rangle, 5)$	$(\langle [-2,8], [0,4], [2,2] \rangle, 6)$	$(\langle [-\infty,8], [0,4], [2,2] \rangle, 6)$
7	$(\langle [0,8], [1,5], [2,2] \rangle, 5)$	$(\langle [-2,8], [1,5], [2,2] \rangle, 6)$	$(\langle [-\infty,8], [1,5], [2,2] \rangle, 6)$
8	$(\langle [2,10], \bot_I, [2,2] \rangle, 5)$	$(\langle [0,10], [5,5], [2,2] \rangle, 6)$	$(\langle [-\infty,10], [5,5], [2,2] \rangle, 6)$

Table 5: Previously non-convergent analysis reaching a fixpoint via widening ($w_k = 6$). The 8^{th} iteration is omitted since it is the same at the 7^{th} iteration.

is weak. Now suppose a w_k is chosen which is greater than 6. Then since the solving of S'_4 to S'_8 encounters infinite chains, and because the outcome of solving with any $w_k \geq 6$ is the same, solving would perform unnecessary iterations, thus slowing the solving process. The desired w_k value varies on a per-program basis, depending upon the length of the chains therein. Since it is not easy to know how long ascending chains could be, a suitable w_k is sometimes determined manually.

2.5 Chapter Summary

In this chapter, abstract interpretation was formally introduced. A series of interpretations were designed to infer the signedness of variables at each point in a $\mathcal{G}$ program. The first example abstracted numeric values as a set of signs. It was then shown that the sign abstraction domain could generate false positives due to the coarse nature of the domain. To rectify this, the abstract domain was revised to incorporate intervals. It was shown that by this new abstraction, unnecessary false positives can be avoided. Because Kleene iteration could encounter long or infinite ascending chains, a simple fixpoint acceleration technique that used

widening was presented. Using this widening on top of the existing abstraction, the analysis yields results even in the presence of infinite ascending chains. It was then pointed out that, whilst widening is often necessary, the quality of the results greatly varies depending upon the choice of w_k, which varies on a per-program basis and may not be known up-front.

Chapter 3

Ranges and Sets for Boolean Formulae

Although the fundamental ideas in abstract interpretation were laid down over thirty years ago [31], abstract interpretation has only entered its industrialisation phase comparatively recently [33]. This new phase is not only characterised by an increased focus on tooling and systems building, but also by work on designing and implementing new abstract domains. For example, domains for improved scalability i.e. the class of weakly-relational domains [86, 111], and domains that better match the structure of real programs i.e. symbolic decision trees that correlate the relationship between status flags and numeric variables [14]. This chapter offers a new means by which to automatically infer ranges and sets of values directly from a Boolean formula. The method naturally takes into account bitwise details and since the method is not underpinned by a decision tree, it sidesteps the need for a canonical form and variable ordering.

3.1 Motivation

Blanchet et al. [14] illustrate the need for mixed symbolic and numeric abstractions with the following pseudo-code:

```
B := (X = 0);
if (!B) Y := 1/X;
```

Listing 3.1: The relationship between `B` and `X` must be tracked.

This code is correct in the sense that it does not give a division by zero error if `X = 0`, but to deduce this it is necessary to track the relationship between `B` and `X`. The authors state:

> "In order to deal precisely with those examples, we implemented a simple relational domain consisting in a decision tree with leaf an arithmetic abstract domain. The decision trees are reduced by ordering Boolean variables and by performing some opportunistic sharing of sub-trees. The only problem with this approach is that the size of a decision tree can be exponential in the number of Boolean variables, and the code contains thousands of global ones"

The problem of relating Booleans to numeric values is particularly acute in binary reverse engineering and verification, though in these cases the Booleans are CPU status flags. Since the runtime values of the status flags determine which branches are taken, the flags must be taken into account when attempting to recover the control flow graph (CFG) of a binary. This is a task which has recently been explored in the literature [71]. The problem here is the so-called "chicken and egg" problem [38]. To derive the CFG it is necessary to collect register values to decide possible indirect jump targets. However, in order to collect register values, the CFG is required. Kinder resolves this cyclic dependency by applying a data-flow analysis in conjunction with a CFG that itself grows monotonically [71]. He illustrates these ideas with an idealised assembler language. In practice the problem is considerably harder to solve, partly because of the problem of relating status flags to ranges. To illustrate, consider the following x86 assembler code for a switch table:

```
mov eax, [ebp-0x8]   ; eax := *(ebp - 8)
sub eax, 0x2         ; eax := eax - 2
cmp eax, 0x5         ; cf := (0 =< eax < 5)
                     ; zf := (eax = 5)
ja  0xd8             ; jump to 0xd8 if cf = 0 and zf = 0
jmp [0x8048a0c + eax*4]
```

Listing 3.2: Use of an indirect jump in a jump table.

To determine the CFG it is necessary to ascertain that `eax` $\in [0, 5]$ when the indirect jump is reached. This range information, and the table itself, permits

the CFG to be over-approximated. However, inferring the range on `eax` itself requires careful reasoning about the value of the carry (`cf`) and zero (`zf`) flags — a problem which is analogous to that addressed by Blanchet et al. [14].

In this chapter it is shown that an over-approximate range or set of the satisfying models of a Boolean formula can be automatically derived. This represents a significant step towards the automated abstraction of binary code, where bit-wise details, such as status flags, must be carefully considered. The approach is attractive in that a range analysis could be formulated directly from a concrete description of a binary program without the worry of a canonical representation (as with a decision tree) and without the need to define or synthesise abstract transfer functions. The following contributions are presented in this chapter:

- It is shown that an over-approximate range of Boolean satisfiability (SAT) models can be efficiently extracted from a Boolean formula by repeated calls to a SAT solver.
- It is shown that the range can be incrementally refined into successively more precise over-approximate sets of SAT models. The technique relies on computing over- and under-approximate sub-ranges of models. Eventually, the set converges onto the precise set of SAT models, however, the algorithm may be terminated early and still give a sound over-approximation.
- It is shown that the techniques dovetail with incremental SAT and experimental results are presented which suggest that the techniques are viable.

3.2 Range Abstraction

Suppose that the goal is to compute a range of values for a bit vector $\boldsymbol{x}$ which is constrained by a Boolean formula f, where the variables over which f is defined are exactly the variables of $\boldsymbol{x}$[1]. To compute the range, the maximum and minimum values of $\boldsymbol{x}$ need to be determined. In principle, this can be achieved by applying a SAT solver in conjunction with blocking clauses.

[1] In the next chapter an extension is proposed which lifts this restriction. This allows one to compute ranges for a subset of the variables over which f is defined.

Definition 21 (Blocking Clause)**.** *Given a model* $\boldsymbol{x} = \langle x_0, \ldots, x_{n-1} \rangle$ *of a formula* f *under which the propositional variables of* $\boldsymbol{x}$ *are bound to the truth values* $b_0, \ldots, b_{n-1} \in \{0, 1\}$*, a blocking clause is defined as:*

$$\bigvee_{i=0}^{n-1} y_i \quad \text{where } y_i = \begin{cases} x_i & \text{if } b_i = 0 \\ \neg x_i & \text{otherwise} \end{cases}$$

Thus, by adding a blocking clause to a SAT instance, any subsequent solution differs from the truth values $b_0, \ldots, b_{n-1}$ on at least one b_i. By repeating this technique until the SAT instance is unsatisfiable, it is possible to enumerate all solutions, hence all values that $\boldsymbol{x}$ can assume. From the set of satisfying models, the maximum and minimum can then be extracted. The limitation of this technique is that the number of invocations of the solver is linear in the number of solutions (which may be large) and moreover, the size of the SAT instance grows as blocking clauses are added. Instead, a method is proposed which finds the minimum and maximum models directly.

3.2.1 Computing the Minimum

Algorithm 2 presents an algorithm for computing a minimum model that requires only n calls to a SAT solver. If the Boolean flag $s = 1$ then the bit vector $\boldsymbol{x} = \langle x_0, \ldots, x_{n-1} \rangle$ is interpreted as a signed integer, represented using two's complement, where x_{n-1} is the sign bit and x_0 is the least significant bit. If $s = 0$ then the bit vector $\boldsymbol{x}$ is interpreted as an unsigned integer. The function MINIMUM returns the minimum value expressed as binary vector $\boldsymbol{k} \in \{0, 1\}^n$.

Consider first the unsigned case that is handled in the else branch of the loop body. The bits of $\boldsymbol{k}$ are computed in reverse order: the high bit first and the low bit last. On each iteration of the loop, f is tested to see whether it possesses a solution in which the bit $\boldsymbol{x}[n - |\boldsymbol{k}| - 1]$ is assigned to 0. If so, then the minimum value of $\boldsymbol{x}$ has a 0 in this bit position, hence 0 is prepended to $\boldsymbol{k}$. If not, then every solution of $\boldsymbol{x}$ (including the minimum) has a 1 in this position, hence 1 is prepended to $\boldsymbol{k}$. Note that as the loop progresses, f is itself modified so as to clamp the high bits of $\boldsymbol{x}$ to the high bits of the partially computed minimum $\boldsymbol{k}$.

The signed case proceeds analogously except for the very first iteration which computes the sign bit of the minimum. If f has a solution with $\boldsymbol{x}[n-1]$ assigned

to 1, then the minimum is negative, which is reflected by setting $\boldsymbol{k}$ to the unary vector $\langle 1 \rangle$, so as to record the sign of the minimum. Otherwise, the minimum is non-negative, hence $\boldsymbol{k}$ is set to $\langle 0 \rangle$. Setting s to 0 ensures that all subsequent loop iterations deduce the lower bits of $\boldsymbol{k}$ in the same manner as in the unsigned case.

Algorithm 2 Computing the minimum value of the bit-vector $\boldsymbol{x}$

```
1:  function MINIMUM(f, x, s)
2:      k ← ⟨⟩
3:      n ← |x|
4:      while |k| < n do
5:          if s then
6:              if SAT(f ∧ x[n − 1]) then
7:                  f ← f ∧ x[n − 1]
8:                  k ← ⟨1⟩
9:              else
10:                 f ← f ∧ ¬x[n − 1]
11:                 k ← ⟨0⟩
12:             end if
13:             s ← 0
14:         else
15:             if SAT(f ∧ ¬x[n − |k| − 1]) then
16:                 f ← f ∧ ¬x[n − |k| − 1]
17:                 k ← ⟨0⟩ :: k
18:             else
19:                 f ← f ∧ x[n − |k| − 1]
20:                 k ← ⟨1⟩ :: k
21:             end if
22:         end if
23:     end while
24:     return k
25: end function
```

3.2.2 Computing the Maximum

Computing a maximum model is analogous to computing a minimum model. To compute the maximum, take Algorithm 2 and make the following amendments:

- Invert the polarities of $\boldsymbol{x}[n-1]$ on lines 6, 7 and 10.
- Invert the polarities of $\boldsymbol{x}[n-|\boldsymbol{k}|-1]$ on lines 15, 16 and 19.

- Invert the truth values prepended onto $\boldsymbol{k}$ on lines 8, 11, 17 and 20.
- Rename the function to MAXIMUM.

3.3 Set Abstraction

Switch tables can in general be hierarchical structures in which a series of tests direct the control into smaller tables that handle indices that are close to one another. Range abstraction alone cannot accurately model such sets of indices and addresses and therefore it is necessary to instead employ set abstraction. Since an n-ary bit-vector $\boldsymbol{x}$ can assume up to 2^n distinct values, the set itself can be large, at least in the pathological case. Therefore, for cautionary reasons, abstraction is deployed to compute an over-approximation (superset) that keeps the size of the set manageable. As a by-product of this construction, it is also able to compute under-approximations (subsets) of the set of values that bit-vector can assume when constrained by a given Boolean function f.

Algorithm 3 presents the function SET for computing a set abstraction for $\boldsymbol{x}$. The Boolean argument s indicates whether $\boldsymbol{x}$ has a signed interpretation. The integer argument c bounds the number of iterations of the loop. Moreover, if c is non-negative and odd then an over-approximation is found, whereas if c is non-negative and even then an under-approximation is derived. If c is negative then the algorithm will run to completion and exactly characterise the values of $\boldsymbol{x}$.

The set S, which starts empty, is refined on each iteration of the loop. The vectors $\boldsymbol{l}$ and $\boldsymbol{u}$ are used to further constrain f; these bounds increase and decrease respectively, until either: a) the c threshold triggers premature termination, or b) the bounds $\boldsymbol{l}$ and $\boldsymbol{u}$ cross and an exact description of the set is found. The special treatment of the most significant bit of $\boldsymbol{l}$ and $\boldsymbol{u}$ on lines 7 and 8 stem from the two's complement representation for the case where $s = 1$. The function VALUE is used to interpret a bit vector as a numeric value:

Definition 22 (Integer Interpretation of a bit-vector $\boldsymbol{b}$)**.**

$$\text{VALUE}(\boldsymbol{b}, s) = (1 - 2s)2^{n-1}\boldsymbol{b}[n-1] + \sum_{i=0}^{n-2} 2^i \boldsymbol{b}[i]$$

Algorithm 3 Computing a set abstraction for the bit-vector $\boldsymbol{x}$

```
1: function SET(f, x, s)
2:     return SET(f, x, s, −1)
3: end function
4: function SET(f, x, s, c)
5:     S ← ∅
6:     p ← 1
7:     l ← ⟨0, ..., 0, s⟩
8:     u ← ⟨1, ..., 1, ¬s⟩
9:     while VALUE(l, s) <VALUE(u, s) ∧ c ≠ 0 do
10:        l ← MINIMUM(f ∧ (l ≤_s x), x, s)
11:        u ← MAXIMUM(f ∧ (x ≤_s u), x, s)
12:        if p then
13:            S ← S ∪ [VALUE(l, s), VALUE(u, s)]
14:        else
15:            S ← S \ [VALUE(l, s), VALUE(u, s)]
16:        end if
17:        p ← ¬p
18:        f ← ¬f
19:        c ← c − 1
20:    end while
21:    return S
22: end function
```

Each iteration of the loop determines a new minimum ($\boldsymbol{l}$) and maximum ($\boldsymbol{u}$) solution to a SAT instance that is obtained by augmenting either f or $\neg f$ with a formula that imposes a less-than-or-equals relation on $\boldsymbol{x}$. This additional formula prevents the previously found ranges from being rediscovered. Although incremental SAT can be used within the functions MINIMUM and MAXIMUM, the different less-than-or-equal-to relations impede incremental SAT being applied in the function SET.

The less-than-or-equal-to relations are defined as follows:

Definition 23 (Less-than-or-equal-to relations for bit-vectors)**.** *For the case of unsigned vectors:*

$$\begin{aligned}\langle\rangle \leq_0 \langle\rangle &= \text{TRUE}\\ \langle \boldsymbol{x}[0]\ldots\boldsymbol{x}[n-1]\rangle \leq_0 \langle \boldsymbol{y}[0]\ldots\boldsymbol{y}[n-1]\rangle &= \\ &(\neg\boldsymbol{x}[n-1]\wedge\boldsymbol{y}[n-1])\vee((\boldsymbol{x}[n-1]\Leftrightarrow\boldsymbol{y}[n-1])\wedge\\ &(\langle\boldsymbol{x}[0]\ldots\boldsymbol{x}[n-2]\rangle \leq_0 \langle\boldsymbol{y}[0]\ldots\boldsymbol{y}[n-2]\rangle))\end{aligned}$$

For the case of signed vectors:

$$\begin{aligned}\langle\rangle \leq_1 \langle\rangle &= \text{TRUE}\\ \langle \boldsymbol{x}[0]\ldots\boldsymbol{x}[n-1]\rangle \leq_1 \langle \boldsymbol{y}[0]\ldots\boldsymbol{y}[n-1]\rangle &= \\ &(\boldsymbol{x}[n-1]\wedge\neg\boldsymbol{y}[n-1])\vee((\boldsymbol{x}[n-1]\Leftrightarrow\boldsymbol{y}[n-1])\wedge\\ &(\langle\boldsymbol{x}[0]\ldots\boldsymbol{x}[n-2]\rangle \leq_0 \langle\boldsymbol{y}[0]\ldots\boldsymbol{y}[n-2]\rangle))\end{aligned}$$

On line 10 of the set abstraction algorithm, $\boldsymbol{l}$ is a vector of truth values, hence the formula $(\boldsymbol{l} \leq_s \boldsymbol{x})$ can be partially evaluated to simplify the comparison (and likewise on line 11 for $(\boldsymbol{x} \leq_s \boldsymbol{u})$). For example consider two 4-bit vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ and suppose $\boldsymbol{y} = \langle 1, 0, 1, 1\rangle$. Then $\boldsymbol{x} \leq_0 \boldsymbol{y}$ can be reduced to the formula $\neg\boldsymbol{x}[3]\vee(\boldsymbol{x}[3]\wedge\neg\boldsymbol{x}[2]\vee(\boldsymbol{x}[2]\wedge\neg\boldsymbol{x}[1]))$.

Example 2 (Set Abstraction)**.** *Suppose a 4-bit vector $\boldsymbol{x}$ is constrained by a formula so that it can only draw an unsigned value from the set:*

$$\{1, 2, 3, 5, 6, 8, 9, 12, 13, 15\}$$

Table 6 shows how set abstraction converges onto this set by finding alternating over- and under-approximations. Each entry in the table corresponds one iteration of the algorithm immediately after S is refined. i.e. prior to the execution of line 17 in Algorithm 3.

Figure 7 shows convergence diagrammatically. The numeric values which may appear in S can be visualised as a bar separated into 16 cells. Each cell corresponds to one numeric value. An open cell represents a numeric value which is a member of the set S, whereas a shaded cell represents a numeric value which is absent in the set S. The range inferred at each iteration is indicated by an arrowed line

underneath each bar.

c	p	$\boldsymbol{l}$	$\boldsymbol{u}$	VALUE $(\boldsymbol{l}, 0)$	VALUE $(\boldsymbol{u}, 0)$	S
-1	1	$\langle 1,0,0,0\rangle$	$\langle 1,1,1,1\rangle$	1	15	$\{1 \ldots 15\}$
-2	0	$\langle 0,0,1,0\rangle$	$\langle 0,1,1,1\rangle$	4	14	$\{1,2,3,15\}$
-3	1	$\langle 1,0,1,0\rangle$	$\langle 1,0,1,1\rangle$	5	13	$\{1,2,3,5,6,7,8,9,10,11,12,13,15\}$
-4	0	$\langle 1,1,1,0\rangle$	$\langle 1,1,0,1\rangle$	7	11	$\{1,2,3,5,6,12,13,15\}$
-5	1	$\langle 0,0,0,1\rangle$	$\langle 1,0,0,1\rangle$	8	9	$\{1,2,3,5,6,8,9,12,13,15\}$
-6	0	$\langle 0,1,0,1\rangle$	$\langle 1,0,0,1\rangle$	10	7	$\{1,2,3,5,6,8,9,12,13,15\}$ ✓

Table 6: Example showing how S converges.

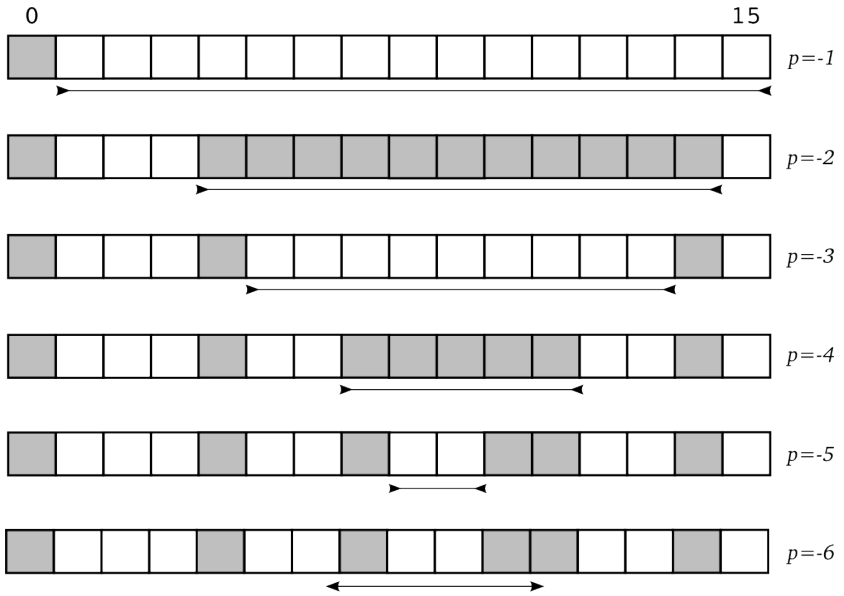


Figure 7: Convergence of set abstraction shown diagramatically.

3.4 Experimental Results

The MINIMUM and MAXIMUM functions at the heart of the range and set abstraction algorithms amount to solving a series of related SAT problems. This suggests the application of incremental SAT. Incremental SAT is the problem of solving a series SAT instances $\{\wedge F_1, \ldots, \wedge F_k\}$ defined over a common set of variables. Each F_i is a set of clauses, and the consecutive instances are related according to

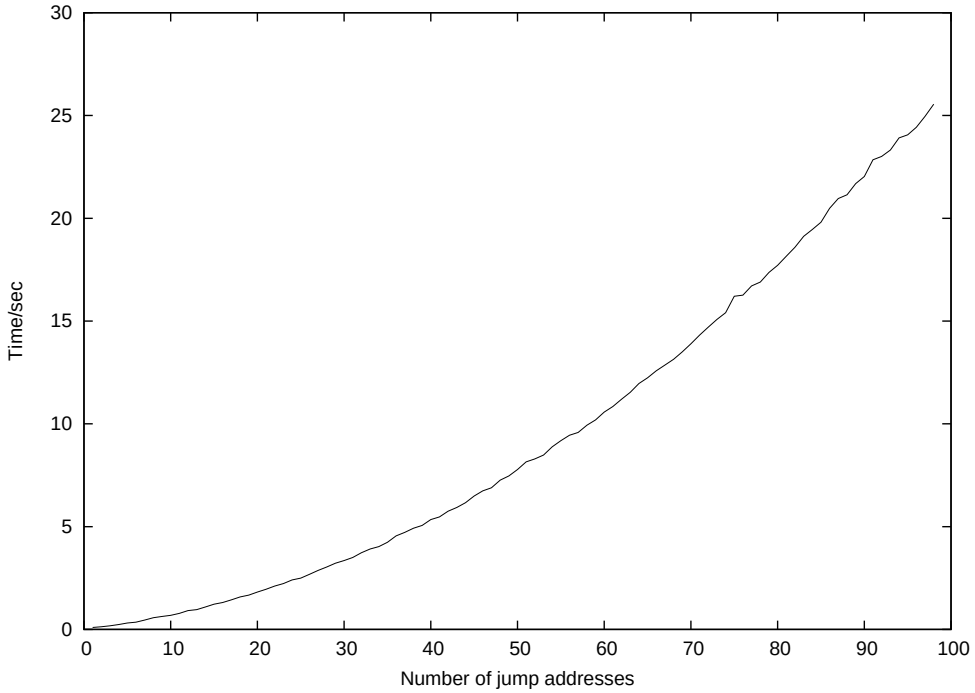


Figure 8: Satisfiable jump addresses vs. time.

$F_{i+1} = (F_i \setminus G_i) \cup H_i$ where G_i and H_i are sets of clauses that are respectively rescinded and added [65, 117]. Incremental SAT is most useful when $|G_i| \ll |F_i|$ and $|H_i| \ll |F_i|$ since then solving $\wedge F_{i+1}$ can take advantage of the clauses learnt when solving $\wedge F_i$, and possibly earlier instances.

In Algorithm 2, unit clauses of $\boldsymbol{x}[n-1]$, $\neg\boldsymbol{x}[n-1]$, $\neg\boldsymbol{x}[n-|\boldsymbol{k}|-1]$ and $\boldsymbol{x}[n-|\boldsymbol{k}|-1]$ are added to f at lines 6, 10, 15 and 19 respectively. Conversely, the unit clauses $\boldsymbol{x}[n-1]$ and $\neg\boldsymbol{x}[n-|\boldsymbol{k}|-1]$ are rescinded when the satisfiability questions posed at lines 6 and 15 are found to have a negative answer. (Note that these removal operations are not reflected in the algorithm but are applied in the else blocks that commence at lines 10 and 19.) Thus whenever a new SAT instance is encountered, $|G_i| \leq 1$ and $|H_i| = 1$, which suggests that the algorithm is ideal for incremental SAT. Moreover, only unit clauses are added and removed, and this specialised form of incremental SAT is supported with the so-called unit assumptions of the popular MiniSat solver [44][2].

[2]Actually, unit assumptions are automatically withdrawn by MiniSat after checking satisfiability and thus those unit assumptions which need to be preserved have to be re-added as immutable clauses.

Of course, incremental SAT is only of value if it actually improves performance. To investigate this, a series of Boolean formulae were generated whose models are representative of the feasible jump targets of multi-level switch tables. Each formula constrains a 64-bit vector $\boldsymbol{x}$ to model up to 98 different branch addresses. To investigate scalability, the set abstraction algorithm was applied to switch tables of increasing size where the branch addresses were non-consecutive (this is because typically branch addresses are 32 or 64 bits in length, meaning that indirect jump targets occur spaced 4 or 8 bytes apart). The set abstraction algorithm was not terminated prematurely, so as to exercise it fully. The graph shown in Figure 8 suggests that the time to compute the precise set abstraction grows smoothly with the size of the switch table. These timings were generated on a 3GHz x86 machine with 4GB of RAM running Linux.

Interestingly, replacing incremental SAT with a series of independent calls to the solver gave a slowdown of two orders of magnitude. Thus incremental SAT both improves performance and avoids the need to invoke the solver 64 times to compute the minimum/maximum of a 64-bit integer. Figure 9 illustrates the variability in the time required to compute the minima and maxima at different iterations of the set abstraction algorithm. Importantly, the time to compute the minima and maxima do not increase in the latter iterations of algorithm, which one might expect as the solution range diminishes.

It should be emphasised that the efficiency of technique can doubtless be improved by standard tactics such as more refined CNF conversion [92]. Furthermore, by changing the search strategy used in the SAT solver, it may be possible to directly derive the maximum (or minimum) model without deploying n separate (albeit incremental) calls to the solver. Although this would require fundamental changes to the SAT solver itself, the speed-up could be very considerable.

3.5 Chapter Summary

The work presented in this chapter has shown that, given a Boolean formula, ranges and sets of satisfying models can be inferred in an efficient manner. This represents a significant step towards the automated abstraction of binary code, where bit-level details must be considered to obtain suitably precise information. For example, the method could be used to compute a range (or set) of jump targets

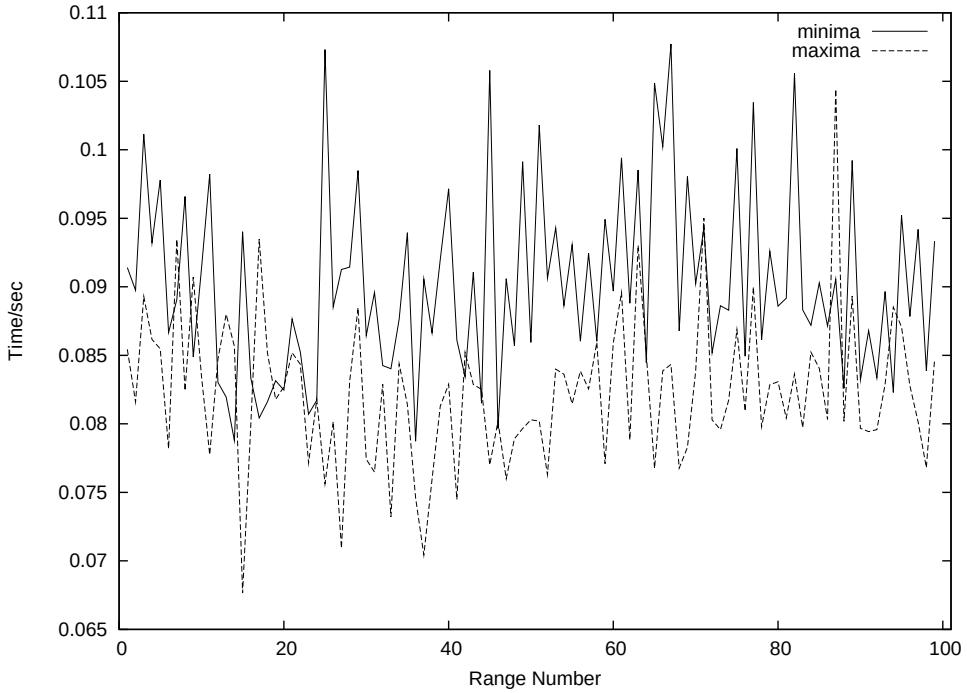


Figure 9: Time taken to solve minima and maxima of each range.

for a switch table, where the status register flags must be taken into account. Potential jump targets are invaluable for the task of CFG recovery, which itself is important for underpinning other analyses. That such information could be derived automatically is encouraging, particularly as it cannot be ensured that range checks and multi-way branches take a regular recognisable structure.

The range and set abstraction methods presented here, however, only provide part of the necessary infrastructure for an automated range analysis. In this setting, the semantics of blocks would be expressed as a Boolean formula corresponding to a mapping $block_i : \mathbb{B}^n \times \ldots \times \mathbb{B}^n \rightarrow \mathbb{B}^n \times \ldots \times \mathbb{B}^n$. The inputs to the function represent n-bit register values prior to execution of the block, and the outputs of the function represent the mutated n-bit register values after execution of the block. To infer a range or set of values for one of the output registers, it is not the SAT models themselves that should be abstracted, but rather the values of a sub-vector. As shall be discussed in the next chapter, to achieve this, a quantified Boolean formula is required.

Chapter 4

Quantifier Elimination with Optimisation

In the previous chapter an algorithm was presented which abstracts the solutions of a Boolean formula as a range. The range can optionally be refined to an over-approximate set. This chapter discusses the application of these algorithms to Boolean encodings of CPU operations so as to recover control flow information from a binary program. As shall become apparent, to achieve this it is necessary to deploy quantifier elimination.

4.1 Motivation

Consider the following sequence of x86 instructions:

```
shl eax, 2
jmp [eax]
```

Listing 4.1: Example usage of an indirect jump.

The first instruction shifts the 32-bit `eax` register left twice, therefore multiplying by four. The second instruction transfers control (indirectly) to the address held in `eax`. Traditionally, the analysis of such code has been troublesome, since indirect jumps make determining the control flow graph (a common preliminary for many analyses) difficult. This stems from the fact that in order to know the targets of an indirect jump, the values that the operand register may assume must be known.

In the case of the above example, it is necessary to know the values that `eax` may assume prior to the indirect jump. However, to infer register values, typically the control flow graph itself is required. This predicament is widely acknowledged as the "chicken and egg problem" and was discussed in Chapter 1 of this thesis.

The work discussed in this chapter aims to overcome the chicken and egg problem through the application of the range and set abstraction algorithms shown in the previous chapter. Since CPU operations can easily be expressed as propositional formulae [16], it should be possible to incrementally resolve indirect jump targets through the application of range and set abstraction. Indeed it is possible, however these algorithms cannot be applied directly; to get the desired result the Boolean formulae must include universal quantifiers, and because range and set abstraction utilise a SAT solver, these quantifiers must be eliminated. This led to the main outcome of this chapter – a new quantifier elimination technique. To appreciate the necessity for quantifier elimination, let us first discuss what happens if range and set abstraction are applied naively.

4.1.1 Applying Range and Set Abstraction Naively

To demonstrate why the naive application of range and set abstraction is insufficient, let us apply these algorithms to the program snippet shown in Listing 4.1. Under this scenario the targets of the indirect jump can be known by inferring the values that `eax` may assume after the left shift operation. As a starting point, the program is encoded as a Boolean formula.

The shift operation can be thought of as a mapping $shl_{eax,2} : \mathbb{B}^{32} \rightarrow \mathbb{B}^{32}$, i.e. a mapping from a bit-vector representing `eax` prior to shifting ($\boldsymbol{eax}$) to a bit-vector representing the mutated `eax` register after shifting ($\boldsymbol{eax'}$). This relationship can be captured with a Boolean formula ξ as follows:

$$\xi = \neg eax'_0 \wedge \neg eax'_1 \wedge (eax'_2 \Leftrightarrow eax_0) \wedge \ldots \wedge (eax'_{31} \Leftrightarrow eax_{29})$$

Following the approach of [16], the semantics of entire blocks or functions can be encoded in this way. For simplicity, the example considers a single operation. Also for simplicity, ξ does not model the registers that are unaffected by the operation (`ebx`, `ecx`, ...) and `eax` is assumed to be unsigned.

Now suppose that the set abstraction algorithm described in the previous

chapter (Algorithm 3 on Page 54) is used to compute a set of possible jump targets characterised by $\boldsymbol{eax'}$. Let $r_i = [\textsc{Value}(\boldsymbol{l_i}, 0), \textsc{Value}(\boldsymbol{u_i}, 0)]$ be the range computed upon iteration i of the set abstraction algorithm. One would be forgiven for proceeding in the following manner:

- Convert ξ to conjunctive normal form (CNF) via the Tseitin transform [115] to derive ξ' where $\xi \equiv \exists T.\ \xi'$ and each $t_i \in T$ is an existentially quantified Tseitin (witness) variable. Crucially, ξ' is equisatisfiable to ξ. Similarly, a CNF formula ξ'' can be constructed which is equisatisfiable to $\neg\xi$ (this can be accomplished by taking ξ' and negating the topmost Tseitin witness variable so that $\neg\xi \equiv \exists T.\ \xi''$).

- Perform the first iteration of set abstraction, thus finding a range r_1, where:

$$\boldsymbol{l_1} = \textsc{Minimum}(\xi' \wedge \langle 0, \ldots, 0\rangle \leq_0 \boldsymbol{eax'}, \boldsymbol{eax'}, 0)$$
$$\boldsymbol{u_1} = \textsc{Maximum}(\xi' \wedge \boldsymbol{eax'} \leq_0 \langle 1, \ldots, 1\rangle, \boldsymbol{eax'}, 0)$$

  This is an odd iteration of set abstraction, so ξ' is used. An over-approximate range of values is computed which corresponds to values of $\boldsymbol{eax'}$ that can be obtained by shifting `eax` left twice. Since the input vector $\boldsymbol{eax}$ is unconstrained, the computed interval is $r_1 = [0, 2^{32} - 4]$.

- Next, perform the second iteration to refine the abstraction with a range r_2:

$$\boldsymbol{l_2} = \textsc{Minimum}(\xi'' \wedge \boldsymbol{l_1} \leq_0 \boldsymbol{eax'}, \boldsymbol{eax'}, 0)$$
$$\boldsymbol{u_2} = \textsc{Maximum}(\xi'' \wedge \boldsymbol{eax'} \leq_0 \boldsymbol{u_1}, \boldsymbol{eax'}, 0)$$

  where $\boldsymbol{l_1}$ and $\boldsymbol{u_1}$ correspond to the lower and upper bounds found in the previous iteration (i.e. 0 and $2^{32} - 4$). Since this is an even iteration of set abstraction, ξ'' is used. An over-approximate range of values (as a subset of $[0, 2^{32} - 4]$) is computed. The range encloses the values of $\boldsymbol{eax'}$ that are non-solutions to shifting `eax` left twice. The range computed is $r_2 = [0, 2^{32} - 4]$. Notice that $r_2 = r_1$.

- Perform the third iteration to further refine the abstraction with a range r_3:

$$\boldsymbol{l_3} = \textsc{Minimum}(\xi' \wedge \boldsymbol{l_2} \leq_0 \boldsymbol{eax'}, \boldsymbol{eax}, 0)$$
$$\boldsymbol{u_3} = \textsc{Maximum}(\xi' \wedge \boldsymbol{eax'} \leq_0 \boldsymbol{u_2}, \boldsymbol{eax}, 0)$$

  The interval computed is $r_3 = [0, 2^{32} - 4]$. Notice that $r_3 = r_2 = r_1$.

- Etcetera. Termination never occurs.

4.1.2 The Need for Quantifier Elimination

One may be surprised to see that set abstraction, when applied to the example in the previous subsection, does not terminate. The set abstraction will continually find the range $[0, 2^{32} - 4]$. Admittedly, at first this was an unexpected outcome, especially since no experiment in the previous chapter exhibited this behaviour.

The problem arises from the fact that some values of $\boldsymbol{eax'}$ are solutions to both ξ and $\neg\xi$, albeit under different assignments to the input vector $\boldsymbol{eax}$. For example, $\textsc{Value}(\boldsymbol{eax}, 0) = 0 \wedge \textsc{Value}(\boldsymbol{eax'}, 0) = 0$ is a satisfying assignment to ξ (and therefore also a satisfying assignment to ξ'). This is because by shifting $\langle 0, \ldots, 0\rangle$ left twice, the outcome would be $\langle 0, \ldots, 0\rangle$. Now consider the assignment $\textsc{Value}(\boldsymbol{eax}, 0) = 1 \wedge \textsc{Value}(\boldsymbol{eax'}, 0) = 0$. This is a satisfying assignment of $\neg\xi$ (and therefore also a satisfying assignment to ξ''), meaning that by shifting $\langle 0, \ldots, 0, 1\rangle$ left twice, the outcome can not be $\langle 0, \ldots, 0\rangle$. So the partial assignment $\textsc{Value}(\boldsymbol{eax'}, 0) = 0$ can either satisfy or falsify ξ depending upon the assignments to the variables of $\boldsymbol{eax}$. In turn, the bounds found for $\boldsymbol{eax'}$ in consecutive iterations of set abstraction may be the same and this compromises termination. In other words, set abstraction cannot be applied directly to compute ranges and sets for sub-vectors of the variables over which a formula is defined. By contrast, the experiments presented in the previous chapter computed ranges and sets for entire SAT models. This meant that each discovered interval bound corresponded to a complete assignment to the variables of the formula, thus each bound either satisfied or falsified the formula, but never both.

Naturally, the question that follows is, could the algorithm be refined to lift this restriction? The answer is yes, it is possible, by adjusting the Boolean formula in even iterations of set abstraction. On even iterations, interval bounds should

be computed that correspond to minimum and maximum values of $\boldsymbol{eax'}$ such that the bounds satisfy $\neg\xi$ for all assignments to the input vector $\boldsymbol{eax}$. Thus for even iterations of set abstraction, the input vector $\boldsymbol{eax}$ should be universally quantified.

To illustrate, the second iteration of the above example should compute a range $r_2 = [\textsc{Value}(\boldsymbol{l_2}, 0), \textsc{Value}(\boldsymbol{u_2}, 0)]$ such that:

$$\boldsymbol{l_2} = \textsc{Minimum}(\forall eax_{31}.\ \ldots \forall eax_0.\ \exists T.\ \xi'' \wedge \boldsymbol{l_1} \leq_0 \boldsymbol{eax'}, \boldsymbol{eax'}, 0)$$
$$\boldsymbol{u_2} = \textsc{Maximum}(\forall eax_{31}.\ \ldots \forall eax_0.\ \exists T.\ \xi'' \wedge \boldsymbol{eax'} \leq_0 \boldsymbol{u_1}, \boldsymbol{eax'}, 0)$$

where T is again the set of Tseitin variables. This poses a problem though. The functions Minimum and Maximum call the SAT solver to obtain a minimum and maximum model respectively, yet a quantified formula of the form $\forall I.\ \exists T.\ f$ cannot be passed directly to the solver. If the universal quantifiers are eliminated, then the formula can be passed to the SAT solver, however, the innermost existential quantifiers must be eliminated first.

