Abstract Interpretation Over Bitvectors

by

Tushar Sharma

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 08/11/2017

The dissertation is approved by the following members of the Final Oral Committee:

Thomas Reps, Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Benjamin Liblit, Associate Professor, Computer Sciences
Aws Albarghouthi, Assistant Professor, Computer Sciences
Vadim Shapiro, Professor, Mechanical Engineering and Computer Sciences

Dedicated to mummy, papa, didi, and bhaiya.

ACKNOWLEDGMENTS

My graduate student journey was a memorable one, and several people made it a meaningful experience. Without the help and support of mentors, co-workers, friends, and family, this dissertation would not have been possible.

First and foremost, I am indebted to my advisor, Prof. Thomas Reps, who has motivated and advised me throughout my journey in the doctoral program. Without his immense knowledge and expertise in program analysis and abstract interpretation, none of my research projects would have been successful. His passion and enthusiasm for research is unparalleled. Tom has given me so much freedom to pursue my research interests, and helped me in converting my half-baked ideas into useful publications. Over the years, he cultivated me into an independent researcher by teaching me the fine art of asking interesting research questions, reading interesting related work, and building the research ideas into concrete projects worthy of publication. Most importantly, he has treated me more like a friend, than as a subordinate, and for that, I will be forever indebted. I hope you enjoyed working with me, as much I did working with you.

I thank my dissertation committee for taking the time to read my dissertation. I thank Prof. Somesh Jha, Prof. Ben Liblit, and Prof. Shan Lu for reading my prelim document, attending my prelim talk, and providing useful and constructive suggestions that guided me through my years as a dissertator. I thank my undergrad mentor, Prof. Sundar Balasubramaniam, who got me excited about static analysis and type systems, and motivated me to pursue graduate school.

Over the last few years, I had the fortune of working and collaborating with the best colleagues. I want to thank Junghee Lim, Matt Elder, Evan Driscoll, and Aditya Thakur, for mentoring me at the earlier years of my graduate school, and answering a whole variety of questions from the

complex subtleties of static analysis to basic coding questions. I want to thank Vijay Chidambaram, Venkatesh Srinivasan, and Tycho Andersen for collaborating with me on several projects and making my graduate school experience much more fun. I have had so many wonderful conversations with Venkatesh Srinivasan, Peter Ohmann, and Jason Breck. It has been a pleasure to have you guys around, on whom I can vent my pessimism and nihilism. I am appreciative for David Bingham, Stephen Lee, Venkatesh Srinivasan, Samuel Drews, Aditya Thakur, Evan Driscoll, Drew Davidson and Jason Breck, who patiently listened to my boring presentations and provided me constructive feedback.

The employees at GrammaTech have helped me tremendously over the years to answer my queries, fix bugs, provide useful technical support in the tools and infrastructure support for machine-code analysis. In particular, Junghee Lim, Evan Driscoll, Suan Yong, Brian Alliet, Tom Johnson, Alexey Loginov and David Melski have promptly answered my questions and provided helpful comments to improve my tools.

I thank my friends for making my life in Madison an exciting and happy one. Suyash Singh was a friend before we started the graduate journey and over our common graduate years at Madison, our friendship has grown stronger (and quirky). Thank you for the laughs and the support! Hemanth Pikkili, Gurdaman Khaira, Kushal Sinha, and Suyash Singh have been amazing roommates over the years, and I have never felt bored in Madison because of you guys. Thank you Suyash, Rishabh, Fulya, and Kushal for being the 'chemical brothers/sisters'; Raja, Rohit, and Sankar for being the 'doods'; Saili, Etienne, Cherry, Bang, Dipto, Saurabh, Keive, Anand, and Amy for the 'spicy balls'; Selah, Obi, Manali, Jason, and Janina for being the voice of reason; and Thanu, Venkatesh, Goldy and Dipto for listening to my graduate school rants.

My mom and dad have unconditionally loved and supported me throughout my life. They have always supported me in my graduate

school journey, even though we have been thousands of miles away. My sister, Sugandha, always provides me with moral support. Her innocence, charisma, and kindness is unparalled. My brother has been my role model, my mentor, my ally, and my competition. He has always believed in me, and challenged me to dream bigger, higher, and better. Words would never be enough to describe the gratitude I have for my family. I want to thank Mrs. and Mr. Kulkarni and Shefali for being my family in US. Thanks to them, thankgiving and christmas holidays have been full of life.

Finally, I want to thanks the love of my life, Saili Kulkarni, for being there for me, for understanding me, and for believing in me, even when I didn't believe in myself.

This dissertation is supported, in part, by a gift from Rajiv and Ritu Batra; by DARPA under cooperative agreement HR0011-12-2-0012; by NSF under grant CCF-{0810053, 0904371}; by ONR under grants N00014-{09-1-0510, 11-C-0447}; by AFRL under contract FA9550-09-1-0279 and FA8650-10-C-7088; by ARL under grant W911NF-09-1-0413, by AFRL under DARPA MUSE award FA8750-14-2- 0270, DARPA STAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author, and do not necessarily reflect the views of the sponsoring agencies. The author's advisor Thomas Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this dissertation.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Most critical applications, such as airplane and rocket controllers, need correctness guarantees. Usually these correctness guarantees can be described as safety properties in the form of assertions. Verifying an assertion amounts to showing that the assertion holds true for all possible runs of an application. Abstract interpretation is a method to automatically verify a program by soundly abstracting the concrete executions of the program to elements in an abstract domain, and checking the correctness guarantees using the abstraction. However, traditional abstract domains treat the machine integers as mathematical integers. As a result, the conclusions drawn from such an abstract interpretation are, in general, unsound. In other words, the assertions shown to be true by traditional abstract interpretation approaches might actually be false because the underlying point space does not faithfully model bit-vector arithmetic.

This dissertation advances the field of abstract interpretation by providing sound abstraction techniques and abstract-domain frameworks that faithfully model bit-vector semantics. We focus on numerical abstract domains for bitvectors, which can provide equality and inequality invariants.

The first part of the dissertation focuses on abstract domains capable of expressing bit-vector-sound *equality* invariants. The performance and precision of two equality domains is compared, and sound inter-conversion methods are provided. Furthermore, we generalize one of the equality domains to develop a new abstract-domain framework that is capable of expressing a certain class of disjunctions over bit-vector-sound equality constraints. This framework can be instantiated with any relational or non-relational base abstract domain over bitvectors.

The second part of the dissertation focuses on abstract domains capable of expressing bit-vector-sound *inequality* invariants. We develop an

abstract domain that is capable of expressing a certain class of bit-vector-sound inequalities over memory variables and registers. Furthermore, we develop an abstract-domain framework that takes an abstract domain that is sound with respect to mathematical integers, and creates an abstract domain whose operations and abstract transformers are sound with respect to machine integers.

1 INTRODUCTION

Nowadays, humans rely on computer systems in almost every aspect of life. Reliance on critical systems such as automated banking systems, flight-control systems, automated cruise control and braking systems, and traffic-light control systems has increased as a result. The presence of bugs in these critical systems have lead to embarrassing issues like the Excel 2007 bug [98] and overflow in youtube view limit [6], unnecessary human intervention [31], losses in millions of dollars [38, 36] or worse, loss of human life [28, 32].