There are standard techniques available to perform quantifier elimination (QE) for Boolean formulae, but unfortunately there are aspects of these methods which hinder their effectiveness. For example, expansion-based quantifier elimination [74] could be used, however, by these methods the size of the formula can grow significantly, particularly when many variables are to be eliminated.

This chapter proposes an entirely new approach to quantifier elimination for Boolean formulae. Specifically, the contributions of this chapter are as follows:

- An algorithm is presented that computes the prime implicates of a CNF formula through the repeated generation of Chvátal cuts. The process is driven by mixed-integer linear programming (MILP). Once the prime implicates are found, it is straightforward to eliminate both universal and existential quantifiers.

- Through the use of a cost function and blocking constraints, it is shown that the method can avoid finding redundant implicates, thus reducing the number of calls to the MILP solver.

- Experimental results are presented which show that the new algorithm finds the prime implicates in much fewer operations than by traditional binary

resolution.

The remainder of this chapter will proceed as follows. Section 4.2 shows that quantifier elimination is straightforward given the prime implicates and that binary resolution could, in principle, be used to find the prime implicates. Section 4.3 describes Chvátal cuts and their relationship to binary resolution. Section 4.4 presents the new algorithm which aims to automatically find the prime implicates in as few Chvátal cuts as possible using mixed-integer linear programming. In Section 4.6, experimental results are presented that evaluate the new approach. Finally, Section 4.7 draws the chapter to a conclusion and suggests some possible improvements to the algorithm.

4.2 Quantifier Elimination by Prime Implicates

This chapter suggests that one way to to eliminate quantifiers is via the prime implicates.

Definition 24 (Implicate [36])**.** *Given a formula f, a clause C is an implicate of f if f implies C is valid.*

Definition 25 (Prime Implicate [36])**.** *Given a formula f, an implicate C is prime if no other implicate of f implies C.*

Every propositional formula can be reduced to an equisatisfiable conjunction of irredundant prime implicates, denoted here as $\bigwedge F_p$, where F_p is the set of prime implicates. Note that $\bigwedge F_p$ is a CNF formula. The prime implicates lend themselves well to the problem of quantifier elimination. To appreciate this, observe that:

- Given the conjunction of prime implicates $\bigwedge F_p$ and a set of variables X, by discarding all clauses of $\bigwedge F_p$ that contain a variable appearing in X, the resulting CNF formula $g \equiv \exists X.\ \bigwedge F_p$. This is described by Lang et al. [77].
- Given a formula f in CNF, by removing all instances of the variable x in either polarity, the resulting CNF formula $g \equiv \forall x.\ f$. This is called $\forall$-reduction [74].

Notice that by dropping clauses from the prime implicates, not only are the existential quantifiers eliminated, but also the formula remains in CNF. This means that $\forall$-reduction can be applied immediately after to eliminate universal quantifiers. This would suggest that given a formula of the form $\forall I.\ \exists T.\ f$, the quantifiers can be eliminated as shown in Algorithm 4.

Algorithm 4 Quantifier Elimination by Prime Implicates (QEPI)

Given a formula $\forall I.\ \exists T.\ f$, where f is in CNF, a quantifier-free equivalent formula h can be computed as follows:

1. First, compute $\bigwedge F_p$ of f.
2. Drop all clauses containing any variable of T, thus eliminating all existential quantifiers. This gives a quantifier-free formula $g \equiv \exists T\ f$.
3. Remove all instances of the variables appearing in I to eliminate the universal quantifiers. This gives a quantifier-free formula $h \equiv \forall I.\ g$.

The formula h is both quantifier free and equivalent to $\forall I.\ \exists T.\ f$, thus the quantifiers have been eliminated. Note that stages 2 and 3 are computationally inexpensive, meaning that the success of such an approach greatly depends upon the feasibility of computing the prime implicates.

Traditionally, the prime implicates of a propositional formula are computed by binary resolution and absorption:

Definition 26 (Binary Resolution [74])**.**

$$\frac{(c_1 \vee \ldots \vee c_n \vee \beta) \quad (d_1 \vee \ldots \vee d_m \vee \neg\beta)}{(c_1 \vee \ldots \vee c_n \vee d_1 \vee \ldots \vee d_m)}$$

where each c_i and d_i are propositional literals and β is a variable. The resulting clause is called the resolvent clause.

Definition 27 (Absorption [64])**.** *A clause C absorbs a clause D if every literal in C also appears in D.*

By repeatedly closing a formula under resolution before simplification by absorption, the prime implicates are found [63], however, the number of resolution steps required can be exponential in the number of the input clauses due to the large

number of redundant implicates generated. Directed resolution algorithms, or bucket resolution algorithms [12], such as the Davis-Putnam algorithm [37] (not to be confused with the DPLL algorithm of a similar name [40]) have been suggested that aim to minimise the number or resolution steps required. Whilst directed resolution is tractable for restricted forms of CNF, such as 2-CNF and Horn clauses [40], in general the number of resolution steps required remains high.

Due to the shortcomings of resolution, the problem is approached from a different angle. The following sections show that that the prime implicates can be computed by linear programming, namely through the generation of Chvátal cuts.

4.3 Chvátal Cuts

The relevance of Chvátal cuts stems from surprising parallels between propositional logic and linear algebra. These parallels were described in a survey by Hooker [63]. The fundamental link between propositional logic and linear algebra is the ability to encode clauses as linear inequalities over binary decision variables.

Definition 28 (Linear Clause Encoding)**.** *Given a propositional clause of the form* $\bigvee_{x_i \in P} x_i \vee \bigvee_{x_i \in N} \neg x_i$ *where* P *and* N *are sets of propositional variables, the clause is encoded as a linear inequality as follows:*

$$\sum_{x_i \in P} x_i + \sum_{x_i \in N} (1 - x_i) \geq 1 \qquad \textit{or equivalently} \qquad \sum_{x_i \in P} x_i + \sum_{x_i \in N} -x_i \geq 1 - |N|$$

where each variable in the resulting linear equality is a binary decision variable assuming a value of either 0 (for False*) or 1 (for* True*).*

Henceforth, the latter encoding is used, as it lends itself well to encoding as a matrix. Further, for the remainder of this chapter, a propositional clause and a corresponding linear constraint are considered equivalent, e.g. $(x_1 \vee \neg x_2) \equiv x_1 - x_2 \geq 0$. The upshot of this relationship is that propositional reasoning can be performed with linear constraints. For example, it is possible to test the satisfiability of a set of clauses via linear constraint solving. Given a CNF formula, a system of linear inequalities can be constructed. If the constraints can be satisfied by a $\{0, 1\}$-assignment to each of the variables, then the original

CNF formula is satisfiable. Of particular interest though, is the ability to perform resolution through the generation of Chvátal cuts.

Chvátal's method is traditionally applied for finding integer solutions to linear programs. In this setting, a linear relaxation of an integer linear program (ILP) is solved to give an initial solution, then if the solution is not already integral, cutting planes (Chvátal cuts) are generated to separate non-integer vertices from the feasible space. Surprisingly, Chvátal cuts can be used to find resolvent clauses [63].

Definition 29 (Chvátal Cut)**.** *Let $A\boldsymbol{x} \geq \boldsymbol{b}$ be a system of linear inequalities where $\boldsymbol{x} = \langle x_1, \ldots, x_m \rangle$ is a vector of $\{0,1\}$-variables, A is an $m \times n$ matrix of integer coefficients and $\boldsymbol{b}$ is a vector of integers. A Chvátal cut is a non-negative linear combination of the constraint system with both sides rounded up, i.e. $\lceil \boldsymbol{u}A \rceil \boldsymbol{x} \geq \lceil \boldsymbol{u}b \rceil$, where each u_i of $\boldsymbol{u}$ is non-negative, but at least one $u_i > 0$.*

By encoding a CNF formula as a system of constraints (as shown in Definition 28) and adding bounding constraints of the form $0 \leq x_i \leq 1$, a subset of Chvátal cuts generated from the system will yield inequalities that can be interpreted as resolvent clauses.

Example 3 (Resolution by Cutting Planes)**.** *Consider the CNF formula $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_3)$. The clauses are encoded as inequalities as shown in Definition 28 to give:*

$$\begin{array}{rrrl} x_1 & +x_2 & +x_3 & \geq 1 \\ & x_2 & -x_3 & \geq 0 \end{array}$$

Bounding constraints are added:

$$\begin{array}{rrrrr} x_1 & +x_2 & +x_3 & \geq & 1 \\ & x_2 & -x_3 & \geq & 0 \\ \hline x_1 & & & \geq & 0 \\ -x_1 & & & \geq & -1 \\ & x_2 & & \geq & 0 \\ & -x_2 & & \geq & -1 \\ & & x_3 & \geq & 0 \\ & & -x_3 & \geq & -1 \end{array}$$

The linear combination obtained by $\boldsymbol{u} = \langle 1/2, 1/2, 1/2, 0, 0, 0, 0, 0 \rangle$ is:

$$
\begin{array}{rrrcr}
1/2 \cdot x_1 & +1/2 \cdot x_2 & +1/2 \cdot x_3 & \geq & 1/2 \\
 & 1/2 \cdot x_2 & -1/2 \cdot x_3 & \geq & 0 \\
1/2 \cdot x_1 & & & \geq & 0 \\
\hline
x_1 & +x_2 & & \geq & 1/2
\end{array}
$$

The combination is rounded up to arrive at the cut: $x_1 + x_2 \geq 1 \equiv (x_1 \vee x_2)$*, which can also be obtained by binary resolution of* $(x_1 \vee x_2 \vee x_3)$ *and* $(x_2 \vee \neg x_3)$*.*

Note that not every Chvátal cut will yield an inequality which can be interpreted as a propositional clause. In the above example, $\boldsymbol{u} = \langle 3/4, 3/4, 3/4, 0, 0, 0, 0, 0 \rangle$ gives a linear combination $1 1/2 \cdot x_1 + 1 1/2 \cdot x_2 \geq 3/4$, which when rounded up gives a cut $2x_1 + 2x_2 \geq 1$ which does not correspond to a propositional clause.

Nevertheless, each inequality generated by a Chvátal cut that *can* be interpreted as a propositional clause corresponds to either an input clause or to a resolvent clause and thus always to an implicate. The same resolvent clauses obtained by binary resolution can be found by generating Chvátal cuts, therefore in principle the prime implicates could be computed by exhaustive generation of Chvátal cuts. There is no benefit in doing so however, as the method will suffer from the same problems as exhaustive binary resolution. Observe though, that the task of computing the prime implicates has been translated from the propositional domain into the domain of linear constraints. This raises the question of whether a linear optimisation problem can be devised to find cuts that correspond to resolvent clauses and furthermore, direct the search towards the prime implicates, thus avoiding the generation of redundant implicates.

The algorithm outlined in the following sections derives the prime implicates of a CNF formula via autonomous generation of Chvátal cuts. The algorithm is coined PIDC (Prime Implicates by Directed Cutting). PIDC uses mixed-integer linear programming to discover the prime implicates in as few cuts as possible and avoids the discovery of redundant implicates. The algorithm will be described with the aid of a worked example.

4.4 Worked Example

Suppose that the goal is to compute the prime implicates of the CNF formula:

$$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$$

Generation of cuts proceeds as follows:

1. The clauses are encoded into a system of inequalities $A\boldsymbol{x} \geq \boldsymbol{b}$ with bounding as shown in Example 3. These constraints are referred to as the clause constraints so as to disambiguate them from MILP constraints.

2. A MILP problem is constructed that finds a linear combination $\boldsymbol{u}A\boldsymbol{x} \geq \boldsymbol{u}\boldsymbol{b}$ such that $\lceil \boldsymbol{u}A \rceil \boldsymbol{x} \geq \lceil \boldsymbol{u}\boldsymbol{b} \rceil$ can be interpreted as a propositional clause. Additionally, a cost function aims to find the cut whose propositional interpretation contains the fewest literals. Intuitively, such a cut corresponds to a short implicate. The problem is passed to a MILP solver and a solution is obtained, giving a linear combination $x_2 - x_3 \geq 0$. The combination is rounded up (this step is ineffectual in this case) giving the first cut: $x_2 - x_3 \geq 0 \equiv (x_2 \vee \neg x_3)$. The cut is recorded and added as a new row into the clause constraints.

3. A second MILP is constructed using the augmented clause constraints. This time, a blocking constraint is added to the MILP instance. The blocking constraint (described later in Section 4.5.1) serves two purposes. Firstly, it prevents the same cut from being generated again. Secondly, it stipulates that subsequently found cuts should not be absorbed by previously generated cuts. The MILP is solved and another solution is found. This time the linear combination is $-x_1 + x_2 \geq -1/2$. The combination is rounded up to give the second cut: $-x_1 + x_2 \geq 0 \equiv (\neg x_1 \vee x_2)$. Again, the cut is recorded and added to the clause constraints.

4. A third MILP is constructed in which blocking constraints are present for the two previously found cuts. The MILP is solved yielding another linear combination: $-x_1 \geq -3/4$. Once again, the combination is rounded up to give a third cut $-x_1 \geq 0 \equiv (\neg x_1)$. The cut is recorded and added to the clause constraints.

5. A fourth MILP is constructed in which blocking constraints are present for the three previously found cuts. The fourth MILP is solved. This time the solver states that the problem is INFEASIBLE, so generation of cuts ceases.

Once cut generation terminates, the accumulated cuts are propositionally interpreted to give a set of implicates: $\{(x_2 \vee \neg x_3), (\neg x_1 \vee x_2), (\neg x_1)\}$. Whilst the algorithm does indeed find all cuts corresponding to non-redundant implicates, in some cases redundant cuts are unavoidable. This can be seen here as $(\neg x_1)$ absorbs $(\neg x_1 \vee x_2)$. Once the absorbed clause has been removed, the remaining implicates are the prime implicates: $\{(\neg x_1), (x_2 \vee \neg x_3)\}$. The conjunction of these implicates would be the input of stage two of QEPI as outlined in Algorithm 4.

For this small example, PIDC finds the prime implicates through the discovery of three cuts, one of which yields a redundant implicate. By contrast, exhaustive binary resolution discovers five new implicates and discards four of them. This is demonstrated by the graph in Figure 10. Furthermore, exhaustive resolution would find the same implicates multiple times. For example, the redundant implicate $(\neg x_1 \vee \neg x_3)$ is derived twice: once by resolution of $(x_2 \vee \neg x_3)$ and $(\neg x_1 \vee \neg x_2)$, then again by resolution of $(x_2 \vee \neg x_3)$ and $(\neg x_1 \vee \neg x_2 \vee \neg x_3)$. By finding short implicates and by blocking weaker or duplicate implicates, PIDC finds the prime implicates in fewer steps. The following section discusses in detail the formulation of the optimisation problems underlying PIDC.

4.5 Mixed Integer Linear Programming

The construction of a MILP begins with the input clauses encoded as a linear constraint system with bounding as demonstrated previously in Example 3. Fresh variables are then introduced to represent a linear combination of the constraint system. In the case of the first iteration of the worked example, PIDC starts with the clause constraints shown in Figure 11.

The u_i variables are rational variables representing the row coefficients of $\boldsymbol{u}$ (see Definition 29). The sum, denoted $\boldsymbol{sx} \geq e$ represents a linear combination of the constraints. The integer variables $\boldsymbol{s} = \langle s_1, s_2, s_3 \rangle$ where $s_i \in \{-1, 0, 1\}$ represent the coefficients of x_i on the left hand side of the linear combination, whereas the rational variable e represents the constant on the right hand side of

the linear combination. The linear combination is expressed as MILP constraints by equating the sum of the $\boldsymbol{u}$ coefficients in each column with the corresponding s_i or e coefficient. For the above clause constraints, the following MILP constraints are generated:

$$\begin{aligned} s_1 &= -u_1 - u_2 - u_4 + u_5 - u_6 \\ s_2 &= -u_1 + u_2 + u_3 - u_4 + u_7 - u_8 \\ s_3 &= -u_1 + u_2 - u_3 + u_4 + u_9 - u_{10} \\ e &= -2 \cdot u_1 - u_4 - u_6 - u_8 - u_{10} \end{aligned}$$

Any solution to the above MILP constraints will yield a linear combination $\boldsymbol{sx} \geq e$, however, additional constraints need to be added to ensure that the linear combination will give a Chvátal cut that can be interpreted as a propositional clause. In other words, a cut of the form $\sum_{x_i \in P} x_i + \sum_{x_i \in N} -x_i \geq 1 - |N|$ should be found (see Definition 28). Further, a cut that corresponds to a short implicate should be found. To enforce these properties, several decision variables are deployed. For each s_i, two binary variables o_i and p_i are introduced. Each o_i variable will indicate whether the corresponding x_i literal occurs in the interpretation of the cut, i.e. $o_i = 1$ if s_i is non-zero. Each p_i variable will indicate the polarity of the corresponding x_i literal in the propositional interpretation of the cut, zero for

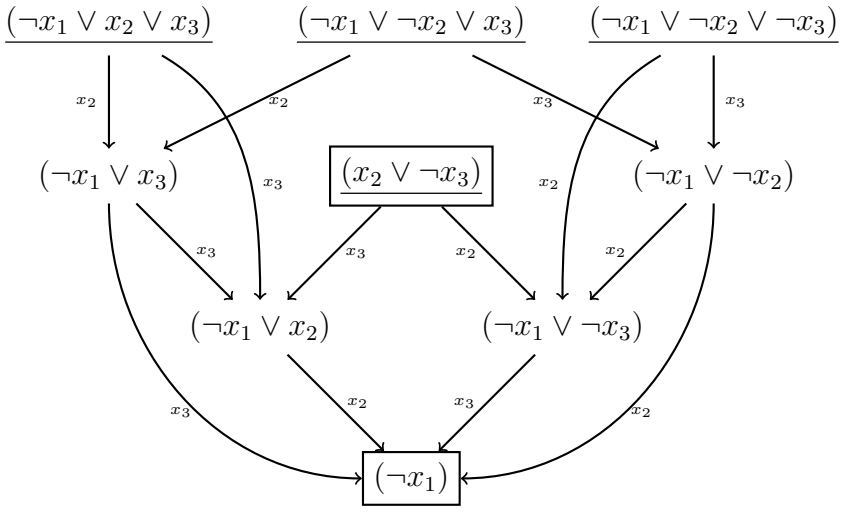


Figure 10: Implicates generated by exhaustive binary resolution of the worked example. Underlined clauses are input clauses. Boxed clauses are prime implicates.

$$
\begin{array}{rrrcr}
-u_1 \cdot x_1 & -u_1 \cdot x_2 & -u_1 \cdot x_3 & \geq & -2 \cdot u_1 \\
-u_2 \cdot x_1 & +u_2 \cdot x_2 & +u_2 \cdot x_3 & \geq & 0 \cdot u_2 \\
 & u_3 \cdot x_2 & -u_3 \cdot x_3 & \geq & 0 \cdot u_3 \\
-u_4 \cdot x_1 & -u_4 \cdot x_2 & +u_4 \cdot x_3 & \geq & -1 \cdot u_4 \\
u_5 \cdot x_1 & & & \geq & 0 \cdot u_5 \\
-u_6 \cdot x_1 & & & \geq & -1 \cdot u_6 \\
 & u_7 \cdot x_2 & & \geq & 0 \cdot u_7 \\
 & -u_8 \cdot x_2 & & \geq & -1 \cdot u_8 \\
 & & u_9 \cdot x_3 & \geq & 0 \cdot u_9 \\
 & & -u_{10} \cdot x_3 & \geq & -1 \cdot u_{10} \\
\hline
s_1 \cdot x_1 & +s_2 \cdot x_2 & +s_3 \cdot x_3 & \geq & e \\
\hline
\end{array}
$$

Figure 11: Initial clause constraints for the worked example.

positive (or not occurring), and one for negative. The assignments to each o_i and p_i are expressed by $s_i = o_i - (2p_i)$. The correctness of this constraint is argued in Section A.2 (Theorem 8, Page 175).

With the p_i variables constrained in this way, the right hand side of the linear combination can be constrained to ensure that a cut corresponding to a propositional clause is found. For the worked example, e is further constrained by the inequality $e \geq \epsilon - p_1 - p_2 - p_3$, where ϵ is a small constant just above zero (10^{-4} will suffice). This ensures that once the linear combination $\boldsymbol{sx} \geq e$ is rounded up, the resulting inequality can be interpreted as a propositional clause. Further, it stipulates that at least one u_i is non-zero, because when $\boldsymbol{u} = \langle 0, \ldots, 0 \rangle$, then it follows that e is zero and each p_i is also zero, thus the constraint is false.

Finally, a cost function is used to give preference to cuts with fewer variables occurring, therefore finding a cut corresponding to a short implicate. This is key to the method, as a short implicate is less likely to be absorbed by subsequently found implicates and is more likely to be prime. To this end, the cost function simply minimises the sum of the o_i variables. For the worked example, the cost function is $\mathit{minimise} : o_1 + o_2 + o_3$.

With this, the MILP of the first iteration of the worked example is complete. The full constraint system is shown in normalised form in Figure 12. The constraints are passed to a MILP solver, which returns a solution that assigns:

$$\boldsymbol{u} = \langle 0, 0, 1, 0, 0, 0, 0, 0, 0, 0 \rangle \qquad \boldsymbol{s} = \langle 0, 1, -1 \rangle \qquad e = 0$$

$$
\begin{aligned}
\mathit{minimise} \quad & o_1 + o_2 + o_3 \ s.t. \\
\mathit{integers} \quad & s_1, s_2, s_3, p_1, p_2, p_3, o_1, o_2, o_3
\end{aligned}
$$

$$
\begin{aligned}
-1 \leq & s_1 \leq 1 \quad -1 \leq s_2 \leq 1 \quad -1 \leq s_3 \leq 1 \\
0 \leq & p_1 \leq 1 \quad 0 \leq p_2 \leq 1 \quad 0 \leq p_3 \leq 1 \\
0 \leq & o_1 \leq 1 \quad 0 \leq o_2 \leq 1 \quad 0 \leq o_3 \leq 1 \\
-\infty \leq & e \leq \infty \\
& -u_1 - u_2 - u_4 + u_5 - u_6 - s_1 = 0 \\
& -u_1 + u_2 + u_3 - u_4 + u_7 - u_8 - s_2 = 0 \\
& -u_1 + u_2 - u_3 + u_4 + u_9 - u_{10} - s_3 = 0 \\
& -2 \cdot u_1 - u_4 - u_6 - u_8 - u_{10} - e = 0 \\
& s_1 + 2 \cdot p_1 - o_1 = 0 \\
& s_2 + 2 \cdot p_2 - o_2 = 0 \\
& s_3 + 2 \cdot p_3 - o_3 = 0 \\
& p_1 + p_2 + p_3 + e \geq \epsilon
\end{aligned}
$$

Figure 12: The complete MILP for the first iteration of the worked example.

By rounding up e in the linear combination, the cut $x_2 - x_3 \geq 0 \equiv (x_2 \vee \neg x_3)$ is found. In this case, the implicate is one of the input clauses. This is because one of the input clauses is a prime implicate (see Figure 10). Note that, for the above, rounding up e was ineffectual, however, this is not always the case. There is no need to round the s_i variables as they may only assume integer values.

4.5.1 Enumerating Cuts

So far, it has been shown that a Chvátal cut corresponding to a short implicate can be automatically generated using mathematical optimisation. However, cuts must be enumerated until the prime implicates have been found. There are several considerations that must be addressed:

1. It must be possible to find deeper cuts, i.e. cuts that only become feasible having found intermediate cuts. Failure to find deeper cuts would mean that only implicates corresponding to input resolution would be found [63]. Input resolution (resolution of clauses where at least one input clause is used at each step) is not sufficient to find the prime implicates. This is

demonstrated by the graph shown in Figure 10. The only way to derive the prime implicate $(\neg x_1)$ is by resolution of clauses which are not input clauses.

2. The solver should not find cuts which correspond to an implicate already known. Not only is a duplicate implicate useless, but the solver may repeatedly find the same implicate, thus compromising termination.

3. Subsequently found cuts should not yield implicates that are absorbed by any previously found implicate. An implicate absorbed in this way must be a weaker implicate than one that is already known and is therefore redundant. The discovery of redundant implicates also wastes solving iterations, thus impacting the performance and space requirements of the algorithm.

To illustrate how PIDC enumerates cuts, consider the beginning of step 3 of the worked example (Section 4.4), where the first cut has been found and a second MILP is about to be constructed. The first cut was $x_2 - x_3 \geq 0$. To make deeper cuts feasible, the constraint $x_2 - x_3 \geq 0$ is added as a new row into the clause constraints as shown in Figure 13[1]. The augmented clause constraints are then encoded into a new MILP as described before. To ensure that cuts corresponding to absorbed or duplicate implicates are not found, for each cut found so far, a blocking constraint is added to the MILP.

Definition 30 (Blocking Constraint)**.** *Given a linear constraint of the form* $\sum_{x_i \in P} x_i + \sum_{x_i \in N} -x_i \geq 1 - |N|$*, where* P *and* N *are sets of decision variables, a blocking constraint is:*

$$\sum_{x_i \in P} (-o_i + p_i) + \sum_{x_i \in N} (-o_i - p_i) \geq 1 - |P| - 2 \cdot |N|$$

At step 3 of the worked example, the cut $x_2 - x_3 \geq 0$ has just been found. For this cut, $P = \{x_2\}$ and $N = \{x_3\}$, so the blocking constraint $-o_2 + p_2 - o_3 - p_3 \geq -2$ is added to the MILP instance. The addition of a blocking constraint asserts that either a) at least one of the occurring variables of the cut must not occur in subsequently found cuts, or b) the occurring variables of the cut may occur in

[1] Actually, in this case it is not strictly necessary to add the cut because $x_2 - x_3 \geq 0$ already exists in the clause constraints. Since this is not always the case, and for purpose of example, the constraint is added anyway.

$$
\begin{array}{rrrcr}
-u_1 \cdot x_1 & -u_1 \cdot x_2 & -u_1 \cdot x_3 & \geq & -2 \cdot u_1 \\
-u_2 \cdot x_1 & +u_2 \cdot x_2 & +u_2 \cdot x_3 & \geq & 0 \cdot u_2 \\
 & u_3 \cdot x_2 & -u_3 \cdot x_3 & \geq & 0 \cdot u_3 \\
-u_4 \cdot x_1 & -u_4 \cdot x_2 & +u_4 \cdot x_3 & \geq & -1 \cdot u_4 \\
u_5 \cot x_1 & & & \geq & 0 \cdot u_5 \\
-u_6 \cdot x_1 & & & \geq & -1 \cdot u_6 \\
 & u_7 \cdot x_2 & & \geq & 0 \cdot u_7 \\
 & -u_8 \cdot x_2 & & \geq & -1 \cdot u_8 \\
 & & u_9 \cdot x_3 & \geq & 0 \cdot u_9 \\
 & & -u_{10} \cdot x_3 & \geq & -1 \cdot u_{10} \\
 & u_{11} \cdot x_2 & -u_{11} \cdot x_3 & \geq & 0 \cdot u_{11} \\
\hline
s_1 \cdot x_1 & +s_2 \cdot x_2 & +s_3 \cdot x_3 & \geq & e
\end{array}
$$

Figure 13: Clause constraints for the second iteration of the worked example.

subsequently found cuts, but at least one literal must have differing polarity. It is easier to see this by taking the propositional interpretation of the blocking constraint, e.g. $-o_2 + p_2 - o_3 - p_3 \geq -2 \equiv (\neg o_2 \vee p_2 \vee \neg o_3 \vee \neg p_3)$. Remember, when $p_i = 1$, x_i is negative.

Once the blocking constraint has been added to the MILP, the problem is passed to the solver to find the next cut. The second MILP of the worked example is shown in Figure 14. The solver returns a solution assigning $\boldsymbol{u} = \langle 1/4, 3/4, 1/2, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$. This gives the linear combination $-x_1 + x_2 \geq -1/2$. The right hand side is rounded up to give a cut: $-x_1 + x_2 \geq 0 \equiv (\neg x_1 \vee x_2)$.

4.5.2 Termination

The enumeration process continues until the solver returns INFEASIBLE. By taking the accumulated cuts, interpreting them as clauses and simplifying them via absorption, the prime implicates are found. The absorption stage is necessary as in some cases redundant implicates must be found in order to find a prime implicate. For example, the worked example finds the redundant implicate $(\neg x_1 \vee x_2)$ which is absorbed by the prime implicate $(\neg x_1)$. The discovery of a redundant implicate is unavoidable in this case, as $(\neg x_1)$ can only be discovered by resolution with a redundant implicate (see Figure 10).

Alternatively, PIDC may terminate upon detecting the empty clause. The empty clause is characterised by a MILP solution where each $s_i = 0$ and $e = \epsilon$. In

$$
\begin{aligned}
minimise \quad & o_1 + o_2 + o_3 \ s.t. \\
integers \quad & s_1, s_2, s_3, p_1, p_2, p_3, o_1, o_2, o_3
\end{aligned}
$$

$$
\begin{aligned}
-1 \leq & s_1 \leq 1 \quad -1 \leq s_2 \leq 1 \quad -1 \leq s_3 \leq 1 \\
0 \leq & p_1 \leq 1 \quad 0 \leq p_2 \leq 1 \quad 0 \leq p_3 \leq 1 \\
0 \leq & o_1 \leq 1 \quad 0 \leq o_2 \leq 1 \quad 0 \leq o_3 \leq 1 \\
-\infty \leq & e \leq \infty \\
-2 \cdot u_1 - u_4 - u_6 - u_8 - u_{10} - e &= 0 \\
-u_1 - u_2 - u_4 + u_5 - u_6 - s_1 &= 0 \\
-u_1 + u_2 + u_3 - u_4 + u_7 - u_8 + u_{11} - s_2 &= 0 \\
-u_1 + u_2 - u_3 + u_4 + u_9 - u_{10} - u_{11} - s_3 &= 0 \\
s_1 + 2 \cdot p_1 - o_1 &= 0 \\
s_2 + 2 \cdot p_2 - o_2 &= 0 \\
s_3 + 2 \cdot p_3 - o_3 &= 0 \\
p_1 + p_2 + p_3 + e &\geq \epsilon \\
p_2 - p_3 - o_2 - o_3 &\geq -2
\end{aligned}
$$

Figure 14: The complete MILP for the second iteration of the worked example.

such a case, the input formula is inconsistent. When the empty clause is detected, PIDC halts, flagging the formula as inconsistent. It is interesting to note that it is the top priority of the solver to find the empty clause. When each s_i is zero, necessarily each o_i must also be zero (see Theorem 8 on Page 175). Because the cost function aims to minimise the sum of the o_i variables, the empty clause is, in the eyes of the solver, the most optimal solution that could ever be found. This means that PIDC "hones in" on the empty clause if the input formula is inconsistent.

4.5.3 Implementation Detail

Notice that upon discovery of each new cut, the clause constraints are amended and a new MILP is encoded. The algorithm was described in this way to aid reader understanding. The implementation from which the experimental results are taken does not amend the clause constraints or encode a fresh MILP for each iteration of the algorithm. Instead, the existing MILP is adjusted to reflect the

necessary changes. Namely, $\boldsymbol{u}$ is extended with a fresh rational variable, the inequalities constraining s_i and e are amended and a blocking constraint is added. Note that although the existing clause constraints are re-used and modified, the solving process is not incremental in the classical SAT/SMT sense [75].

4.6 Experimental Results

To evaluate PIDC, the method was compared with the binary resolution approach to computing the prime implicates. If PIDC is feasible, then it can be used as the first stage of the proposed QEPI algorithm (described in Section 4.2).

The test environment is underpinned by a random test harness that generates random CNF formulae based upon a number of tunable parameters: the number of variables over which a formula is defined, the number of random clauses in a formula, the minimum clause size and the maximum clause size. By providing values for these parameters, the test harness is able to generate random test cases and then compute the prime implicates, first by binary resolution and then once more by PIDC. There are two metrics of interest: the number of operations required to compute prime implicates and the time taken to compute the prime implicates. An operation is defined to be a resolution operation, a Chvátal cut, or an absorption operation. The number of these operations will serve as an indicator of the amount of redundant work expended by either technique. The solving times will indicate the feasibility of PIDC in practice.

The test harness was written in Python. MILP solving was performed using the Python bindings to LPSolve (calling out to C). Binary resolution was performed from within Python using the built-in `frozenset` data structure. The binary resolution algorithm that was used is shown in Algorithm 5. Note that to avoid running out of memory, exhaustive resolution was not used. Instead, a more systematic approach is used, which simplifies the formula between input resolution phases. Alternatively, a method such as bucket elimination could have been deployed [39]. The binary resolution implementation also plays the role of an oracle, verifying the correctness of each solution given by PIDC.

Three sets of experiments were conducted:

1. Random CNFs over 5 variables and 8 clauses of size between 1 and 4 literals.

Algorithm 5 Binary resolution algorithm used in experiments.

```
function BinRes(C)                                  ▷ Takes a set of clauses C.
    v ←Vars(C)
    I ← C                                           ▷ I will be the returned implicates.
    I′ ← ∅                  ▷ The last iteration's implicates, for detecting a fixpoint.
    while I′ ≠ I do
        N ← ∅                                       ▷ N is the new resolvents found this iteration.
        for v_i ∈ v do
                    ▷ Partition the clauses of I by +ve and -ve occurrences of v_i.
            ⟨I⁺, I⁻⟩ ← Partition(I, v_i)
            β ← ∅                ▷ Collects new resolvents found by resolution upon v_i.
            for p ∈ I⁺ do
                for n ∈ I⁻ do
                    r ←ResolveClauses(p, n)
                    if NotTautologyClause(r) then
                        β ← β ∪ r
                    end if
                end for
            end for
            N ← N ∪ β
        end for
        I′ ← I                  ▷ Caches the set of implicates from the last iteration.
        I ←Simplify(N ∪ I)                          ▷ Simplification by absorption.
    end while
    return I                                        ▷ The prime implicates are returned.
end function
```

2. Random CNFs over 10 variables and 16 clauses of size between 1 and 4 literals.

3. Random CNFs over 20 variables and 32 clauses of size between 1 and 4 literals.

A small number of clauses were used for each CNF so that binary resolution could operate within the resource constraints of the test machine (4GB of RAM). The number of literals per clause was chosen so as to reflect the typical clause length of a formula that has undergone the Tseitin transformation [115]. For each set of experiments, 2000 samples were taken. Each individual experiment was given a 20 second time allowance to complete. Any experiment taking longer than this was terminated and disregarded.

The experimental results are shown in Figures 15 to 20. Figures 15, 17 and 19 show the number of operations performed by binary resolution (x-axis)

plotted against the number of operations performed by PIDC (y-axis). On each of these graphs, a line separates the points where binary resolution required more operations than PIDC[2]. For all three sets of experiments, the majority of the points fall beneath the line, indicating that PIDC usually discovers the prime implicates in fewer operations than binary resolution. The effect is most exaggerated by the results of the third set of tests (Figure 19), where binary resolution could require more than 12,000 operations, whereas PIDC requires no more than 60. Also notice that there is always a cluster of points across $x = 1$; upon further inspection, these points correspond to experiments where the randomly generated formula was inconsistent and where the empty clause could be derived in the first iteration of PIDC.

Figures 16, 18 and 20 show the solving times in seconds of binary resolution (x-axis) plotted against the solving times of PIDC (y-axis). On each of these graphs, a line separates the points where binary resolution took the longest from the points where PIDC took the longest. Unfortunately, for all three sets of experiments, most of the points fall above the line, indicating that in most cases PIDC takes longer to find the prime implicates. For the small experiments of the first test set (Figure 16), PIDC is a bit slower, but remains competitive with binary resolution. As the size of the input is increased (Figures 18 and 20), the difference in solving times increases, with PIDC performing significantly worse. In fact, in the third test set, 208 of the 2000 experiments caused the 20 second timeout to fire when solved by PIDC, yet binary resolution managed to find the prime implicates in no more than 0.8 seconds.

4.7 Chapter Summary

This chapter has shown that the prime implicates of a CNF formula can be computed by a series of linear optimisation problems (PIDC). Through this construction, cost functions and blocking constraints can be deployed to search for short implicates, therefore reducing the number of redundant implicates found. Since existential QE and universal QE are straightforward given the prime implicates, the proposed algorithm can be used as part of the process of deriving a quantifier-free formula equivalent to $\forall I.\ \exists T.\ f$.

[2]In Figure 19 this line is on top of the y-axis.

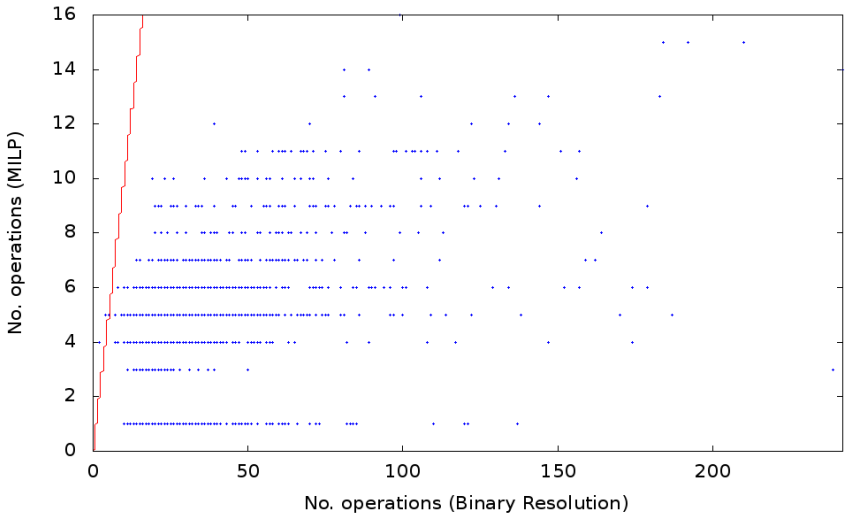


Figure 15: Number of operations required for problems containing 5 variables and 8 clauses of length between 1 and 4.

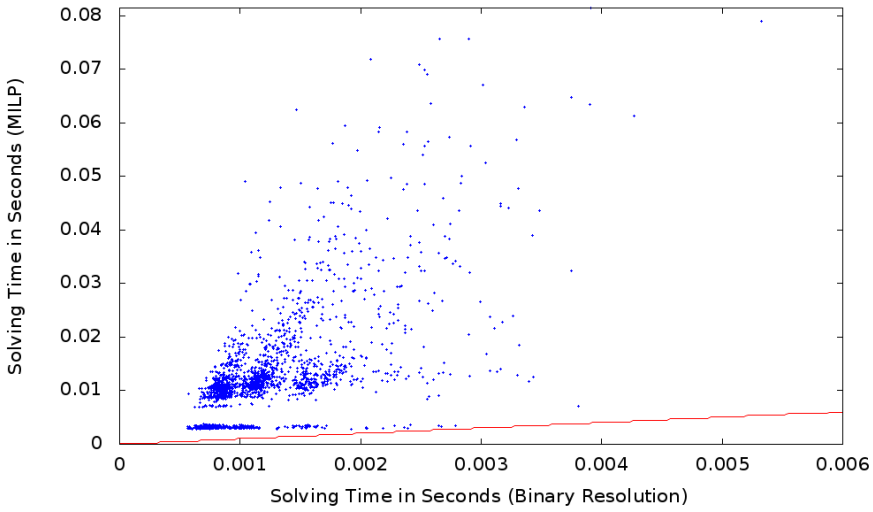


Figure 16: Solving times for problems containing 5 variables and 8 clauses of length between 1 and 4.

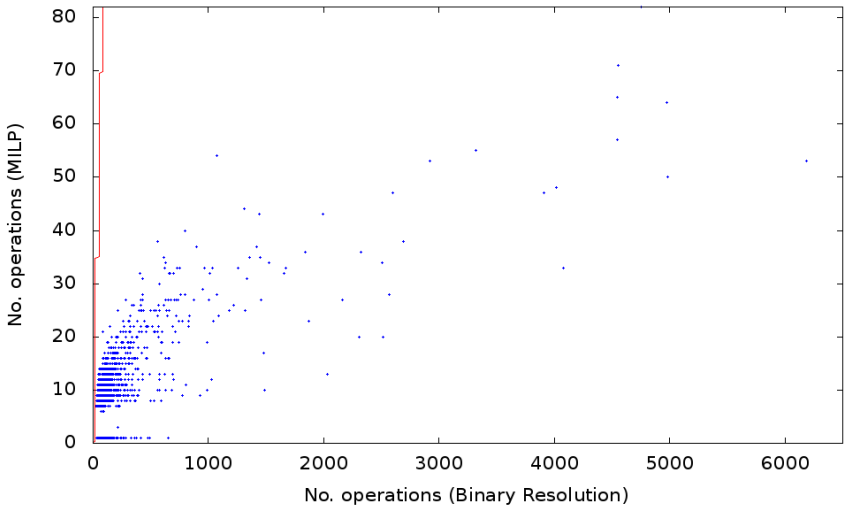


Figure 17: Number of operations required for problems containing 10 variables and 16 clauses of length between 1 and 4.

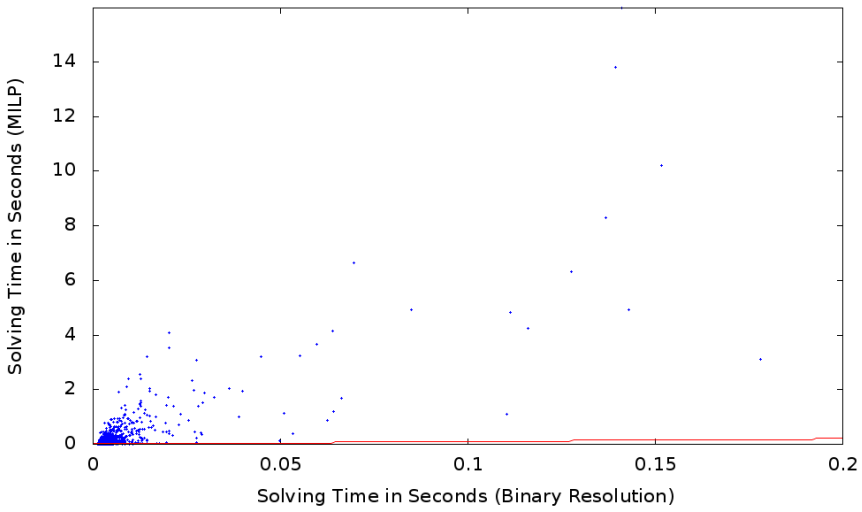


Figure 18: Solving times for problems containing 10 variables and 16 clauses of length between 1 and 4.

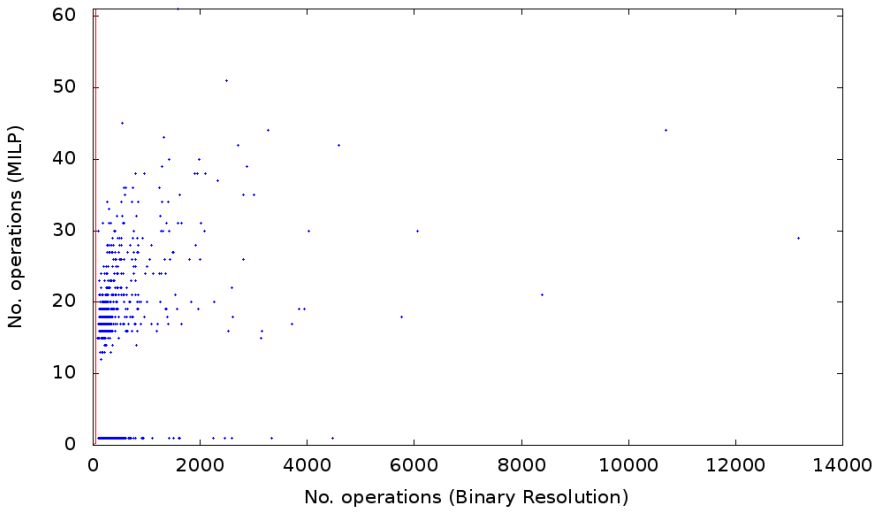


Figure 19: Number of operations required for problems containing 20 variables and 32 clauses of length between 1 and 4.

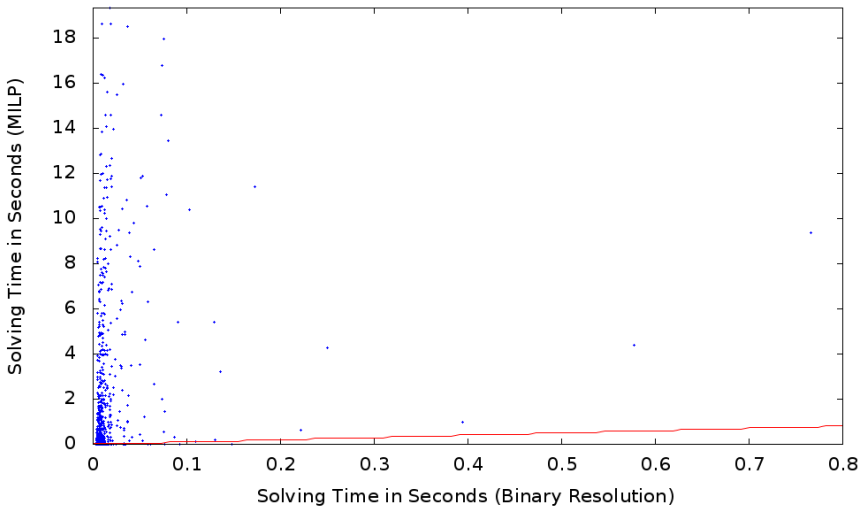


Figure 20: Solving times for problems containing 20 variables and 32 clauses of length between 1 and 4.

Eliminating quantifiers from the prime implicates is inexpensive, so the feasibility of the approach is predicated upon the feasibility of computing the prime implicates. Experimental results show that the number of operations required to compute the prime implicates by PIDC is much fewer than by binary resolution. The technique also has the advantage that it finds the empty clause quickly if the input formula is inconsistent. Unfortunately PIDC does not scale well and binary resolution was able to find the prime implicates faster in most cases. The suboptimal performance of PIDC is likely to be attributed to the use of integer variables in the optimisation problems. A linear optimisation problem with no integer variables is in complexity class P. By enforcing integrality upon the variables of a linear program, the problem is promoted to NP-complete [106]. It follows that ILP must also be NP-hard.

Despite the shortcomings of the new technique, it has at least been shown that it is possible to compute the prime implicates as an optimisation problem. If PIDC could be improved so that the time taken to compute a single cut matched that of a single binary resolution step, then the PIDC would, without a doubt, out-perform binary resolution. This can be seen by comparing the number of operations required to compute the prime implicates by each algorithm. One possible way to speed up the algorithm might involve the relaxation of the integer variables, however, it is not immediately clear how this could be achieved. It may also be possible to port the algorithm to a different decision procedure, where the interplay between decision variables and linear constraints could be handled more efficiently.

In conclusion, it has yet to be seen whether range and set abstraction combined with quantifier elimination would be feasible for inferring ranges and sets of register values.

Chapter 5

Range Analysis using Linear Programming

5.1 Introduction

At this point in the thesis, the focus of study shifts slightly. The work shown in the previous two chapters presented ways by which to infer ranges and sets for any given register at any given program point. Whilst these methods are useful for inferring ranges (and sets) for strategically selected sites in a binary program – for example, indirect jumps – often it is useful to infer ranges and sets for every register at every program point. Perhaps we want to show that a property holds for the entire program's execution. In principal, it is possible to infer ranges and sets for the entire program by range and set abstraction, however this would mean running an analysis once for each register at each program point. Such an approach is unlikely to be practical when analysing binary programs consisting of hundreds or thousands of program points.

In this chapter (and the next), a more general range analysis is described; one which can infer a range for each register at each program point in a single invocation. This range analysis assumes that the control flow graph is already known, meaning that range and set abstraction (with a suitable quantifier elimination strategy) remain relevant, if only as a preliminary analysis to uncover the control flow graph. Of course, if it is assumed that the CFG is known, there is nothing

precluding the application of standard abstract interpretation techniques. However, as has already been discussed, such techniques often suffer from problems relating to fixpoint convergence. The range analysis described in the following sections aims to sidestep these hindrances by using an alternative solving strategy, namely, linear optimisation.

5.2 Motivation

In the introductory chapter of this thesis, several applications for binary analysis were identified. Amongst these applications, and probably the most prolific at this time, are the applications relating to security. Of particular interest to the government, to the military and to penetration testers, is the problem of auditing commercial off-the-shelf software (COTS) packages for security vulnerabilities. COTS is being increasingly deployed since it is seen to reduce development times. However, these components are written by third-parties, typically with an eye towards functionality rather than security and reliability [41]. This motivates consumers to audit COTS components prior to deployment. Specifically, it is important to know whether a software product is susceptible to (amongst others) buffer overflows, integer overflows, race conditions, etc. If exploited, these vulnerabilities could be used maliciously to corrupt the system on which the code is running, to obtain sensitive data or to execute arbitrary code. Sadly, exploits for high profile software products are published all too often. Even now, a large portion of the exploits published on the Internet still involve a buffer overflow [1].

Unfortunately, auditing for security vulnerabilities is not straight-forward where COTS components are concerned. These packages are commercial products whose source code is kept secret. This prevents competing software houses from examining the inner workings and also makes "cracking" more difficult[1]. For these reasons, COTS packages are usually delivered as pre-compiled binaries and linkable shared libraries only [122]. Some companies go as far as to obfuscate their binary code, making it only more opaque. Because the components are distributed in a binary-only form, standard software engineering tools, like linters and static analysers built into compilers or the development environment itself, are no longer of use for auditing because they require the source code. As a consequence, analyses

[1]See Page 2 for a description of crackers and cracking.

must occur at the binary level.

When examining binary code, range information can often help to identify possible vulnerabilities by, for example, inferring that an indirect memory access may be out of bounds; this may correspond to an out of bounds array access in the source code. Whilst it is recognised that range information can aid the security engineer in the auditing process [42], most industrial decompilers do not currently infer ranges for the values stored in registers and memory. This is surprising because the interval domain has been studied in great detail and is often the subject of pedagogical discussions relating to static analysis and abstract interpretation.

Perhaps the reason why intervals have not been adopted for use in decompilers relates to the fact that, in order for an interval analysis to be fast, a widening strategy must be devised [32]. Even for finite-precision signed 32-bit computer integers, the domain of intervals, $D = \{\emptyset\} \cup \{[l, u] \mid -2^{31} \leq l \leq u \leq 2^{31} - 1\}$, admits long ascending chains such as $d_0 = \emptyset$ and $d_{n+1} = [0, n]$ where $n \in [0, 2^{32} - 1]$. As discussed in Chapter 2, widening can be integrated into an abstract interpretation to guarantee fast termination, albeit at the cost of precision.

To illustrate, observe that the lower bound in the chain $d_0, d_1, d_2, \ldots$ is stable after d_1, whilst the upper bound is strictly increasing. Widening would typically enlarge, literally widen, d_3 to $[0, 2^{31} - 1]$ to preserve the lower bound of 0 whilst relaxing the upper bound to the maximum representable signed integer. This side-steps the generation of the intermediates $d_4, \ldots, d_{2^{31}-2}$. A fixpoint is reached very quickly and the abstraction remains sound (in that all possible states are captured by the over-approximate interval). It could be argued however, that this widening scheme is overly aggressive. Suppose that the chain's least-fixpoint is $d_{31} = [0, 31]$. A widening strategy that over-approximates the possible values as $[0, 2^{32} - 1]$ is hardly precise.

In response to this shortcoming, variations of widening have been proposed which aim to find more precise information. One variation prescribes a set of increasing thresholds which are widened to in a series of steps. If relaxing a bound to one threshold is not sufficient for stability, then the next threshold is tried, and so on. This is called widening with thresholds [14]. Note that the threshold values themselves must be specified a priori and it is not always obvious what appropriate thresholds may be. With a view towards automation, widenings [57, 109] have

been suggested that infer the thresholds based on the structure of a program, in particular, where a transition in a chain from one interval to the next flips an abstract semantic equation from being unsatisfiable to being satisfiable. This usually signifies a change in program behaviour, such as the exit condition of a loop being satisfied for the first time. By widening to these semantic thresholds, the analysis is much more likely to find a precise fixpoint. The counter argument for these methods is that they are either speculative (as with [109]) or require two analyses to be run side-by-side (as with [57]).

The method in this chapter takes a different approach entirely. Namely, computation of the least-fixpoint is performed by mathematical optimisation. The method is heavily inspired by the pioneering work of Rugina et al. [99], who showed that range analysis by optimisation is not only possible, but also advantageous. The constraint solver effectively replaces Kleene iteration, so the need for widening is dispelled completely and the least-fixpoint is computed directly. Such an approach is much more suited to the application of decompilers, where ranges are expected to be precise, but also inferred automatically. Unfortunately, it was found that Rugina's method was unsound with regards to certain kinds of conditional constructs. The method described in this chapter shows that for soundness (and precision) non-linear constraints are required to correctly model conditionals. Specifically the contributions are:

- It is shown that range analysis can be formulated in terms of *min* and *max* constraints.
- By contrast to traditional Kleene iteration, a method is shown which computes the least-fixpoint of the constraints by repeatedly calling a linear programming solver.
- Heuristics are given to minimise the number of calls to the solver required. These heuristics significantly improve the performance of the algorithm.
- Experimental results indicate that the method is feasible and can be used to find potential buffer overflow vulnerabilities in binary code.

The structure of the remainder of this chapter is as follows. Section 5.3 gives a high-level overview of the method with a worked example, then Section 5.5

shows how to solve systems of inequalities containing disjunctions using repeated linear programming (LP). Experimental results are shown in Section 5.6 and in Section 5.7 the shortcomings of Rugina's method are discussed. The chapter is concluded in Section 5.8.

5.3 Worked Example

This section shows how ranges can be derived as an optimisation problem and without resorting to widening. Although the method is targeted at the binary level, the following worked example analyses a high-level pseudo-code program so as to aid reader comprehension. Later, experimental results for binary programs are shown.

```
1 i := 10;
2 while (i ≥ m)
3         m := m + 1;
4 endwhile
5 // last program point
```

Listing 5.1: Worked Example Program.

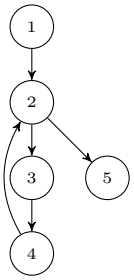


Figure 21: Control flow graph for the worked example program.

5.3.1 Collecting Semantics

Listing 5.1 shows a small program where the program points are annotated 1 through to 5. Suppose that each variable is a signed 32-bit integer and that $m \in [5, 20]$ prior to execution (at program point 1). The overall goal of the analysis is to summarise the values of each variable at each of the program points

using intervals, but without actually running the program. First the concrete domain is specified, over which the collecting semantics is defined. This semantics must capture the possible values of the two variables `i` and `m` at each program point.

To this end, let the set of possible 32-bit signed integers be $Z = \{x \mid -2^{31} \leq x \leq 2^{31} - 1\}$. The possible values that any given variable could assume at any single program point is then a subset of Z. So as to account for multiple variables, the possible states at any given program point are expressed as a set of n-vectors drawn from $\wp(Z^n)$, where n parameterises the concrete domain according to the number of program variables. The worked example program is concerned with the abstraction of two variables per program point, so in this case $n = 2$ and the concrete state at each program point is a set of 2-vectors. To simplify notation, let $U = \wp(Z^n)$. This shall serve as the concrete domain. The ordering and the domain operations of U are defined in the usual manner, so as to form a complete and finite lattice:

Definition 31 (Ordering and domain operations for U)**.**

$$\begin{array}{rcl} S_i \subseteq_U S_j & \iff & S_i \subseteq S_j \\ S_i \cup_U S_j & \triangleq & S_i \cup S_j \\ S_i \cap_U S_j & \triangleq & S_i \cap S_j \end{array}$$

Once the concrete domain is specified, the collecting semantics of can be defined. For each program point, P_k, a set $S_k \in U$ represents the possible concrete states. A set of recursive equations defines and relates each of these sets. For the worked example program, the collecting semantics are as follows:

$$\begin{array}{rl} S_1 = & \{\langle i, m\rangle \mid -2^{31} \leq i \leq 2^{31} - 1 \wedge 5 \leq m \leq 20\} \\ S_{2\star} = & \{\langle 10, m\rangle \mid \langle i, m\rangle \in S_1\} \\ S_2 = & S_{2\star} \cup S_4 \\ S_3 = & \{\langle i, m\rangle \mid \langle i, m\rangle \in S_2 \wedge i \geq m\} \\ S_4 = & \{\langle i, \min(m + 1, 2^{31} - 1)\rangle \mid \langle i, m\rangle \in S_3\} \\ S_5 = & \{\langle i, m\rangle \mid \langle i, m\rangle \in S_2 \wedge i < m\} \end{array}$$

The dependencies between the program points, hence the dependencies between the sets, are illustrated by the control flow graph (CFG) shown in Figure 21. Note

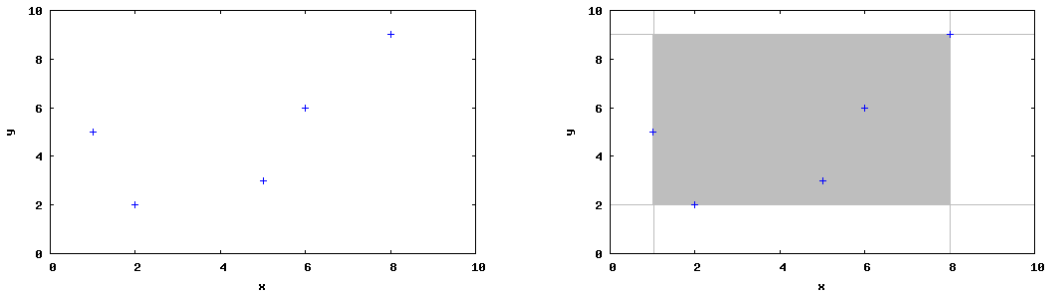


Figure 22: (a) $S = \{\langle 2,2\rangle, \langle 5,3\rangle, \langle 1,5\rangle, \langle 6,6\rangle, \langle 8,9\rangle\}$ (b) $\alpha_U(S) = [1,8] \times [2,9]$

how S_2 is defined in terms of S_4 and $S_{2\star}$ which, in turn, is defined in terms of S_1. This is because control passes from program point 1 and 4 to program point 2. The set $S_{2\star}$ is merely introduced as a calculational device (an intermediate set) that is used to decompose S_2 into an update operation and a merge operation, that define $S_{2\star}$ and S_2 respectively. Note too that for now the collecting semantics do not faithfully reflect the possibility of integer overflows. Instead, the increment operation at program point 3 is assumed to be a saturating computation. In actuality, an overflow may not occur for this particular example. In the next chapter, however, an extension to the method described here is proposed which takes care of integer overflow scenarios.

5.3.2 Abstract Semantics

Every subset of Z can be efficiently abstracted by an interval drawn from the abstract domain $D = \{\emptyset\} \cup \{[l,u] \mid -2^{31} \leq l \leq u \leq 2^{31}-1\}$. It follows that each element of the concrete domain can be abstracted by n intervals, i.e. an element drawn from D^n. For the worked example, two variables will be abstracted per program point, so each element of the abstract domain will be drawn from D^2. The ordering and domain operations upon D^n are lifted from D in the obvious way:

Definition 32 (Ordering and domain operations for D^n)**.**

$$\begin{array}{lcll} \langle d_1,\ldots,d_n\rangle & \sqsubseteq_{D^n} & \langle d'_1,\ldots,d'_n\rangle & \iff \quad d_1 \sqsubseteq_D d'_1 \wedge \ldots \wedge d_n \sqsubseteq_D d'_n \\ \langle d_1,\ldots,d_n\rangle & \sqcup_{D^n} & \langle d'_1,\ldots d'_n\rangle & \triangleq \quad \langle d_1 \sqcup_D d'_1,\ldots,d_n \sqcup_D d'_n\rangle \\ \langle d_1,\ldots,d_n\rangle & \sqcap_{D^n} & \langle d'_1,\ldots d'_n\rangle & \triangleq \quad \langle d_1 \sqcap_D d'_1,\ldots,d_n \sqcap_D d'_n\rangle \end{array}$$

where $\sqsubseteq_D, \sqcup_D$ and $\sqcap_D$ are the standard ordering, join operation and meet operation of the interval domain [32]. Similarly, $\alpha_Z : Z \to D$ and $\gamma_D : D \to Z$ are the standard interval abstraction and concretisation mappings. The correspondence between the concrete domain U and the abstract domain D^n is then formalised as follows:

Definition 33 (Correspondence between U and D^n)**.**

$$\begin{aligned} \alpha_U &: U \to D^n \\ \alpha_U(S) &= \langle \alpha_Z(v_1), \ldots, \alpha_Z(v_n) \rangle \text{ where } v_i = \{p_i \mid \langle p_1, \ldots, p_n \rangle \in S\} \end{aligned}$$

$$\begin{aligned} \gamma_{D^n} &: D^n \to U \\ \gamma_{D^n}(\langle d_1, \ldots, d_n \rangle) &= \{\langle v_1, \ldots, v_n \rangle \mid v_1 \in \gamma_D(d_1) \wedge \ldots \wedge v_n \in \gamma_D(d_n)\} \end{aligned}$$

Each n-tuple of intervals $\langle d_1, \ldots, d_n \rangle \in D^n$ is interpreted as its Cartesian product $d_1 \times \ldots \times d_n$ which defines a hyper-rectangle in n-dimensional space. Observe also how $\alpha_U(S)$ computes the least hyper-rectangle that encloses each n-vector in S. This is illustrated graphically for $n = 2$ in Figure 22.

With this relationship in place, the collecting semantics can be relaxed to a system of recursive equations that operate over hyper-rectangles drawn from D^n. For each program point, P_k, an abstraction $S'_k \in D^n$ aims to best characterise $S_k \in U$. From here onward, subscripts on domain operations ($\sqcup$ and $\sqcap$) are omitted for readability[2]. The following recursive equations are the abstract semantics for the worked example:

$$\begin{aligned} S'_1 &= \langle [-2^{31}, 2^{31} - 1], [5, 20] \rangle \\ S'_{2\star} &= \langle [10, 10], m \rangle \text{ where } \langle i, m \rangle = S'_1 \\ S'_2 &= S'_{2\star} \sqcup S'_4 \\ S'_3 &= \langle [l_i, u_i] \sqcap [l_m, 2^{31} - 1], [l_m, u_m] \sqcap [-2^{31}, u_i] \rangle \\ &\qquad \text{where } \langle [l_i, u_i], [l_m, u_m] \rangle = S'_2 \\ S'_4 &= \langle i, [min(l_m + 1, 2^{31} - 1), min(u_m + 1, 2^{31} - 1)] \rangle \\ &\qquad \text{where } \langle i, [l_m, u_m] \rangle = S'_3 \\ S'_5 &= \langle [l_i, u_i] \sqcap [-2^{31}, u_m - 1], [l_m, u_m] \sqcap [l_i + 1, 2^{31} - 1] \rangle \\ &\qquad \text{where } \langle [l_i, u_i], [l_m, u_m] \rangle = S'_2 \end{aligned}$$

[2]The exact operation can be inferred from the operand types.

5.3.3 Direct Calculation of the Abstract Semantics

Traditionally, the above abstract semantics would be solved to a fixpoint using Kleene iteration. As discussed in Chapter 2, this amounts to the repeated evaluation of the abstract semantic equations until the intervals stabilise. However, a widening operator is typically required to ensure fast termination in case long ascending chains are encountered. Whilst it is possible to find the least-fixpoint (the tightest hyper-rectangles) with widening, in general there is no guarantee that this will be the case. This is because widening may incur a further loss of precision, giving a post-fixpoint solution. A post-fixpoint is a sound but imprecise solution.

Alternatively, the best hyper rectangles can be found directly using mathematical optimisation. By this method there is no explicit iteration process, meaning that there is no need to specify a widening operator. Let $S'_1 = \langle [l_{i,1}, u_{i,1}], [l_{m,1}, u_{m,1}] \rangle \wedge \ldots \wedge S'_5 = \langle [l_{i,5}, u_{i,5}], [l_{m,5}, u_{m,5}] \rangle$, where each pair of intervals correspond to the possible values of `i` and `m` at a specific program point. The abstract semantics undergoes a transformation (detailed in the next section) so as to arrive at the following optimisation problem:

$$minimise \quad \sum_{j=1}^{5}(u_{i,j} - l_{i,j}) + \sum_{j=1}^{5}(u_{m,j} - l_{m,j}) \; s.t.$$

$$\begin{array}{rclcrcll}
l_{i,1} & = & -2^{31} & \wedge & u_{i,1} & = & 2^{31}-1 & \wedge \\
l_{i,2\star} & = & 10 & \wedge & u_{i,2\star} & = & 10 & \wedge \\
l_{i,2} & = & min(l_{i,2\star}, l_{i,4}) & \wedge & u_{i,2} & = & max(u_{i,2\star}, u_{i,4}) & \wedge \\
l_{i,3} & = & max(l_{i,2}, l_{m,2}) & \wedge & u_{i,3} & = & u_{i,2} & \wedge \\
l_{i,4} & = & l_{i,3} & \wedge & u_{i,4} & = & u_{i,3} & \wedge \\
l_{i,5} & = & l_{i,2} & \wedge & u_{i,5} & = & min(u_{i,2}, u_{m,2}-1) & \wedge \\
l_{m,1} & = & 5 & \wedge & u_{m,1} & = & 20 & \wedge \\
l_{m,2\star} & = & l_{m,1} & \wedge & u_{m,2\star} & = & u_{m,1} & \wedge \\
l_{m,2} & = & min(l_{m,2\star}, l_{m,4}) & \wedge & u_{m,2} & = & max(u_{m,2\star}, u_{m,4}) & \wedge \\
l_{m,3} & = & l_{m,2} & \wedge & u_{m,3} & = & min(u_{m,2}, u_{i,2}) & \wedge \\
l_{m,4} & = & min(l_{m,3}+1, 2^{31}-1) & \wedge & u_{m,4} & = & min(u_{m,3}+1, 2^{31}-1) & \wedge \\
l_{m,5} & = & max(l_{m,2}, l_{i,2}+1) & \wedge & u_{m,5} & = & u_{m,2} &
\end{array}$$

Figure 23: Optimisation problem for the worked example.

The constraint system is, for most part, composed of linear equalities. The

only exceptions to this are the min and max constraints which are used to realise control flow joins, conditional branching and saturating arithmetic. For example, the constraints $l_{i,2} = min(l_{i,2\star}, l_{i,4})$ and $u_{i,2} = max(u_{i,2\star}, u_{i,4})$ assert that $[l_{i,2}, u_{i,2}]$ is the smallest interval that encloses both $[l_{i,2\star}, u_{i,2\star}]$ and $[l_{i,4}, u_{i,4}]$. Likewise $l_{m,2} = min(l_{m,2\star}, l_{m,4})$ and $u_{m,2} = max(u_{m,2\star}, u_{m,4})$ assert tight bounds on m. In combination, these four constraints symbolically define S'_2 as the merge of the hyper-rectangles $S'_{2\star}$ and S'_4. Modelling the loop condition $i \geq m$ is a particular subtlety. Note how $l_{i,3} = max(l_{i,2}, l_{m,2})$ and $u_{i,3} = u_{i,3}$ strengthen (not weaken) the lower bound of i but preserve its upper bound. Conversely, $l_{m,3} = l_{m,2}$ and $u_{m,3} = min(u_{i,2}, u_{m,2})$ refine the upper bound of m but preserve its lower bound. An analogous construction is used to model the loop exit condition. The cost function asserts that the desired solution is the least (best) hyper-rectangle that satisfies all of the constraints.

By solving the above (with the technique outlined in Section 5.5) the following ranges are inferred, which characterise the least-fixpoint of the abstract semantics:

$$\begin{array}{lcl} S'_1 & = & \langle [-2^{31}, 2^{31}-1], [5, 20] \rangle \\ S'_2 & = & \langle [10, 10], [5, 20] \rangle \\ S'_3 & = & \langle [10, 10], [5, 10] \rangle \end{array} \qquad \begin{array}{lcl} S'_4 & = & \langle [10, 10], [6, 11] \rangle \\ S'_5 & = & \langle [10, 10], [11, 20] \rangle \end{array}$$

5.4 Deriving the Initial Optimisation Problem

As previously mentioned, the technique described in this chapter relies upon the generation of an optimisation problem. This optimisation problem is then used as a basis for finding the least-fixpoint of the abstract semantics. This section describes how the optimisation problem is derived.

Given a set of fixpoint equations defined over the abstract domain D^n, constraint generation begins by assigning each interval bound a symbolic name. For an interval representing the possible states of the variable v at program point p, the symbolic names $l_{v,p}$ and $u_{v,p}$ are introduced. For the worked example program, symbolic names are defined for the interval bounds of `i` and `m` at each program point such that: $S'_1 = \langle [l_{i,1}, u_{i,1}], [l_{m,1}, u_{m,1}] \rangle \wedge \ldots \wedge S'_5 = \langle [l_{i,5}, u_{i,5}], [l_{m,5}, u_{m,5}] \rangle$. Next, the abstract semantics is rewritten in terms of the symbolic names to arrive at a revised abstract semantics. The revised abstract semantics for the worked example program is shown in Figure 24.

$$
\begin{array}{rcl}
\langle [l_{i,1}, u_{i,1}], [l_{m,1}, u_{m,1}] \rangle = & \langle [-2^{31}, 2^{31}-1], [5, 20] \rangle \\
\langle [l_{i,2\star}, u_{i,2\star}], [l_{m,2\star}, u_{m,2\star}] \rangle = & \langle [10, 10], [l_{m,1}, u_{m,1}] \rangle \\
\langle [l_{i,2}, u_{i,2}], [l_{m,2}, u_{m,2}] \rangle = & \langle [l_{i,2\star}, u_{i,2\star}], [l_{m,2\star}, u_{m,2\star}] \rangle \sqcup \langle [l_{i,4}, u_{i,4}], [l_{m,4}, u_{m,4}] \rangle \\
\langle [l_{i,3}, u_{i,3}], [l_{m,3}, u_{m,3}] \rangle = & \langle [l_{i,2}, u_{i,2}] \sqcap [l_{m,2}, 2^{31}-1], [l_{m,2}, u_{m,2}] \sqcap [-2^{31}, u_{i,2}] \rangle \\
\langle [l_{i,4}, u_{i,4}], [l_{m,4}, u_{m,4}] \rangle = & \langle [l_{i,3}, u_{i,3}], [min(l_{m,3}+1, 2^{31}-1), min(u_{m,3}+1, 2^{31}-1)] \rangle \\
\langle [l_{i,5}, u_{i,5}], [l_{m,5}, u_{m,5}] \rangle = & \langle [l_{i,2}, u_{i,2}] \sqcap [-2^{31}, u_{m,2}-1], [l_{m,2}, u_{m,2}] \sqcap [l_{i,2}+1, 2^{31}-1] \rangle
\end{array}
$$

Figure 24: Revised abstract semantics for the worked example.

Each of the revised abstract semantic equations is then transformed into a conjunction of constraints. The transformation occurs in a syntactically driven manner as follows:

- For a semantic equation of the form:

$$\langle [l_1, u_1], \ldots, [l_n, u_n] \rangle = \langle [l'_1, u'_1], \ldots, [l'_n, u'_n] \rangle$$

  the following constraints are generated:

$$\bigwedge_{x=1}^{n} l_x = l'_x \wedge u_x = u'_x$$

- For a semantic equation of the form:

$$\langle [l_1, u_1], \ldots, [l_n, u_n] \rangle = \langle [l'_1, u'_1], \ldots, [l'_n, u'_n] \rangle \sqcup \langle [l''_1, u''_1], \ldots, [l''_n, u''_n] \rangle$$

  the following constraints are generated:

$$\bigwedge_{x=1}^{n} l_x = min(l'_x, l''_x) \wedge u_x = max(u'_x, u''_x)$$

- Finally, for a semantic equation of the form:

$$\langle [l_1, u_1], \ldots, [l_n, u_n] \rangle = \langle [l'_1, u'_1] \sqcap [l''_1, u''_1], \ldots, [l'_n, u'_n] \sqcap [l''_n, u''_n] \rangle$$

the following constraints are generated:

$$\bigwedge_{x=1}^{n} l_x = max(l'_x, l''_x) \wedge u_x = min(u'_x, u''_x)$$

By applying this transformation to the revised abstract semantics, a system of (non-linear) constraints is derived. The constraints are accompanied by an objective function to form an optimisation problem. The objective function aims to find the tightest possible interval bounds and thus the least solution corresponds with the least-fixpoint of the abstract semantics.

The optimisation problem for the worked example program is shown in Figure 23. Note that some constraints have been simplified. For example, the abstract semantic equation:

$$\langle [l_{i,3}, u_{i,3}], [l_{m,3}, u_{m,3}] \rangle = \langle [l_{i,2}, u_{i,2}] \sqcap [l_{m,2}, 2^{31} - 1], [l_{m,2}, u_{m,2}] \sqcap [-2^{31}, u_{i,2}] \rangle$$

is transformed into the following constraints:

$$\begin{aligned} l_{i,3} = max(l_{i,2}, l_{m,2}) \quad &\wedge \quad u_{i,3} = min(u_{i,2}, 2^{31} - 1) \\ l_{m,3} = max(l_{m,2}, -2^{31}) \quad &\wedge \quad u_{m,3} = min(u_{m,2}, u_{i,2}) \end{aligned}$$

Recall that each interval of the form $[l_{x,k}, u_{x,k}]$ describes the possible values of a 32-bit signed integer. Since the upper bound $u_{i,2}$ can be no greater than $2^{31} - 1$, and since the lower bound $l_{m,2}$ can be no less than -2^{31}, the above constraints collapse to:

$$\begin{aligned} l_{i,3} = max(l_{i,2}, l_{m,2}) \quad &\wedge \quad u_{i,3} = u_{i,2} \\ l_{m,3} = l_{m,2} \quad &\wedge \quad u_{m,3} = min(u_{m,2}, u_{i,2}) \end{aligned}$$

5.5 Solving Minimum and Maximum Constraints

The *min* and *max* terms in the initial optimisation problem are non-convex, yet convexity is a prerequisite of classical linear programming. It is, however, possible to solve the non-linear constraint system through repeated linear programming. This strategy is accompanied by heuristics to improve solving performance. In this section, the solving strategy is described in detail.

5.5.1 Constraint Decomposition

The first stage in solving the non-linear constraint system involves decomposing the *min* and *max* constraints using the following equivalences:

$$\begin{aligned} x = min(y,z) &\equiv \quad (x \leq y) \wedge (x \leq z) \wedge (x = y \vee x = z) \\ x = max(y,z) &\equiv \quad (x \geq y) \wedge (x \geq z) \wedge (x = y \vee x = z) \end{aligned}$$

It should be clear that these equivalences are correct. Take the definition of *min* for example. The result should be less than or equal to both of the operands, but it must equal one of them. A similar argument holds for *max*.

After the decomposition of the *min* and *max* terms, the constraint system is partitioned between a set, L, of linear constraints and a vector, $\boldsymbol{C}$, of complementary constraints. As the name might suggest, the complementary constraints are disjunctive. Specifically a complementary constraint takes the form $(a = b) \vee (c = d)$, where each of a, b, c and d are linear terms involving optimisation variables and constants. The remaining linear constraints are added to L.

Example 4 (Constraint decomposition)**.** *The constraint* $u_{m,3} = \min(u_{i,2}, u_{m,2})$ *is decomposed into the linear system* $L = \{u_{m,3} \leq u_{i,2}, u_{m,3} \leq u_{m,2}\}$ *and the complementary constraints* $\boldsymbol{C} = \langle (u_{m,3} = u_{i,2} \vee u_{m,3} = u_{m,2}) \rangle$.