Most critical applications, such as airplane and rocket controllers, need correctness guarantees. Usually these correctness guarantees can be described as safety properties in the form of assertions. Verifying an assertion amounts to showing that the assertion's condition holds true for all possible runs of an application. Proving an assertion is, in general, an undecidable problem. Nevertheless, there exist static-analysis techniques that are able to verify automatically some kinds of program assertions. One such technique is abstract interpretation [20], which soundly abstracts the concrete executions of the program to elements in an abstract domain, and checks the correctness guarantees using the abstraction.

Abstract interpretation has seen a lot of progress in last 40 years since it was introduced by Cousot and Cousot in 1977. Many numeric abstract domains have been introduced to verify the correctness of assertions by the programmer, as well as divide-by-zero, array-bounds checks, buffer-overflow behavior, etc. Arguably, abstract interpretation has made the most progress in numeric program analysis, that is, discovering numerical properties about the program. The material in this dissertation is centered on discovering numerical properties on the machine integers through abstract interpretation.

The building block for the abstraction of the program is an *abstract*

domain. An abstract domain defines the level and detail of the abstraction. An abstract domain can be seen as a logic fragment [86] that is capable of expressing only a certain class of constraints. For instance, an interval value $v \in [0, 5]$, where v is a variable, expresses the constraint: $0 \leqslant v \leqslant 5$. The richer the abstract domain, the more intricate the constraints it can express. Richer abstract domains can prove more assertions possibly at the cost of worse performance.

Mathematically, abstract domains are (usually) partial orders, where the order is given by set containment: the smaller the set denoted by an abstract-domain element, the more precise it is. For instance, the interval $i_1 : v \in [0, 5]$ is more precise than the interval $i_2 : v \in [-10, 10]$.

Often $\mathcal{A}$ is really a family of abstract domains, in which case $\mathcal{A}[U]$ denotes the specific instance of $\mathcal{A}$ that is defined over vocabulary U. The two important steps in abstract interpretation (AI) are:

1. Abstraction: The abstraction of the program is constructed using the abstract domain and abstract semantics.
2. Fixpoint analysis: Iteration until a fixpoint is reached is performed on the abstraction of the program to identify invariants of the program.

In the typical setup of AI, the set of states that can arise at each point in the program is safely represented by the abstract-domain element found for that program point in the fixed point. This setup can be used to prove assertions. Let us denote the set of variables in a program by V. In this case, the abstract-domain elements abstract the set of values at a program point, thus they are abstracting the values for V, and therefore $U = V$. For the purpose of this thesis, abstract-domain elements are abstract transformers, that is, they are abstractions of concrete transformers describing the control-flow graph edges in the program. When abstract-domain elements are abstract transformers, the results provide function summaries or loop summaries [19, 92]. In principle, summaries can be computed offline for large libraries of code so that client static analyses can use them

to obtain verification results more efficiently. Abstract transformers are abstracting the set of values at two program points: one is represented by the pre-transformation variables V, and the other is represented by the post-transformation variables V', and therefore $U = V \cup V'$. A static analyzer needs a way to construct abstract transformers for the concrete operations in the programs. Semantic reinterpretation (see §2.3.1) and symbolic abstraction (see §2.3.2) provide automatic ways to construct abstract transformers.

Past literature has introduced many non-relational and relational abstract domains. Examples of non-relational domains include constant propagation [51] ($v_i = \alpha_i$), signs [19] ($\pm v_i \leqslant 0$), intervals [19] ($\alpha_i \leqslant v_i \leqslant \beta_i$), congruences [39] ($v_i \equiv \alpha_i \bmod \beta_i$), and interval congruences [65] ($v_i \in [\alpha_i, \beta_i] \bmod \gamma_i$). Here, v_i refers to a variable in the program, and the symbols α_i, β_i and γ_i refer to constants.

The classical example of a relational domain is the domain of linear equalities introduced by Karr [49] ($\Sigma_i \alpha_{ij} v_i = \beta_j$). Granger [40] introduced the linear-congruence domain ($\Sigma_i \alpha_{ij} v_i = \beta_j \bmod \gamma_j$). One of the most widely used relational abstract domains is the polyhedral domain [23], which is capable of expressing relational affine inequalities ($\Sigma_i \alpha_{ij} v_i \leqslant \beta_j$). While the polyhedral domain is useful, it is also very slow, and not scalable to large systems. With that in mind, previous research [68, 69, 97, 58, 89, 73] has also provided weaker variants of the polyhedra domain that are capable of expressing some; but not all; affine inequalities. For instance, the octagon abstract domain [68] can express only relational inequalities involving at most two variables where the coefficients on the variables are only allowed to be plus or minus one ($\pm v_i \pm v_j \leqslant \alpha_{ij}$).

1.1 Motivation.

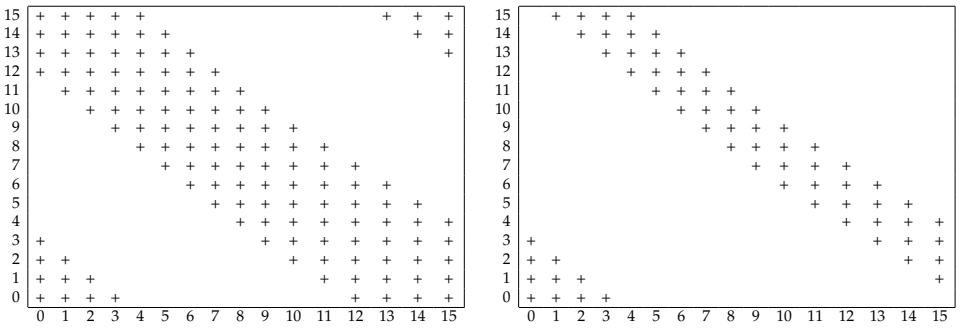
These abstract domains suffer from one huge limitation: they treat the program variables as mathematical integers or rationals. However, the native machine-integer data-types used in programs (e.g., `int`, `unsigned int`, `long`, etc.) perform bit-vector arithmetic, and arithmetic operations wrap around on overflow. Thus, the underlying point space used in the aforementioned abstract domains does not faithfully model bit-vector arithmetic, and consequently the conclusions drawn from an analysis based on these domains are, in general, unsound, unless special steps are taken [96][15].

Example 1.1. *The following C-program fragment incorrectly computes the average of two* `int`*-valued variables [9]:*

```
int low, high, mid;
assume(0 <= low <= high);
mid = (low + high)/2;
assert(0<=low<=mid<=high);
```

A static analysis based on polyhedra or octagons would draw the wrong conclusion that the assertion always holds. In particular, assuming 32-bit `int`*s, when the sum of* `low` *and* `high` *is greater than* 2^{31}-1*, the sum overflows, and the resulting value of* `mid` *is smaller than* `low`*. Consequently, there exist runs in which the assertion fails. These runs are overlooked when the polyhedral domain is used for static analysis because the domain fails to take into account the bit-vector semantics of program variables.* □

The problem that we wish to solve is not one of merely *detecting* overflow—e.g., to restrain an analyzer from having to explore what happens after an overflow occurs. On the contrary, our goal is to be able to track soundly the effects of arithmetic operations, including wrap-around effects of operations that overflow. This ability is useful, for instance, when analyzing code generated by production code generators, such as dSPACE TargetLink [25], which use the "compute-through-overflow" tech-



(a) $x + y + 4 \leqslant 7$ (b) $x + y \leqslant 3$

Figure 1.1: Each + represents a solution of the indicated inequality in 4-bit unsigned bit-vector arithmetic.

nique [35]. Furthermore, clever idioms for bit-twiddling operations, such as the ones explained in [102], sometimes rely on overflow [24].