The decomposed constraints for the worked example are as follows. L is a set containing the following linear constraints:

$$\begin{array}{rclrclrclrcl}
l_{i,1} & = & -2^{32} & u_{i,1} & = & 2^{31}-1 & & & & & & \\
l_{i,2\star} & = & 10 & u_{i,2\star} & = & 10 & & & & & & \\
l_{i,2} & \leq & l_{i,2\star} & l_{i,2} & \leq & l_{i,4} & u_{i,2} & \geq & u_{i,2\star} & u_{i,2} & \geq & u_{i,4} \\
l_{i,3} & \geq & l_{i,2} & l_{i,3} & \geq & l_{m,2} & u_{i,3} & = & u_{i,2} & & & \\
l_{i,4} & = & l_{i,3} & u_{i,4} & = & u_{i,3} & & & & & & \\
l_{i,5} & = & l_{i,2} & u_{i,5} & \leq & u_{i,2} & u_{i,5} & \leq & u_{m,2}-1 & & & \\
l_{m,1} & = & 5 & u_{m,1} & = & 20 & & & & & & \\
l_{m,2\star} & = & l_{m,1} & u_{m,2\star} & = & u_{m,1} & & & & & & \\
l_{m,2} & \leq & l_{m,2\star} & l_{m,2} & \leq & l_{m,4} & u_{m,2} & \geq & u_{m,2\star} & u_{m,2} & \geq & u_{m,4} \\
l_{m,3} & = & l_{m,2} & u_{m,3} & \leq & u_{i,2} & u_{m,3} & \leq & u_{m,2} & & & \\
l_{m,4} & \leq & l_{m,3}+1 & l_{m,4} & \leq & 2^{31}-1 & u_{m,4} & \leq & u_{m,3}+1 & u_{m,4} & \leq & 2^{31}-1 \\
l_{m,5} & \geq & l_{m,2} & l_{m,5} & \geq & l_{i,2}+1 & u_{m,5} & = & u_{m,2} & & &
\end{array}$$

and $\boldsymbol{C}$ is a vector containing the following complementary constraints:

$$\begin{array}{rcl} (l_{i,2} = l_{i,2\star} & \vee & l_{i,2} = l_{i,4}) \\ (l_{i,3} = l_{i,2} & \vee & l_{i,3} = l_{m,2}) \\ (l_{m,2} = l_{m,2\star} & \vee & l_{m,2} = l_{m,4}) \\ (u_{m,3} = u_{i,2} & \vee & u_{m,3} = u_{m,2}) \\ (u_{m,4} = u_{m,3} + 1 & \vee & u_{m,4} = 2^{31} - 1) \end{array} \qquad \begin{array}{rcl} (u_{i,2} = u_{i,2\star} & \vee & u_{i,2} = u_{i,4}) \\ (u_{i,5} = u_{i,2} & \vee & u_{i,5} = u_{m,2} - 1) \\ (u_{m,2} = u_{m,2\star} & \vee & u_{m,2} = u_{m,4}) \\ (l_{m,4} = l_{m,3} + 1 & \vee & l_{m,4} = 2^{31} - 1) \\ (l_{m,5} = l_{m,2} & \vee & l_{m,5} = l_{i,2} + 1) \end{array}$$

5.5.2 Constraint Solving

Although the non-linear disjunctions of $\boldsymbol{C}$ disallow conventional linear programming solvers from being directly applied, the optimal solution can be found through repeated solving of smaller linear constraint systems. To see this, observe that a complementary constraint $(a = b) \vee (c = d)$ has one of two states, according to whether the first or the second equality holds. Therefore there are $2^{|\boldsymbol{C}|}$ possible combinations under which the disjunctions of $\boldsymbol{C}$ may be satisfied. The optimal solution to the non-linear constraint system could, in principle, be found through solving $2^{|\boldsymbol{C}|}$ smaller linear programs, each consisting of the linear constraints of L, augmented with one combination under which $\boldsymbol{C}$ may be satisfied. By this method, solving the worked example would amount to solving 1024 LPs:

$$\begin{array}{l} \mathit{minimise}(F)\ \mathit{s.t.}\ L \wedge \boldsymbol{C}_1^l \wedge \boldsymbol{C}_2^l \wedge \boldsymbol{C}_3^l \wedge \boldsymbol{C}_4^l \wedge \boldsymbol{C}_5^l \wedge \boldsymbol{C}_6^l \wedge \boldsymbol{C}_7^l \wedge \boldsymbol{C}_8^l \wedge \boldsymbol{C}_9^l \wedge \boldsymbol{C}_{10}^l \\ \mathit{minimise}(F)\ \mathit{s.t.}\ L \wedge \boldsymbol{C}_1^l \wedge \boldsymbol{C}_2^l \wedge \boldsymbol{C}_3^l \wedge \boldsymbol{C}_4^l \wedge \boldsymbol{C}_5^l \wedge \boldsymbol{C}_6^l \wedge \boldsymbol{C}_7^l \wedge \boldsymbol{C}_8^l \wedge \boldsymbol{C}_9^l \wedge \underline{\boldsymbol{C}_{10}^r} \\ \mathit{minimise}(F)\ \mathit{s.t.}\ L \wedge \boldsymbol{C}_1^l \wedge \boldsymbol{C}_2^l \wedge \boldsymbol{C}_3^l \wedge \boldsymbol{C}_4^l \wedge \boldsymbol{C}_5^l \wedge \boldsymbol{C}_6^l \wedge \boldsymbol{C}_7^l \wedge \boldsymbol{C}_8^l \wedge \underline{\boldsymbol{C}_9^r} \wedge \boldsymbol{C}_{10}^l \\ \mathit{minimise}(F)\ \mathit{s.t.}\ L \wedge \boldsymbol{C}_1^l \wedge \boldsymbol{C}_2^l \wedge \boldsymbol{C}_3^l \wedge \boldsymbol{C}_4^l \wedge \boldsymbol{C}_5^l \wedge \boldsymbol{C}_6^l \wedge \boldsymbol{C}_7^l \wedge \boldsymbol{C}_8^l \wedge \underline{\boldsymbol{C}_9^r} \wedge \underline{\boldsymbol{C}_{10}^r} \\ \vdots \\ \mathit{minimise}(F)\ \mathit{s.t.}\ L \wedge \underline{\boldsymbol{C}_1^r} \wedge \underline{\boldsymbol{C}_2^r} \wedge \underline{\boldsymbol{C}_3^r} \wedge \underline{\boldsymbol{C}_4^r} \wedge \underline{\boldsymbol{C}_5^r} \wedge \underline{\boldsymbol{C}_6^r} \wedge \underline{\boldsymbol{C}_7^r} \wedge \underline{\boldsymbol{C}_8^r} \wedge \underline{\boldsymbol{C}_9^r} \wedge \boldsymbol{C}_{10}^l \\ \mathit{minimise}(F)\ \mathit{s.t.}\ L \wedge \underline{\boldsymbol{C}_1^r} \wedge \underline{\boldsymbol{C}_2^r} \wedge \underline{\boldsymbol{C}_3^r} \wedge \underline{\boldsymbol{C}_4^r} \wedge \underline{\boldsymbol{C}_5^r} \wedge \underline{\boldsymbol{C}_6^r} \wedge \underline{\boldsymbol{C}_7^r} \wedge \underline{\boldsymbol{C}_8^r} \wedge \underline{\boldsymbol{C}_9^r} \wedge \underline{\boldsymbol{C}_{10}^r} \end{array}$$

where F is the objective function finding the tightest bounds and where $\boldsymbol{C}_i^l$ and $\boldsymbol{C}_i^r$ refer to the left and right equality of the i^{th} complementary constraint. The right hand equalities are underlined in the above to aid reader comprehension. Each of the above linear programs could be solved independently to obtain a vector of solutions. A solution is either:

- A pair $\langle o, n \rangle$, where o is the objective function value and n is a mapping from interval bounds to their assignments.

- or $\perp$, meaning that the LP was INFEASIBLE.

The solution $s_i \neq \perp$ whose cost function gives the overall least value would be selected, as this corresponds to the optimal solution of the overarching non-linear constraint system. However, since the number of LPs that must be solved by this method grows exponentially with the number of complementary constraints in $\boldsymbol{C}$, an alternative strategy is suggested.

The search space can be thought of as a binary decision tree, where the edges represent the selection of either the left or right equality of a complementary constraint. Under this model, each path from the root node to a leaf node represents a sequence of decisions which may satisfy the complementary constraints. The proposed method walks the tree in a depth first fashion, at each node solving a linear relaxation. The boilerplate of the algorithm is shown in Algorithm 6.

Before the algorithm commences, L is augmented with constraints to ensure that each interval bound is within the feasible integer range. This is required because the complementary constraints normally provide the necessary bounding. By relaxing these constraints, bounding is no longer guaranteed. In the case of the worked example, each interval bound, $b_i \in \{l_{i,1}, u_{i,1}, l_{m,1}, u_{m,1}, \ldots\}$, is limited to the range of signed 32-bit integers such that $-2^{31} \leq b_i \leq 2^{31} - 1$. This augmented system will henceforth be denoted $\bar{L}$.

The search starts at the root node of the tree with $\tau =$TRUE. Each node of the tree equates to a linear program, $\bar{L} \wedge \tau$, which is tested for satisfiability with a solver. τ is the conjunction of equalities selected so far from $\boldsymbol{C}$ (as illustrated in Figure 25). If $\bar{L} \wedge \tau$ is INFEASIBLE, then there is no solution for this choice of τ. Furthermore, the search down the current branch of the tree is aborted, therefore pruning the search space. Searching deeper would only augment τ with additional equalities from $\boldsymbol{C}$, so if $\bar{L} \wedge \tau$ is already INFEASIBLE, then constraining it further will make no difference. If, on the other hand, $\bar{L} \wedge \tau$ is satisfiable, then another equality is selected from $\boldsymbol{C}$ from a disjunction that has not already been considered. This is the role of CHOOSENEXTDECISION. The selection of a new equality corresponds to the transition from the current node to another, one level deeper in the tree. If exactly one equality has been selected from each disjunction of $\boldsymbol{C}$ and $\bar{L} \wedge \tau$ is still feasible (i.e. the LP at a leaf node is satisfiable), then a solution is recorded. The search continues exploring the tree in this way until the search space is exhausted, at which point the solution with the least

objective function value is reported as the overall minimum. The assignment to the interval bounds in the least solution characterises the least-fixpoint of the abstract semantics.

Algorithm 6 Binary search algorithm.

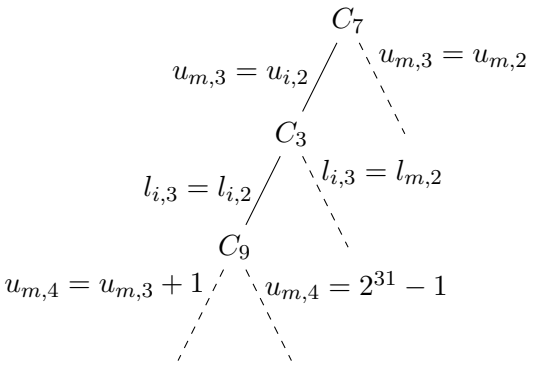
```
1: function BinSearch(L̄, F, C, τ)
2:     r ← MinimizeLp(F, L̄ ∧ τ)
3:     if ¬ Sat(r) then
4:         return []                                  ▷ No solutions here, prune.
5:     else if AllDecisionsMade(C, τ) then
6:         return [(r, τ)]                            ▷ Found a leaf with a solution.
7:     end if
8:     (e1 ∨ e2) ←ChooseNextDecision(C, r, τ)
9:     sl ← BinSearch(L̄, F, C, τ ∧ e1)
10:    sr ← BinSearch(L̄, F, C, τ ∧ e2)
11:    return Append(sl, sr)
12: end function
```

The benefit of this strategy is that if inconsistency is detected when τ contains relatively few equalities from $\boldsymbol{C}$, then many branches through the search space can be dismissed simultaneously. The effectiveness of this pruning strategy is dependent upon the ordering of decisions, and in particular the equalities that are selected from $\boldsymbol{C}$. For the search to be effective, inconsistencies need to be found early in the search, at a shallow depth in the tree. The earlier inconsistencies are found, the earlier pruning can occur, and therefore the more potential solutions can be ruled out at once. If, on the other hand, an inconsistency is found later, then it is likely to be duplicated down alternative paths, nullifying the effect of pruning. Like many combinatoric search problems, the worst case complexity is high; the worst case number of linear programs is $2^{|\boldsymbol{C}|+1} - 1$. In practice performance can be significantly improved with the use of heuristics.

5.5.3 Heuristics

In order to improve upon the worst case complexity of the search space, the following heuristics are deployed:

H1: Prune Inconsistencies Early. This heuristic suggests which disjunction $C_n \in \boldsymbol{C}$ is a good candidate from which an equality should be selected. Suppose



$\bar{L}_x$	Relaxation ($\bar{L} \wedge \tau$)
1	$\bar{L}$
2	$\bar{L} \wedge (u_{m,3} = u_{i,2})$
3	$\bar{L} \wedge (u_{m,3} = u_{i,2}) \wedge (l_{i,3} = l_{i,2})$
⋮	⋮

Figure 25: First three linear relaxations of the worked example program.

solving $\bar{L} \wedge \tau$ returns a solution for which $(u_{m,3} = -2^{31}) \wedge (u_{i,2} = 10) \wedge (u_{m,2} = 20)$. Observe that the disjunction $C_7 = (u_{m,3} = u_{i,2} \vee u_{m,3} = u_{m,2})$ is unsatisfiable under this assignment. Then the heuristic suggests that τ should next be extended with an equality from C_7. The intuition behind this selection strategy is that if C_7 is unsatisfiable for the current solution to $\bar{L} \wedge \tau$, then extending τ with one of the equalities of $\boldsymbol{C}_7$ is likely to detect an inconsistency at the next node, thereby pruning the search space earlier, rather than later. If no complementary constraint can be found which is unsatisfied by the current solution to $\bar{L} \wedge \tau$, then an arbitrary $C_n \in \boldsymbol{C}$ is chosen from which the next equality is selected. The selected C_n is literally chosen at random, thus introducing non-determinism into the algorithm. The proposed definition of ChooseNextDecision, which implements H1, is shown in Algorithm 7.

Algorithm 7 Heuristic 1

```
1: function ChooseNextDecision(C, r, τ)
2:     v ← GetViolatedCCs(C, τ)
3:     if |v| > 0 then
4:         let n ∈ v
5:     else
6:         n ← ChooseArbitraryNextDecision(C, r)
7:     end if
8:     return n
9: end function
```

H2: Block Weak and Duplicate Solutions. A solution is found once an equality is selected from each disjunction of $\boldsymbol{C}$ such that $\bar{L} \wedge \tau$ remains satisfiable. In other words, a solution is found when the LP at a leaf node of the tree is satisfiable. Note that the first solution found may not be the minimal solution. There may be many solutions which, whilst they satisfy the min/max constraints, yield imprecise intervals compared to those of the least solution. For this reason, ideally the search space should be exhausted, although the search could be stopped early and the current best solution can be taken as a post-fixpoint. It is also possible for both equalities of a complementary constraint to evaluate true, e.g. $(l_{i,3} = l_{i,2} \vee l_{i,3} = l_{m,2})$ where $l_{i,2} = l_{i,3} = l_{m,2} = 1$. In this case ineffectual decisions exist in the search space, meaning that duplicate solutions must also exist.

There is no value in finding a duplicate solution or a solution which is weaker than another already found, so the addition of an extra linear constraint is proposed. The extra constraint ensures that any solution that is subsequently found is strictly better than any solution already known. Suppose that, during the solving of the worked example, a solution is found whose objective function value is o_{min}. A constraint is added which literally asserts that the value of the objective function is strictly less than o_{min}. Therefore, subsequent linear programs are solved under the additional constraint : $\sum_{j=1}^{5}(u_{i,j} - l_{i,j}) + \sum_{j=1}^{5}(u_{m,j} - l_{m,j}) < o_{min}$. Through this construction, only solutions yielding a strictly smaller objective are feasible. This has the effect of further pruning the search space and in turn minimising the number of LPs that must be solved overall.

5.6 Experimental Results

Tooling was developed which, given a control flow graph and a description of CFG edge operations, generates $\bar{L}$ and $\boldsymbol{C}$, before proceeding with the binary search as described in the previous section. Linear programs were solved by the open-source LpSolve solver interfaced using Python language bindings. The individual search heuristics, H1 and H2, may be switched on and off, allowing performance comparisons to be drawn under different heuristics configurations. Evaluation of the complementary constraints (for heuristic 1) was performed by SymPy, a computer algebra library for Python. Experiments were run on a 3GHz 64-bit Intel machine with 4GB of RAM and running OpenBSD.

Interval	$m_1 \in$		
	$[2,2]$	$[63,71]$	$[5,20]$
m_{2*}	$[2,2]$	$[63,71]$	$[5,20]$
m_2	$[2,11]$	$[63,71]$	$[5,20]$
m_3	$[2,10]$	$[63,10]$	$[5,10]$
m_4	$[3,11]$	$[64,11]$	$[6,11]$
m_5	$[11,11]$	$[63,71]$	$[11,20]$
i_1	$[0,255]$	$[0,255]$	$[0,255]$
i_{2*}	$[10,10]$	$[10,10]$	$[10,10]$
i_2	$[10,10]$	$[10,10]$	$[10,10]$
i_3	$[10,10]$	$[63,10]$	$[10,10]$
i_4	$[10,10]$	$[63,10]$	$[10,10]$
i_5	$[10,10]$	$[10,10]$	$[10,10]$

(a) Intervals determined.

$m_1 \in$	H1	H2	Mean #LPs	Mean Time (s)
$[2,2]$	✗	✗	208	0.2
	✓	✗	183	0.9
	✗	✓	152	0.1
	✓	✓	38	0.2
$[63,71]$	✗	✗	200	0.2
	✓	✗	125	0.6
	✗	✓	105	0.1
	✓	✓	45	0.2
$[5,20]$	✗	✗	207	0.2
	✓	✗	211	1.0
	✗	✓	143	0.1
	✓	✓	44	0.2

(b) Avg. # of LPs required, Avg. times.

Figure 26: Experimental results for the worked example (Listing 5.1).

The tables in Figure 26 show some experimental results for the worked example (Listing 5.1) under different initial values of m_1 and under different heuristics configurations. Because the algorithm is non-deterministic, each experiment configuration was run 10 times and averages were taken. The tables show the intervals inferred, the mean number of linear programs required (out of a possible worst case number of $2^{10+1} - 1 = 2047$) to find the best solution and (for indicative purposes only) the average amount of time spent finding the solution (in seconds). The intervals of the best solution are precise and, in all cases, the heuristics reduced the number of LPs required to find the best solution. Further, when $m_1 \in [63, 71]$, the loop body is not entered and this is reflected in the results by the empty intervals at program points 3 and 4. Interestingly, run-times appear to be longer when heuristic 1 is enabled. Profiling revealed that the evaluation of complementary constraints (GetViolatedCCs) accounts for a large portion of solving time for this small example. This can probably be attributed to the symbolic evaluation of equalities in an interpreted runtime system (i.e. Python). This subsystem could doubtlessly be optimised, but this is out of the scope of this chapter.

A second program was analysed, this time at the binary level. Listing 5.2 shows the disassembly of a defective implementation of `memcpy(3)` for the x86-64 architecture. As operands, the function takes a pointer to a destination buffer (`rdi`), a pointer to a source buffer (`rsi`) and a length argument (`rdx`). The `r15` register is used as both a loop counter and as an index into the source and destination buffers. The program is defective because it always copies one too

rdx ∈	$r15_1$	H1	H2	MLP	MT
[8, 8]	[0, 8]	✗	✗	25143	75
		✓	✗	18596	178
		✗	✓	11940	45
		✓	✓	69	1
[8, 4096]	[0, 4096]	✗	✗	31045	116
		✓	✗	18963	198
		✗	✓	8989	45
		✓	✓	62	1
[31, 66]	[0, 66]	✗	✗	28639	107
		✓	✗	18963	194
		✗	✓	13885	55
		✓	✓	68	1

(a) Second set of experiments.

rdi ∈	rax_2	H1	H2	MLP	MT
[8, 8]	[1, 8]	✗	✗	36621	219
		✓	✗	20342	302
		✗	✓	7891	34
		✓	✓	85	1
[7, 13]	[1, 13]	✗	✗	35856	151
		✓	✗	19977	258
		✗	✓	8701	37
		✓	✓	99	1
[4, 128]	[1, 128]	✗	✗	40352	166
		✗	✓	19696	252
		✓	✗	7948	34
		✓	✓	105	1

(b) Third set of experiments.

Figure 27: Results for the second and third experiments (MLP is the mean number of linear programs required and MT is the mean time in seconds).

many bytes, thus overshooting the destination buffer. The `jg` instruction on the third line should have been a `jge` instruction.

Let $r15_1 \in [l_{r15,1}, u_{r15,1}]$ be the interval representing `r15` at the program point marked `p1`, where bytes are written into the destination buffer. In order to model the conditional behaviours of binary programs, high-level predicates must be extracted from sequences of assembler instructions. Typically two instructions are involved; one which defines the CPU status flags and another which reads the status flags [7]. For example, ⟨`cmp r15, rdx; jg return`⟩ causes a control flow despatch if `r15` > `rdx`. This inference is performed statically prior to constraint generation.

```
memcpy: xor r15, r15                 # loop counter
loop:   cmp r15, rdx
        jg return
        mov byte ptr cl, [rsi+r15]   # read out src
p1:     mov byte ptr [rdi+r15], cl   # write in dest
        inc r15
        jmp loop
return: mov rax, rdi                 # return ptr to dest
        ret
```

Listing 5.2: A function to copy buffers.

Figure 27a shows the results of the analysis upon the `memcpy(3)` implementation. If a buffer size of between 8 and 4096 is passed to this function, then the analysis indeed infers $r15_1 \in [0, 4096]$. This would suggest that one byte is potentially written outside of the allocated buffer. Again, the number of LPs the analysis is required to solve is improved through the use of heuristics. Evaluation of complementary constraints is especially expensive when heuristic 1 alone is enabled, but the overall time spent searching is vastly improved through the use of heuristics 1 and 2 combined. This experiment utilises 18 complementary constraints, so the theoretical worst case number of LPs required is $2^{18+1}-1 = 524287$.

Listing 5.3 shows an algorithm to byte-swap 16-bit words in a memory buffer. As operands, the function takes a buffer length (`rdi`) and a pointer to a buffer to swap (`rsi`). The register `rax` is being used as an index into the buffer pointed to by `rsi`. The program points of interest are marked `p1` and `p2`; these points are where the buffer contents are written. Let $rax_1 \in [l_{rax,1}, u_{rax,1}]$ and $rax_2 \in [l_{rax,2}, u_{rax,2}]$ be the intervals of `rax` at `p1` and `p2` respectively. The results of the analysis of this program (Figure 27b) highlight an interesting deficiency in the analysis. If the function is called with an odd buffer size argument, then the function indeed writes one byte outside its allocated buffer, this is reflected by the experimental results. Yet, if a buffer size argument of 8 is passed to the analysis, then it infers $rax_2 \in [1, 8]$. This would suggest that a byte was written outside of the allocated buffer, however, in reality this is untrue. Due to the approximation introduced by the interval domain, the analysis is unable to take into account the strided nature of the loop counter. As a consequence the analysis over-approximates the upper bound of `rax` upon entry to the loop. Nevertheless, the solution is sound (it safely over-approximates all possible register values). Further, the solution is found quickly and in a fraction of the worst case number of linear programs ($2^{22+1} - 1 = 8388607$).

5.6.1 Comparison with Kleene Iteration

To reflect further upon the feasibility of the analysis described in this chapter, it may be beneficial to compare its performance with that of traditional Kleene iteration. Although such a comparison is not offered here, it should be noted that making this comparison may not be as straightforward as one might expect.

```
endswap: xor r15, r15   # loop counter
loop:    cmp r15, rdi
         jge return
         mov rax, r15   # rax is used as a write index
         mov byte ptr dl, [rsi+r15]
         inc r15
         mov byte ptr cl, [rsi+r15]
         inc r15
p1:      mov byte ptr [rsi+rax], cl
         inc rax
p2:      mov byte ptr [rsi+rax], dl
         jmp loop
return:  ret
```

Listing 5.3: A 16-bit byte swap.

Perhaps the most obvious approach would be to compare how long it takes each method to find a least-fixpoint solution. The linear programming method guarantees that the solution found is the least-fixpoint, and the same can be said for Kleene iteration just so long as no fixpoint acceleration technique is used. However, this raises an important issue; that typically when using Kleene iteration, fixpoint acceleration techniques are often required for a tractable analysis. Furthermore, by deploying fixpoint acceleration, the precision of the analysis may be compromised (a post-fixpoint may be found).

An approach which may be more insightful therefore, may be to compare the new approach with several runs of Kleene iteration with varying widening thresholds. Inevitably, a range of post-fixpoint solutions would be found. To rank the quality of each of these solutions, one could devise a fitness function which takes into account: a) the precision of the solution (how far it deviates from the least-fixpoint determined by linear programming), and b) the time taken to find the solution. Of course, this then raises the question of what a suitable fitness function could be.

5.7 Discussion

The analysis presented in this paper was mostly inspired by the pioneering work by Rugina et al. [99]. The extension to Rugina's work diverges in some aspects in

response to shortcomings that are not mentioned in the literature. In this section, a brief discussion of these shortcomings is offered.

5.7.1 Conditional Semantics

It would appear that Rugina's branching semantics are unable to model a class of loop constructs correctly. One such example is the program:

```
B1: int i = 10;
B2: while (i >= m)
B3:         m := m + 1;
    endwhile
```

Listing 5.4: A program which may not be analysed by Rugina's method.

Following Rugina's constraint generation scheme, the program is reduced to the following constraint system, which is infeasible:

$$
\begin{array}{lllllllll}
l_{i,2} \leq 10 & \wedge & 10 \leq u_{i,2} & \wedge & l_{m,2} \leq l_{m,1} & \wedge & u_{m,1} \leq u_{m,2} & \wedge \\
l_{i,3} \leq l_{m,2} & \wedge & u_{i,2} \leq u_{i,3} & \wedge & l_{m,3} \leq l_{m,2} & \wedge & \mathbf{u_{m,2} \leq u_{m,3}} & \wedge \\
l_{i,2} \leq l_{i,3} & \wedge & u_{i,3} \leq u_{i,2} & \wedge & l_{m,2} \leq l_{m,3} + 1 & \wedge & \mathbf{u_{m,3} + 1 \leq u_{m,2}} &
\end{array}
$$

The constraints set in bold are inconsistent. Since the constraint system is infeasible, no interval bounds may be inferred. This means that the results of the analysis are inconclusive. The construction using *min* and *max* constraints remedies this shortcoming, meaning that the analysis always yields conclusive results.

5.7.2 Junk propagation

In both the method proposed in this chapter and in Rugina's analysis, unreachable code manifests itself as empty intervals (i.e. intervals of the form $[l, u]$, where $l > u$). The empty interval in the abstract domain corresponds to the empty set in the concrete domain. Consider the following program snippet, in which B_3 is unreachable:

```
B1: int i = 12;
B2: while (i <= 10)
B3:        i := i + 1;
    endwhile
```

Listing 5.5: Program demonstrating junk propogation.

Analysis of this program via Rugina's method, infers the following intervals: $i_2 \in [12, 15], i_3 \in [12, 10]$. The interval at i_3 is empty, correctly indicating that this program point is unreachable. Unfortunately, the loop's back-arc propagates bounding information from B_3 back to B_2, thereby compromising the precision of the upper bound of i_2. We refer to this phenomenon as "junk propagation". As far as is known at the time of writing, junk propagation is not mentioned in the literature.

The method proposed earlier in this chapter overcomes imprecisions incurred through junk propagation by treating the false branch of the loop check (or of any conditional for that matter) as a conditional whose predicate is the negation of that of the true branch. For the above counter-example, a loop exit block, B_4, would be introduced which is connected to B_2 by a conditional edge asserting that $i > 11$. Through this construction, the precision of the upper bound of i_2 is retained.

5.8 Chapter Summary

The method presented in this chapter has shown how range analysis can be computed, not as the solution to a system of recursive equations, but as the solution of a system of constraints over *min* and *max* expressions. Such constraints can be reduced to a system of linear constraints, augmented with complementary constraints, and thus solved by repeated linear programming. The method can be implemented with an off-the-shelf linear programming package, which can be used as a black-box. Furthermore, the number of calls to the solver can be reduced by using search heuristics. The result is an analysis that does not depend on classical fixpoint acceleration methods such as widening, since the least-fixpoint is found directly. It was shown that by using the range information inferred by

the analysis, it is possible to identify possible buffer overflow vulnerabilities in binaries. Further, the experimental results suggest that the analysis would perform adequately for deployment in, for example, an automated disassembler.

The approach is limited by the fact that it cannot take into account integer overflow scenarios. Integer overflows are often the source of subtle software deficiencies, so it is important that these scenarios are taken into consideration. The saturating addition semantics that were shown earlier in this chapter are, strictly speaking, unsound (for general purpose CPU architectures). Unfortunately, it is not trivial to model the overflow scenarios in a linear program because to do so requires modular arithmetic. By nature, modular arithmetic is non-linear. The following chapter shows that, whilst it is difficult to faithfully approximate these modulo behaviours with LP, it is not impossible with a mixed-integer linear program (MILP).

On a related note, it may be possible to solve non-linear constraint systems implementing *min* and *max* constraints with policy iteration. By such a method, the choice of the left or right inequality from a complementary constraint could form the basis for policy selection. Such an approach would, of course, re-introduce the need for an iteration strategy and widening, but may yield valuable insights nevertheless. See Section 7.4 for a discussion on policy iteration.

Chapter 6

Modelling Integer Overflow with MILP

6.1 Introduction

In the last chapter, a method was presented that allows range analysis to be formulated as a series of linear optimisation problems. The method shows promise; it is conceptually simple and experimental results suggest that, in terms of performance, it would be feasible for industrial application.

Yet, in its current state, the method fails to model integer wrapping scenarios, i.e. integer overflow and underflow. For example, consider the x86-64 instruction $\langle$`add rax,` c$\rangle$, where c is assumed to be a positive integer constant. The instruction simply adds c to the current value held in `rax`, storing the result back in `rax`. The `rax` register itself is a general purpose 64-bit register, meaning that it can express unsigned values between 0 and $2^{64} - 1$, or it may express signed values between -2^{63} and $2^{63} - 1$. Actually, registers have no notion of signedness and merely store bytes that may be interpreted in either way. An overflow (or underflow) occurs when an arithmetic operation causes a register value to fall outside of what is expressible. In the case of the above example, if `rax` is interpreted as unsigned, then an overflow occurs when the result of adding c to `rax` exceeds $2^{64} - 1$. Similarly, for a signed interpretation of `rax`, if $rax + c > 2^{63} - 1$, then an overflow occurs. Note that, on most CPUs, when computing the result of the addition, the `add` instruction is oblivious to the two numeric interpretations of

`rax`. The two's complement encoding means that the result is correct for both signed and unsigned interpretations.

Exactly what happens when an integer overflow occurs is defined by the underlying CPU architecture, however it is usually the case that the result literally wraps, as though it were a modulo system (whilst this is the norm for general purpose CPU architectures, note that other overflow models do exist, such as saturating arithmetic, as used in digital signal processing [73]). For example, if `rax` holds the unsigned number $2^{64}-1$, and then the CPU executes $\langle$`add rax, 1`$\rangle$, the resulting value of `rax` is 0. Similarly, the signed value $2^{63}-1$ would wrap to -2^{63} when incremented. This poses a problem for the method presented in the previous chapter. Wrapping means that arithmetic operations cannot be expressed as a convex linear function and thus cannot be encoded directly as linear constraints. Furthermore, it is not clear how wrapping operations can be expressed by min/max constraints, meaning that the method from the previous chapter does not help to overcome this problem. Observe however, that the potential outcomes of the operation can, in fact, be described by a piecewise linear function:

$$add_{u64}(rax, c) = \begin{cases} rax + c & \text{if } rax + c \leq 2^{64-1} \\ rax + c - 2^{64} & \text{otherwise} \end{cases}$$

The method described in this chapter shows that it is possible to model functions of this form as a mixed-integer linear program (MILP). The approach relies on the introduction of Boolean decision variables to encode the problem. The use of decision variables was inspired by Goubault et al. [58], who use decision variables to encode reachability in range analyses. Unlike their approach, however, the method described here emphasises the ability to model integer wrapping scenarios with as little precision loss as possible. To summarise, the contributions of this chapter are as follows:

- An extension to the method presented in the previous chapter is shown, which uses Boolean decision variables to model the effects of integer wrapping in the interval analysis.
- It is shown that control flow reachability and min/max constraints can be recast using decision variables too, thus sidestepping the need for repeated solving of linear programs.

- It is shown that by inferring register signedness a priori, the number of optimisation variables required can be reduced to improve performance. Experimental results are shown to support this claim.

The rest of this chapter is arranged as follows. Section 6.2 shows a worked example demonstrating the fundamental analysis, then Section 6.3 presents the technical aspects underpinning the modelling of integer wraps. Section 6.4 demonstrates how to model control flow and reachability using decision variables, then Section 6.5 shows how to extend the analysis to efficiently model mixed signedness. In Section 6.6 the performance of the analysis is evaluated, before Section 6.7 ends the chapter with some concluding remarks.

6.2 Worked Example

In this section, an overview of the proposed analysis is presented through the use of a worked example. In later sections, the more technical aspects of the analysis are described.

6.2.1 Collecting Semantics

Figure 28 shows the control flow graph (CFG) and the disassembly of an x86-64 function which shall serve as a worked example. The function dynamically allocates a memory buffer for storage of a variable-size UTF-32 string and returns a pointer to the buffer. The function accepts, as its sole argument, the number of characters for which space should be allocated. This argument is passed through the 64-bit `rdi` register. Since UTF-32 characters are of fixed size (four bytes), the function computes the size of the storage buffer by multiplying `rdi` by four and then adding a further four to accommodate a UTF-32 `null` sentinel. Once the buffer size has been computed, the buffer is allocated with a call to the system's memory allocator, `malloc(3)`, and the resulting pointer is checked in case of an allocation failure. If the operating system was unable to allocate the requested number of bytes, then the result of this function call is a `null` pointer, which is in turn passed back to the callee. Assuming that this is not the case and that `malloc(3)` succeeds, the remainder of the function initialises the allocated buffer by filling it with zeroes. This is accomplished with a loop that writes double-word

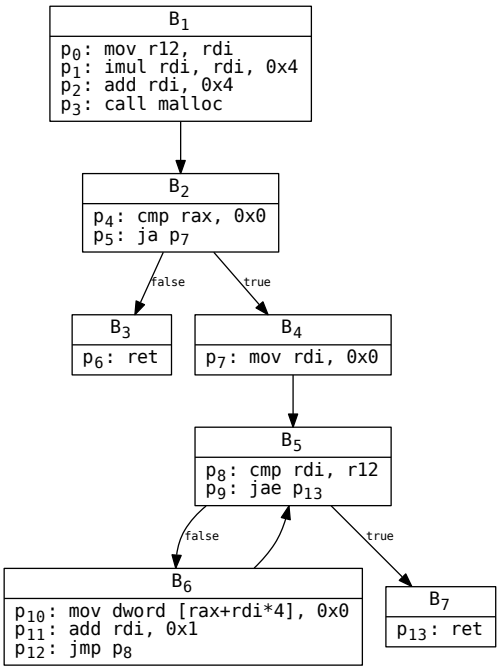


Figure 28: Allocating a UTF-32 string buffer.

zeroes (four `0x00` bytes) into each UTF-32 character slot in the buffer. Finally, a pointer to the allocated buffer is returned to the callee. For now, assume that all registers are to be interpreted as unsigned integers. The program uses only 64-bit registers, meaning that each register value may be between 0 and $2^{64} - 1$. Later, a type inference is presented to improve the handling of mixed sign interpretations.

The analysis starts by identifying code blocks. A block is a straight line sequence of instructions terminated by either a control flow despatch (e.g. `jmp` or `ret`), or an incoming control flow edge. In Figure 28 the blocks are annotated $B_1, \ldots, B_7$. The program instructions themselves are labeled $p_0, \ldots, p_{13}$. The aim of the analysis is to compute a range of values for each register at each program point without actually executing any code. Following the standard procedure (outlined in Chapter 2), a concrete domain and collecting semantics is defined.

The concrete domain is defined analogously to before in Chapter 5. Namely, a single possible state is a tuple drawn from Z^n, where $Z = \{0, \ldots, 2^{64} - 1\}$ and n is the number of registers to model. For this particular example, a single state

$$
\begin{array}{rcl}
S_1 & = & \{\langle rdi, rdi, rax\rangle \mid \langle rdi, r12, rax\rangle \in S_0\} \\
S_2 & = & \{\langle 4 \cdot rdi \bmod 2^{64}, r12, rax\rangle \mid \langle rdi, r12, rax\rangle \in S_1\} \\
S_3 & = & \{\langle rdi + 4 \bmod 2^{64}, r12, rax\rangle \mid \langle rdi, r12, rax\rangle \in S_2\}
\end{array}
$$

$$
\begin{array}{rclrcl}
S_4 & = & S_{3:e} & S_5 & = & S_4 \\
S_6 & = & S_{5:f} & S_7 & = & S_{5:t} \\
S_8 & = & S_{7:e} \cup S_{12:e} & S_9 & = & S_8 \\
S_{10} & = & S_{9:f} & S_{11} & = & S_{10} \\
S_{12} & = & \{\langle rdi + 1 \bmod 2^{64}, r12, rax\rangle \mid \langle rdi, r12, rax\rangle \in S_{11}\} & & & \\
S_{13} & = & S_{9:t} & & &
\end{array}
$$

$$
\begin{array}{rcl}
S_{3:e} & = & \{\langle rdi, r12, rax'\rangle \mid \langle rdi, r12, rax\rangle \in S_3 \ \wedge 0 \leq rax' \leq 2^{64-1}\} \\
S_{5:t} & = & \{\langle rdi, r12, rax\rangle \mid \langle rdi, r12, rax\rangle \in S_5 \ \wedge rax > 0\} \\
S_{5:f} & = & \{\langle rdi, r12, rax\rangle \mid \langle rdi, r12, rax\rangle \in S_5 \ \wedge rax \leq 0\} \\
S_{6:e} & = & S_6 \\
S_{7:e} & = & \{\langle 0, r12, rax\rangle \mid \langle rdi, r12, rax\rangle \in S_7\} \\
S_{9:t} & = & \{\langle rdi, r12, rax\rangle \mid \langle rdi, r12, rax\rangle \in S_9 \ \wedge rdi \geq r12\} \\
S_{9:f} & = & \{\langle rdi, r12, rax\rangle \mid \langle rdi, r12, rax\rangle \in S_9 \ \wedge rdi < r12\} \\
S_{12:e} & = & S_{12} \qquad S_{13:e} = S_{13}
\end{array}
$$

Figure 29: Collecting semantics.

must express the values of three registers, $R = \{rdi, r12, rax\}$, thus $n = 3$ and a state will be represented by a 3-tuple. For a given program point, p_k, the set of all possible states, S_k, is an element drawn from $\wp(Z^n)$.

To aid the composition of block semantics, additional sets are introduced to express the possible register values at the exits of each block. Observe that each block has either one outgoing edge (e.g. B_1), or two outgoing edges (e.g. B_2). For a block with a single outgoing edge, a set $S_{k:e}$ describes the set of possible states upon exit of the block, i.e. immediately after p_k. If, on the other hand, a block is terminated by a conditional jump, it has two outgoing edges and therefore two sets, $S_{k:t}$ and $S_{k:f}$, are introduced to represent states in the true and false arms of the conditional respectively. Figure 29 gives recursive definitions of these sets.

For now the analysis is intra-procedural, meaning that the analysis makes no attempt to follow `call` instructions. As a consequence, the return values of function calls cannot be known. In the worked example, the call to `malloc(3)` will return a pointer value in `rax`. Therefore, `rax` is handled conservatively in $S_{3:e}$, assuming that it may take any value. It is also assumed that functions correctly

implement callee-save. Of course, there is no guarantee that this is the case and indeed registers other than `rax` may be mutated. Recently, an analysis called side-effect analysis has been proposed for addressing this very problem [46].