1.1.1 Challenges in dealing with bit-vectors.

Some of the ideas used in designing an inequality domain for reals do not carry over to ones designed for bit-vectors. First, in bit-vector arithmetic, additive constants cannot be canceled on both sides of an inequality, as illustrated in the following example.

Example 1.2. *Let* x *and* y *be 4-bit unsigned integers. Fig. 1.1 depict the solutions in bit-vector arithmetic of the inequalities* $x + y + 4 \leqslant 7$ *and* $x + y \leqslant 3$*. Although* $x + y + 4 \leqslant 7$ *and* $x + y \leqslant 3$ *are syntactically quite similar, their solution spaces are quite different. In particular, because of wrap-around of values computed on the left-hand sides using bit-vector arithmetic, one cannot just subtract* 4 *from both sides to convert the inequality* $x+y+4 \leqslant 7$ *into* $x+y \leqslant 3$*.* □

Second, in bit-vector arithmetic, positive constant factors cannot be canceled on both sides of an inequality; for example, if x and y are 4-bit bit-vectors, then $(4, 4)$ is in the solution set of $2x + 2y \leqslant 4$, but not of $x + y \leqslant 2$.

These properties do not carry over because bitvectors are not a field, but rather a ring $\mathbb{Z}_m$ of integers modulo $m = 2^w$ for $w > 1$. Moreover, the ring has zero divisors, and consequently even numbers in bit-vector arithmetic do not have a multiplicative inverse. Because of these properties, results from linear algebra over fields do not apply to the abstract domains over $\mathbb{Z}_m$.

While some simple domains do exist that are capable of representing certain kinds of inequalities over bit-vectors (e.g., intervals with a congruence constraint, sometimes called "strided-intervals" [83, 91, 5, 72]), such domains are non-relational; that is, they are not capable of expressing relations among several variables. On the other hand, there exist relational bit-precise domains such as the domains introduced by Müller-Olm/Seidl (denoted by MOS) [78] and King/Søndergaard (denoted by KS) [53], but they cannot express inequalities. Moreover, it is not clear how MOS and KS relate to each other.

1.2 Thesis Contributions

In this thesis, we provide sound abstract domains, and abstract-transformer construction techniques, that allow one to prove assertions and provide function summaries for programs with machine integers. The communication with existing analyzers is done strictly through a generic abstract-domain interface. Thus, our work should be compatible with most existing analysis engines. As a result, our abstraction techniques can be applied to source-code analysis as well as machine-code analysis.

1.2.1 Quick Summary

Broadly, our contributions to the numeric analysis of programs with machine integers fall into two categories: i) Bit-Vector Precise Equality Ab-

stract Domains, ii) Bit-Vector Precise Inequality Abstract Domains. The thesis makes the following contributions:

- **Bit-Vector Precise Equality Abstract Domains**
  - **Abstract Domains of Affine Relations [29, 30]:** An affine relation is a *linear-equality* constraint over a given set of variables that hold machine integers. In this work, we compare the MOS and KS abstract domains, along with several variants. These domains are capable of expressing affine relations over bitvectors. We show that MOS and KS are, in general, *incomparable* and give sound interconversion methods for KS and MOS. We introduce a third domain for representing affine relations, called AG, which stands for *affine generators*. Furthermore, we present an experimental study comparing the precision and performance of analyses with the KS and MOS domains. (See §1.2.2 for more details.)
  - **Abstraction Framework for Affine Transformers [93]:** In this work, we define the Affine-Transformers Abstraction Framework, which represents a new family of numerical abstract domains. This framework is parameterized by a base numerical abstract domain, and allows one to represent a set of affine transformers (or, alternatively, certain disjunctions of transition formulas). Specifically, this framework is a generalization of the MOS domain. The choice of the base abstract domain allows the client to have some control over the performance/precision trade-off. (See §1.2.3 for more details.)
- **Bit-Vector Precise Inequality Abstract Domains**
  - **An Abstract Domain for Bit-vector Inequalities [95]:** This work describes the design and implementation of a new abstract domain, called the *Bit-Vector Inequality* domain, which is capable of capturing certain inequalities over bit-vector-valued

variables (which represent a program's registers and/or its memory variables). This domain tracks properties of the values of selected registers and portions of memory via *views*, and provides automatic heuristics to gather equality and inequality views from the program. Furthermore, experiments are provided to show the usefulness of the Bit-Vector Inequality domain. (See §1.2.4 for more details.)

– **Sound Bit-Precise Numerical Domains Framework for Inequalities [94]:** This work introduces a class of abstract domains, parameterized on a base domain, that is sound with respect to bitvectors whenever the base domain is sound with respect to mathematical integers. The base domain can be any numerical abstract domain. We also describe how to create abstract transformers for this framework that incorporate lazy wrap-around to achieve more precision, without sacrificing soundness with respect to bitvectors. We use a finite number of disjunctions of base-domain elements to help retain precision. Furthermore, we present experiments to empirically demonstrate the usefulness of the framework. (See §1.2.5 for more details.)

1.2.2 Abstract Domains of Affine Relations

As mentioned before, the abstract domains MOS and KS can express relational equalities over bitvectors. An element in the King/Søndergaard domain (KS) is an affine-closed set of linear-equality constraints over bitvectors ($\Sigma_i \alpha_{ij} v_i + \Sigma_i \alpha'_{ij} v'_i = \beta_j$, where $\alpha_{ij}, \alpha'_{ij}, \beta_j \in \mathbb{Z}_m$). An element in the Müller-Olm/Seidl domain (MOS) is an affine-closed set of affine transformers. An affine transformer is a relation on states, defined by $\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}$, where $\overrightarrow{v}'$ and $\overrightarrow{v}$ are row vectors that represent the post-transformation state and the pre-transformation state, respectively.

C is the linear component of the transformation and $\overrightarrow{d}$ is a constant vector. For example, $[\ x'\quad y'\ ] = [\ x\quad y\ ]\begin{bmatrix} 1 & 0 \\ 2 & 0 \end{bmatrix} + [\ 10\quad 0\ ]$ denotes the affine transformation $(x' = x + 2y + 10 \wedge y' = 0)$ over variables $\{x, y\}$. We denote such an affine transformation by $C : \overrightarrow{d}$. An element of these domains represents a set of points that satisfies an affine relation over variables that hold machine integers.

Chapter 3 considers MOS and KS abstract domains, along with several variants, and studies how they relate to each other. We show that MOS and KS are, in general, *incomparable*. In particular, we show that KS can express transformers with affine guards, which MOS cannot express. On the other hand, MOS can express some non-affine-closed sets, which are not expressible by KS. We give sound interconversion methods for KS and MOS. That is, we give an algorithm to convert a KS element to an over-approximating MOS element, as well as an algorithm to convert an MOS element to an over-approximating KS element.

We introduce a third domain for representing affine relations, called AG, which stands for *affine generators*. Whereas an element in the KS domain consists of a set of *constraints* on the values of variables, AG represents a collection of allowed values of variables via a set of *generators*. We show that AG is the generator counterpart of KS: a KS element can be converted to an AG element, and vice versa, with no loss of precision.

Furthermore, Chapter 3 presents an experimental study with the Intel IA32 (x86) instruction set in which the symbolic abstraction (§2.3.2) method and two reinterpretation methods (§2.3.1)—KS-reinterpretation and MOS-reinterpretation —are compared in terms of their performance and precision. The precision comparison is done by comparing the affine invariants obtained at branch points, as well as the affine procedure summaries obtained for procedures. For KS-reinterpretation and MOS-reinterpretation, we also compare the abstract transformers generated for individual x86 instructions.

1.2.3 A New Abstraction Framework for Affine Transformers

Abstractions of affine transformers can be used to obtain affine-relation invariants at each program point in the program [75]. An affine relation is a linear-equality constraint between numeric-valued variables of the form $\sum_{i=1}^{n} a_i v_i + b = 0$. For a given set of variables $\{v_i\}$, affine-relation analysis (ARA) identifies affine relations that are invariants of a program. The results of ARA can be used to determine a more precise abstract value for a variable via *semantic reduction* [21], or detect the relationship between program variables and loop-counter variables. While the MOS domain is useful for finding affine-relation invariants in a program, the join operation used at confluence points can lose precision in many cases, leading to imprecise function summaries. Furthermore, the analysis does not scale well as the number of variables in the vocabulary increases. In other words, it has one baked-in performance-versus-precision aspect.

Chapter 4 provide analysis techniques to abstract the behavior of the program as a set of affine transformations over bit-vectors. Remember that an affine transformation has the following form: $\vec{v}' = \vec{v} \cdot C + \vec{d}$). In this work, we generalize the ideas used in the MOS domain—in particular, to have an abstraction of *sets of affine transformers*—but to provide a way for a client of the abstract domain to have some control over the performance/precision trade-off. Toward this end, we define a new family of numerical abstract domains, denoted by ATA[$\mathcal{B}$]. (ATA stands for Affine-Transformers Abstraction.) Following our observation, ATA[$\mathcal{B}$] is parameterized by a base numerical abstract domain $\mathcal{B}$, and allows one to represent a set of affine transformers (or, alternatively, certain disjunctions of transition formulas).

The overall contribution of our work is the framework ATA[$\mathcal{B}$], for which we present

- methods to perform basic abstract-domain operations, such as equality and join.
- a method to perform abstract composition, which is needed to perform abstract interpretation.
- a faster method to perform abstract composition when the base domain is non-relational.

1.2.4 An Abstract Domain for Bit-vector Inequalities

Two of the biggest challenges in machine-code analysis are: (1) identifying inequality invariants while handling overflow in arithmetic operations over bit-vector data-types, and (2) identifying invariants that capture properties of values in memory.

When analyzing machine code, memory is usually modeled as a flat array. When analyzing Intel x86 machine code, for instance, memory is modeled as a map from 32-bit bit-vectors to 8-bit bit-vectors. Consequently, an analysis has to deal with complications arising from the little-endian addressing mode and aliasing, as we now illustrate.

Example 1.3. *Consider the following machine-code snippet:*

```
mov   eax, [ebp]
mov   [ebp+2], ebx
```

The first instruction loads the four bytes pointed to by register `ebp` *into the 32-bit register* `eax`*. Suppose that the value in register* `ebp` *is* A*. After the first instruction, the bytes of* `eax` *contain, in least-significant to most-significant order, the value at memory location* A*, the value at location* $A + 1$*, the value at location* $A + 2$*, and the value at location* $A + 3$*. The second instruction stores the value in register* `ebx` *into the memory pointed to by* `ebp+2`*. Due to this instruction, the values at memory locations* $A+2$ *through* $A+5$ *are overwritten, after which the value in register* `eax` *no longer equals (the little-endian interpretation of) the bytes in memory pointed to by* `ebp`*.* □

These challenges lead to the following problem statement:

> *Design an abstract domain of relational bit-vector affine-inequalities over memory-values and registers.*

Chapter 5 expands the set of techniques available for abstract interpretation and model checking of machine code. We describe the design and implementation of a new abstract domain, called the *Bit-Vector Inequality* ($\mathcal{BVI}$) domain, that is capable of capturing certain inequalities over bit-vector-valued variables. We also consider some variants of the $\mathcal{BVI}$ domain.

The key insight used to design $\mathcal{BVI}$ domain (and its variants) involves a new domain combinator (denoted by $\mathcal{V}$), called the *view-product combinator*. $\mathcal{V}$ constructs a reduced product of two domains [21], using shared *view-variables* to communicate information between the domains.

The *Bit-Vector Memory-Equality Domain* $\mathcal{BVME}$, a domain of bit-vector affine-equalities over variables and memory-values, is created by applying the view-product combinator $\mathcal{V}$ to the bit-vector memory domain (§5.3) and the bit-vector equality domain. The *Bit-Vector Inequality Domain* $\mathcal{BVI}$, a domain of bit-vector affine-inequalities over variables, is created by applying $\mathcal{V}$ to the bit-vector equality domain and a bit-vector interval domain. The *Bit-Vector Memory-Inequality Domain* $\mathcal{BVMI}$, a domain of relational bit-vector affine-inequalities over variables and memory, is then created by applying $\mathcal{V}$ to the $\mathcal{BVME}$ domain and the bit-vector interval domain. The latter construction illustrates that $\mathcal{V}$ composes: the $\mathcal{BVMI}$ domain is created via two applications of $\mathcal{V}$.

This work makes the following contributions:

- The bit-vector memory domain, a non-relational memory domain capable of expressing invariants involving memory accesses.
- The view-product combinator $\mathcal{V}$, a general procedure to construct more expressive domains.
- Three domains for machine-code analysis constructed using $\mathcal{V}$:

  - The bit-vector memory-equality domain $\mathcal{BVME}$, which captures equality relations among bit-vector variables and memory.
  - The bit-vector inequality domain $\mathcal{BVI}$, which captures inequality relations among bit-vector variables.
  - The bit-vector memory-inequality domain $\mathcal{BVMI}$, which captures inequality relations among bit-vector variables and memory.
- A procedure for synthesizing best abstract operations for reduced products of domains that meet certain requirements.
- Experimental results that illustrate the effectiveness of the $\mathcal{BVI}$ domain applied to machine-code analysis. On average (geometric mean), our $\mathcal{BVI}$-based analysis is about 3.5 times slower than an affine-equality-based analysis, while finding improved (more-precise) invariants at 29% of the branch points.

1.2.5 Sound Bit-Precise Numerical Domains Framework for Inequalities

Chapter 6 describes a second approach to designing and implementing a bit-precise relational domain capable of expressing inequality invariants. This work presents the design and implementation of a new framework for abstract domains, called the *Bit-Vector-Sound Finite-Disjunctive* (*BVSFD*) domains, which are capable of capturing useful program invariants such as inequalities over bit-vector-valued variables.