6.2.2 Abstract Semantics

Analogous to before in Chapter 5, abstraction amounts to representing each unsigned 64-bit register at each program point as an element of the interval domain: $D_{64}^{uint} = \{\emptyset\} \cup \{[l,u] \mid 0 \leq l \leq u \leq 2^{64}-1\}$. This definition is lifted into n dimensions so as to abstract each set S_k with an element drawn from $(D_{64}^{uint})^n$ (a triple of intervals). The ordering, join, meet and domain correspondence are defined similarly to as shown in the previous chapter. The abstract semantics are then defined as shown in Figure 30.

$$
\begin{array}{rcl}
S'_1 & = & \langle rdi, rdi, rax\rangle \text{ where } \langle rdi, r12, rax\rangle = S'_0 \\
S'_2 & = & \begin{cases} \langle I, r12, rax\rangle & \text{if } I \sqsubseteq [0, 2^{64}-1] \\ \langle [l_{rdi}\cdot 4 - 2^{64}, u_{rdi}\cdot 4 - 2^{64}], r12, rax\rangle & \text{if } I \sqsubseteq [2^{64}, 2\cdot(2^{64})-1] \\ \langle \top, r12, rax\rangle & \text{otherwise} \end{cases} \\
 & & \text{where } \langle [l_{rdi}, u_{rdi}], r12, rax\rangle = S'_1 \text{ and } I = [4\cdot l_{rdi}, 4\cdot u_{rdi}] \\
S'_3 & = & \begin{cases} \langle I, r12, rax\rangle & \text{if } I \sqsubseteq [0, 2^{64}-1] \\ \langle [l_{rdi}+4-2^{64}, u_{rdi}+4-2^{64}], r12, rax\rangle & \text{if } I \sqsubseteq [2^{64}, 2\cdot(2^{64})-1] \\ \langle \top, r12, rax\rangle & \text{otherwise} \end{cases} \\
 & & \text{where } \langle [l_{rdi}, u_{rdi}], r12, rax\rangle = S'_2 \text{ and } I = [l_{rdi}+4, u_{rdi}+4]
\end{array}
$$

$$
\begin{array}{rclrclrcl}
S'_4 & = & S'_{3:e} & S'_5 & = & S'_4 & S'_6 & = & S'_{4:f} \\
S'_7 & = & S'_{4:t} & S'_8 & = & S'_{7:e} \sqcup S'_{12:e} & S'_9 & = & S'_8 \\
S'_{10} & = & S'_{9:f} \sqcup S'_{12:e} & S'_{11} & = & S'_{10} & & &
\end{array}
$$

$$
\begin{array}{rcl}
S'_{12} & = & \begin{cases} \langle I, r12, rax\rangle & \text{if } I \sqsubseteq [0, 2^{64}-1] \\ \langle [l_{rdi}+1-2^{64}, u_{rdi}+1-2^{64}], r12, rax\rangle & \text{if } I \sqsubseteq [2^{64}, 2\cdot(2^{64})-1] \\ \langle \top, r12, rax\rangle & \text{otherwise} \end{cases} \\
 & & \text{where } \langle [l_{rdi}, u_{rdi}], r12, rax\rangle = S'_{11} \text{ and } I = [l_{rdi}+1, u_{rdi}+1]
\end{array}
$$

$$
\begin{array}{rcl}
S'_{13} & = & S'_{9:t} \\
S'_{3:e} & = & \langle rdi, r12, \top\rangle \text{ where } \langle rdi, r12, rax\rangle = S'_3 \\
S'_{5:t} & = & \langle rdi, r12, rax \sqcap [1, 2^{64}-1]\rangle \text{ where } \langle rdi, r12, rax\rangle = S'_5 \\
S'_{5:f} & = & \langle rdi, r12, rax \sqcap [0, 0]\rangle \text{ where } \langle rdi, r12, rax\rangle = S'_5 \\
S'_{6:e} & = & S'_6 \\
S'_{7:e} & = & \langle [0,0], r12, rax\rangle \text{ where } \langle rdi, r12, rax\rangle = S'_7 \\
S'_{9:t} & = & \langle rdi \sqcap [l_{r12}, 2^{64}-1], [l_{r12}, u_{r12}], rax\rangle \text{ where } \langle rdi, [l_{r12}, u_{r12}], rax\rangle = S'_9 \\
S'_{9:f} & = & \langle rdi \sqcap [0, u_{r12}], [l_{r12}, u_{r12}], rax\rangle \text{ where } \langle rdi, [l_{r12}, u_{r12}], rax\rangle = S'_9
\end{array}
$$

$$
\begin{array}{rclrcl}
S'_{12:e} & = & S'_{12} & S'_{13:e} & = & S'_{13}
\end{array}
$$

Figure 30: Abstract semantics for the worked example.

Of particular interest are the definitions of S'_2, S'_3 and S'_{12} (Figure 30), in which the different modes of arithmetic are modelled. For example, the abstract equation S'_3 models the three different overflow cases of the addition operation:

- The first case stipulates that if after adding 4 to the bounds of `rdi`, both bounds are between 0 and $2^{64}-1$, then the addition operates in exact mode (with no overflow).
- The second stipulates that if after adding 4 to the bounds of `rdi`, both bounds are between 2^{64} and $2 \cdot 2^{64}-1$, then both bounds overflow as a result of the addition. To compute the resulting bounds, 2^{64} must be subtracted from each bound.
- The third case accounts for situations where only the upper bound overflows. In this case the addition is modelled conservatively.

Note that the semantics of the `imul` instruction as described by the equation for S'_2 is a simplified approximation, since it is entirely possible for $4 \cdot rdi$ to be greater than $2 \cdot (2^{64}) - 1$. Further cases could be added to obtain higher precision at the cost of a larger constraint system with more variables. For example, the abstract semantics of S'_2 could be defined as:

$$S'_2 = \begin{cases} \langle I, r12, rax \rangle & \text{if } I \sqsubseteq [0, 2^{64}-1] \\ \langle [l_{rdi} \cdot 4 - 2^{64}, u_{rdi} \cdot 4 - 2^{64}], r12, rax \rangle & \text{if } I \sqsubseteq [2^{64}, 2 \cdot (2^{64}) - 1] \\ \langle [l_{rdi} \cdot 4 - 2 \cdot (2^{64}), u_{rdi} \cdot 4 - 2 \cdot (2^{64})], r12, rax \rangle & \text{if } I \sqsubseteq [2 \cdot (2^{64}), 3 \cdot (2^{64}) - 1] \\ \langle [l_{rdi} \cdot 4 - 3 \cdot (2^{64}), u_{rdi} \cdot 4 - 3 \cdot (2^{64})], r12, rax \rangle & \text{if } I \sqsubseteq [3 \cdot (2^{64}), 4 \cdot (2^{64}) - 1] \\ \ldots & \ldots \\ \langle \top, r12, rax \rangle & \text{otherwise} \end{cases}$$
$$\text{where } \langle [l_{rdi}, u_{rdi}], r12, rax \rangle = S'_1 \text{ and } I = [4 \cdot l_{rdi}, 4 \cdot u_{rdi}]$$

The addition of further cases, however, is not required for safety, as the final case will catch any scenario not previously considered.

An initial state, S'_0, is then specified so as to mimic the passing of the function's operands. For example, the following would be used to pass the value $(2^{64}-4)/4$ as the function's sole operand:

$$S'_0 = \langle [(2^{64}-4)/4, (2^{64}-4)/4], \top, \top \rangle$$

As seen later, this particular input manifests an interesting behaviour. Here a single value is passed as the `rdi` argument, but there is no reason why an interval could not also be used if so desired. With this, the abstract semantics is complete.

6.2.3 Solving via Mathematical Optimisation

The system of abstract semantic equations has a best solution which corresponds to the least-fixpoint. This solution can be found by traditional Kleene iteration. As discussed in the previous chapter, solving by this method may suffer from long convergence times and widening may be required, potentially incurring a further loss of precision. Instead, the abstract semantics are encoded as a mixed-integer linear program (MILP) and the least-fixpoint is found directly.

The decomposition of the abstract semantics into a MILP is not dissimilar from the approach shown in the previous chapter. Again, for each interval, two continuous optimisation variables are introduced, one for each bound. For example, an abstract set S'_k is modelled using $2n$ optimisation variables, where n is the number of registers to account for. In the case of the worked example, for each program point, p_k, six optimisation variables are used to represent interval bounds: $l_{rdi,k}, u_{rdi,k}, l_{r12,k}, u_{r12,k}, l_{rax,k}$ and $u_{rax,k}$. Each interval is constrained to fall within the numeric range of its associated register. Since for the worked example program, all registers are 64-bit and assumed to be interpreted unsigned, each interval is constrained between 0 and $2^{64} - 1$. The variables are then constrained to mirror the abstract semantics, for example, the abstract state $S'_{7:e}$ is modelled with the following linear constraints:

$$\begin{aligned} l_{rdi,7:e} = 0 \quad &\wedge \quad u_{rdi,7:e} = 0 \quad \wedge \\ l_{r12,7:e} = l_{r12,7} \quad &\wedge \quad u_{r12,7:e} = u_{r12,7} \wedge \\ l_{rax,7:e} = l_{rax,7} \quad &\wedge \quad u_{rdi,7:e} = u_{rdi,7} \end{aligned}$$

This expresses the update to the interval bounds of `rdi` to [0, 0], but also asserts that the other interval bounds do not change.

An objective function is used, which again, aims to find the tightest hyper-rectangles that enclose all possible concrete states. The result is a MILP of the form:

$$\text{minimise} \sum_{k \in P} \sum_{r \in R} (u_{r,k} - l_{r,k}) \text{ s.t. } C$$

where P is a set of program points, R is a set of registers and C is the constraint system expressing the abstract semantics. The interval bounds of the optimal solution to this problem characterises the least-fixpoint of the abstract semantics.

Solving the MILP generated from the worked example program gives a solution that corresponds to the following interval bounds:

$$\begin{aligned}
S'_1 &= \langle [(2^{64}-4)/4, (2^{64}-4)/4], [(2^{64}-4)/4, (2^{64}-4)/4], \top \rangle \\
S'_2 &= \langle [2^{64}-4, 2^{64}-4], [(2^{64}-4)/4, (2^{64}-4)/4], \top \rangle \\
S'_3 &= \langle [0,0], [(2^{64}-4)/4, (2^{64}-4)/4], \top \rangle \\
&\dots
\end{aligned}$$

This would suggest that the first argument register, `rdi`, is zero immediately prior to the call to `malloc(3)` at p_3. Closer inspection shows that the addition operation at p_2 causes an integer overflow. Between p_2 and p_3 the abstraction of `rdi` transitions from $[2^{64}-4, 2^{64}-4]$ to $[0,0]$ as a result of adding four to each of the interval bounds. As a consequence, when the program is executed under this input, `malloc(3)` is asked to allocate 0 bytes. Oddly enough, most implementations of `malloc(3)` do allow this, and a pointer is returned, but to a block of memory which is access protected. Any read or write to this memory will raise a memory exception. In fact, the memory write at p_{10} will cause such an exception. Since the `rdi` component of S'_3 is a singleton, it is easy to see that this is not a false positive. Hence, the analysis deduces that the code in question is not fit for purpose. Indeed the code has a heap overflow vulnerability and will crash under the assignment: $S'_0 = \langle [(2^{64}-4)/4, (2^{64}-4)/4], \top, \top \rangle$.

Encoding Differences

Whilst the method shares much of its infrastructure with the method described in the previous chapter, some aspects differ substantially. Specifically:

- Instead of assuming that arithmetic saturates, decision variables are used to model integer wrapping operations. This mechanism is described in Section 6.3.

- Block reachability is expressed with decision variables, rather than by the non-canonical empty interval $[l, u]$, where $l > u$. Conditional constructs are also realised through decision variables, instead of through min/max constraints. This means that the problem can be solved as a single MILP rather than as many LP relaxations. These aspects are described in Section 6.4.

6.3 Piecewise Linear Functions in MILP

It is not obvious how the multi-modal wrapping cases used in S'_2, S'_3 and S'_{12} in the worked example should be encoded as MILP constraints. In this section, a general framework is presented, dubbed decide-and-impose, that leverages decision variables to model piecewise linear functions within a MILP problem. As the name may suggest, the framework is broken down into two distinct phases: the decision phase and the impose phase. The update of `rdi` in S'_3 shall be used to exemplify the framework.

6.3.1 The Decision Phase

Consider the abstract semantic equation for S'_3. The update of the interval bounds for `rdi` can be thought of as a piecewise linear function, $U : D_{64}^{uint} \to D_{64}^{uint}$, such that $[l_{rdi,3}, u_{rdi,3}] = U([l_{rdi,2}, u_{rdi,2}])$:

$$U([l_{rdi,2}, u_{rdi,2}]) = \begin{cases} [l_{rdi,2} + 4, u_{rdi,2} + 4] & \text{if } u_{rdi,2} + 4 \leq 2^{64} - 1 \\ [l_{rdi,2} + 4 - 2^{64}, u_{rdi,2} + 4 - 2^{64}] & \text{if } l_{rdi,2} + 4 \geq 2^{64} \\ [0, 2^{64} - 1] & \text{otherwise} \end{cases}$$

The function takes advantage of the fact that, for any given interval $[l, u] \in D_{64}^{uint}$, it is assumed that $l \leq u$. This assumption is safe because in the revised MILP encoding, the empty interval (and thus unreachable code) is expressed using decision variables, instead of an interval $[l, u]$ where $l > u$. This mechanism is described later in Section 6.4.

It is straightforward to encode each of the three possible updates independently as linear constraints, but a mechanism is required which can select which case

should apply. To this end, three decision variables δ_1, δ_2 and δ_3 are introduced, one for each case. The three cases are mutually exclusive but one case must apply, hence $\delta_1 + \delta_2 + \delta_3 = 1$. The meaning of these variables is as follows:

$$\begin{aligned} (u_{rdi,2} + 4 \leq 2^{64} - 1) &\iff \delta_1 = 1 \\ (l_{rdi,2} + 4 \geq 2^{64}) &\iff \delta_2 = 1 \\ \delta_1 + \delta_2 + \delta_3 &= 1 \end{aligned}$$

The third constraint can be used as is and the former two constraints are decomposed into MILP constraints using the following equivalence:

$$(x \leq y \iff \delta_i = 1) \equiv (x \leq y + M \cdot (1 - \delta_i) \wedge x + M \cdot \delta_i \geq y + 1)$$

where M is an integer constant that exceeds the maximum absolute difference between x and y. For example, suppose the equivalence is used to encode $(u_{rdi,2} + 4 \leq 2^{64} - 1) \iff \delta_1 = 1$. Here $x = u_{rdi,2} + 4$ taking a value between 4 and $2^{64} - 1 + 4$, and y is the constant $2^{64} - 1$. In this case, M should exceed $2^{64} - 5$ since this is the maximum absolute difference between x and y. The correctness of the equivalence is shown by Corollary 3 on Page 177. The equivalence gives the following constraints for the decision phase of the update of `rdi` in S_3':

$$\begin{aligned} u_{rdi,2} + 4 \leq 2^{64} - 1 + M \cdot (1 - \delta_1) &\wedge \\ u_{rdi,2} + 4 + M \cdot \delta_1 \geq 2^{64} &\wedge \\ 2^{64} \leq l_{rdi,2} + 4 + M \cdot (1 - \delta_2) &\wedge \\ 2^{64} + M \cdot \delta_2 \geq l_{rdi,2} + 4 + 1 &\wedge \\ \delta_1 + \delta_2 + \delta_3 = 1 \end{aligned}$$

Note that it is acceptable to use a single instantiation of M just as long as M is sufficiently large enough for each inequality in isolation.

6.3.2 The Impose Phase

Now that decision variables are constrained so as to indicate which update case should apply, the impose phase can now be implemented. Based upon the truth values of δ_1, δ_2 and δ_3, constraints will be put in place to ensure that only the

correct update occurs. The impose phase must express the following:

$$\begin{array}{lcrcll}
(\delta_1 = 1) & \Longrightarrow & l_{rdi,3} & \leq & l_{rdi_2} + 4 & \wedge \\
(\delta_1 = 1) & \Longrightarrow & u_{rdi,3} & \geq & u_{rdi_2} + 4 & \wedge \\
(\delta_2 = 1) & \Longrightarrow & l_{rdi,3} & \leq & l_{rdi_2} + 4 - 2^{64} & \wedge \\
(\delta_2 = 1) & \Longrightarrow & u_{rdi,3} & \geq & u_{rdi_2} + 4 - 2^{64} & \wedge \\
(\delta_3 = 1) & \Longrightarrow & l_{rdi,3} & \leq & 0 & \wedge \\
(\delta_3 = 1) & \Longrightarrow & u_{rdi,3} & \geq & 2^{64} - 1 &
\end{array}$$

For example, suppose that the assignment to $\langle \delta_1, \delta_2, \delta_3 \rangle$ is $\langle 0, 1, 0 \rangle$. This indicates that the second case applies and that the interval bounds should be updated to reflect the overflow condition. Under this assignment, the above constraint system collapses to $l_{rdi,3} \leq l_{rdi_2} + 4 - 2^{64} \quad \wedge \quad u_{rdi,3} \geq u_{rdi_2} + 4 - 2^{64}$. This is sufficient to express the update of the interval bounds with the second case. At first the constraints may seem too weak, however, recall that the objective function seeks the tightest intervals. This means that the optimal solution to the above would infer $rdi_3 = [l_{rdi_2} + 4 - 2^{64}, u_{rdi_2} + 4 - 2^{64}]$. To encode the bounds updates as MILP constraints, the following equivalence is used:

$$(\delta_i = 1) \implies (x \leq y) \equiv x \leq y + M \cdot (1 - \delta_i)$$

Here M takes on the same role as before. The correctness of the equivalence is shown by Corollary 4 on Page 178. This leads to the following constraints for the impose phase of the update to `rdi` in S'_3:

$$\begin{array}{rcll}
l_{rdi,3} & \leq & l_{rdi,2} + 4 + M \cdot (1 - \delta_1) & \wedge \\
u_{rdi,2} + 4 & \leq & u_{rdi,3} + M \cdot (1 - \delta_1) & \wedge \\
l_{rdi,3} & \leq & l_{rdi,2} + 4 - 2^{64} + M \cdot (1 - \delta_2) & \wedge \\
u_{rdi,2} + 4 - 2^{64} & \leq & u_{rdi,3} + M \cdot (1 - \delta_2) & \wedge \\
l_{rdi,3} & \leq & 0 + M \cdot (1 - \delta_3) & \wedge \\
2^{64} - 1 & \leq & u_{rdi,3} + M \cdot (1 - \delta_3) &
\end{array}$$

This concludes the encoding of the update to `rdi` in S'_3. Other abstract operations, even interval multiplication, can be encoded analogously.

6.4 Encoding Control Flow and Reachability

Recall that some other aspects of the analysis were reformulated so as to take advantage of decision variables. Namely, block reachability and control flow branches are now modelled with decision variables. In this section, these mechanisms are described.

6.4.1 Intra-Block Reachability

Each block B_i has an associated decision variable ϵ_i which indicates whether a block is possibly reachable ($\epsilon_i = 1$), or otherwise definitely unreachable ($\epsilon_i = 0$). Let these variables be referred to as block entry decisions. The block in which execution starts is assumed to be reachable, thus in the worked example, $\epsilon_1 = 1$. A block also has associated with it, at least one further decision variable to indicate the reachability of each outgoing edge of the block. These decision variables are referred to as block exit decisions. Each block's entry and exit decisions must be related.

When a block B_i has a single outgoing edge, then a single block exit decision $\eta_{i:e}$ is used to indicate whether execution may pass out of the block. In this case, the analysis assumes that if execution could enter the block, then execution could pass out of the block, i.e. $\epsilon_i = \eta_{i:e}$. If, on the other hand, a block B_i is terminated with a conditional jump, then two block exit decisions, $\eta_{i:t}$ and $\eta_{i:f}$, are defined. These decision variables indicate whether execution may pass out of the block by taking the conditional jump ($\eta_{i:t} = 1$) or by the fall-through edge ($\eta_{i:f} = 1$). Note that if $\eta_{i:t} = 1$, then it is quite possible that $\eta_{i:f} = 1$ also. Indeed, any combination is possible.

In the case where a block, B_i, is terminated by a conditional jump, relating the block entry decision to the block exit decisions is slightly more involved. Before constraints may be generated, a high-level predicate must be extracted from a backward slice. This procedure was described in Section 5.6. For example, the instructions at p_8 and p_9 of B_5 in the worked example program (Figure 28) correspond to the high-level (unsigned) predicate $rdi \geq r12$. Therefore, $\eta_{5:t}$ should

be expressed as follows:

$$\begin{aligned} l_{r12,9} \leq u_{rdi,9} &\iff \delta_t = 1 \\ \eta_{5:t} = 1 &\iff \delta_t = 1 \wedge \epsilon_5 = 1 \end{aligned}$$

The former of the two constraints uses a fresh decision variable, δ_t, to indicate whether the branching predicate could be satisfied based upon the intervals prior to the jump. This constraint can be decomposed into MILP constraints using the equivalence shown in Section 6.3. The latter constraint then encapsulates the exit decision. The true exit decision is only considered reachable if the branching predicate is satisfied and B_5 itself is reachable. This constraint is decomposed into MILP constraints using the following equivalence:

$$((\delta_m = 1 \wedge \delta_n = 1) \iff \delta_i = 1) \equiv ((\delta_m + \delta_n - 2 \cdot \delta_i \leq 1) \wedge (\delta_m + \delta_n - 2 \cdot \delta_i \geq 0))$$

The correctness of the equivalence is shown in Theorem 13 on Page 178. The following constraints are generated to relate the block entry and true block exit decision of B_5:

$$\begin{aligned} l_{r12,9} &\leq u_{rdi,9} + M \cdot (1 - \delta_t) &\wedge \\ l_{r12,9} + M \cdot \delta_t &\geq u_{rdi,9} + 1 &\wedge \\ \delta_t + \epsilon_5 - 2 \cdot \eta_{5:t} &\leq 1 &\wedge \\ \delta_t + \epsilon_5 - 2 \cdot \eta_{5:t} &\geq 0 \end{aligned}$$

The reachability of the false branch is encoded analogously.

6.4.2 Inter-Block Reachability

The reachability flags have been related within each block, but now constraints should be introduced to propagate reachability between the blocks. To propagate these reachability decisions forward through the control flow of the program, the block exit decisions of each block are related to the block entry decisions of successor blocks. Specifically, a block is reachable if at least one of the exit decisions of a direct predecessor block is reachable. For example, B_5 is reachable if either the exit decision of B_4 is set ($\eta_{4:e} = 1$) or if the exit decision of B_6 is set ($\eta_{6:e} = 1$).

Logically, this is expressed as:

$$(\epsilon_5 = 1) \iff (\eta_{4:e} = 1 \vee \eta_{6:e} = 1)$$

By applying De Morgan's law, the above can be put in a form which can be decomposed into MILP constraints, i.e. $(\epsilon_5 = 0) \iff (\eta_{4:e} = 0 \wedge \eta_{6:e} = 0)$ is encoded into MILP constraints using the equivalence shown in the previous section:

$$\begin{aligned}(1 - \eta_{4:e}) + (1 - \eta_{6:e}) - 2 \cdot (1 - \epsilon_5) &\leq 1 \quad \wedge \\ (1 - \eta_{4:e}) + (1 - \eta_{6:e}) - 2 \cdot (1 - \epsilon_5) &\geq 1\end{aligned}$$

6.4.3 Interval Partitioning for Conditional Jumps

Recall that each outgoing edge of a block has an associated abstract state. In the case where a block is terminated by a conditional jump, the intervals before the jump must be partitioned between the states corresponding to the true and false branches. For example, the abstract states $S'_{9:t}$ and $S'_{9:f}$ describe the intervals in the true and false arm of the conditional at p_9 respectively. Notice that these intervals are based upon a partitioning, $rdi \geq r12$ of S'_9. Again $rdi \geq r12$ is the predicate extracted from the reverse slice $\langle$`cmp rdi, r12; jae ...`$\rangle$. Constraints must be introduced to reflect this partitioning.

Chapter 5 showed that it is possible to model this partitioning using min/max constraints. The update to the interval bounds of `rdi` in the true branch of the conditional at p_9 can be expressed as follows:

$$\begin{aligned}l_{rdi,9:t} &= max(l_{rdi,9}, l_{r12,9}) \wedge \\ u_{rdi,9:t} &= u_{rdi,9}\end{aligned}$$

The max constraint is of course non-convex. Rather than using a binary search, as suggested in Chapter 5, this chapter proposes that min/max constraints be encoded directly as MILP constraints through the use of decision variables. Notice that the max constraint can be expressed as a piecewise linear function:

$$max(x, y) = \begin{cases} x & \text{iff } x \leq y \\ y & \text{otherwise} \end{cases}$$

Therefore there is no need to perform a binary search, since the interval partitioning can be encoded into MILP constraints using the decide-and-impose framework described in Section 6.3. The interval bounds for the false arm of the conditional may be expressed analogously. The advantage of this approach is that only a single MILP program need be solved.

6.4.4 Control Flow Joining

At points in the program where control flow converges, a join of interval bounds should occur. In the worked example, a control flow join occurs at p_8. The intervals of S'_8 should therefore be defined in terms of $S'_{7:e}$ and $S'_{12:e}$. However, to retain precision, abstract information should only be joined from block exits that are reachable. If it is not possible for execution to reach p_{12}, then the abstract information $S'_{12:e}$ should not be considered in the computation of S'_8.

The following constraints express the desired effect of the join at p_8:

$$
\begin{array}{rcrcl}
\eta_{6:e} = 1 & \Longrightarrow & (l_{rdi,8} & \leq & l_{rdi,12:e}) \\
\eta_{6:e} = 1 & \Longrightarrow & (u_{rdi,12:e} & \leq & u_{rdi,8}) \\
\eta_{6:e} = 1 & \Longrightarrow & (l_{r12,8} & \leq & l_{r12,12:e}) \\
\eta_{6:e} = 1 & \Longrightarrow & (u_{r12,12:e} & \leq & u_{r12,8}) \\
\eta_{6:e} = 1 & \Longrightarrow & (l_{rax,8} & \leq & l_{rax,12:e}) \\
\eta_{6:e} = 1 & \Longrightarrow & (u_{rax,12:e} & \leq & u_{rax,8})
\end{array}
$$

$$
\begin{array}{rcrcl}
\eta_{4:e} = 1 & \Longrightarrow & (l_{rdi,8} & \leq & l_{rdi,7:e}) \\
\eta_{4:e} = 1 & \Longrightarrow & (u_{rdi,7:e} & \leq & u_{rdi,8}) \\
\eta_{4:e} = 1 & \Longrightarrow & (l_{r12,8} & \leq & l_{r12,7:e}) \\
\eta_{4:e} = 1 & \Longrightarrow & (u_{r12,7:e} & \leq & u_{r12,8}) \\
\eta_{4:e} = 1 & \Longrightarrow & (l_{rax,8} & \leq & l_{rax,7:e}) \\
\eta_{4:e} = 1 & \Longrightarrow & (u_{rax,7:e} & \leq & u_{rax,8})
\end{array}
$$

Each constraint expresses the conditional propagation of an interval bound between two blocks. Each bound propagation is predicated upon the reachability of the exit node from which the bounds are propagated. If a block exit decision, $\eta_{i:j}$, indicates unreachability, i.e. $\eta_{i:j} = 0$, then propagation constraints based upon this decision are rendered vacuous.

The above constraints are reduced to MILP constraints using the equivalence shown in Section 6.3.2. Note that blocks with a single incoming edge are in fact treated as a join. This is necessary, since if the predecessor's exit edge decision indicates unreachability, then interval bounds should not be propagated.

The prerequisites for the analysis of binaries using only unsigned integer interpretations has now been fully detailed. However, realistically binaries interpret registers not only as unsigned integers, but sometimes as signed integers, or even as both signed and unsigned integers simultaneously. The following section explains how the analysis can be extended to account for these behaviours.

6.5 Modelling Mixed Signedness

Until now it has been assumed that all registers are interpreted in an unsigned context, yet in reality programs use a mix of signed and unsigned interpretations. An unsigned 64-bit number ranges between 0 and $2^{64} - 1$, where as a signed 64-bit number ranges between -2^{63} and $2^{63} - 1$. This means that certain arithmetic instructions will have different outcomes depending upon the interpretation of the operand register(s). It could be argued that the analysis only needs to care about the unsigned context, since the computation of the result of arithmetic operations is sign agnostic. Whilst this is true, comparison operations such as `cmp` set the flags in the status register, some flags under the assumption that the operations are signed and others assuming the operands are signed. Numeric comparisons are usually followed by one of the conditional jump instructions, which reads a subset of the previously set flags, therefore possibly indicating a signed or unsigned interpretation.

The upshot of this is that the analysis must have some notion of signed and unsigned interpretations in order to decide whether a conditional jump is taken. Ideally, a mechanism could be devised which is able to interpret a register's interval bounds as either signed or unsigned. This is not straightforward, as by necessity, the numeric bounds of an interval have already committed to either a signed or an unsigned interpretation and it is not clear how interval bounds could be cast to a different sign interpretation within the optimisation problem. Another approach is to model every register twice, once for the signed interpretation and once for unsigned. This would mean that each register at each program point

would be represented by four optimisation variables, a lower and upper bound for the unsigned context and a lower and upper bound for the signed context. However, this doubles the number of optimisation variables required to express intervals in the MILP.

Instead, a compromise is proposed. Type inference is used to gain insights on the possible interpretations of each register throughout the program. Using this information it is possible to model interval bounds for only the sign interpretations that actually occur within the program. This means that the MILP can be encoded utilising fewer optimisation variables. To illustrate, Figure 31 shows the outcome of the type inference for a snippet of synthetic x86-64 code. The code uses both unsigned numeric comparisons (`cmp` followed by `ja`) and signed numeric comparisons (`cmp` followed by `jg`). For each program point, a set of possible interpretations of the registers is inferred. From this outcome it is possible to decide which sign contexts must be modelled in the MILP program. For example at p_{16} it is only necessary to model the unsigned interpretation of `rax`. Also note that it is not necessary to model `rcx` at all since neither the signed nor the unsigned interpretation are required at any program point. This is because the value stored in `rcx` is not used and this is reflected by the type inference.

Type information is not explicit in binary code, so the intended interpretation(s) of each register must be inferred. To simplify terminology, henceforth let the intended interpretation(s) of a register be referred to as simply the types of a register. The proposed type inference is a flow analysis inspired by, amongst others, liveness analyses used in compiler construction [4]. Like these analyses, the algorithm uses an ***in*** and ***out*** state for each program instruction to propagate information entering and leaving each program point. These states are used to propagate information throughout the control flow of the program[1]. Unlike similar flow analyses that infer types for, for example, C or Java, it is necessary to propagate information both backwards and forwards. This is because the types of a register are not known at the definition site and must be inferred from status flag usage patterns.

To demonstrate, in the code snippet shown in Figure 31, although `rbx` is defined at p_1, the type the register is not known until the status register flags assigned by $\langle$`cmp rbx, 0x1`$\rangle$ at p_3 have been paired with the unsigned status flag

[1]The control flow of the program must be known a priori.

p_i	Instruction	rax	rbx	rcx
p_0	`mov rax, 0x0`	$\{s, u\}$	$\emptyset$	$\emptyset$
p_1	`mov rbx, 0x666`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_2	`mov rcx, 0x1`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_3	`cmp rbx, 0x1`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_4	`ja p6`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_5	`jmp p11`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_6	`cmp rax, 0x10`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_8	`jg p15`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_9	`inc rcx`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_{10}	`jmp p6`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_{11}	`cmp rax, 0x10`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_{12}	`ja p15`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_{13}	`inc rcx`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_{14}	`jmp p11`	$\{s, u\}$	$\{u\}$	$\emptyset$
p_{15}	`mov rax, 0x0`	$\emptyset$	$\{u\}$	$\emptyset$
p_{16}	`mov rax, rbx`	$\{u\}$	$\{u\}$	$\emptyset$
p_{17}	`ret`	$\{u\}$	$\{u\}$	$\emptyset$

Figure 31: Example outcome of type inference.

use of $\langle$`ja p6`$\rangle$ at p_4. The former instruction assigns (defines) the status flags based upon both the signed and unsigned interpretations of `rbx`, then the `ja` instruction reads the flags to ascertain if the previous comparison was true in the unsigned case. The type of `rbx` cannot be ascertained from either instruction in isolation, i.e. $\langle$`cmp rbx, 0x1`$\rangle$ does not allude to the sign of `rbx` and $\langle$`ja p6`$\rangle$ does not indicate which register the unsigned comparison was based upon. Thus the flag definitions (flag-defs) must be paired with the flag uses (flag-uses). The pairing is complicated by the fact that some instructions set only a subset of the flags in the status register.

With these considerations in mind, the pairing of flag-uses with flag-defs is accomplished by propagating flag-uses backwards towards flag-defs. When the flag-uses meet their corresponding flag-defs, the flag-uses cease to be propagated further backwards and a register type may be inferred. Once inferred, type information is propagated forwards until the register is redefined, and backwards to the definition site of the register.

6.5.1 Algebraic Inference Structures

First, the data structures over which the analysis operates are defined.

Flag State ($\mathcal{F}$ mapping)

Let $C = \{o, s, z, c\}$ be a set of status flags which can reveal information pertaining to register types. These flags are: the overflow flag, the sign flag, the zero flag and the carry flag. For each of these flags, at each program point, the analysis keeps track of whether the current definition of the flag is used later by another instruction. Here "used" means that the flag is read, but not mutated. By contrast, when a flag is mutated, it is said to be "defined". Let $\mathcal{B}$ be the totally ordered set of Booleans, where TRUE means that the current definition of a flag may be used before its redefinition and FALSE means that the current definition of a flag is never used before its redefinition. The domain forms a two-element complete lattice $\langle \mathbb{B}, \sqsubseteq_{\mathcal{B}}, \sqcup_{\mathcal{B}}, \sqcap_{\mathcal{B}} \rangle$, where:

$$\begin{array}{lcl} b_1 \sqsubseteq_{\mathcal{B}} b_2 & \iff & b_1 \wedge b_2 = b_1 \\ b_1 \sqcup_{\mathcal{B}} b_2 & \triangleq & b_1 \vee b_2 \\ b_1 \sqcap_{\mathcal{B}} b_2 & \triangleq & b_1 \wedge b_2 \end{array}$$

For each program point, a mapping $C \to \mathcal{B}$ is held indicating whether the current definition of each of the status flags is used before its redefinition. The set of all such mappings $\mathcal{F}$ is also partially ordered so as to form a complete lattice $\langle \mathcal{F}, \sqsubseteq_{\mathcal{F}}, \sqcup_{\mathcal{F}}, \sqcap_{\mathcal{F}} \rangle$, where:

$$\begin{array}{lcl} f_1 \sqsubseteq_{\mathcal{F}} f_2 & \iff & \forall c \in C.\ f_1(c) \sqsubseteq_{\mathcal{B}} f_2(c) \\ f_1 \sqcup_{\mathcal{F}} f_2 & \triangleq & f_3 \text{ where } \forall c \in C.\ f_3(c) = f_1(c) \sqcup_{\mathcal{B}} f_2(c) \\ f_1 \sqcap_{\mathcal{F}} f_2 & \triangleq & f_3 \text{ where } \forall c \in C.\ f_3(c) = f_1(c) \sqcap_{\mathcal{B}} f_2(c) \end{array}$$

The bottom element $\bot_{\mathcal{F}}$ indicates that no status flag may be used before its redefinition, i.e. $\forall c \in C.\ \bot_{\mathcal{F}}(c) = \text{FALSE}$. The top element $\top_{\mathcal{F}}$ is defined analogously to mean that all status flags may be used before their redefinition, i.e. $\forall c \in C.\ \top_{\mathcal{F}}(c) = \text{TRUE}$.

Register State ($\mathcal{R}$ mapping)

Next, a data structure is introduced to track the possible types of a single register. A register may be interpreted: signed, unsigned, both signed and unsigned, or neither. To this end, the type of a single register is an element drawn from $\mathcal{L} = \wp(\{s, u\})$. An ordering is defined so as to form a complete lattice, $\langle \mathcal{L}, \sqsubseteq_{\mathcal{L}}, \sqcup_{\mathcal{L}}, \sqcap_{\mathcal{L}} \rangle$, where:

$$\begin{array}{rcl} l_1 \sqsubseteq_{\mathcal{L}} l_2 & \iff & l_1 \subseteq l_2 \\ l_1 \sqcup_{\mathcal{L}} l_2 & \triangleq & l_1 \cup l_2 \\ l_1 \sqcap_{\mathcal{L}} l_2 & \triangleq & l_1 \cap l_2 \end{array}$$

The top element $\top_{\mathcal{L}} = \{s, u\}$ indicates that a register is interpreted both signed and unsigned, whereas the bottom element, $\bot_{\mathcal{L}} = \emptyset$, means that there is nothing to indicate that the register is either signed or unsigned.

Since it it necessary to track the possible interpretations of each register, mappings are used to associate each register to its types. Let R be a set of registers to track. For example purposes, assume that $R = \{\texttt{rax}, \texttt{rbx}, \texttt{rcx}\}$. Then $\mathcal{R}$ is the set of all mappings $R \to \mathcal{L}$ ordered as follows:

$$\begin{array}{rcl} a_1 \sqsubseteq_{\mathcal{R}} a_2 & \iff & \forall r \in R.\ a_1(r) \sqsubseteq_{\mathcal{L}} a_2(r) \\ a_1 \sqcup_{\mathcal{R}} a_2 & \triangleq & a_3 \text{ where } \forall r \in R.\ a_3(r) = a_1(r) \sqcup_{\mathcal{L}} a_2(r) \\ a_1 \sqcap_{\mathcal{R}} a_2 & \triangleq & a_3 \text{ where } \forall r \in R.\ a_3(r) = a_1(r) \sqcap_{\mathcal{L}} a_2(r) \end{array}$$

The bottom element $\bot_{\mathcal{R}}$ indicates that no type information is known for any register, i.e. $\forall r \in R.\ \bot_{\mathcal{R}}(r) = \emptyset$. Similarly, for $\top_{\mathcal{R}}$ where $\forall r \in R.\ \top_{\mathcal{R}}(r) = \{s, u\}$.

Overall State

Finally, the previously defined structures are composed, thus forming the overall state that shall ultimately be propagated by the type inference. For each program point, a tuple drawn from $\mathcal{K} : \mathcal{R} \times \mathcal{F}$ represents the register types and the status flag usage. The orderings are lifted in the obvious manner:

$$\begin{array}{rcl} \langle a_1, f_1 \rangle \sqsubseteq_{\mathcal{K}} \langle a_2, f_2 \rangle & \iff & (a_1 \sqsubseteq_{\mathcal{R}} a_2) \wedge (f_1 \sqsubseteq_{\mathcal{F}} f_2) \\ \langle a_1, f_1 \rangle \sqcup_{\mathcal{K}} \langle a_2, f_2 \rangle & \triangleq & \langle a_1 \sqcup_{\mathcal{R}} a_2, f_1 \sqcup_{\mathcal{F}} f_2 \rangle \\ \langle a_1, f_1 \rangle \sqcap_{\mathcal{K}} \langle a_2, f_2 \rangle & \triangleq & \langle a_1 \sqcap_{\mathcal{R}} a_2, f_1 \sqcap_{\mathcal{F}} f_2 \rangle \end{array}$$

The bottom and top elements of $\mathcal{K}$ are $\bot_{\mathcal{K}} = \langle \bot_{\mathcal{R}}, \bot_{\mathcal{F}} \rangle$ and $\top_{\mathcal{K}} = \langle \top_{\mathcal{R}}, \top_{\mathcal{F}} \rangle$

respectively. The first and second elements of a tuple drawn from $\mathcal{K}$ are referred to as the register state and the flag state respectively. The register and flag state of some $k \in \mathcal{K}$ are addressed $fst(k)$ and $snd(k)$.

6.5.2 Algorithm

The type inference algorithm is shown in Algorithm 8. To aid readability, subscripts of join and meet operations are omitted; the exact operation being used can be inferred from the operands.

The analysis iterates over a vector of instructions ***instrs*** that constitutes the program. Each instruction is uniquely identified by the program point immediately prior. Instructions shall be denoted $\langle p_x : op\ arg_1,\ ... \rangle$, where p_x is the program point, op is an assembler operation and arg_i is an operand. Let the set of all instructions be denoted I. The set of predecessors and the set of successors of an instruction i is denoted $pred(i)$ and $suc(i)$ respectively. For each instruction i, the $\boldsymbol{in}[i]$ and $\boldsymbol{out}[i]$ states are elements of $\mathcal{K}$. The states initially begin empty ($\bot_{\mathcal{K}}$). The algorithm iteratively propagates flag and register states between the ***in*** and ***out*** states whilst inferring register types. Flag states are propagated backwards only, whereas register states are propagated in both directions. Care is taken to ensure that flag state does not propagate backwards beyond the definition site of each flag. Similarly, register state should not propagate outside of the scope of each register's definition. Propagation continues until the ***in*** and ***out*** states achieve a fixpoint. Since most information is being propagated backwards, a fixpoint is reached quicker if the program is iterated backwards, hence the call to Reverse.