We introduce a class of abstract domains, called $BVS(\mathcal{A})$, that is sound with respect to bitvectors whenever $\mathcal{A}$ is sound with respect to mathematical integers. The $\mathcal{A}$ domain can be any numerical abstract domain. For example, it can be the polyhedral domain, which can represent useful program invariants as inequalities. We also describe how to create abstract transformers for $BVS(\mathcal{A})$ that are sound with respect to bitvectors. For $v \subseteq V$ and $a_v \in \mathcal{A}$, we denote the result by $WRAP_v(a_v)$; the operation

performs wraparound on $a\nu$ for variables in ν. We give an algorithm for $WRAP_{\nu}(a\nu)$ that works for any relational abstract domain. We use a finite number of disjunctions of $\mathcal{A}$ elements—captured in the domain $\mathcal{FD}_{k}(\mathcal{A})$—to help retain precision. Our contributions are the following:

- We propose a framework for abstract domains, called $BVSFD_{d}(\mathcal{A})$, to express bit-precise relational invariants by performing wrap-around over abstract domain $\mathcal{A}$ and using disjunctions to retain precision. This abstract domain is parameterized by a positive value d, which provides the maximum number of disjunctions that the abstract domain can make use of.
- We provide a generic technique via reinterpretation to create the abstract transformer for the path through a basic block to a given successor, such that the transformer incorporates lazy wrap-around.
- We present experiments to show how the performance and precision of $BVSFD_{d}$ analysis changes with the tunable parameter d.

1.3 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 provides background on concrete semantics, abstract interpretation and automatic generation of abstract transformers via semantic reinterpretation and symbolic abstraction. Chapter 3 presents the theoretical and experimental comparison of MOS and KS domains. Chapter 4 presents the Affine-Transformers Abstraction framework, which generalizes the MOS domain. Chapter 5 describe the design and implementation of the Bit-Vector Inequality ($\mathcal{BVI}$) domain, that is capable of capturing certain inequalities over bit-vector-valued variables (which represent a program's registers and/or its memory variables). Chapter 6 describes the design and implementation of the Bit-Vector-Sound Finite-Disjunctive framework. Chapter 7 concludes and describes possible future work.

This chapter presents a gentle introduction to concrete semantics (§2.1), abstract interpretation (§2.2, §2.3, and §2.4) and weighted pushdown systems used to perform abstract interpretation (§2.5). The notion of the best abstract transformer is also introduced (§2.3.2).

2.1 Concrete Semantics

A semantics is a mathematical characterization of program behavior. The concrete semantics is the most precise semantics describing the actual execution of the program. Let us consider a very simple concrete language to illustrate concrete semantics. An SLang program is a sequence of basic blocks with execution starting from the first block. Each basic block consists of a sequence of statements and a list of control-flow instructions.

The statements are restricted to an assignment of a linear expression. A control-flow instruction consists of either a jump statement or a conditional statement. The guard in the conditional statement can be true, false, negation of a condition, conjunction/disjunction of two conditions, an equality/inequality on two bitvector expressions, or a modulus check operation. For the sake of simplicity, we assume that each variable in the program is a 32-bit unsigned integer.

$$
\begin{array}{rcl}
\langle \mathtt{SLang} \rangle & :: & (\mathtt{Block})* \\
\langle \mathtt{Block} \rangle & :: & \mathtt{l} : (\langle \mathtt{Stmt} \rangle \ \mathbf{;})^* \ \langle \mathtt{Next} \rangle \\
\langle \mathtt{Next} \rangle & :: & \mathbf{jump}\ \mathtt{l}\mathbf{;} \\
 & | & \mathbf{if}\ \phi_{\mathtt{Cond}}\ \mathbf{then\ jump}\ \mathtt{l}_1\ \mathbf{else\ jump}\ \mathtt{l}_2 \\
\langle \mathtt{Expr} \rangle & :: & \mathtt{n} \mid \mathtt{n} * \mathtt{v} + \langle \mathtt{Expr} \rangle \\
\langle \mathtt{Stmt} \rangle & :: & \mathtt{v} = \langle \mathtt{Expr} \rangle \\
\langle \mathtt{Cond} \rangle & :: & \mathtt{True} \mid \mathtt{False} \mid \mathtt{Not}(\langle \mathtt{Cond} \rangle) \mid \langle \mathtt{BCond} \rangle \mid \langle \mathtt{ECond} \rangle \\
\langle \mathtt{ECond} \rangle & :: & \langle \mathtt{Expr} \rangle \ \langle \mathtt{ExprOp} \rangle \ \langle \mathtt{Expr} \rangle \mid \langle \mathtt{Expr} \rangle \ \% \ \langle \mathtt{Expr} \rangle == \langle \mathtt{Expr} \rangle \\
\langle \mathtt{BCond} \rangle & :: & \langle \mathtt{Cond} \rangle \ \langle \mathtt{CondOp} \rangle \ \langle \mathtt{Cond} \rangle \\
\langle \mathtt{CondOp} \rangle & :: & \mathtt{And} \mid \mathtt{Or} \\
\langle \mathtt{ExprOp} \rangle & :: & == \mid < \mid \leqslant \mid \%
\end{array}
$$

A concrete state σ for an SLang program is a tuple of concrete values, $\sigma : \Pi_{v \in V} BV^{s(v)}$, where $s(v)$ is the size of variable v in bits and BV^b is a bitvector with b bits. We use k to denote the size of the vocabulary V. We use the row vector $\overrightarrow{v} = (v_1, v_2, \ldots, v_k)$ to denote the tuple of variables in the vocabulary V (which makes explicit the implicit order in "$\sigma : \Pi_{v \in V} BV^{s(v)}$"). A concrete state σ is described by the tuple of bitvectors $(bv_1, bv_2, \ldots bv_k)$, where the value of variable v_i is bv_i. The concrete states described by a tuple of length k are one-vocabulary concrete states. Let $\mathbb{C}$ be the set of all one-vocabulary concrete states.

The concrete transformer τ_E for a basic-block edge $E = B \rightarrow B'$ is described by $[\![E]\!]_{Block}$. A concrete transformer is a *two-vocabulary* transition relation, denoted by $R[\overrightarrow{v}; \overrightarrow{v}']$, where $\overrightarrow{v}' = (v_1', v_2', \ldots, v_k')$ is a row vector of length k. $\overrightarrow{v}$ and $\overrightarrow{v}'$ represent the pre-transformation state and the post-transformation state, respectively. τ_E is set of tuples of bitvectors of the form $(bv_1, bv_2, \ldots bv_k, bv_1', bv_2', \ldots bv_k')$. Thus, a concrete transformer is a set of two-vocabulary concrete states. A (two-vocabulary) concrete state is sometimes called an *assignment* to the variables of the pre-state and the post-state vocabulary. Let $\mathbb{C}$ be the set of all concrete transformers. The concrete domain $\mathcal{C} = \mathcal{P}[\mathbb{C}]$ is the powerset of the set of concrete transformers.

Fig. 2.1 provides the concrete semantics for the SLang program. The concrete semantics for an edge $B \rightarrow B'$ from the basic block B to basic block B' is denoted by $[\![B \rightarrow B']\!]$, where $[\![]\!] : \mathbb{C}$ represents the concrete evaluator. Rule 2.1 in Fig. 2.1 specifies how the concrete evaluation for basic-block pairs feeds into concrete evaluation on a sequence of statements. The concrete evaluator takes a one-vocabulary concrete state σ as input. Note that l' is the label corresponding to the basic block B'. Rule 2.2 states that the concrete evaluation of a sequence of instructions can be broken down into a concrete evaluation of a smaller sequence of instructions, by recursively performing statement-level concrete evaluation

$[\![.]\!]_{\mathtt{Stmt}}$ on the first instruction in the sequence. Rules 2.3, 2.4, 2.5, and 2.6 handle control-flow statements. Rule 2.3 delegates the responsibility of executing the last instruction in the statement sequence to $[\![.]\!]_{\mathtt{Next}}$. Rule 2.4 deals with unconditional-jump instructions. Rules 2.5 and 2.6 deal with conditional-jump instructions; both delegate to the assume evaluation function $[\![.]\!]_{\mathtt{Assume}}$. The assume is given the guard condition $\phi_{\mathtt{Cond}}$ for the true case and $\mathtt{Not}(\phi_{\mathtt{Cond}})$ for the false case. Rule 2.7 deals with assumes. Assume operations filters out the concrete states that do not match the assume condition. If the evaluation of ϕ on the concrete state σ is true, then the state σ is returned; otherwise, a special empty concrete state $\{\}$ is returned. The result $[\![E]\!]_{\mathtt{Block}}\sigma = \{\}$ specifies that the concrete state σ cannot concretely execute along the edge E.

Rule 2.8 handles assignment statements. Assignment $[\![v = exp]\!]_{\mathtt{Stmt}}$ returns a concrete state that is the same as σ except the variable v is updated with the evaluated value of exp. Rules 2.9-2.12 specify the concrete evaluation of linear expressions. Similarly, 2.13-2.17 specify the concrete evaluation of conditional expressions.

2.2 Abstract Interpretation

Abstract Interpretation is a process of discovering properties about the program by "running" a safe approximation of the program [20]. This safe approximation of the program is called an *abstraction* of the program. The building block for the abstraction of the program is an *abstract domain*, denoted by $\mathcal{A}$. An abstract domain defines the level and detail of the abstraction. The program properties inferred by means of abstract interpretation are a safe approximation of actual program properties, and hence they are invariants for the program.

Basic Block:

$$[\![B \to B']\!]_{Block}\sigma = [\![[s_1;...;s_n;nxt]]\!]_{Seq}(\sigma, l') \tag{2.1}$$

$$[\![[s_1;...;s_n;nxt]]\!]_{Seq}(\sigma, l') = [\![[s_2;...;s_n;nxt]]\!]_{Seq}(([\![s_1]\!]_{Stmt}\sigma), l') \tag{2.2}$$

Control Flow:

$$[\![[nxt]]\!]_{Seq}(\sigma, l') = [\![nxt]\!]_{Next}(\sigma, l') \tag{2.3}$$

$$[\![\textbf{jump } l']\!]_{Next}(\sigma, l') = \sigma \tag{2.4}$$

$$[\![\textbf{if } \phi_{Cond} \textbf{ then jump } l_1 \textbf{ else jump } l_2]\!]_{Next}(\sigma, l_1) = [\![assume(\phi_{Cond})]\!]_{Assume}\sigma \tag{2.5}$$

$$[\![\textbf{if } \phi_{Cond} \textbf{ then jump } l' \textbf{ else jump } l_2]\!]_{Next}(\sigma, l_2) = [\![assume(Not(\phi_{Cond}))]\!]_{Assume}\sigma \tag{2.6}$$

$$[\![\phi]\!]_{Assign}\sigma = \begin{cases} \sigma & \textit{if } [\![\phi]\!]_{Cond}\sigma = True \\ \{\} & \textit{otherwise} \end{cases} \tag{2.7}$$

Assignments:

$$[\![v = exp]\!]_{Stmt}\sigma = \sigma[v \leftarrow [\![exp]\!]_{Expr}\sigma] \tag{2.8}$$

Expressions:

$$[\![n]\!]_{Expr}\sigma = n \tag{2.9}$$

$$[\![v]\!]_{Expr}\sigma = \sigma[v] \tag{2.10}$$

$$[\![exp_1 * exp_2]\!]_{Expr}\sigma = [\![exp_1]\!]_{Expr}\sigma * [\![exp_2]\!]_{Expr}\sigma \tag{2.11}$$

$$[\![exp_1 + exp_2]\!]_{Expr}\sigma = [\![exp_1]\!]_{Expr}a + [\![exp_2]\!]_{Expr}\sigma \tag{2.12}$$

Conditions:

$$[\![True/False]\!]_{Cond}\sigma = True/False \tag{2.13}$$

$$[\![exp_1\ Op\ exp_2]\!]_{Cond}\sigma = [\![exp_1]\!]_{Expr}\sigma\ \ Op\ \ [\![exp_2]\!]_{Expr}\sigma \tag{2.14}$$

$$[\![Not\ (exp)]\!]_{Cond}\sigma = Not([\![exp]\!]_{Expr}\sigma) \tag{2.15}$$

$$[\![exp_1\ \%\ exp_2 == exp_3]\!]_{Cond}\sigma = [\![exp_1]\!]_{Expr}\sigma\ \ \%\ \ [\![exp_2]\!]_{Expr}\sigma == [\![exp_3]\!]_{Expr}\sigma \tag{2.16}$$

$$[\![cond_1\ Op\ cond_2]\!]_{Cond}\sigma = [\![cond_1]\!]_{Cond}\sigma\ \ Op\ \ [\![cond_2]\!]_{Cond}\sigma \tag{2.17}$$

Figure 2.1: Concrete semantics for $\mathcal{L}(SLANG)$.

2.2.1 Abstract Domain

As mentioned in §2.1, the *concrete domain*, denoted by $\mathcal{C}$, is the powerset of the set of concrete transformers. Often, but not always, the concrete

domain $\mathcal{C}$ and the abstract domain $\mathcal{A}$ form a Galois connection $\mathcal{G} = \mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ [20]. γ takes an abstract-domain element A, and gives back the concrete-domain element $C \in \mathcal{C}$, that A represents. α takes a concrete domain value $C \in \mathcal{C}$, and abstracts C to the least abstract-domain element A such that the set of concrete elements represented by A is a superset of C. For a given abstract domain $\mathcal{A}$, $\mathcal{A}[V]$ denotes the specific instance of $\mathcal{A}$ that is defined over vocabulary V.

Parity Abstract Domain

As an illustrative example of an abstract domain, consider the *parity* abstract domain that only tracks whether each variable in the program is even or odd. A parity map is a mapping from variables to their parity, $pm : V \to \{e, o\}$, where e represents the set of even integers and o represents the set of odd integers. Thus, $\gamma(e) = \{0, 2, 4, \ldots, 2^{32} - 2\}$ and $\gamma(o) = \{1, 3, 5, \ldots, 2^{32} - 1\}$ for 32-bit unsigned integers. The concretization $\gamma(pm) = \{(bv_1, bv_2, \ldots, bv_k) : \bigwedge_{0 \leqslant i \leqslant k} bv_i \in \gamma(pm(v_i))\}$. Let us denote the set of all parity maps by $\mathcal{PM}$. We define the the parity abstract domain, denoted by $\mathcal{A}_{Par}$, as the powerset of $\mathcal{PM}$. Therefore, $\mathcal{A}_{Par} = \mathcal{P}[\mathcal{PM}]$. The concretization $\gamma(a)$ for an element $a \in \mathcal{A}_{Par}$ is defined as the union over the concretizations of the parity maps in a.

$$\gamma(a) = \bigcup_{pm \in a} \gamma(pm)$$

Example 2.1. *Consider* $V = \{v, v'\}$ *and* $a \in \mathcal{A}_{Par}[V]$, *such that* $a = \{(e, e), (o, o)\}$. *The abstract element* a *represent all concrete states such that parity(v) = parity(v'). Consider the abstraction for the concrete state set* $css = \{(1, 3), (11, 13), (211, 215)\}$. $\alpha(css) = \{(o, o)\}$. *Notice that some precision is lost in the abstraction process. For instance, the concrete state* $(1, 1) \in \gamma(\alpha(css))$, *but* $(1, 1) \notin css$.