For each instruction, the analysis first calls upon the functions UsesFlags, DefinesFlags and DefinesRegs (at line 8):

- The function UsesFlags : $I \rightarrow \mathcal{F}$ is an oracle indicating the flag usage of an instruction. For example, UsesFlags($\langle p_x$: `jge rdi`$\rangle$) returns the mapping $\{o \mapsto \text{True}, s \mapsto \text{True}, z \mapsto \text{False}, c \mapsto \text{False}\}$, thus indicating that the instruction uses the overflow and sign flags. This function is used to add information to the flag states of the analysis. This information is propagated backwards to flag definition sites.

Algorithm 8 Inferring register types.

```
 1: function InferTypeDefs(instrs)
 2:     for i in instrs do
 3:         in[i] ← out[i] ← ⊥_K
 4:     end for
 5:     repeat
 6:         out' ← out
 7:         in' ← in
 8:         for i in Reverse(instrs) do
 9:             ⟨f_def, f_use, r_def⟩ ← ⟨DefinesFlags(i), UsesFlags(i), DefinesRegs(i)⟩
10:
11:             fst(in[i]) ← (⨆_{p∈pred(i)} fst(out[p])) ⨆ (fst(out[i]) ⨅ r_def)
12:             snd(in[i]) ← (snd(out[i]) ⨅ f_def) ⨆ f_use
13:
14:             ⟨r_inf, r_xfer⟩ ← ⟨InferredRegs(i, snd(out[i])), XferredRegs(i, fst(out[i]))⟩
15:             fst(out[i]) ← (⨆_{p∈suc(i)} fst(in[p])) ⨆ r_xfer ⨆ r_inf
16:             snd(out[i]) ← (⨆_{p∈suc(i)} snd([p]))
17:         end for
18:     until out' = out ∧ in' = in
19:     return out
20: end function
```

- The function DefinesFlags : $I \to \mathcal{F}$ is another oracle that indicates which flags are defined by an instruction. Take for example the `cmc` instruction (complement carry flag). DefinesFlags$(\langle p_x : \texttt{cmc} \rangle) = \{o \mapsto \text{True}, s \mapsto \text{True}, z \mapsto \text{True}, c \mapsto \text{False}\}$, meaning that the carry flag is mutated whilst the other flags are untouched. At first this may seem counter intuitive. However, at the point where a flag $c \in C$ is redefined, flag state for c should not be propagated backwards any further. The above result of DefinesFlags is constructed in such a way that the meet ($\sqcap_{\mathcal{F}}$) with an element of $\mathcal{F}$ erases the usage information of the carry flag.

- The function DefinesRegs : $I \to \mathcal{R}$ indicates which registers are defined by an instruction. For example, DefinesRegs$(\langle p_x : \texttt{mov rax, 9} \rangle) = \{rax \mapsto \emptyset, rbx \mapsto \{s, u\}, rcx \mapsto \{s, u\}\}$, therefore indicating that `rax` is defined by the instruction. The result is constructed in such a way that the meet ($\sqcap_{\mathcal{R}}$) with a register state will erase known types of the `rax` register. This enables the analysis to limit register types to the scope of each register's definition.

Lines 10 and 11 update the register ***in*** state of the current instruction. The new ***in*** register state is the join of the register state held by predecessor ***out***

states and the current instruction's ***out*** state. This has the effect of propagating register state both backwards and forwards. Note however, that the new register state is met ($\sqcap_{\mathcal{R}}$) with the register definitions of the current instruction. This has the effect of preventing type information from being propagated back beyond the definition site of the register. Line 11 then propagates flag state backwards from the current instruction's ***out*** state, in the process introducing any flag uses of the current instruction. If the current instruction defines any flags, then their usage information is also erased at this point, thus preventing them from being propagated backwards further.

The remainder of the inner loop deals with the update of $\boldsymbol{out}[i]$. This is achieved using two more functions:

- Given an instruction and a flag state, the function InferredRegs : $I \times \mathcal{F} \rightarrow \mathcal{R}$ determines which (if any) types are inferred by an instruction. This is how register type information enters the analysis. The inferred type information varies depending upon the instruction, for example InfersTypes($\langle p_x$: `cmp rax, 2`$\rangle$, f) infers signed type upon `rax` if $f(o) \vee f(s)$. This is because when either the overflow flag or the sign flag is set in the mapping, a successor instruction has used the flags for a signed interpretation of `rax`. Similarly, the same instruction infers unsigned type upon `rax` if $f(c)$.

- Some instructions suggest that the types of its operands are the same. For example, the instruction $\langle$`cmp rax, rbx`$\rangle$, suggests that the type of `rax` is the same as the type of `rbx` and vice versa. The types are said to be transferred. In such a case, the function XferredTypes : $I \times \mathcal{R} \rightarrow \mathcal{R}$ returns a register state such that the types of `rax` and `rbx` have been merged.

Line 14 updates the register state of the $\boldsymbol{out}[i]$. This is the join of the successor ***in*** register states with any newly inferred types or types transferred between the current instruction's operands. Line 15 then propagates flag use information from the ***in*** states of successors into the current instruction's ***out*** state.

Once the analysis reaches a fixpoint, the ***out*** state is returned, from which the types of each register at each program point can be extracted.

6.6 Experimental Results.

ILP, hence MILP, is NP-hard, which begs the question of what scale of problem can be tackled by the techniques proposed in this chapter? To investigate the feasibility of the approach and the impact of type inference, tooling was developed. Given a binary, a function name (or address) and some input intervals, the tooling automatically performs the intra-procedural interval analysis described in this chapter. The bulk of the tooling is written in Python. The PyElfTools module was used to extract function addresses from a binary and the Distorm3 module was used for x86-64 disassembly. Ocaml bindings to the Parma Polyhedra Library (PPL) were used for MILP solving [5]. Note that PPL was chosen for its arbitrarily large integer support, which is required to accommodate the large M constants in the constraint systems.

Experiments were conducted for six small x86-64 functions:

bubble A bubble sort algorithm.

worked The worked example program shown in Figure 28.

fib An iterative implementation of a Fibonacci number generator.

euclid A subtractive implementation of Euclid's GCD algorithm.

fold A function to fold a list of integers with addition.

listsum A function to compute the pointwise addition of two equally sized lists.

Input intervals were devised for each of the above experiments and the problem was passed to the tooling. In all cases, the intervals computed (not shown) were precise. Since it is the feasibility of approach at question, each problem was solved twice, once without type inference and once with type inference. Each experiment is given 180 seconds in which to terminate. This is to investigate the value of type recovery in the interval analysis.

Figure 32 shows typical solving times of each of the experiments both with type inference (indicated by a ✓) and without type inference (indicated by a ✗). The number of variables in the MILP problem are also shown for comparison. In general, the solving times are worse than expected, although it is evident that

Func.	# Regs	# Instrs	Type Inf.	# Vars	Time (s)
bubble	5	19	✗	787	> 180
			✓	484	135
worked	5	19	✗	332	60
			✓	183	3
fib	6	15	✗	572	> 180
			✓	342	24
euclid	3	14	✗	347	> 180
			✓	239	24
fold	5	9	✗	311	35
			✓	254	14
listsum	7	10	✗	451	76
			✓	388	40

Figure 32: Experimental results

type inference does improve the solving times. This is most evident for the worked example program, where the solving time with type inference is 5% of that without type inference. In all cases type inference also reduces the number of optimisation variables required. This would suggest that the improved solving times are a direct consequence of using fewer optimisation variables.

Note that without type inference some experiments timed-out (took greater than 180 seconds to solve). Note too that the solving times do not appear to grow linearly with the number of optimisation variables (reflecting the NP-hard nature of ILP).

6.7 Chapter Summary

The methods presented in this chapter have shown that it is possible to encode piecewise linear functions into mixed-integer linear programming problems. In turn, this allows the encoding of modular arithmetic behaviours into a range analysis underpinned by mixed-integer linear programming. This means that integer overflow scenarios that arise in binary programs can be soundly and faithfully modelled. The ability to model these scenarios is paramount when auditing sensitive binary code, where an integer overflow could lead to security vulnerabilities, or worse, published exploits. It was also shown that decision variables can be used to eliminate the need for a binary search over many smaller linear programs (this

was the approach used in Chapter 5). A type inference was also shown which minimises the number of optimisation variables required to model mixed-signedness and as a consequence, improves the performance of the analysis.

Experimental results indicate that the performance of the method needs to be improved before it can be considered feasible. One possible approach would be to use a more compact representation of the abstract semantics, which would in turn reduce the size of the constraint system and number of optimisation variables used. For example, single static assignment (SSA) could be used to achieve this. As it stands, the method introduces interval bounds for every register at every program point, regardless of whether the registers were mutated or not.

Chapter 7

Related Work

Range analysis has a long history in compilers and verification, dating back to the seminal work of Harrison [61] who leveraged ranges for compiler optimisations. Whilst this work is not formalised by abstract interpretation (which is hardly surprising since AI was formalised that same year) Harrison's ideas are strongly reminiscent of the fundamental underpinnings of the framework. His analysis operates over ranges of possible values, i.e. the interval domain. Information regarding ranges is propagated and "narrowed" (not in the sense of fixpoint acceleration, but rather from $\top$ down) using an iterative solving algorithm which uses a work-list, ψ. When describing his termination argument, he explains that:

> "Additions are made to ψ only when the range of a variable at some *def* point is narrowed. But since computer representations allow for only a finite number of values, this narrowing can only be performed a finite number of times."

In retrospect, this is very similar to the termination argument behind the Galois connection approach to abstract interpretation, where, because the abstract lattice is finite and the transfer functions are monotonic, termination is guaranteed under the ascending chain condition. In Harrison's case however, termination is guaranteed under the descending chain condition, as ranges are initialised to $[-\infty, \infty]$ and iteratively strengthened. Further, he explains that "the guarantee that a loop will only execute a few trillion times is not comforting" and then proceeds to propose a mechanism not dissimilar to widening, thereby ensuring fast termination.

Of course since Harrison's work, range analysis has been studied thoroughly, especially since the acceptance of abstract interpretation in both academic and industrial circles. This chapter discusses recent developments in and around the topic of range analysis. Specifically, literature of relevance to the work presented in Chapters 3, 4, 5 and 6 is discussed.

7.1 Abstraction of Boolean Formulae

Of particular relevance to the work shown in Chapters 3 and 4, where the aim is to directly abstract Boolean formulae, is the problem of automated transfer function synthesis. Conventionally, when designing an abstract interpretation, a set of abstract transfer functions parameterised by predecessor states is defined. The result of each transfer function should over-approximate the actual states that may arise. Of course, it is hoped that the transfer functions compute the best over-approximations of the concrete states, and that more importantly, the transfer functions are sound. Because transfer functions are usually defined manually, there is no such intrinsic guarantee of either soundness or precision. For this reason, many have devised techniques by which to automatically generate transfer functions from concrete descriptions of program semantics.

Brauer et al. show that transfer functions for an interval abstraction can be synthesised automatically from a Boolean formula using universal quantification [16]. The aim of the method is to compute approximate transfer functions that characterise the possible register ranges for each basic block in a binary program. The method begins with Boolean formulae that model the concrete semantics of each individual CPU instruction (as described in Chapter 4). Each formula maps bit-vectors representing input registers to bit-vectors that represent the mutated output registers. Then the semantics of a block can be expressed by composing the smaller single-instruction formulae to give a composite block semantics formula. Next, bit-vectors are introduced to represent the lower and upper bounds of the input and output registers of the block, then using universal quantification, the bounds are constrained so as to describe the block semantics as an update of interval bounds. The formula is is converted into conjunctive normal form (CNF) using the Tseitin transform [115], then quantifiers are eliminated, first

the existentially quantified Tseitin variables and then universally quantified variables. This is essentially the same quantifier elimination problem that is addressed in Chapter 4. The authors eliminate the quantifiers using binary resolution and $\forall$-reduction. As previously mentioned, this approach to quantifier elimination has complexity issues. Recently, a new quantifier elimination technique, DSS, has surfaced [56] which may be applicable to Brauer's work. DSS will be discussed shortly in Section 7.2.

Reps et al. show that transfer functions can be synthesised with the aid of almost any decision procedure [96]. Reps' idea is that, if a decision procedure is available that can be called upon for solutions to a concrete transformer (an oracle), then the best abstract transfer function can be iteratively derived. The decision procedure, of course, could easily be a SAT solver, thus the method can be used to abstract Boolean formulae. Upon each iteration, the oracle is asked for a solution, which is then abstracted and joined ($\sqcup$) with the abstractions obtained from the previous iterations. For each solution given by the oracle, a blocking constraint is added to the solver instance; this ensures that subsequent solutions are not included in the already accumulated abstraction. When the oracle has no further solutions to give, the abstraction characterises the most precise transfer function possible, also referred to as the best transformer. The method has the advantage that it is parameterised by both the concrete and abstract domains, so the framework could easily be used to abstract Boolean formulae into the domain of choice. Note however that, because each solution obtained from the oracle is an under-approximation, the transfer function is refined from the bottom of the abstract lattice upwards. It follows that the method must be run to completion before a sound over-approximation has been derived. In other words, the algorithm is not anytime. Premature termination of the algorithm will yield an under-approximated transfer function, whereas in static analysis, an over-approximation is typically required for soundness.

By comparison with the previously mentioned transfer function synthesis techniques, the ideas presented in Chapters 3 and 4 aim to compute ranges from a Boolean formula without explicitly synthesising intermediate transfer functions. Rather, given a Boolean formula representing a sequence of CPU instructions, the algorithms aim to directly compute a range for a register at a given program point. The range may then be refined into an over-approximate set should more

precision be required. Furthermore, the set abstraction algorithm, unlike Reps', may be halted prematurely to yield a sound over-approximation.

Outside of transfer function synthesis, symbolic decision trees offer an alternative way by which to relate numeric values to Boolean expressions [14], for example in a reduced product abstract domain. Blanchet et al. deploy this tactic to abstract an expression such as `x = (y < 3)` in high-level C programs. The authors propose that a decision tree should be used to relate Boolean decisions, then each leaf node corresponds to an element drawn from a numeric abstract domain (the authors use the interval). This construction allows a sequence of Boolean decisions to be related to numeric properties. Unfortunately, the size of a decision tree varies greatly depending upon the ordering of the variables within. In the worst case, the size of the tree is exponential in the number of variables. To work around this issue, the authors implement variable packing [34], where Boolean decision variables are grouped into small packs whose members are determined through inter-variable dependencies. Since the variables of any two given packs need not be related, several smaller and unrelated decision trees may then be constructed, whose aggregate size is much smaller than a large decision tree relating all Boolean variables. Unfortunately, to be tractable, the size of the packs must be limited to at most four variables. Whilst a similar approach could be used for the goal of relating status flags to numeric register values, instead the work in Chapters 3 and 4 approaches the problem by using a SAT solver. This sidesteps problems relating to variable orderings, therefore techniques like variable packing are not required. Although the Boolean satisfiability problem is NP-hard, modern SAT solvers such as MiniSat [43] often perform adequately and can be used as a black box.

Using a bit-level decision procedure like SAT has the advantage that bitwise details, like those of the status flags, can be easily modelled. However, others have taken approaches which do not explicitly take into account the semantics surrounding the status flags. Cifuentes and Van Emmerik [27] have shown how to compute numeric ranges for switch tables using reverse slicing. The idea is that high level constructs can be recognised by syntactic matching against known compiler idioms. The advantage of such an approach is that no bit-level model of the program is required. For example a switch table is usually implemented as follows. First a bounds check is performed and a conditional jump will skip

the remainder of the switch code if the operand is out of range. If the operand is in range, an address is computed (usually by addition and shifting), followed by an indirect jump to this address. Cifuentes suggests scanning backwards upon reaching an indirect jump so as to extract a range of possible jump targets from the bounds check.

Similarly, in their strided-interval analysis, Balakrishnan et al. [7] use slicing to extract high-level predicates from binary code. A high-level predicate is required to correctly partition an abstract state between the true and false branches of a conditional jump. For example, consider the instruction sequence ⟨`cmp rax, 5; jb` *addr*⟩. Whether or not the conditional jump is taken depends upon whether `rax` < 5 in an unsigned context. Balakrishnan proposes that a lookup table be used to map an instruction that defines the status flags and an instruction that reads the status flags to a high-level predicate. In fact, this was the same technique used by the work in Chapters 5 and 6 to partition abstract states at control flow divergence points.

Approaches that depend upon slicing to extract ranges or predicates may be effective for binaries conforming to a known standard complication model, but outside of this restricted domain they are fragile. Since slicing is entirely syntactic, diversified or obfuscated binaries, or even binaries generated by unfamiliar tool-chains, are likely to give inconclusive or incorrect results. For example, consider the task of specifying a high-level predicate for the following sequence of instructions: ⟨`shl eax, 4; inc eax; ja 0x1234`⟩. The meanings of these instructions are: shift `eax` left four times, increment `eax`, and jump-if-above to `0x1234`. It may be tempting to devise a high-level predicate for the pairing of `inc` and `ja`. This is troublesome for two reasons. Firstly, the `ja` instruction (jump-if-above) only holds true to its name if it is preceded by a `cmp` instruction, which in this case it is not. Actually, `ja` transfers control flow if both the carry and zero flags are clear (`cf`$= 0 \wedge$ `zf`$= 0$). Secondly, and more importantly, to disregard the `shl` instruction is incorrect because `inc` does not set the carry flag. This means that when `ja` comes to read the status register, it is reading the carry flag as defined by `shl` and the zero flag as defined by `inc`. Therefore, specifying a high-level predicate for each possible instruction sequence is a non-trivial task in itself.

7.2 Quantifier Elimination

The problem of quantifier elimination (QE) for Boolean formulae is by now well recognised. Given a quantified Boolean formula f, a formula g must be found which is quantifier-free, yet logically equivalent to f. Elimination of quantifiers is often a necessary step should a quantified formula need to be passed to a SAT solver. As discussed in Chapter 4, the standard quantifier elimination techniques are impractical in cases where a large number of variables are quantified. Several new methods have been proposed.

Biere showed that the standard QE techniques (resolution and expansion) can be feasible if operations are carefully scheduled so as to keep the formula size small [11]. In this work the author operates upon a formula with alternating quantifier scopes, for example $\forall W.\ \exists X.\ \forall Y.\ \exists Z.\ f$, where W, X, Y and Z are sets of variables and f is a CNF formula. The aim is to derive an equisatisfiable formula which is either quantifier free, or contains only existential quantifiers, thus allowing the application of a SAT solver. To this end, quantifiers are eliminated one by one. The quantifier scopes are ranked from 1 (outermost) to m (innermost). At each stage Biere eliminates either one of the existentially quantified variables at scope m using resolution, or he eliminates one of the universally quantified variables at scope $m-1$ using a slight variation of expansion-based universal QE. When the latter elimination occurs, fresh variables and further existential quantifiers must be introduced to preserve equisatisfiability. For example, given a formula $\forall\{x_1\}.\ \exists\{x_3\}.\ (x_1 \vee x_2 \vee x_4) \wedge (\neg x_3 \vee x_4)$, the outer universal quantifier is eliminated by expansion as follows: $\exists\{x_3, x_3'\}.\ (0 \vee x_2 \vee x_4) \wedge (\neg x_3 \vee x_4) \wedge (1 \vee x_2 \vee x_4) \wedge (\neg x_3' \vee x_4)$. Notice that the fresh existentially quantified variable x_3' is used in one half of the expanded formula. This effectively works around the need to eliminate the innermost existential quantifiers first, albeit at the cost of introducing further existential quantifiers. After each elimination step, the formula is simplified (by $\forall-$reduction, absorption, etc.) to reduce its overall size. The exact sequence of resolve/expand operations is determined by a scheduling algorithm which aims to minimise the size of the CNF at each elimination step. For each possible next elimination step, a cost function computes the number of literals by which the formula would grow and the step with the least cost is chosen. Experimental results show, not surprisingly, that the method fares well for quantified boolean

formulae (QBF) with structure, but much worse for randomly generated QBF.

Brauer et al. [19] have shown that the task of existential quantifier elimination itself can be almost entirely delegated to a SAT solver. The method works by first taking a formula $\exists X.\ f$ in CNF. To eliminate the existentially quantified variables, the formula must be projected upon the set of variables $Y = vars(f) \backslash X$. The clauses of f are encoded into so-called dual-rail form. This is achieved by replacing the positive and negative occurrences of each $y_i \in Y$ with fresh Boolean variables y_i^+ and y_i^-. Extra constraints are added to ensure that each y_i^+ and y_i^- are not simultaneously true. This encoding allows a model to express the absence of a variable in an implicant. Brauer et al. show that by augmenting the dual-rail encoded formula with a sorting network, a set of short implicants of the input formula can be found by incremental SAT. By finding short implicants, the need to enumerate each and every possible SAT model is avoided and in the process the existential quantifiers are eliminated. A secondary dualisation step can be used to convert the set of implicants into a set of implicates. The approach is attractive, since it derives a quantifier-free equivalent formula in either disjunctive normal form (DNF) or CNF using SAT as a black box. Further, because the method aims to avoid enumerating redundant models, solving times should be fast. The authors claim that the method finds the minimal set of shortest implicants (the prime implicants), however, due to an unfortunate error in the formulation of the method, it is entirely possible for the method to find redundant implicants. The authors assume that they are able to find the shortest implicants due to the fact that any implication holding with respect to the original input formula also holds within the corresponding dual-rail encoding, however, this is not true. For example $x_1 \implies \exists x_3.\ (x_1) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$, yet $x_1^+ \not\implies \exists x_3.\ (x_1^+) \wedge (x_2^+ \vee \neg x_3) \wedge (\neg x_2^- \vee x_3) \wedge (\neg x_2^+ \vee \neg x_2^-)$. The consequence of this is that the prime implicates may not be found and because redundant implicants may be found, the performance of the method may also be impacted.

Most recently, Goldberg et al. [56] proposed quantifier elimination by the derivation of dependency sequents (DDS). The aim of DDS is to find a quantifier free formula equivalent to $\exists X.F$, where F is a CNF formula and X is a set of variables. Clauses are systematically added to F which are implied by F, but that are not dependent upon the variables of X. Then, at the final stage,

any clause containing a variable in X may then be dropped to give a quantifier-free equivalent formula in CNF. Under the hood, the method is actually based upon binary resolution, but the beauty of the method is the way in which very few resolvent clauses are generated. The possible assignments to the variables over which $\exists X.\ F$ is defined forms a binary decision tree. Each node in the tree represents a partial assignment to the variables. Suppose the current partial assignment is denoted $\boldsymbol{q}$. An edge in the tree represents the extension of $\boldsymbol{q}$ with a further assignment of one more as-of-yet unassigned variable. The algorithm begins at the root node of the tree with $\boldsymbol{q} = \langle\rangle$ and searches the tree in a depth first fashion. At each node, the algorithm specialises the formula to reflect the new variable assignment (denoted F_q), before attempting to prove the redundancy of each unassigned quantified variable. A quantified variable is redundant under the current assignment, F_q, if it appears monotonically or if F_q is unsatisfiable. For each redundant quantified variable a dependency sequent (D-seq) of the form $(\boldsymbol{q}) \rightarrow Z$ is generated, meaning that under the assignment, $\boldsymbol{q}$, the set of variables Z is redundant in F. When all unassigned quantified variables are proven redundant, the algorithm need not search deeper down the current branch, thus the search space is pruned. Binary resolution only occurs when both choices of a branching decision yield the formula unsatisfiable. In such a case, two falsified clauses, one from each decision, are resolved and the resolvent clause is added to F. Once the search space is exhausted, clauses containing any quantified variables are removed to arrive at a quantifier-free equivalent formula. Whilst the method is sensitive to the order in which decisions are placed in the search tree, the authors' experimental results suggest that the algorithm fares well for medium to large-sized problems even with random variable ordering.

The work undertaken in Chapter 4 differs from the aforementioned approaches in that it explores the feasibility of performing quantifier elimination as a mathematical optimisation problem. This approach permits the use of an objective function, which can be used to find short implicates. In combination with blocking constraints, the method aims to minimise the number of redundant implicates found. However, this work suffered from poor solving times, so for now either Goldberg's method or Biere's method is preferred. Although Goldberg's algorithm can only perform existential quantifier elimination, once the existential quantifiers have been removed, $\forall$-reduction could be used to find a quantifier-free

formula equivalent to $\forall I.\ \exists T.\ f$, as proposed in Chapter 4.

7.3 Fixpoint Acceleration

Fixpoint acceleration is often a necessary step in making an abstract interpretation tractable in practice. Cousot and Cousot documented the need for widening in their seminal paper describing the AI framework [31]. Defining a bare-minimum widening operator is simple, however, crafting an operator which gives adequately precise fixpoints is non-trivial and is a topic which is still being explored.

In its purest form, widening is distinct from the Galois connection approach to abstract interpretation [32]. Typically, however, widening is used as a fixpoint acceleration technique to enhance a standard Galois-connection-style interpretation. The widening for $\mathcal{G}$ programs, as discussed in Chapter 2, worked in this way. In this example, the widening scheme was rather crude and merely extrapolated any unstable interval bound to a conservative bound when a chain of abstract iterates exceeded a given length.

A better approach is widening with thresholds [13]. By this method, a number of intermediate widening thresholds are defined which are tried in turn. To illustrate, consider the finite chain of iterates $\langle [0,0], [0,1], [0,2], \ldots, [0,41], [0,42] \rangle$. The chain is long enough to warrant widening. Suppose the sequence is accelerated after three iterations of instability. Instead of using aggressive widening to extrapolate to $[0, +\infty]$, widening to intermediate thresholds spaced 20 apart could be used in between normal iterations to give the iterates:

$$\langle [0,0], [0,1], [0,2], [0,3], [0,20], [0,21], [0,40], [0,41], [0,50], [0,50] \rangle$$

In this instance, widening with thresholds gives a much more precise post-fixpoint than by aggressive widening. This raises the question of what a suitable choice of thresholds is. On one hand, the further apart the thresholds are spaced, the faster convergence will be. On the other hand, the closer together the thresholds are, the more precise the fixpoint will be. Of course, there is no need for thresholds to be evenly spaced apart either.

In response to the problem of deciding suitable widening thresholds, Simon et al. [109] show an approach based on the observation that useful thresholds often

occur at the point where an abstract semantic equation flips from being unsatisfiable to being satisfiable. This change often signifies a change in control flow behaviour which is likely to propagate meaningful abstract information. Simon et al. strive to automatically infer useful thresholds by observing the rate of change of an increasing (or decreasing) bound between two consecutive abstract iterates. The number of iterations that would be required to activate a dormant abstract equation can then be estimated and adopted as a candidate widening threshold (called a landmark in this context). By leapfrogging to landmarks the analysis can be accelerated whilst retaining better precision than by regular threshold widening or by aggressive widening. The authors implement their widening scheme for the polyhedral domain and note that the approach works well for typical counted loops, e.g. `for (i = 0; i < 64; i++)`. Simon found, however, that imprecision is incurred if the loop is not counted linearly.

Around the same time as the discovery of widening with landmarks, Gopan et al. [57] made a similar observation, that often loops are modal in nature. They exemplify this with a loop that executes 100 times; in the first 50 iterations, incrementing a variable, then in the remaining 50 iterations, decrementing that same variable. The loop can thus be seen as being composed of two distinct behavioral phases. The authors raise the issue that traditional widening will typically extrapolate abstract states prior to the enablement of the second phase of the loop, and thus beyond the ideal threshold. By contrast to widening with landmarks, Gopan suggests that, in addition to a main interpretation, a pilot interpretation is run alongside the main analysis. It is the role of the pilot to "look ahead" in isolation at each of the behavioural phases. Iterates of the pilot analysis are accelerated with standard widening and narrowing operators so as to meet a fixpoint quickly. The abstract information discovered by the pilot is then integrated into the main analysis. The pilot analysis then advances to the next unexplored behaviour. Through exploring the loop behaviours in isolation, weak abstract information is not propagated across as-of-yet disabled behaviours.

Recently, Bouissou et al. [15] proposed a novel fixpoint acceleration technique which is not based upon widening, but rather upon the classical theory of sequence transformation. The authors show that the iterates of an interval analysis can be accelerated to within a close vicinity of a precise fixpoint. The method works by running a standard Kleene iteration in parallel with a second analysis which

uses sequence transformation to compute accelerated iterates. The accelerated iterates are computed by flattening the abstract states into vectors which are then transformed by either the Aitken method or the ϵ-algorithm. If the accelerated iterates reach a fixpoint, or if the improvement between the accelerated iterates falls below a δ constant, then the accelerated iterates are joined with the iterates of the standard Kleene iteration. Unlike a widening operator, which extrapolates based upon the last two abstract iterates, the acceleration operator in Bouissou's method works over a larger history of iterates. Also, the authors make it clear that their method is not a replacement for widening, as the accelerated iterates may not include the least-fixpoint (hence δ). Actually, the method is more accurately described as means to infer widening thresholds. Note that in some cases, the accelerated iterates may not stabilise at all, meaning that conventional widening may be required.

By contrast to fixpoint acceleration, the work in Chapter 5 (and Chapter 6 for that matter) uses a somewhat novel solving method. Since mathematical optimisation is used in place of Kleene iteration, the problem of finding the least-fixpoint is delegated to a constraint solver. In turn, because there is no iteration strategy (other than the pivot operations of the solver, which are not exposed) and because optimisation problems are always guaranteed to eventually terminate, the concept of widening no longer applies. This can be seen as an advantage, just as long as the solver terminates within reasonable time.

7.4 Novel Solving Strategies

It was the pioneering work of Rugina et al. [99] that showed that the least solution of a system of abstract fixpoint equations can be formulated as a linear program (LP). By this approach they show that it is possible to perform a symbolic range analysis. First the lower and upper bounds of each variable at the end of each each block are expressed as a system of linear inequalities. A cost function is used to minimise the distance between the lower and upper bounds, therefore finding tightest interval bounds that satisfy the constraints, i.e. the least-fixpoint. The inequalities then undergo a transformation into a second system of inequalities that expresses the interval bounds as linear combinations of the initial states of the program variables. For example, if a program were to deploy two variables,

`x` and `y`, then each interval bound would be expressed as a linear combination of the form $c_1x_0 + c_2y_0 + c_3$, where x_0 and y_0 represent the initial values of `x` and `y`. The constraint system is solved to find symbolic interval bounds. The advantages of such an approach are twofold. Firstly, the least-fixpoint is computed directly, so there is no explicit iteration strategy and therefore, no need for fixpoint acceleration techniques. Secondly, the approach effectively computes a symbolic summary of a collection of blocks. These block summaries could be used in a bottom-up inter-procedural analysis, where block summaries could be composed to model function calls. Their approach is ingenious, but only considers a subset of conditional branches. The work in Chapter 5 shows a method by which to remedy this.

Goubault et al. also leverage the fact that the computation of ranges can be spun as an optimisation problem. The approach is conceptually similar to the work of Rugina, but differs in a couple of aspects. Firstly, the outcome of the analysis is a collection of concrete interval values rather than symbolic bounds. Secondly, each basic block has associated with it a Boolean decision variable, e_i, which is constrained in such a way that it indicates the reachability of the basic block. Since this construction relies upon discrete integer values and products of variables, the optimisation problem is a MINLP (Mixed-Integer Non-Linear Program). The least solution of the resulting MINLP characterises the least solution of the abstract semantics. The introduction of integer variables into mathematical optimisation problems can be problematic in terms of performance, so the authors provide two alternative solving methods. By choosing an assignment for each of the e_i decision variables, the MINLP problem is reduced to a LP. Each different assignment to the decision variables yields a post-fixpoint, the smallest of which is the least-fixpoint. The first solving approach entails solving all of the possible LPs, thus enumerating the post-fixpoints. Finally the fixpoint with the least solution is selected as the least-fixpoint. This approach is conceptually similar to the approach taken in Chapter 5, where the complementary constraints form a disjunctive binary search space. However, since Goubault et al. lack heuristics, an exponential number of LPs may need to be solved. In order to avoid solving a large number of LPs, the authors also provide a solving approach based upon policy iteration, where the e_i decision variables form a basis for policy selection.

Policy Iteration (PI) itself is not commonly associated with static analysis, so

a brief description of the topic is offered. PI has its roots in artificial intelligence, where it is widely understood as a method for solving Markov Decision Problems (MDPs). An MDP typically involves the transition of an agent through a finite set of states to a goal state. A reward is assigned to each goal state and optionally a cost is associated with each state transition. State transitions may be stochastic so as to model uncertainty. A policy is a mapping from each state to one of the possible transitions. Since there are many possible policies, the goal of the exercise is to choose an optimal policy according to a cost function. A textbook example would be an animal (the agent), in a grid environment (the states), moving in one of four predefined directions (transitions) to a food source (positive reward goal) as quickly as possible (transition cost) whilst avoiding being killed by predators (negative reward goals) and without getting lost (stochastic transitions). The optimal policy is a mapping from each state to the transition that probabilistically yields the highest reward. The optimal policy of a MDP may be found in one of two ways. The first, value iteration, involves computing the utility of each and every state; that is, the long-term reward associated with a state, considering all possible future transitions. To this end, a set of fixpoint equations are solved before the optimal policy is selected based upon the utility of each state. A second solving technique, policy iteration, does not find the optimal policy by computing the utility of all of the states upfront, but rather by solving repeated smaller fixpoint problems and incrementally choosing an improved policy.

Recently, ideas from MDPs, and specifically policy iteration, have made an appearance in the abstract interpretation literature. Costan et al. [30] observe that Kleene iteration is a kind of value iteration and that many of the fundamental concepts behind MDPs can be mapped to abstract interpretation. By approaching abstract interpretation from this new angle, a solving strategy more akin to policy iteration may be applied as a replacement to Kleene iteration. The authors provide a framework that can be used for any arbitrary numerical domain that forms a complete lattice. They convey the idea exemplified by a standard interval analysis. Monotonic abstract semantics are constructed in the usual manner, before a number of states are identified; these states will provide a basis for policy selection. In the case of the authors' example, a state is any semantic equation containing an abstract lattice meet, $\sqcap$. A policy assigns to each of the intersection operations, $G_1 \sqcap G_2$, a meaning. For example the meaning of an intersection

operation could be:

- $l([a,b],[c,d])$ – The left side of the intersection, i.e. $[a,b]$.
- $r([a,b],[c,d])$ – The right side of the intersection, i.e. $[c,d]$
- $m([a,b],[c,d])$ – The merge of the two intervals, i.e. $[a,d]$.

When solving begins, the abstract states begin at the top of the lattice and an initial policy is selected using heuristics. One of the intersection operations is selected and solved to a fixpoint under the current policy. This is called the value determination phase and can be performed by either Kleene iteration or by an external decision procedure. Note that during value determination, a single abstract semantic equation is solved in isolation of the rest. After value determination terminates, the entire system of semantic equations is evaluated once to decide if a global fixpoint has been met and if not, policy improvement is invoked. The intersection point whose bounds are still unstable is chosen and the meaning of the intersection is changed. Value determination is then invoked upon the semantic equation whose policy was improved. The process continues until a fixpoint is met. The interval bounds at the time of termination are a post-fixpoint of the abstract semantics. Experimental results show that often the least-fixpoint is found and in far fewer operations than with Kleene iteration plus widening. The quality of the obtained fixpoint greatly depends upon the choice of initial policy, but the authors provide a method by which to detect and recover from a post-fixpoint.

Gaubert et al. [49] showed that the technique could just as easily be applied to relational domains that form a complete lattice. In this work, the authors show that PI can be applied to the zone abstract domain [85] to capture relationships of the form $x - y \leq c$, and to the template constraint matrix abstract domain (TCM) [102] to capture relationships of the form $\boldsymbol{ax} + c \geq 0$ where each a_i is fixed a priori. Several other works extending this idea have surfaced [50, 51, 2] and most recently the work of Gawlitza et al. [52] shows that the idea can even be applied to arbitrary polynomial template domains.

The work presented in Chapters 5 and 6 could be used to enhance the approaches underpinned by PI. For example, Costan's analysis uses Kleene iteration with widening for the value determination phase, but this could be replaced by linear programs akin to those described in this thesis.

7.5 Modulo Arithmetic Abstraction

One way to model modulo arithmetic in an abstract interpretation is to make the modulo behaviour inherent in the abstract domain. Amongst the first of the domains to take this approach was the congruence domain [59]. A congruence is a relational domain, meaning that it can capture the relationship between different program variables. Numeric domains, by contrast to relational domains, can only express numeric values regardless of the values of other variables. The simplest congruence is of the form $x \equiv_m d$, where x, m and d are all integers. Here x is said to be congruent to d modulo m, sometimes written $x \bmod m = d$. Such a congruence is useful for expressing a set of strided values. For example $x \equiv_5 0$, where x is a non-negative integer, represents a set $\{0, 5, 10, 15, \ldots\}$. More generally, congruences can express arbitrary modulo relationships between any number of variables, i.e. $\boldsymbol{cx} \equiv_m d$, for example $3x + 2y \equiv_{256} 4$. Granger's main contribution was the formalisation of the congruence as an abstract domain for use in the AI framework. Since Granger's pioneering work on the congruence domain, several novel uses of the congruence have surfaced, for example as an inter-procedural analysis [87] and for bit-level reasoning [72].

Congruences are useful for capturing variable relationships and strides, but fall short with regards to range analysis, i.e. the congruence domain possesses no ability to express that a modulo relationship can only hold between a given range. The result is a loss of precision. Brauer et al. [18] note that the precision loss of congruences and intervals manifest differently. The interval domain can only represent a range of consecutive values, whereas the congruence domain can only express modulo relationships between variables. The authors point out that the intersection of the two abstractions yields an expressive and more precise means of abstraction. For example, suppose the set of non-negative integers $x = \{4, 6, 8\}$ is to be abstracted; the congruence $x \equiv_2 0$ expresses that x must be even, but is unable to place lower and upper bounds upon x. On the other hand, the interval $x = [4, 8]$ can express the desired bounding, but must introduce spurious numeric values to do so; namely the odd integers 5 and 7. Yet the intersection of the two abstractions gives a precise abstraction, i.e. $\{0, 2, 4, 6, 8, 10, 12, \ldots\} \cap \{4, 5, 6, 7, 8\} = \{4, 6, 8\}$. Brauer uses this observation to infer more precise register values for embedded AVR micro-controller code. Given a Boolean formula which

characterises a sequence of AVR instructions, decision procedures based upon SAT [72, 29] are used to compute separately: a) a lower and upper numeric bound (an interval) for each register at each program point, and b) the congruent closure describing register relationships at each program point. An operator `reduce` is then used to compute the numeric intersection of the two abstractions. The range analysis presented in Chapter 5 could have benefited from the introduction of congruent information in this way, as the analysis was unable to account for variable strides (see Section 5.6 on Page 102).