Table 2.1: Abstract-domain operations.

Result Type	Operation	Description
$\mathcal{A}$	$\bot$	*bottom element*
bool	$(a_1 == a_2)$	*equality*
$\mathcal{A}$	$(a_1 \sqcup a_2)$	*join*
$\mathcal{A}$	$(a_1 \nabla a_2)$	*widen*
$\mathcal{A}$	Id	*identity element*
$\mathcal{A}$	$(a_1 \circ a_2)$	*composition*

2.2.2 Abstract-Domain Operations

A static analyzer needs a way to construct abstract transformers for the concrete operations in the programs. *Semantic reinterpretation* and *symbolic abstraction* provide automatic ways to construct abstract transformers. We provide a more detailed discussion of these automatic techniques in §2.3.

For an analysis that provides function summaries or loop summaries, the fixpoint analysis is performed using equality, join ($\sqcup$), and abstract-composition ($\circ$) operations on abstract transformers. We provide a more detailed description of the fixpoint analysis in §2.4.

For the analyses discussed in this dissertation, the program is abstracted to a control-flow graph, where each edge in the graph is labeled with an abstract transformer. As described in §2.1, an abstract transformer is a *two-vocabulary* transition relation $R[\overrightarrow{v}; \overrightarrow{v}']$, where $\overrightarrow{v}$ and $\overrightarrow{v}'$ are row vectors of length k that represent the pre-transformation state and the post-transformation state, respectively.

Tab. 2.1 lists the abstract-domain operations needed to generate the program abstraction and perform fixpoint analysis on it. The bottom element is the abstract-domain element at the bottom of the abstract domain. The concretization of the bottom element is the empty set: $\gamma(\bot) = \varnothing$. Equality is the standard lattice equality operation. Join is the least upper bound operation on the abstract-domain elements, that is, $a_1 \sqcup a_2 \sqsupseteq a_1, a_2$ and $\forall_{a_3} : a_3 \sqsupseteq a_1, a_2 \Rightarrow a_3 \sqsupseteq a_1 \sqcup a_2$. The *widen* operation is needed

for domains with infinite ascending chains to ensure termination [23]. For abstract domains with finite ascending chains, the widen operation can be safely replaced with the join operation. $\mathtt{Id}$ is the identity element, which represents the identity transformation $(\bigwedge_{i=1}^{k} v_i' = v_i)$. Finally, the abstract-composition operation $a_1 \circ a_2$ returns a sound overapproximation of the composition of abstract transformer a_1 with abstract transformer a_2.

For the parity abstract domain, join is set union and meet is set intersection. The empty set $\{\}$ is the bottom element for $\mathcal{A}_{\mathtt{Par}}$. Consider $\overrightarrow{v} = (v, v')$. The abstract-domain element $\{(e,e),(o,o)\}$ is the identity transformer, specifying that the parity of the variable v is preserved.

Example 2.2. *Consider* $a_1 = \{(e,e),(o,o)\}$ *and* $a_2 = \{(e,o),(o,e)\}$. *The abstract transformer* a_1 *states that the parity of the variable* v *is preserved. Similarly, the abstract transformer* a_2 *states that the parity of variable* v_2 *is flipped. The join of* a_1 *and* a_2, *denoted by* $a_1 \sqcup a_2$ *is* $\{(e,e),(o,o),(e,o),(o,e)\}$. $a_1 \sqcup a_2$ *represents the set of all concrete states because all the permutations of parity transformations are possible. The meet of* a_1 *and* a_2, *denoted by* $a_1 \sqcap a_2$ *is* $\{\}$.

Consider the abstract composition of $a_1 \circ a_2$. *Intuitively, the abstract transformer* $a_1 \circ a_2$ *takes the parity of the variable* v *and,*

1. *applies the abstract transformation* a_1 *to* v*'s parity (which preserves the parity) to get a temporary parity* t_p,
2. *applies the abstract transformation* a_2 *to the parity* t_p *(which flips the parity).*

The overall effect is that the parity of the variable v *is flipped. Thus,* $a_1 \circ a_2 = \{(e,o),(o,e)\} = a_2$.

2.3 Creation of Abstract Transformers.

To perform program analysis, the program-state transitions that are associated with the edges of a control-flow graph also need to be abstracted. We will use the map $[\![]\!]^\sharp : E \to \mathcal{A}$ to denote the map that specifies abstract trans-

formers for each edge in the CFG. We say that $[\![]\!]^\sharp$ is a sound approximation of $[\![]\!]$ if the following condition holds for every edge $e \in E$:

$$\gamma([\![e]\!]^\sharp) \supseteq \gamma([\![e]\!])$$

Example 2.3. *The following example is used to illustrate abstract interpretation with the parity abstract domain:*

```
L0:    v=v+1
L1:    while(*) {
L2:        v=v+2
       }
L3:    if(v%4==2) {
L4:        v=v+1
L5:    }
END:
```

For instance, the abstract transformer for the concrete operation $v = v + 2$ *starting from node* L2 *and ending at node* L1*, denoted by* $\tau^\sharp_{L2\to L1}$*, is defined as* $\{(e, e), (o, o)\}$*. Thus, the abstract transformer* $\tau^\sharp_{L2\to L1}$ *is the identity function on parity maps because* $v = v + 2$ *does not change the parity of variable* v*.*

More interesting is the abstract transformer for $v = v + 1$ *which flips the parity of* v*.* $\tau^\sharp_{L0\to L1}$ *is defined as* $\{(e, o), (o, e)\}$*.* □

2.3.1 Semantic Reinterpretation

In this section, we describe how the abstract transformers are generated using semantic reinterpretation [47, 79, 81, 64, 60, 30]. Semantic reinterpretation is an automatic and efficient method to create abstract transformers. Semantic reinterpretation makes use of abstract components of each of the concrete operations and concrete data types used to specify the semantics of a programming language. These abstract components are collectively referred to as the semantic core [60] (sometimes called a semantic algebra [90]). Specifically, the reinterpretation consists of a domain of abstract transformers $\mathcal{A}[V, V']$, a domain of abstract integers $\mathcal{A}^{INT}[t, V]$, and operations to lookup a variable's value in the post-state

of an abstract transformer and to create an updated version of a given abstract transformer. Here t denotes a temporary variable not in V or V'. Given blocks $B : [l : s_1; ...s_n; nxt]$ and $B' : [l' : s'_1; ...s'_n; nxt']$ in an SLang program (see §2.1), where B' is a successor of B, reinterpretation of B can provide an abstract transformer for the transformation that starts from the first instruction in B and ends in the first instruction in B', denoted by $B \rightarrow B'$.