Instead of combining abstract domains, Simon and King [110] propose the modification of the classical convex polyhedral domain to incorporate modulo behaviour into a relational analysis for C code. The authors observe that implicit wrapping occurs when dynamic type casts occur in C and as a result subtle programming errors can be introduced. Instead of reporting each integer overflow as a potential error, which burdens the analyst who must decide if each case is a false positive, a different approach is proposed. First, the concretisation mapping, which in this instance maps polyhedra to bit-vectors, is adjusted to perform implicit wrapping. This means that the arithmetic operations of the abstract semantics themselves need not be littered with explicit two's complement considerations. Only when the abstract state is concretised, does wrapping occur. There is but one exception to this rule; numeric comparisons. Consider a polyhedral abstraction of a program state, P, and a guard $x \leq y$. In the classic polyhedral setting, the set of points in P which satisfy the guard is simply expressed by the intersection of P with the half-space imposed by $x \leq y$ (denoted $P \sqcap [\![x \leq y]\!]$). This is correct for unbounded integers, but because computer integers wrap, simple half-space intersection is no longer correct. To model guards, Simon proposes that the feasible space of P be partitioned into smaller polyhedra according to a number of quadrants, where each quadrant represents a different wrapping mode. By mapping each of the smaller polyhedra back into the valid integer range and taking the convex hull, the authors arrive at a new polyhedron P', whose feasible space describes the integer values that may arise after wrapping. It is now safe to compute $P' \sqcap [\![x \leq y]\!]$ conventionally. By this method the consequences of a mistaken integer overflow, for example an out of bounds array write, can be reported, rather than the integer overflow itself, which is more likely to be dismissed as a false positive. As with any polyhedral analysis, the success of the approach

is predicated on the judicious application of widening [109, 76].

More recently, numeric modulo abstract domains have been proposed. The circular linear progression (CLP), as proposed by Sen et al. [107], is one such example. A CLP is a 3-tuple of the form (l, u, δ) where l is a start value, u is an end value and δ is a step value. The interpretation of a CLP, C, is then $\gamma(C) = \{a_i = l + i\delta \mid 0 \leq i \leq s\}$, where $i \in \mathbb{Z}$ and s is the smallest non-negative integer such that $a_s = u$. For example, the CLP $(0, 8, 6)$ when modelling unsigned 4-bit numbers corresponds to the concrete set $\{0, 2, 6, 8, 12\}$. Sen shows that the set of CLPs forms a complete lattice and thus can be used directly within the abstract interpretation framework. Although at first the CLP is reminiscent of the strided interval [47], by contrast, integer wrap is inherent in the construction of the CLP. The abstract semantics of an analysis for CLPs operate in two phases. The first computes the update as though wrapping does not occur, then the second computes the result of wrapping if it is deemed possible. The wrapping itself is separated into disjoint cases similarly to as shown in Chapter 6.

Most recently, Navas et al. [90] devised a signedness-agnostic wrapping interval analysis for LLVM IR (Low Level Virtual Machine Intermediate Representation). The authors use a concrete domain of bit-vectors. The interpretation of a wrapped interval is then:

$$\begin{aligned} \gamma(\bot) &= \emptyset \\ \gamma(\llparenthesis x, y \rrparenthesis) &= \begin{cases} \{x, \ldots, y\} & \text{if } x \leq y \\ \{0^w, \ldots, y\} \cup \{x, \ldots, 1^w\} & \text{otherwise} \end{cases} \\ \gamma(\top) &= \mathcal{B} \end{aligned}$$

where 0^w and 1^w describe the w-bit vectors $\langle 0, 0, 0, \ldots \rangle$ and $\langle 1, 1, 1, \ldots \rangle$ respectively. Note that the case where $x \not\leq y$ expresses a set which straddles the unsigned overflow boundary. For example, $\llparenthesis 15, 1 \rrparenthesis$ for $w = 4$ expresses the set of vectors $\{\langle 0, 0, 0, 0 \rangle, \langle 0, 0, 0, 1 \rangle\} \cup \{\langle 1, 1, 1, 1 \rangle\}$. The same set abstracted by the classical interval is $[0, 15]$, which is much weaker by comparison. Abstract transfer functions are then developed, including ones which take into consideration signedness specific numeric comparisons.

Interestingly, the authors show that their wrapped interval coincides with a CLP of step value one ($\delta = 1$) and they further dismiss Sen's claim, that the set

of CLPs forms a lattice:

> "They [Sen et al.] give detailed abstract operations and refer to their abstract domain as a lattice. Setting the stride in their strided intervals to 1 results in precisely the concept of wrapped intervals that we use in this paper. Hence, as will become clear, their claim that the domain of (wrapped) strided intervals has lattice structure is not correct."

Note that this is not to say that the domains are not partially ordered. Nevertheless, the lack of lattice structure poses some rather irregular challenges. There is no join (or meet) operator available, so to realise control flow joins Navas uses a pseudo-meet operator, which unfortunately fails to be monotone. Furthermore, the domain correspondence does not form a Galois connection. The upshot of these quirks is that solving must deploy widening even though the abstract domain is finite.

The methods described in Chapters 5 and 6 are distinct from the work discussed here in that they explore the possibility of encoding modulo behaviours into linear optimisation problems whilst avoiding the need for widening at all. For now, the methods operate over the standard interval domain, however, future work may find that it is also possible to integrate domains such as the wrapped interval or the congruence into optimisation problems.

Chapter 8

Future Work and Conclusions

In summary, this thesis has surveyed the application of decision procedures to the problem of static range analysis of binaries. The work was presented in contrast to the widely used abstract interpretation framework and with emphasis on the applications to security and verification; two topics of growing importance in the industrial, military and governmental sectors.

8.1 Reflection upon Chapters 3 and 4

The first body of work showed how to abstract Boolean formulae as ranges to overcome the so called "chicken and egg" problem of control flow graph (CFG) recovery. This work was presented in two parts.

In Chapter 3, a method was shown which abstracts the satisfying models of a Boolean formula as a range. The range could then be refined into an incrementally more precise over-approximate set of Boolean satisfiability (SAT) models. This was achieved by repeatedly calling a SAT solver to systematically find the minimum and maximum satisfying models that, when interpreted as interval bounds, correspond to a sound and tight under- or over-approximation of the satisfying models. Experiments were conducted showing that the method was able to efficiently recover sets of models that were representative of a set of indirect jump targets.

Chapter 4 then showed how range and set abstraction could be applied to sub-vectors of a SAT model (a register at a given program point) using a quantified Boolean formula of the form $\forall I.\ \exists T.\ f$. Because range and set abstraction

is underpinned by SAT, and because SAT cannot work directly on a quantified formula containing universal quantifiers, the quantifiers must be eliminated. The standard quantifier elimination (QE) methods are hindered by complexity issues, so a new quantifier elimination method was proposed, based upon mathematical optimisation. By this approach, it was shown that a cost function and blocking constraints could be used to minimise the amount of redundant computation required. Experimental results were shown to back this claim.

It is particularly encouraging that ranges and sets can be automatically abstracted from a Boolean formula. CPU instructions lend themselves well to Boolean encoding, since for soundness and precision, bitwise details (such as the status flags) must be captured. By the proposed method, the need for explicit transfer functions is dispelled, as the abstraction is formulated directly from concrete Boolean formulae. Unlike traditional range analyses though, the method is only able to infer a range or set of values for a specific register at a specific program point. In the case where a program utilises multiple indirect jumps, several independent analyses would be required to collect ranges for each of the target registers at each indirect jump site. Although this was not an avenue that was explored in this thesis, little extra engineering is required to accommodate this. In fact, the same Boolean formula can be used repeatedly for each of the desired jump targets; even the quantifier elimination stage need only occur once. Only the sub-vector that is being minimised/maximised would change each time. This does highlight the requirement for the method to be fast however. Unfortunately the performance of the approach as a whole was not evaluated because the quantifier elimination component of the analysis did not perform sufficiently well. Instead, DDS [56] or Biere's method [11] could be used as a replacement for the QE algorithm proposed in Chapter 4. Assuming that by one of these approaches, QE is no longer a performance issue, future work could see the evaluation of the overarching method (range and set abstraction of sub-vectors) as a whole. If the method fares well, then it could be used as part of the CFG recovery process to incrementally grow an under-approximate control flow graph.

It is unfortunate that the performance of the QE method proposed in Chapter 4 is below par. The likely cause of the poor performance is the discrete integer variables that are used in the formulation of the mixed-integer linear programs

(MILP) underpinning the approach. By using integer variables in a linear program, the problem is promoted to NP-hard. There is no obvious way to remedy the unsatisfactory solving times, but it is hoped that the work presented here will inspire future work in this area. For example it may be possible to reformulate the problem using a different decision procedure, such as SMT (Satisfiability Modulo Theories). Another possible route would be to relax the MILPs to a series of plain linear programs (without integer variables) in an approach more akin to that of Chapter 5.

8.2 Reflection upon Chapters 5 and 6

Chapters 5 and 6 then assume that the CFG has been recovered by the methods proposed before or otherwise. The remaining work explores the feasibility of decision procedures as a replacement for Kleene iteration in an abstract interpretation. Again, the work is partitioned between two chapters.

Based upon the pioneering work of Rugina et al. [99], Chapter 5 showed that sound range analysis of binary code can be undertaken as a series of linear optimisation problems. The method begins with a standard abstract interpretation defined over intervals. Instead of solving the abstract semantics via Kleene iteration, a reformulation into an optimisation problem is proposed. To preserve soundness, *min* and *max* terms are used when modelling conditional branching constructs. This renders the optimisation problem non-linear, but it is shown that the least solution (and therefore the best over-approximation) of the non-linear constraints can be found by solving a series of simpler linear optimisation problems. By this approach the *min* and *max* terms are decomposed into linear constraints and so-called complementary constraints. The possible assignments to the complementary constraints form a disjunctive search space, therefore the best solution can be found through the repeated solving of linear relaxations of the overarching non-linear optimisation problem. Although the worst case number of LPs that need to be solved is high, heuristics were developed that in practice allow the problems to be solved quickly and in a fraction of the worst case number of LPs. Experimental results were presented to support this claim.

The work shown in Chapter 6 extends the range analysis proposed in Chapter 5 to allow the modelling of integer overflow scenarios. This was motivated by the

fact that unforeseen integer overflows often have serious security implications. The work was based upon the realisation that possibly overflowing arithmetic operations can be expressed as piecewise linear updates. The update cases can easily be encoded as linear constraints and then decision variables can be used to select the case that should apply. Further, since *min* and *max* themselves can be considered piecewise linear functions, the optimal solution to the overarching MILP can be found without the need for complementary constraints or a binary search. Control flow and reachability was also reformulated to take advantage of decision variables, thereby allowing a uniform representation of an unreachable block. Because signed an unsigned integers have differing overflow behaviours, each interpretation of a register was modelled separately, and in doing so, more optimisation variables were introduced. In the interest of reducing the number of variables necessary, a type inference was devised to infer which signedness interpretations are actually required on a per-program basis. Experimental results suggested that the type inference does indeed improve the performance of the analysis.

The biggest advantage of the proposed approaches is that, because Kleene iteration is not used, fixpoint acceleration (such as widening) is not required. Instead the method uses an automated linear constraint solver as a black-box to find the least-fixpoint directly. From an engineering standpoint this is beneficial, as many good off-the-shelf linear constraint solvers exist. Also, the analyst need not worry about the design of a widening operator, which is also an advantage. From a performance standpoint, the analysis proposed in Chapter 5 shows that decision procedures are at least competitive with traditional Kleene iteration. In fact, when both heuristics are enabled, the method appears to solve the experimental samples in constant time. Future work could try scaling the method up to larger problems, perhaps in an inter-procedural setting, to see if the solving times remain small.

The extension proposed in Chapter 6 suffers from poor performance, even when the proposed type inference was enabled. Again, this can probably be attributed to the use of integer optimisation variables. A different method could be used that, instead of using explicit decision variables, uses a search tree similar to the method presented in Chapter 5. In addition to the complementary constraints that are used for *min* and *max* constraints, disjunctive constraints could be used

to model the different overflow cases that arise in arithmetic operations. By this approach, case selection is lifted outside of the decision procedure and into the searching of the tree. Of course, this would mean that the search tree would no longer be composed of binary decisions (complementary constraints), but rather of n-ary decisions, depending upon the number of overflow modes that are modelled for each operation. The use of heuristics would be paramount in this method, as by increasing the depth and arity of the tree, the worst case number of LPs that must be solved is substantially increased.

An alternative approach is to delegate overflow mode case selection to policy iteration [30]. By this approach each potentially overflowing arithmetic operation would correspond to a state, and a policy would assign each an overflow mode. Although this appears to be a possible direction for future work, note that by re-introducing a system of fixpoint equations, widening will once again be required, which is counter-intuitive in the context of this thesis.

8.3 Final Remarks

All in all, the question of whether decision procedures are a worthy drop-in replacement for traditional methods for binary analysis remains partially unanswered. It has been shown that SAT and linear optimisation do offer the necessary level of expression to formulate range analyses. Furthermore, when a decision procedure is used to replace Kleene iteration, there is no need to specify a widening operator, since a least-fixpoint can be found directly.

The main shortcomings with the methods presented were related to performance. SAT and LP when used in isolation (Chapters 3 and 5) performed adequately, but integer variables in optimisation problems have proven to be particularly problematic. This strongly motivates the development of new solving strategies for integer linear programming (ILP). The discreteness of ILP makes it a flexible platform; indeed it has been shown that even certain non-linear properties can be expressed by ILP. However, this power of expression comes at the cost of poor performance. It could be argued that because ILP is NP-hard, it is unlikely that better solving strategies could exist. On the other hand, the same could be said for SAT, which too is NP-hard, but several good solving strategies exist that perform very well for the majority of real-world SAT problems.

Of course, there are many other aspects of binary analysis not considered in this thesis that could benefit from the application of decision procedures. Most notably, the methods presented only tracked general purpose integer registers. Many instruction set architectures implement specialised fixed point and floating point registers, such as the SSE and x87 registers found in Intel CPUs. Furthermore, the modelling of overlapping registers was not considered. This is of particular importance as there is interplay between the values that each register may assume. For example, the 64-bit `rax` register found in x86-64 architectures shares its lower 32-bits with the 32-bit `eax` register. This means that, for example, any operation that mutates `eax` will also mutate `rax`. Quirks like this are merely an artefact of legacy, but compilers readily take advantage of the overlap to optimise code.

Another area that may be opportune for the application of decision procedures is the modelling of memory contents. Whilst it was shown that inferring values of memory offsets can be beneficial in a range analysis, the actual contents of the memory cells themselves were not modelled. The ability to track the contents of memory would undoubtedly give a much more precise range analysis, particularly for execution environments using a stack-based calling convention or for register starved CPU architectures. Some architectures, such as PIC, have but a single general purpose register, meaning that values are constantly being swapped in and out of memory. A memory model would also be useful for a side-effect analysis which decides if a function respects callee save conventions. Before returning, callee save functions are expected to restore certain registers to their original state, as they were prior to the function call. Usually a compiler will temporarily store the register values on the stack in the function prelude and restore them later in the function epilogue, but the ability to verify this would be useful, especially since compiler tool-chains often contain bugs.

Some areas of static binary analysis are likely to remain problematic for the foreseeable future, with or without the support of decision procedures. Obfuscation deployed in malware is one such example. Far fewer assumptions can be made in this setting, as foul play is to be expected. Obfuscated binaries may conform to no particular compilation model or calling convention, functions may not return, instructions may be fabricated on the fly (as in self-modifying code), packers and virtual machines may be embedded into the code and so on. Of course, malware

writers make a conscious effort to make their binary code difficult to reverse. Unfortunately, the analysis of malware is likely to remain difficult in the long-term because as new reverse engineering methods are published, malware authors are likely to reactively adjust their tactics.

Appendix A

Proofs

A.1 Proofs for Chapter 2.

Definition 34 (Projection from L)**.** *The function* $proj_L : L \times \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \to V$ *projects out the value-set of a single variable from a concrete state:*

$$proj_L(l, var) = \begin{cases} \{x \mid \langle x, y, z\rangle \in l\} & \text{if } var = \mathtt{x} \\ \{y \mid \langle x, y, z\rangle \in l\} & \text{if } var = \mathtt{y} \\ \{z \mid \langle x, y, z\rangle \in l\} & \text{if } var = \mathtt{z} \end{cases}$$

Definition 35 (Projection from M)**.** *The function* $proj_M : M \times \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \to W$ *projects out the set of signs for a single variable from an abstract state:*

$$proj_M(\langle x, y, z\rangle, var) = \begin{cases} x & \text{if } var = \mathtt{x} \\ y & \text{if } var = \mathtt{y} \\ z & \text{if } var = \mathtt{z} \end{cases}$$

Definition 36 (Shorthand projection operator)**.** *Given an element* d *drawn from either* L *or* M*, the shorthand syntax* $d_{[var]}$ *refers to either* $proj_L(d, var)$ *or* $proj_M(d, var)$ *depending upon the type of* d*:*

$$d_{[var]} = \begin{cases} proj_L(d, var) & \text{if } d \text{ is of type } L \\ proj_M(d, var) & \text{if } d \text{ is of type } M \end{cases}$$

Definition 37 (Ordering on M in terms of projection)**.** *The ordering on M shown in Definition 4 (Page 27) can be written in terms of projection:*

$$\begin{array}{ccc} \langle x,y,z\rangle \sqsubseteq_M \langle x',y',z'\rangle & \iff & x \sqsubseteq_W x' \wedge y \sqsubseteq_W y' \wedge z \sqsubseteq_W z' \\ & \Vert\Vert & \\ s \sqsubseteq_M t & \iff & \bigwedge_{i\in\{\mathtt{x},\mathtt{y},\mathtt{z}\}} s_{[i]} \sqsubseteq_W t_{[i]} \end{array}$$

Definition 38 (Ordering and domain operations of V)**.**

$$\begin{array}{ccc} s \subseteq_V s' & \iff & s \subseteq s' \\ s \cup_V s' & \triangleq & s \cup s' \\ s \cap_V s' & \triangleq & s \cap s' \end{array}$$

Definition 39 (Correspondence between V and W)**.**

$$\begin{array}{rcl@{\qquad\qquad}rcl} \alpha_V : V & \to & W & \alpha_V(v) & = & \{sign(j) \mid j \in v\} \\ \gamma_W : W & \to & V & \gamma_W(w) & = & from\_sign(w) \end{array}$$

Lemma 1. *If an $s \in L$ is a subset of a $t \in L$, then it follows that a single-variable projection of a variable i from s is a subset of the projection of i from t, i.e.:*

$$\forall s \in L.\ \forall t \in L.\ s \subseteq_L t \implies \bigwedge_{i\in\{\mathtt{x},\mathtt{y},\mathtt{z}\}} s_{[i]} \subseteq_V t_{[i]}$$

Proof. Since $\subseteq_L \triangleq \subseteq$, it follows that any $\langle x',y',z'\rangle$ drawn from s must also be a member of t. Therefore, the following properties hold:

$$x' \in s_{[\mathtt{x}]} \wedge x' \in t_{[\mathtt{x}]} \qquad y' \in s_{[\mathtt{y}]} \wedge y' \in t_{[\mathtt{y}]} \qquad z' \in s_{[\mathtt{z}]} \wedge z' \in t_{[\mathtt{z}]}$$

Because this holds for every $\langle x',y',z'\rangle \in s$ and because $\subseteq_V \triangleq \subseteq$, it follows that $s_{[\mathtt{x}]} \subseteq_V t_{[\mathtt{x}]} \wedge s_{[\mathtt{y}]} \subseteq_V t_{[\mathtt{y}]} \wedge s_{[\mathtt{z}]} \subseteq_V t_{[\mathtt{z}]}$.

□

Theorem 1. *The correspondence between L and M is a Galois connection $L \underset{\alpha_L}{\overset{\gamma_M}{\leftrightarrows}} M$ if the correspondence between V and W is a Galois connection $V \underset{\alpha_V}{\overset{\gamma_W}{\leftrightarrows}} W$.*

Proof. The correspondence between L and M is a Galois connection if:

$$\forall l \in L.\ \forall m \in M.\ \alpha_L(l) \sqsubseteq_M m \iff l \subseteq_L \gamma_M(m)$$

Using the definition of $\sqsubseteq_M$ (Definition 37, Page 162), the above can be rewritten as follows:

$$\forall l \in L.\ \forall m \in M.\ \bigwedge_{i \in \{\mathbf{x},\mathbf{y},\mathbf{z}\}} \left(\alpha_L(l)_{[i]} \sqsubseteq_W m_{[i]}\right) \iff l \subseteq_L \gamma(m)$$

Then, through the application of Lemma 1 (Page 163) and since $\alpha_L(l)_{[i]} = \alpha_V(l_{[i]})$ and $\gamma_M(m)_{[i]} = \gamma_W(m_{[i]})$, the above is equivalent to:

$$\forall l \in L.\ \forall m \in M.\ \bigwedge_{i \in \{\mathbf{x},\mathbf{y},\mathbf{z}\}} \left(\alpha_V(l_{[i]}) \sqsubseteq_W m_{[i]}\right) \iff \bigwedge_{i \in \{\mathbf{x},\mathbf{y},\mathbf{z}\}} \left(l_{[i]} \subseteq_V \gamma_W(m_{[i]})\right)$$

The above must be true if the correspondence between single-variable projections of L and M forms a Galois connection, i.e.:

$$\forall v \in V.\ \forall w \in W.\ \alpha_V(v) \sqsubseteq_W w \iff v \subseteq_V \gamma_W(w)$$

□

Theorem 2. *The correspondence between single variable projections of L and M form a Galois connection, $V \underset{\alpha_V}{\overset{\gamma_W}{\leftrightarrows}} W$, i.e. $\forall v \in V.\ \forall w \in W.\ \alpha_V(v) \sqsubseteq_W w \iff v \subseteq_V \gamma_W(w)$*

Proof. Starting with the forward direction:

$$\forall v \in V.\ \forall w \in W.\ \alpha_V(v) \sqsubseteq_W w \implies v \subseteq_V \gamma_W(w)$$

By Definition 3 (Page 27) and Definition 38 (Page 163), we have $\sqsubseteq_W \triangleq \subseteq$ and $\subseteq_V \triangleq \subseteq$, so the above is equivalent to:

$$\forall v \in V.\ \forall w \in W.\ \alpha_V(v) \subseteq w \implies \forall n \in v.\ n \in \gamma_W(w)$$

Then, via case analysis it can then be shown that the implication holds for an arbitrary choice of $n \in v$:

Case $n < 0$: From the definition of α_V (Definition 39, Page 163) it follows that

$(-) \in \alpha_V(v)$. Then $(-) \in w$ also, since $\alpha_V(v) \subseteq w$. Therefore, by Definition 39 (Page 163), $\gamma_W(w) \supseteq [-\infty, -1]$, so it must be the case that $n \in \gamma_W(w)$.

Case $n = 0$**:** From the definition of α_V it follows that $0 \in \alpha_V(v)$. Then $0 \in w$ also, since $\alpha_V(v) \subseteq w$. Therefore $\gamma_W(w) \supseteq \{0\}$, so it must be the case that $n \in \gamma_W(w)$.

Case $n > 0$**:** From the definition of α_V it follows that $(+) \in \alpha_V(v)$. Then $(+) \in w$ also, since $\alpha_V(v) \subseteq w$. Therefore $\gamma_W(w) \supseteq [1, +\infty]$, so it must be the case that $n \in \gamma_W(w)$.

Because the implication holds for any arbitrary $n \in v$, it follows that the implication is true for all $n \in v$. Now the converse must be shown to hold:

$$\forall v \in V.\ \forall w \in W.\ \alpha_V(v) \sqsubseteq_W w \Longleftarrow v \subseteq_V \gamma_W(w)$$

By Definition 3 (Page 27) and Definition 38 (Page 163), we have $\sqsubseteq_W \triangleq \subseteq$ and $\subseteq_V \triangleq \subseteq$, so the above is equivalent to:

$$\forall v \in V.\ \forall w \in W.\ (\forall s \in \alpha_V(v).\ s \in w \Longleftarrow v \subseteq \gamma_W(w))$$

By cases it can be shown that the implication holds for any arbitrary $s \in \alpha_V(v)$:

Case $s = (-)$**:** From the definition of α_V (Definition 39, Page 163), it follows that $\exists n \in v.\ n < 0$. Then since $v \subseteq \gamma_W(w)$ it must be the case that $n \in \gamma_W(w)$. From the definition of γ_W (Definition 39, Page 163) there must be $(-) \in w$, thus $s \in w$.

Case $s = 0$**:** From the definition of α_V it follows that $0 \in v$ and since $v \subseteq \gamma_W(w)$ it follows that $0 \in \gamma_W(w)$. From the definition of γ_W there must be $0 \in w$, thus $s \in w$.

Case $s = (+)$**:** From the definition of α_V it follows that $\exists n \in v.\ n > 0$ and since $v \subseteq \gamma_W(w)$ it follows that $n \in \gamma_W(w)$. From the definition of γ_W there must be $(+) \in w$, thus $s \in w$.

Because the implication holds for any arbitrary $s \in \alpha_V(v)$, it follows that the implication is true for all $s \in \alpha_V(v)$. $\square$

Corollary 1 (L $\underset{\alpha_L}{\overset{\gamma_M}{\leftrightarrows}}$ M, described on Page 29)**.** *The correspondence between L and M forms a Galois connection, $L \underset{\alpha_L}{\overset{\gamma_M}{\leftrightarrows}} M$.*

Proof. Theorem 1 shows that the correspondence between L and M forms a Galois connection iff the correspondence between single-variable projections forms a Galois connection. Theorem 2 then shows that indeed, the correspondence between single-variable projections forms a Galois connection. □

Lemma 2. *Sign addition and subtraction are monotonic.*

Proof. Consider sign addition, i.e. $+_W : W \times W \to W$ (Definition 11, Page 31). Since the the function is binary, an ordering is defined for pairs of W:

$$\langle a, b\rangle \sqsubseteq_{W^2} \langle c, d\rangle \iff a \sqsubseteq_W c \wedge b \sqsubseteq_W d$$

Then the following must be shown:

$$\forall\langle l_1, r_1\rangle \in W^2.\ \forall\langle l_2, r_2\rangle \in W^2.\ \langle l_1, r_1\rangle \sqsubseteq_{W^2} \langle l_2, r_2\rangle \implies l_1 +_W r_1 \sqsubseteq_W l_2 +_W r_2$$

From the definitions of W^2 (above), $\sqsubseteq_W$ (Definition 3, Page 27) and $+_W$ (Definition 11, Page 31), the above is equivalent to:

$$\begin{aligned}\forall\langle l_1, r_1\rangle \in W^2.\ \forall\langle l_2, r_2\rangle \in W^2.\ l_1 \subseteq l_2 \wedge r_1 \subseteq r_2 &\implies k_1 \subseteq k_2\\ \textit{where } k_1 &= \bigcup\{s +'_W t \mid s \in l_1 \wedge t \in r_1\}\\ \textit{and } k_2 &= \bigcup\{u +'_W v \mid u \in l_2 \wedge v \in r_2\}\end{aligned}$$

Since $l_1 \subseteq l_2$ and $r_1 \subseteq r_2$, each element of $\{s +'_W t \mid \ldots\}$ is also an element of $\{u +'_W v \mid \ldots\}$. It follows that each element of k_1 is also in k_2, thus $k_1 \subseteq k_2$.

$-_W$ is monotonic by analogous reasoning. □

Definition 40 (Single-variable update for a self map of M)**.** *Given a transfer function $f : M \to M$, the single-variable update of a variable $v \in \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ is a function $f_{[v]} : M \to W$ such that $f_{[v]}(m) = f(m)_{[v]}$.*

Lemma 3. *An abstract transfer function $f : M \to M$ is monotonic if each single-variable update of which f is composed is a monotonic function $f_{[i]} : M \to W$.*

Proof. Recall that, to show a transfer function $f : M \to M$ is monotonic it must be shown that:

$$\forall a \in M.\ \forall b \in M.\ a \sqsubseteq_M b \implies f(a) \sqsubseteq_M f(b)$$

By the definition of $\sqsubseteq_M$ (Definition 37, Page 162) and because $f_{[i]}(m) = f(m)_{[i]}$, the above is equivalent to:

$$\forall a \in M.\ \forall b \in M.\ a \sqsubseteq_M b \implies \bigwedge_{i \in \{\mathtt{x},\mathtt{y},\mathtt{z}\}} f_{[i]}(a) \sqsubseteq_W f_{[i]}(b)$$

The above is true if every single-variable update is monotonic, i.e. $\forall i \in \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}.\ \forall s \in M.\ \forall t \in M.\ s \sqsubseteq_M t \implies f_{[i]}(s) \sqsubseteq_W f_{[i]}(t)$. □

Lemma 4. *The projection of a variable $v \in \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ from any $m \in M$, i.e. $m_{[v]}$, is monotonic.*

Proof. It must be shown that:

$$\forall v \in \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}.\ \forall a \in M.\ \forall b \in m.\ a \sqsubseteq_M b \implies a_{[v]} \sqsubseteq_W b_{[v]}$$

Consider the case where $v = \mathtt{x}$. By expansion of $\sqsubseteq_M$ (Definition 37, Page 162) the above is equivalent to:

$$\forall a \in M.\ \forall b \in M.\ \bigwedge_{i \in \{\mathtt{x},\mathtt{y},\mathtt{z}\}} (a_{[i]} \sqsubseteq_W b_{[i]}) \implies a_{[\mathtt{x}]} \sqsubseteq_W b_{[\mathtt{x}]}$$

Thus because the updates of $\mathtt{y}$ and $\mathtt{z}$ are independent of the update of $\mathtt{x}$, it follows that $\forall a \in M.\ \forall b \in M.\ a_{[\mathtt{x}]} \sqsubseteq_W b_{[\mathtt{x}]} \implies a_{[\mathtt{x}]} \sqsubseteq_W b_{[\mathtt{x}]}$. A similar argument holds for when $v = \mathtt{y}$ and when $v = \mathtt{z}$. □

Theorem 3. *The semantic equations shown in Section 2.2.3 (Page 29) are monotonic.*

Proof. The functional interpretation F'_i of each S'_i can be shown to be monotonic:

- $F'_1(m) = \top_M$**:** The function is constant and therefore monotonic.

- $F'_2(m) = \langle m_{[\mathtt{x}]}, \{-\}, m_{[\mathtt{z}]} \rangle$**:** According to Lemma 3 (Page 166), F'_2 is monotonic if the following three functions are monotonic:

$$F'_{2[\mathtt{x}]}(m) = m_{[\mathtt{x}]} \qquad F'_{2[\mathtt{y}]}(m) = \{-\} \qquad F'_{2[\mathtt{z}]}(m) = m_{[\mathtt{z}]}$$

  $F'_{2[\mathtt{x}]}$ and $F'_{2[\mathtt{z}]}$ are simple projections from m and are monotonic according to Lemma 4 (Page 167). The remaining function, $F'_{2[y]}$, is constant and therefore monotonic.

- $F'_3(m) = \langle m_{[\mathtt{x}]}, m_{[\mathtt{y}]}, \{+\} \rangle$**:** Similar argument as for F'_2.

- $F'_4(\langle m, n \rangle) = \langle \{0\}, m_{[\mathtt{y}]}, m_{[\mathtt{z}]} \rangle \sqcup_M n$**:** The calculation of the left- and right-hand side of the join can be thought of as independent functions of the shape $M \to M$:

$$F'_{4*}(m) = \langle \{0\}, m_{[\mathtt{y}]}, m_{[\mathtt{z}]} \rangle \qquad F'_{4**}(n) = n$$

  The join of two elements drawn from a join-semilattice (which M is) is monotonic. Therefore F'_4 itself is monotonic if the functions from which it is composed are monotonic. To that end, F'_{4*} is monotonic through a similar argument to that of F'_2. Then F'_{4**} is the ID mapping, which is monotonic.

- $F'_5(m) = m$**,** $F'_6(m) = m$**:** The functions are the ID mapping and are therefore monotonic.

- $F'_7(m) = \langle m_{[\mathtt{x}]} -_W m_{[\mathtt{y}]}, m_{[\mathtt{y}]}, m_{[\mathtt{z}]} \rangle$**:** According to Lemma 3 (Page 166), the following functions must be shown to be monotonic:

$$F'_{7[\mathtt{x}]}(m) = m_{[\mathtt{x}]} -_W m_{[\mathtt{y}]} \qquad F'_{7[\mathtt{y}]}(m) = m_{[\mathtt{y}]} \qquad F'_{7[\mathtt{z}]}(m) = m_{[\mathtt{z}]}$$

  The first is monotonic according to Lemma 2 (Page 166), then $F'_{7[\mathtt{y}]}$ and $F'_{7[\mathtt{z}]}$ are monotonic projections (Lemma 4, Page 167).

- $F'_8(m) = \langle m_{[\mathtt{x}]} \sqcap_W \{+\}, m_{[\mathtt{y}]}, m_{[\mathtt{z}]} \rangle$**:** According to Lemma 3 (Page 166), F'_8 is monotonic if the following three functions are monotonic:

$$F'_{8[\mathtt{x}]}(m) = m_{[\mathtt{x}]} \sqcap_W \{+\} \qquad F'_{8[\mathtt{y}]}(m) = m_{[\mathtt{y}]} \qquad F'_{8[\mathtt{z}]}(m) = m_{[\mathtt{z}]}$$

For the first, it must be shown that:

$$\forall a \in M.\ \forall b \in M.\ a \sqsubseteq_M b \implies (a_{[\mathtt{x}]} \sqcap_W \{+\}) \sqsubseteq_W (b_{[\mathtt{x}]} \sqcap_W \{+\})$$

Using the definitions of $\sqsubseteq_W$, $\sqcap_W$ and $\sqsubseteq_M$ (Definitions 3 and 37, Pages 27 and 162), this is equivalent to:

$$\forall a \in M.\ \forall b \in M.\ \bigwedge_{i \in \{\mathtt{x},\mathtt{y},\mathtt{z}\}} a_{[i]} \subseteq b_{[i]} \implies a_{[\mathtt{x}]} \cap \{+\} \subseteq b_{[\mathtt{x}]} \cap \{+\}$$

The above can be shown to hold via case analysis upon the existence or non-existence of a $(+)$ element in $a_{[\mathtt{x}]}$ and $b_{[\mathtt{x}]}$. First assume $(+) \notin a_{[\mathtt{x}]} \wedge (+) \notin b_{[\mathtt{x}]}$. In this case, the right-hand side is $\emptyset \subseteq \emptyset$. Now assume $(+) \notin a_{[\mathtt{x}]} \wedge (+) \in b_{[\mathtt{x}]}$. In this case, the right-hand side is $\emptyset \subseteq \{+\}$. Now assume $(+) \in a_{[\mathtt{x}]} \wedge (+) \notin b_{[\mathtt{x}]}$. In this case, $a_{[\mathtt{x}]} \not\subseteq b_{[\mathtt{x}]}$, which is inconsistent with the left-hand side. Finally assume $(+) \in a_{[\mathtt{x}]} \wedge (+) \in b_{[\mathtt{x}]}$. In this case, the-right hand side is $\{+\} \subseteq \{+\}$.

$F'_{8[\mathtt{y}]}$ and $F'_{8[\mathtt{z}]}$ are monotonic projections (Lemma 4, Page 167).

- $F'_9(m) = \langle m_{[\mathtt{x}]} +_W m_{[\mathtt{z}]}, m_{[\mathtt{y}]}, m_{[\mathtt{z}]} \rangle$**:** This function is monotonic through a similar line of reasoning to that of F'_7.

- $F'_{10}(\langle m_1, m_2 \rangle) = m_1 \sqcup_M m_2$**:** Since the join of any two elements drawn from a join-semilattice (which M is) is monotonic, F'_{10} is also monotonic.

- $F'_{11}(m) = \langle m_{[\mathtt{x}]} \sqcap_W \{+\}, m_{[\mathtt{y}]}, m_{[\mathtt{z}]} \rangle$**:** This function is monotonic through a similar line of reasoning to that of F'_8.

□

Definition 41 (Projection from K)**.** *Given a tuple $\langle x, y, z \rangle \in K$, the function $proj_K : K \times \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \to I$ gives the projection of a single variable:*

$$proj_K(\langle x, y, z \rangle, v) = \begin{cases} x & \text{if } v = \mathtt{x} \\ y & \text{if } v = \mathtt{y} \\ z & \text{if } v = \mathtt{z} \end{cases}$$

The shorthand notation given in Definition 36 (Page 162) is also extended so as to accommodate K.

Definition 42 (Ordering on K in terms of projection)**.** *The ordering upon K (Definition 14, Page 36) can be rewritten in terms of projection:*

$$\begin{array}{rcl} \langle x, y, z\rangle \sqsubseteq_K \langle x', y', z'\rangle & \iff & (x \sqsubseteq_I x') \wedge (y \sqsubseteq_I y') \wedge (z \sqsubseteq z') \\ & \Vert & \\ a \sqsubseteq_K b & \iff & \bigwedge_{i\in\{\mathtt{x},\mathtt{y},\mathtt{z}\}} a_{[i]} \sqsubseteq_I b_{[i]} \end{array}$$

Theorem 4. *The domain correspondence between L and K is a Galois connection ($L \underset{\alpha_L}{\overset{\gamma_K}{\leftrightarrows}} K$) if the correspondence between single-variable projections is a Galois connection ($V \underset{\alpha_V}{\overset{\gamma_I}{\leftrightarrows}} I$).*

Proof. The correspondence between L and K is a Galois connection when:

$$\forall l \in L.\ \forall k \in K.\ \alpha_L(l) \sqsubseteq_K k \iff l \subseteq_L \gamma_K(k)$$

By the definition of $\sqsubseteq_K$ (Definition 42, Page 170) the above is equivalent to:

$$\forall l \in L.\ \forall k \in K.\ \bigwedge_{j\in\{\mathtt{x},\mathtt{y},\mathtt{z}\}} (\alpha_L(l)_{[j]} \sqsubseteq_I k_{[j]}) \iff l \subseteq_L \gamma_K(k)$$

Then through the application of Lemma 1 (Page 163) and since $\alpha_L(l)_{[j]} = \alpha_V(l_{[j]})$ and $\gamma_K(k)_{[j]} = \gamma_I(k_{[j]})$, the following is also equivalent:

$$\forall l \in L.\ \forall k \in K.\ \bigwedge_{j\in\{\mathtt{x},\mathtt{y},\mathtt{z}\}} (\alpha_V(l_{[j]}) \sqsubseteq_I k_{[j]}) \iff \bigwedge_{j\in\{\mathtt{x},\mathtt{y},\mathtt{z}\}} (l_{[j]} \subseteq_V \gamma_I(k_{[j]}))$$

Therefore, if it can be shown that the correspondence between V and I is a Galois connection, i.e. $\forall v \in V.\ \forall i \in I.\ \alpha_V(v) \sqsubseteq_I i \iff v \subseteq_V \gamma_I(i)$, then the above holds and the correspondence between L and K forms a Galois connection. □

Theorem 5. *The correspondence between single-variable projections of L and K forms a Galois connection, $V \underset{\alpha_V}{\overset{\gamma_I}{\leftrightarrows}} I$, i.e. $\forall v \in V.\ \forall i \in I.\ \alpha_V(v) \sqsubseteq_I i \iff v \subseteq_V \gamma_I(i)$*

Proof. The proposition can be shown to be true by case analysis upon v and i.

Case $v = \bot_V \wedge i = \bot_I$: Therefore, $\bot_I \sqsubseteq_I \bot_I \iff \bot_V \subseteq_V \bot_V$.

Case $v = \bot_V \wedge i \neq \bot_I$**:** Therefore, $\bot_I \sqsubseteq i \iff \bot_V \subseteq_V \gamma_I(i)$. Since no $i \in I$ is less than $\bot_I$ and no $v \in V$ is less than $\bot_V$, this must hold.

Case $v \neq \bot_V \wedge i = \bot_I$**:** Therefore, FALSE $\iff$ FALSE.

Case $v \neq \bot_V \wedge i \neq \bot_I$**, i.e.** $v \neq \emptyset \wedge i = [l_i, u_i]$**:** From the definition of α_V and γ_I (Definition 15, Page 36), the proposition is equivalent to:

$$\forall v \in V.\ \forall [l_i, u_i] \in I.\ [min(v), max(v)] \sqsubseteq_I [l_i, u_i] \iff v \subseteq_L \{x \in \mathbb{Z} \mid l_i \leq x \leq u_i\}$$

Then by the definitions of $\sqsubseteq_I$ and $\subseteq_V$ (Definitions 13 and 38, Pages 35 and 163), the above is equivalent to:

$$\forall v \in V.\ \forall [l_i, u_i] \in I.\ l_i \leq min(v) \wedge max(v) \leq u_i \iff v \subseteq \{x \in \mathbb{Z} \mid l_i \leq x \leq u_i\}$$

□

Corollary 2. *The correspondence between L and K forms a Galois connection $L \underset{\alpha_L}{\overset{\gamma_K}{\leftrightarrows}} K$.*

Proof. Theorem 4 (Page 170) states that the correspondence between L and K is a Galois connection if it can be shown that the correspondence between V and I forms a Galois connection. Theorem 5 (above) shows that the correspondence between V and I forms a Galois connection. □

Lemma 5. *Interval subtraction is monotonic.*

Proof. Since interval subtraction is binary ($-_I : I \times I \to I$), first a lifted ordering is defined for pairs of intervals: $\langle a, b\rangle \sqsubseteq_{I^2} \langle c, d\rangle \iff a \sqsubseteq_I c \wedge b \sqsubseteq_I d$. Then it must be shown that:

$$\forall \langle a, b\rangle \in I^2.\ \forall \langle c, d\rangle \in I^2.\ \langle a, b\rangle \sqsubseteq_{I^2} \langle c, d\rangle \implies a -_I b \sqsubseteq_I c -_I d$$

Using the definitions of $\sqsubseteq_{I^2}$ (above), $\sqsubseteq_I$ (Definition 13, Page 35) and $-_I$ (Definition 17, Page 38), the above proposition can be shown to hold via case analysis upon the emptiness of a, b, c and d:

Case $a = b = c = d = \emptyset$**:**

$$\emptyset \sqsubseteq_I \emptyset \wedge \emptyset \sqsubseteq_I \emptyset \implies \emptyset -_I \emptyset \sqsubseteq_I \emptyset -_I \emptyset$$

Case $a = \emptyset \wedge b = [l_b, u_b] \wedge c = \emptyset \wedge d = [l_d, u_d]$**:**

$$\begin{array}{rrcl} & \emptyset \sqsubseteq_I \emptyset \wedge [l_b, u_b] \sqsubseteq_I [l_d, u_d] & \implies & \emptyset -_I [l_b, u_b] \sqsubseteq_I \emptyset -_I [l_d, u_d] \\ \therefore & [l_b, u_b] \sqsubseteq_I [l_d, u_d] & \implies & \emptyset \sqsubseteq_I \emptyset \end{array}$$

Case $a = \emptyset \wedge b = [l_b, u_b] \wedge c = [l_c, u_c] \wedge d = [l_d, u_d]$**:**

$$\begin{array}{rrcl} & \emptyset \sqsubseteq_I [l_c, u_c] \wedge [l_b, u_b] \sqsubseteq_I [l_d, u_d] & \implies & \emptyset -_I [l_b, u_b] \sqsubseteq_I [l_c, u_c] -_I [l_d, u_d] \\ \therefore & [l_b, u_b] \sqsubseteq_I [l_d, u_d] & \implies & \emptyset \sqsubseteq_I [l_c, u_c] -_I [l_d, u_d] \end{array}$$

Case $a = [l_a, u_a] \wedge b = \emptyset \wedge c = [l_c, u_c] \wedge d = [l_d, u_d]$**:** Similarly.

Case $a = [l_a, u_a] \wedge b = \emptyset \wedge c = [l_c, u_c] \wedge d = \emptyset$**:** Similarly.

Case $a = b = \emptyset \wedge c = [l_c, u_c] \wedge d = [l_d, u_d]$**:** Similarly.

Case $a = [l_a, u_a] \wedge b = [l_b, u_b] \wedge c = [l_c, u_c] \wedge d = [l_d, u_d]$**:**

$$\begin{array}{rrcl} & [l_a, u_a] \sqsubseteq_I [l_c, u_c] \wedge [l_b, u_b] \sqsubseteq_I [l_d, u_d] & \implies & [l_a, u_a] -_I [l_b, u_b] \sqsubseteq_I [l_c, u_c] -_I [l_d, u_d] \\ \therefore & l_c \leq l_a \wedge u_a \leq u_c \wedge l_d \leq l_b \wedge u_b \leq u_d & \implies & [l_a - u_b, u_a - l_b] \sqsubseteq_I [l_c - u_d, u_c - l_d] \\ \therefore & l_c \leq l_a \wedge u_a \leq u_c \wedge l_d \leq l_b \wedge u_b \leq u_d & \implies & l_c - u_d \leq l_a - u_b \wedge u_a - l_b \leq u_c - l_d \\ \therefore & l_c \leq l_a \wedge u_a \leq u_c \wedge l_d \leq l_b \wedge u_b \leq u_d & \implies & l_c - l_a \leq u_d - u_b \wedge u_a - u_c \leq l_b - l_d \end{array}$$

Then since $l_c \leq l_a$ it follows that $l_c - l_a$ must be ≤ 0, and because $u_b \leq u_d$ it follows that $u_d - u_b \geq 0$. Further, since $u_a \leq u_c$ it follows that $u_a - u_c \leq 0$, and because $l_d \leq l_b$ it follows that $l_b - l_d \geq 0$. Thus the proposition is satisfied:

$$\underbrace{l_c - l_a}_{\leq 0} \leq \underbrace{u_d - u_b}_{\geq 0} \quad \wedge \quad \underbrace{u_a - u_c}_{\leq 0} \leq \underbrace{l_b - l_d}_{\geq 0}$$

Any other case is inconsistent with the assumption $\langle a, b\rangle \sqsubseteq_{I^2} \langle c, d\rangle$, therefore making the proposition true. □

Definition 43 (Single-variable update for a self map of K)**.** *Given a transfer function $f : K \to K$, the single-variable update of a variable $v \in \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$ is a function $f_{[v]} : K \to I$ such that $f_{[v]}(m) = f(m)_{[v]}$.*

Lemma 6. *A transfer function $f : K \to K$ is monotonic if each single-variable update is a monotonic function $f_{[v]} : K \to I$.*

Proof. A transfer function $f : K \to K$ is monotonic when:

$$\forall a \in K.\ \forall b \in K.\ a \sqsubseteq_K b \implies f(a) \sqsubseteq_K f(b)$$

By the definition of $\sqsubseteq_K$ (Definition 42, Page 170), and since $f_{[v]}(k) = f(k)_{[v]}$, the above is equivalent to:

$$\forall a \in K.\ \forall b \in K.\ a \sqsubseteq_K b \implies \bigwedge_{v \in \{\mathtt{x},\mathtt{y},\mathtt{z}\}} f_{[v]}(a) \sqsubseteq_I f_{[v]}(b)$$

Thus, if the single-variable updates are monotonic, i.e. $\forall v \in \{\mathtt{x},\mathtt{y},\mathtt{z}\}.\ \forall s \in K.\ \forall t \in K.\ s \sqsubseteq_K t \implies f_{[v]}(s) \sqsubseteq_I f_{[v]}(t)$, then the above also holds. □

Lemma 7. *The projection of a variable $v \in \{\mathtt{x},\mathtt{y},\mathtt{z}\}$ from any $k \in K$ is monotonic, i.e. $k_{[v]}$ is monotonic:*

$$\forall v \in \{\mathtt{x},\mathtt{y},\mathtt{z}\}.\ \forall s \in K.\ \forall t \in K.\ s \sqsubseteq_K t \implies s_{[v]} \sqsubseteq_I t_{[v]}$$

Proof. Consider the case where $v = \mathtt{x}$. Recall that in this instance $s_{[\mathtt{x}]}$ and $t_{[\mathtt{x}]}$ are synonyms for $proj_K(k, \mathtt{x})$ and $proj_K(k, \mathtt{x})$. Therefore, by Definition 41 (Page 169) the proposition is equivalent to:

$$\forall \langle x, y, z\rangle \in K.\ \forall \langle x', y', z'\rangle \in K.\ \langle x, y, z\rangle \sqsubseteq_K \langle x', y', z'\rangle \implies x \sqsubseteq_I x'$$

By the definition of $\sqsubseteq_K$ (Definition 14, Page 36), the above is equivalent to:

$$\forall \langle x, y, z\rangle \in K.\ \forall \langle x', y', z'\rangle \in K.\ x \sqsubseteq_I x' \wedge y \sqsubseteq_I y' \wedge z \sqsubseteq_I z' \implies x \sqsubseteq_I x'$$

Thus the projection of $\mathtt{x}$ from a $k \in K$ is monotonic. Projections of $\mathtt{y}$ and $\mathtt{z}$ are monotonic by symetric reasoning. □

Theorem 6. *The semantic equations shown in Section 2.3.2 (Page 37) are monotonic.*

Proof. The functional interpretation F'_i of each S'_i can be shown to be monotonic:

- $F'_1(k) = \top_K$**:** The function is constant and is therefore monotonic.

- $F_2'(\langle p,q,r\rangle, \langle s,t,u\rangle) = \langle [5,5], q, r\rangle \sqcup_K \langle s,t,u\rangle$**:** The function can be thought of as the join of two smaller independent functions:

$$F'_{2*}(\langle p,q,r\rangle) = \langle [5,5], q, r\rangle \qquad F'_{2**}(\langle s,t,u\rangle) = \langle s,t,u\rangle$$

The join of two elements drawn from a join-semilattice (which indeed K is) is monotonic. Therefore $\sqcup_K : K \times K \to K$ is monotonic, meaning that F'_2 is monotonic if it can be shown that F'_{2*} and F'_{2**} are monotonic. F'_{2**} is the ID mapping and is trivially monotonic. According to Lemma 6 (Page 172), F'_{2*} is monotonic if the following functions are monotonic:

$$F'_{2*[\mathtt{x}]}(\langle p,q,r\rangle) = [5,5] \qquad F'_{2*[\mathtt{y}]}(\langle p,q,r\rangle) = q \qquad F'_{2*[\mathtt{z}]}(\langle p,q,r\rangle) = r$$

The first of the three functions is constant and thus monotonic. The remaining two functions are simple projections from K, which according to Lemma 7, are monotonic.

- $F'_3(\langle x,y,z\rangle) = \langle x \sqcap_I [1,+\infty], y, z\rangle$**:** According to Lemma 6, it must be shown that the following functions are monotonic:

$$F'_{3[\mathtt{x}]}(\langle x,y,z\rangle) = x \sqcap_I [1,+\infty] \qquad F'_{3[\mathtt{y}]}(\langle x,y,z\rangle) = y \qquad F'_{3[\mathtt{z}]}(\langle x,y,z\rangle) = z$$

$F'_{3[\mathtt{x}]}$ is monotonic because the meet of two elements drawn from a meet-semilattice (which indeed I is) is monotonic. $F'_{3[\mathtt{y}]}$ and $F'_{3[\mathtt{z}]}$ are simple projections and are thus monotonic according to Lemma 7 (Page 173).

- $F'_4(\langle x,y,z\rangle) = \langle x -_I [1,1], y, z\rangle$**:** According to Lemma 6 (Page 172), it must be shown that the following functions are monotonic:

$$F'_{4[\mathtt{x}]}(\langle x,y,z\rangle) = x -_I [1,1] \qquad F'_{4[\mathtt{y}]}(\langle x,y,z\rangle) = y \qquad F'_{4[\mathtt{z}]}(\langle x,y,z\rangle) = z$$

The first is monotonic according to Lemma 5 (Page 171), then $F'_{4[\mathtt{y}]}$ and $F'_{4[\mathtt{z}]}$ are simple projections and are thus monotonic by Lemma 7 (Page 173).

- $F'_5(\langle x,y,z\rangle) = \langle x \sqcap_I [-\infty,0], y, z\rangle$**:** The function is monotonic via a similar argument to that of F'_3.

□

Theorem 7. *∇_K (defined on Page 43) is a correct widening operator:*

$$\forall a \in K.\ \forall b \in K.\ a \sqsubseteq_K (a\ \nabla_K\ b) \quad \bigwedge \quad \forall a \in K.\ \forall b \in K.\ b \sqsubseteq_K (a\ \nabla_K\ b)$$

Proof. Let $a = \langle x, y, z\rangle$ and $b = \langle x', y', z'\rangle$. By the definition of ∇_K (Definition 20, Page 43) the proposition is equivalent to:

$$\begin{array}{lllll} \forall \langle x,y,z\rangle \in K.\ \forall \langle x',y',z'\rangle \in K. & \langle x,y,z\rangle & \sqsubseteq_K & \langle x\ \nabla_I\ x', y\ \nabla_I\ y', z\ \nabla_I\ z'\rangle & \wedge \\ \forall \langle x,y,z\rangle \in K.\ \forall \langle x',y',z'\rangle \in K. & \langle x',y',z'\rangle & \sqsubseteq_K & \langle x\ \nabla_I\ x', y\ \nabla_I\ y', z\ \nabla_I\ z'\rangle & \end{array}$$

Then by the definition of $\sqsubseteq_K$ (Definition 14, Page 36):

$$\begin{array}{ll} (\forall \langle x,y,z\rangle \in K.\ \forall \langle x',y',z'\rangle \in K. & x\ \sqsubseteq_I (x\ \nabla_I\ x') \wedge y\ \sqsubseteq_I (y\ \nabla_I\ y') \wedge z\ \sqsubseteq_I (z\ \nabla_I\ z')) \quad \wedge \\ (\forall \langle x,y,z\rangle \in K.\ \forall \langle x',y',z'\rangle \in K. & x' \sqsubseteq_I (x\ \nabla_I\ x') \wedge y' \sqsubseteq_I (y\ \nabla_I\ y') \wedge z' \sqsubseteq_I (z\ \nabla_I\ z')) \end{array}$$

It is taken for granted that Cousot's interval widening operator ∇_I is correct [32], therefore the above holds because:

$$\forall s \in I.\ \forall t \in I.\ s \sqsubseteq_I (s\ \nabla_I\ t) \quad \bigwedge \quad \forall s \in I.\ \forall t \in I.\ t \sqsubseteq_I (s\ \nabla_I\ t)$$

□

A.2 Proofs for Chapter 4.

Theorem 8 (The occurrence and polarity flags described on Page 72 are correctly constrained)**.** *The constraint $s_i = o_i - 2p_i$ correctly relates $s_i \in \{-1, 0, 1\}$ with the occurrence flag $o_i \in \{0, 1\}$, and the polarity flag $p_i \in \{0, 1\}$:*

$$\begin{array}{rll} s_i = o_i - 2p_i \implies (s_i = -1 \iff o_i = 1 \wedge p_i = 1) & \wedge \\ (s_i = 0 \iff o_i = 0) & \wedge \\ (s_i = 1 \iff o_i = 1 \wedge p_i = 0) & \end{array}$$

Proof. The following truth table enumerates all possible combinations of assignments to o_i, p_i and s_i within their domains. The table also indicates whether each assignment satisfies the premise $s_i = o_i - 2p_i$:

o_i	p_i	s_i	$o_i - 2p_i$	$s_i = o_i - 2p_i$
0	0	-1	0	✗
0	0	0	0	✓
0	0	1	0	✗
0	1	-1	-2	✗
0	1	0	-2	✗
0	1	1	-2	✗
1	0	-1	1	✗
1	0	0	1	✗
1	0	1	1	✓
1	1	-1	-1	✓
1	1	0	-1	✗
1	1	1	-1	✗

By discarding the cases which do not satisfy the premise, it is easy to see that the remaining three cases satisfy the conclusion: $(s_i = -1 \iff o_i = 1 \ \wedge \ p_i = 1) \wedge (s_i = 0 \iff o_i = 0) \ \wedge \ (s_i = 1 \iff o_i = 1 \ \wedge \ p_i = 0)$:

o_i	p_i	s_i	$o_i - 2p_i$	$s_i = o_i - 2p_i$
0	0	0	0	✓
1	0	1	1	✓
1	1	-1	-1	✓

□

A.3 Proofs for Chapter 6

Theorem 9. *Given a decision variable $\delta_i \in \{0, 1\}$ and two expressions $x, y \in \mathbb{Z}$ where x and y differ by at most $M - 1$:*

$$((x \leq y) \iff (\delta_i = 1)) \implies ((x \leq y + M \cdot (1 - \delta_i)) \wedge (x + M \cdot \delta_i \geq y + 1))$$

Proof. There are two cases under which the assumption is satisfied. First assume $(x \leq y) \ \wedge \ (\delta_i = 1)$ holds. In this case $(x + M \geq y + 1)$, which is true because x and y differ by at most $M - 1$. Now assume that $(x > y) \ \wedge \ (\delta_i = 0)$. In this case $(x \leq y + M) \wedge (x \geq y + 1)$. The left hand side of the conjunction is true because

x and y differ by at most $M-1$, then the right hand side of the conjunction is true because for integers $(x > y) \implies (x \geq y+1)$. □

Theorem 10. *Given a decision variable $\delta_i \in \{0,1\}$ and two expressions $x, y \in \mathbb{Z}$ where x and y differ by at most $M-1$:*

$$(x \leq y + M \cdot (1-\delta_i)) \wedge (x + M \cdot \delta_i \geq y+1)) \implies ((x \leq y) \iff (\delta_i = 1))$$

Proof. The proposition can be shown to be true by case analysis upon δ_i. First assume $\delta_i = 0$ holds. Then $((x \leq y + M) \wedge (x \geq y+1)) \implies ((x \leq y) \iff$ FALSE). Because x and y differ by at most $M-1$, it follows that $(x \leq y + M)$ is always satisfied. This leaves $(x \geq y+1) \implies ((x \leq y) \iff$ FALSE), which is satisfied because $(x \geq y+1) \implies (x \not\leq y)$.

Now assume $\delta_i = 1$ holds. Then $(x \leq y) \wedge (x + M \geq y+1)) \implies (x \leq y)$. Since x and y differ by at most $M-1$, it follows that $(x + M \geq y+1)$ is always satisfied. This leaves $(x \leq y) \implies (x \leq y)$.

□

Corollary 3 (Equivalence for the decision phase, described on Page 120, Chapter 6.)**.** *From Theorems 9 and 10, it follows that given a decision variable $\delta_i \in \{0,1\}$ and two expressions $x, y \in \mathbb{Z}$ where x and y differ by at most $M-1$:*

$$((x \leq y) \iff (\delta_i = 1)) \iff ((x \leq y + M \cdot (1-\delta_i)) \wedge (x + M \cdot \delta_i \geq y+1))$$

Theorem 11. *Given a decision variable $\delta_i \in \{0,1\}$ and two expressions $x, y \in \mathbb{Z}$ such that the difference between x and y is at most $M-1$:*

$$((\delta_i = 1) \implies (x \leq y)) \implies (x \leq y + M(1-\delta_i))$$

Proof. There are two cases under which the assumption is satisfied. First assume $\delta_i = 0$ holds. In this case $x \leq y + M$. This must be satisfied because x and y differ by at most $M-1$. Now assume that $(\delta_i = 1) \wedge (x \leq y)$. In this case the implication is trivially satisfied. □

Theorem 12. *Given a decision variable $\delta_i \in \{0,1\}$ and two expressions $x, y \in \mathbb{Z}$ such that the difference between x and y is at most $M-1$:*

$$(x \leq y + M(1-\delta_i)) \implies ((\delta_i = 1) \implies (x \leq y))$$

Proof. The proposition can be shown to be correct by case analysis upon δ_i. First suppose $\delta_i = 0$. In this case $x \leq y + M \implies$ TRUE. Now suppose $\delta_i = 1$. In this case $x \leq y \implies x \leq y$. □

Corollary 4 (Equivalence used in the impose phase, described on Page 121, Chapter 6)**.** *By Theorems 11 and 12, it follows that given a decision variable,* $\delta_i \in \{0, 1\}$ *and two expressions* $x, y \in \mathbb{Z}$*, such that the difference between* x *and* y *is at most* $M - 1$*:* $((\delta_i = 1) \implies (x \leq y)) \iff (x \leq y + M(1 - \delta_i))$.

Theorem 13 (Encoding of exit reachability, described on Page 123, Chapter 6)**.** *Given three decision variables* $\delta_i, \delta_m, \delta_n \in \{0, 1\}$*:*

$$((\delta_m = 1 \wedge \delta_n = 1) \iff (\delta_i = 1)) \equiv ((\delta_m + \delta_n - 2 \cdot \delta_i \leq 1) \wedge (\delta_m + \delta_n - 2 \cdot \delta_i \geq 0))$$

Proof. The following truth table enumerates all possible combinations of assignments to the decision variables δ_m, δ_n and δ_i. The table also indicates whether the right hand side of the equivalence is satisfied:

δ_m	δ_n	δ_i	$\delta_m + \delta_n - 2\delta_i$	$0 \leq \delta_m + \delta_n - 2\delta_i \leq 1$
0	0	0	0	✓
0	0	1	-2	✗
0	1	0	1	✓
0	1	1	-1	✗
1	0	0	1	✓
1	0	1	-1	✗
1	1	0	2	✗
1	1	1	0	✓

By discarding all assignments that do not satisfy $0 \leq \delta_m + \delta_n - 2\delta_i \leq 1$, this leaves four cases:

δ_m	δ_n	δ_i	$\delta_m + \delta_n - 2\delta_i$	$0 \leq \delta_m + \delta_n - 2\delta_i \leq 1$
0	0	0	0	✓
0	1	0	1	✓
1	0	0	1	✓
1	1	1	0	✓

The only remaining case where $\delta_i = 1$ is where $\delta_m = 1$ and $\delta_n = 1$ and vice versa. Therefore $((\delta_m = 1 \wedge \delta_n = 1) \iff (\delta_i = 1)) \equiv ((\delta_m + \delta_n - 2 \cdot \delta_i \leq 1) \wedge (\delta_m + \delta_n - 2 \cdot \delta_i \geq 0))$. □

Appendix B

Bibliography

[1] National Vulnerability Database. `http://nvd.nist.gov`.

[2] A. Adjé, S. Gaubert, and E. Goubault. Coupling Policy Iteration with Semi-definite Relaxation to Compute Accurate Numerical Invariants in Static Analysis. In *ESOP*, volume 6012 of *LNCS*, pages 23–42. Springer, 2010.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[4] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[5] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *SCP*, 72(1–2):3–21, 2008.

[6] G. Balakrishnan and T. Reps. DIVINE: Discovering Variables in Executables. In *VMCAI*, volume 4349 of *LNCS*, pages 1–28. Springer, 2007.

[7] G. Balakrishnan and T. W. Reps. WYSINWYX: What You See Is Not What You eXecute. *TOPLAS*, 32(6), 2010.

[8] S. Bardin and P. Herrmann. Structural Testing of Executables. In *ICST*, pages 22–31. IEEE, 2008.

[9] E. Barrett and A. King. Range and Set Abstraction using SAT. *ENTCS*, 267(1):17–27, 2010.

[10] E. Barrett and A. King. Range Analysis of Binaries with Minimal Effort. In *FMICS*, volume 7437 of *LNCS*, pages 93–107. Springer, 2012.

[11] A. Biere. Resolve and Expand. In *Theory and Applications of Satisfiability Testing*, volume 3542 of *LNCS*, pages 59–70. Springer, 2005.

[12] A. Biere, M. Heule, H. Van Maaren, and T. Walsh. *Handbook of Satisfiability*. IOS Press, 2009.

[13] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In *The Essence of Computation*, LNCS, pages 85–108. Springer, 2002.

[14] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *PLDI*, volume 38, pages 196–207. ACM, 2003.

[15] O. Bouissou, Y. Seladji, and A. Chapoutot. Abstract Fixpoint Computations with Numerical Acceleration Methods. *ENTCS*, 267(1):29–42, 2010.

[16] J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In *SAS*, volume 6337 of *LNCS*, pages 167–183. Springer, 2010.

[17] J. Brauer and A. King. Transfer Function Synthesis without Quantifier Elimination. *Logical Methods in Computer Science*, 8(3), 2012.

[18] J. Brauer, A. King, and S. Kowalewski. Range Analysis of Microcontroller Code Using Bit-Level Congruences. In *FMICS*, volume 6371 of *LNCS*, pages 82–98. Springer, 2010.

[19] J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In *CAV*, volume 6806 of *LNCS*, pages 191–207. Springer, 2011.

[20] J. Brauer, B. Schlich, T. Reinbacher, and S. Kowalewski. Stack Bounds Analysis for Microcontroller Assembly Code. In *Workshop on Embedded Systems Security*, pages 5:1–5:9. ACM, 2009.

[21] R. Bryant, D. Kroening, J. Ouaknine, S. A. Seshin, O. Strichman, and B. Brady. Deciding Bit-Vector Arithmetic with Abstraction. In *TACAS*, volume 4424 of *LNCS*, pages 358–372. Springer, 2007.

[22] R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):318, 1992.

[23] C. Cadar, V. Ganesh, P. M. Pawlowski, D .L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *CCS*, pages 322–335. ACM, 2006.

[24] K. Chatterjee, D. Ma, R. Majumdar, T. Zhao, T. Henzinger, and J. Palsberg. Stack Size Analysis for Interrupt-Driven Programs. In *Static Analysis*, volume 2694 of *LNCS*, pages 1075–1075. Springer, 2003.

[25] L. Chen, A. Miné, J. Wang, and P. Cousot. Linear Absolute Value Relation Analysis. In *ESOP*, volume 6602 of *LNCS*, pages 156–175. Springer, 2011.

[26] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983.

[27] C. Cifuentes and M. Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. *SCP*, 40(2-3):171–188, 2001.

[28] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. IEEE, 2004.

[29] M. Codish, V. Lagoon, and P. J Stuckey. Logic programming with satisfiability. *Theory and Practice of Logic Programming*, 8:121–128, 2008.

[30] A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs. In *CAV*, volume 6174, pages 462–475. Springer, 2005.

[31] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[32] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *PLILP*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.

[33] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée Scale Up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[34] P. Cousot, R. Cousot, and L. Mauborgne. A Scalable Segmented Decision Tree Abstract Domain. In *Time for Verification: Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*, pages 72–95. Springer, 2010.

[35] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL*, pages 84–96. ACM, 1978.

[36] Y. Crama and P. L. Hammer. *Boolean Functions: Theory, Algorithms, and Applications*. Cambridge University Press, 2011.

[37] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[38] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *PDPTA*, volume 2, pages 1013–1019. CSREA Press, 2000.

[39] R. Dechter. Bucket Elimination: a Unifying Framework for Processing Hard and Soft Constraints. *Constraints*, 2(1):51–55, 1997.

[40] R. Dechter and I. Rish. Directional resolution: The davis-putnam procedure, revisited. In *Knowledge Representation and Reasoning*, pages 134–145. Morgan Kaufmann, 1994.

[41] D. Doan. Commercial Off the Shelf (COTS) Security Issues and Approaches. Master's thesis, Naval Postgraduate School, Monterey, California, 2006. `www.dtic.mil/cgi-bin/GetTRDoc?AD=ADA456996`.

[42] T. Durden. Automated Vulnerability Auditing in Machine Code. *Phrack Magazine*, #64, 2007.

[43] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004.

[44] N. Een and N. Sörensson. MiniSat. `www.minisat.se`, 2010.

[45] M. Fähndrich and F. Logozzo. Static Contract Checking with Abstract Interpretation. In *Formal Verification of Object-Oriented Software*, volume 6528 of *LNCS*, pages 10–30. Springer, 2011.

[46] A. Flexeder, M. Petter, and H. Seidl. Side-Effect Analysis of Assembly Code. In *SAS*, volume 6887 of *LNCS*, pages 77–94. Springer, 2011.

[47] G. Balakrishnan and T. Reps. WYSINWYX: What You See is Not What You Execute. *TOPLAS*, 32(6):23:1–23:84, 2010.

[48] J. Garrido, D. Brosnan, J. A. de la Puente, A. Alonso, and J. Zamorano. Analysis of WCET in an experimental satellite software development. In *International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OpenAccess Series in Informatics (OASIcs)*, pages 81–90. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012.

[49] S. Gaubert, E. Goubault, A. Taly, and S. Zennou. Static Analysis by Policy Iteration on Relational Domains. In *ESOP*, volume 4421 of *LNCS*, pages 237–252. Springer, 2007.

[50] T. Gawlitza and H. Seidl. Precise Fixpoint Computation Through Strategy Iteration. In *ESOP*, volume 4421 of *LNCS*, pages 300–315. Springer, 2007.

[51] T. Gawlitza and H. Seidl. Precise Relational Invariants Through Strategy Iteration. In *Computer Science and Logic*, volume 4646 of *LNCS*, pages 23–40. Springer, 2007.

[52] T. M. Gawlitza, H. Seidl, A. Adjé, S. Gaubert, and É. Goubault. Abstract Interpretation Meets Convex Optimization. *Journal of Symbolic Computation*, 47(12):1416–1446, 2012.

[53] P. Glasscock. An 80x86 to C Reverse Compiler. Master's thesis, Computing Laboratory, Cambridge University, 1998.

[54] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, pages 213–223. ACM, 2005.

[55] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*. The Internet Society, 2008.

[56] E. Goldberg and P. Manolios. Quantifier elimination by dependency sequents. In *FMCAD*, pages 34–43. IEEE, 2012.

[57] D. Gopan and T. W. Reps. Lookahead Widening. In *CAV*, volume 4144 of *LNCS*, pages 452–466. Springer, 2006.

[58] E. Goubault, S. Le Roux, J. Leconte, L. Liberti, and F. Marinelli. Static Analysis by Abstract Interpretation: A Mathematical Programming Approach. *ENTCS*, 267(1):73–87, 2010.

[59] P. Granger. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.

[60] N. Halbwachs. *Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme.* PhD thesis, Thèse de 3ème cicle d'informatique, 1979.

[61] W. H. Harrison. Compiler Analysis for the Value Ranges of Varibles. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, 1977.

[62] C. Hathhorn, M. Becchi, W. L. Harrison, and A. M. Procter. Formal Semantics of Heterogeneous CUDA-C: A Modular Approach with Applications. In *SSV*, volume 102 of *EPTCS*, pages 115–124. arXiv.org, 2012.

[63] J. N. Hooker. A quantitative approach to logical inference. *Decision Support Systems*, 4(1):45–69, 1988.

[64] J. N. Hooker. Logical Inference and Polyhedral Projection. In *Computer Science Logic*, volume 626 of *LNCS*, pages 184–200. Springer, 1992.

[65] J. N. Hooker. Solving the Incremental Satisfiability Problem. *Journal of Logic Programming*, 15(1&2):177–186, 1993.

[66] P. Jackson. Computing Prime Implicates. In *Proceedings of the 1992 ACM Annual Conference on Communications*, CSC, pages 65–72. ACM, 1992.

[67] R. J. Jiang. Quantifier Elimination via Functional Composition. In *CAV*, volume 5643, pages 383–397. Springer, 2009.

[68] D. Kapur. Automatically Generating Loop Invariants using Quantifier Elimination. In *International Conference on Applications of Computer Algebra*, volume 05431 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2004.

[69] M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[70] J. Kinder and H. Veith. Precise Static Analysis of Untrusted Driver Binaries. In *FMCAD*, pages 43–50. IEEE, 2010.

[71] J. Kinder, F. Zuleger, and H. Veith. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *VMCAI*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[72] A. King and H. Søndergaard. Automatic Abstraction for Congruences. In *VMCAI*, volume 5944 of *LNCS*, pages 197–213. Springer, 2010.

[73] V. Kotlyar and M. Moudgill. Detecting overflow detection. In *Conference on Hardware/software codesign and system synthesis*, CODES+ISSS'04, pages 36–41. ACM, 2004.

[74] D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.

[75] S. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT Techniques for Fast Predicate Abstraction. In *CAV*, volume 4144 of *LNCS*, pages 424–437. Springer, 2006.

[76] L. Lakhdar-Chaouch, B. Jeannet, and A. Girault. Widening with Thresholds for Programs with Complex Control Graphs. In *ATVA*, volume 6996 of *LNCS*, pages 492–502. Springer, 2011.

[77] J. Lang, P. Liberatore, and P. Marquis. Propositional Independence: Formula-Variable Independence and Forgetting. *CoRR*, abs/1106.4578, 2011.

[78] K. R. M. Leino and F. Logozzo. Using Widenings to Infer Loop Invariants Inside an SMT Solver, Or: A Theorem Prover as Abstract Domain. In *WING*, pages 70–84. Microsoft, 2007.

[79] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Foundations of Software Engineering*, FSE, pages 187–196. ACM, 2010.

[80] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *CCS*, pages 290–299. ACM, 2003.

[81] B. Lisper, A. Ermedahl, D. Schreiner, J. Knoop, and p. Gliwa. Practical Experiences of Applying Source-Level WCET Flow Analysis on Industrial Code. In *Leveraging Applications of Formal Methods, Verification, and Validation*, volume 6416 of *LNCS*, pages 449–463. Springer, 2010.

[82] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *ESOP*, volume 3444 of *LNCS*, pages 5–20. Springer, 2005.

[83] K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[84] K. L. McMillan. Applications of Craig Interpolants in Model Checking. In *TACAS*, volume 3440 of *LNCS*, pages 1–12. Springer, 2005.

[85] A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *PADO*, volume 2053 of *LNCS*, pages 155–172. Springer, 2001.

[86] A. Miné. The Octagon Abstract Domain. *Higher-Order Symbolic Computation*, 19(1):31–100, 2006.

[87] M. Müller-Olm and H. Seidl. Analysis of Modular Arithmetic. *TOPLAS*, 29(5), 2007.

[88] A. Mycroft. Type-Based Decompilation. In *ESOP*, volume 1576, pages 208–223. Springer, 1999.

[89] G. J. Myers. *The Art of Software Testing*. Wiley InterScience, 1979.

[90] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code. In R. Jhala and A. Igarashi, editors, *APLAS*, volume 7705 of *LNCS*, pages 115–130. Springer, 2012.

[91] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM*, 53(6):937–977, 2006.

[92] D. A. Plaisted and S. Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[93] W. V. Quine. A Way to Simplify Truth Functions. *The American Mathematical Monthly*, 62(9):627–631, 1955.

[94] A. Ramesh, G. Becker, and N. Murray. CNF and DNF Considered Harmful for Computing Prime Implicants/Implicates. *Journal of Automated Reasoning*, 18:337–356, 1997.

[95] J. Regehr and U. Duongsaa. Deriving Abstract Transfer Functions for Analyzing Embedded Software. In *ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, LCTES'06, pages 34–43. ACM, 2006.

[96] T. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 3–25. Springer, 2004.

[97] T. W. Reps, G. Balakrishnan, and J. Lim. Intermediate-Representation Recovery from Low-Level Code. In *PEPM*, pages 100–111. ACM, 2006.

[98] E. Rodriguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. In *SAS*, volume 3148 of *LNCS*, pages 280–295. Springer, 2004.

[99] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *TOPLAS*, 27:185–235, 2005.

[100] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint Solving for Interpolation. *Journal of Symbolic Computation*, 45:1212–1233, 2010.

[101] P. Samuelson and S. Scotchmer. The Law and Economics of Reverse Engineering. *Yale Law Journal*, 111:1575, 2001.

[102] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable Analysis of Linear Systems Using Mathematical Programming. In *VMCAI*, volume 3385 of *LNCS*, pages 25–41. Springer, 2005.

[103] R. A. Sayle. A Superoptimizer Analysis of Multiway Branch Code Generation. In *GCC Developers Summit*, pages 103–116. Linux Symposium Inc., 2008.

[104] B. Schlich. Model Checking of Software for Microcontrollers. *ACM Transactions in Embedded Computing Systems*, 9:1–27, 2010.

[105] B. Schlich, J. Löll, and S. Kowalewski. Application of Static Analyses for State Space Reduction to Microcontroller Assembly Code. In *FMICS*, volume 4916, pages 21–37. Springer, 2007.

[106] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1998.

[107] R. Sen and Y. N. Srikant. Executable Analysis using Abstract Interpretation with Circular Linear Progressions. In *International Conference on Formal Methods and Models for Codesign*, MEMCODE, pages 39–48. IEEE, 2007.

[108] J. Seward and N. Nethercote. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *USENIX Annual Technical Conference*, pages 17–30. USENIX, 2005.

[109] A. Simon and A. King. Widening Polyhedra with Landmarks. In *APLAS*, volume 4279 of *LNCS*, pages 166–182. Springer, 2006.

[110] A. Simon and A. King. Taming the Wrapping of Integer Arithmetic. In *SAS*, volume 4634 of *LNCS*, pages 121–136. Springer, 2007.

[111] A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In *LOPSTR*, volume 2664 of *LNCS*, pages 71–89. Springer, 2002.

[112] A. Singh. *Identifying Malicious Code Through Reverse Engineering*. Advances in Information Security. Springer, 2009.

[113] Z. Su and D. Wagner. A Class of Polynomially Solvable Range Constraints for Interval Analysis without Widenings. *TCS*, 345(1):122–138, 2005.

[114] S. Thompson and A. Mycroft. Bit-level Partial Evaluation of Synchronous Circuits. In *PEPM*, pages 29–37. ACM, 2006.

[115] G. S. Tseitin. On the Complexity of Derivation in Propositional Calculus. *Studies in Constructive Mathematics and Mathematical Logic*, 2(115-125):10–13, 1968.

[116] G. Weissenbacher. *Program Analysis with Interpolants*. PhD thesis, Magdalen College, 2010.

[117] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A New Incremental Satisfiability Engine. In *Design Automation Conference*, pages 542–545, 2001.

[118] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Transactions in Embedded Computing Systems*, 7(3):36:1–36:53, 2008.

[119] R. Wille, G. Fey, and R. Drechsler. Building Free Binary Decision Diagrams Using SAT Solvers. *Facta Universitatis-Series: Electronics and Energetics*, 20(3):381–394, 2007.

[120] Y. Xie and A. Aiken. Saturn: A Scalable Framework for Error Detection using Boolean Satisfiability. *TOPLAS*, 29(3), 2007.

[121] A. Zaks, Z. Yang, I. Shlyakhter, F. Ivancic, S. Cadambi, M. K. Ganai, A. Gupta, and P. Ashar. Bitwidth Reduction via Symbolic Interval Analysis for Software Model Checking. *IEEE TACAD*, 27(8):1513–1517, 2008.

[122] Q. Zhong and N. Edward. Security Control COTS Components. *IEEE Computer Society*, 31:67–73, 1998.