Fig. 2.2 provide the reinterpretation semantics for the $\mathcal{A}_{Par}$ domain. Rule 2.18 specifies how abstract-transformer evaluation for basic-block pairs feeds into abstract-transformer evaluation on a sequence of statements. The evaluation on a sequence of statements starts with the identity abstract transformer, denoted by Id. Rule 2.19 states that the abstract transformer for a sequence of instruction can be broken down into an abstract transformer for a smaller sequence of instruction, by recursively performing statement-level abstract interpretation $[\![.]\!]^\sharp_{Stmt}$ on the first instruction in the sequence. In this rule and subsequent $[\![.]\!]^\sharp_{Next}$ and $[\![.]\!]^\sharp_{Stmt}$ rules, "a" denotes the intermediate abstract transformer value. It starts as Id at the beginning of the instruction sequence, and gets updated or accessed by assignment and control-flow statements in the sequence.

Rules 2.20, 2.21, 2.22 and 2.23 handle control-flow statements. Rule 2.20 delegates the responsibility of executing the last instruction in the statement sequence to $[\![.]\!]^\sharp_{Next}$. Rule 2.21 deals with unconditional-jump instructions. Rules 2.22 and 2.23 handle conditional branching by delegating the responsibility to $[\![assume(\phi_{Cond})]\!]^\sharp_{Assume}a$ and $[\![assume(Not(\phi_{Cond}))]\!]^\sharp_{Assume}a$ respectively. Rule 2.24 handles assume by delegating the abstract execution of the conditional ϕ to $[\![.]\!]^\sharp_{Cond}$ (see rules 2.32-2.35) and then performing a meet operation.

Rule 2.25 handles the assignment statement by merely performing a post-state-vocabulary update on the current abstract transformer "a."

Rules 2.26-2.29 handle reinterpretation of expressions. The abstract

Basic Block:

$$[\![B \to B']\!]^\sharp_{Block} = [\![[s_1;...;s_n;nxt]]\!]^\sharp_{Seq}(Id, l') \tag{2.18}$$

$$[\![[s_1;...;s_n;nxt]]\!]^\sharp_{Seq}(a, l') = [\![[s_2;...;s_n;nxt]]\!]^\sharp_{Seq}(([\![s_1]\!]^\sharp_{Stmt}(a), l') \tag{2.19}$$

Control Flow:

$$[\![[nxt]]\!]^\sharp_{Seq}(l') = [\![nxt]\!]^\sharp_{Next}(l') \tag{2.20}$$

$$[\![\textbf{jump } l']\!]^\sharp_{Next}(a, l') = a \tag{2.21}$$

$$[\![\textbf{if } \phi_{Cond} \textbf{ then jump } l_1 \textbf{ else jump } l_2]\!]^\sharp_{Next}(a, l_1) =$$
$$[\![assume(\phi_{Cond})]\!]^\sharp_{Assume} a \tag{2.22}$$

$$[\![\textbf{if } \phi_{Cond} \textbf{ then jump } l' \textbf{ else jump } l_2]\!]^\sharp_{Next}(a, l_2) =$$
$$[\![assume(Not(\phi_{Cond}))]\!]^\sharp_{Assume} a \tag{2.23}$$

$$[\![\phi]\!]^\sharp_{Assume} a = a \sqcap [\![\phi]\!]^\sharp_{Cond} a \tag{2.24}$$

Assignments:

$$[\![v = exp]\!]^\sharp_{Stmt} a = update(a, v', [\![exp]\!]^\sharp_{Expr} a) \tag{2.25}$$

Expressions:

$$[\![n]\!]^\sharp_{Expr} a = \{(parity(n), p_1, p_2, \ldots, p_k) : p_i \in \{e, o\}\} \tag{2.26}$$

$$[\![v]\!]^\sharp_{Expr} a = lookup(a, v') \tag{2.27}$$

$$[\![exp_1 * exp_2]\!]^\sharp_{Expr} a = [\![exp_1]\!]^\sharp_{Expr} a *^\sharp [\![exp_2]\!]^\sharp_{Expr} a \tag{2.28}$$

$$[\![exp_1 + exp_2]\!]^\sharp_{Expr} a = [\![exp_1]\!]^\sharp_{Expr} a +^\sharp [\![exp_2]\!]^\sharp_{Expr} a \tag{2.29}$$

Abstract Integers:

$$i *^\sharp i' = \{(parity(p_t) *^\sharp parity(p'_t), p_1, p_2, \ldots, p_k) :$$
$$(p_t, p_1, p_2, \ldots, p_k) \in i, (p'_t, p_1, p_2, \ldots, p_k) \in i'\} \tag{2.30}$$

$$i +^\sharp i' = \{(parity(p_t) +^\sharp parity(p'_t), p_1, p_2, \ldots, p_k) :$$
$$(p_t, p_1, p_2, \ldots, p_k) \in i, (p'_t, p_1, p_2, \ldots, p_k) \in i'\} \tag{2.31}$$

Conditions:

$$[\![True/False]\!]^\sharp_{Cond} a = \top/\bot \tag{2.32}$$

$$[\![exp \,\%\, 2 == 0]\!]^\sharp_{Cond} a = \{(p_1, \ldots, p_k, p_1, \ldots, p_k) :$$
$$(e, p_1, \ldots, p_k) \in [\![exp]\!]_{Expr} a\} \tag{2.33}$$

$$[\![exp \,\%\, 2 == 1]\!]^\sharp_{Cond} a = \{(p_1, \ldots, p_k, p_1, \ldots, p_k) :$$
$$(o, p_1, \ldots, p_k) \in [\![exp]\!]_{Expr} a\} \tag{2.34}$$

$$[\![\phi_{Other}]\!]^\sharp_{Cond} a = \top \tag{2.35}$$

Variable lookup and update:

$$lookup(a, v'_i) = \{(p'_i, p_1, \ldots, p_k) : (p_1, \ldots, p_k, p'_1, \ldots, p'_k) \in a\} \tag{2.36}$$

$$update(a, v'_i, ai) = \{(p_1, p_2, \ldots, p_k, p'_1, p'_2, \ldots, p_{i-1}, p_t, p_{i+1} \ldots, p'_k) :$$
$$(p_1, p_2, \ldots, p_k, p'_1, p'_2, \ldots, p'_k) \in a, (p_t, p_1, p_2, \ldots, p_k) \in ai\} \tag{2.37}$$

Figure 2.2: Reinterpretation semantics for $\mathcal{L}(\text{SLANG})$ for domain $\mathcal{A}_{Par}$.

p_1	p_2	$*^\sharp$	$+^\sharp$
e	e	e	e
e	o	e	o
o	e	e	o
o	o	o	e

Table 2.2: Truth table for multiplication and addition of two parity values.

interpretation of an expression gives back an abstract integer. An abstract integer is an abstract-domain element over $(t, \overrightarrow{v})$, where t is the temporary variable not in $\overrightarrow{v}$ referring to the value of abstract integer. For example, the abstract integer $i = \{(e, e), (o, o)\}$, with $\overrightarrow{v} = \{v_1\}$, is an abstract integer whose parity is the same as $\{v_i\}$. Rule 2.26 creates an abstract integer from a constant integer n by setting the *parity* of the t variable to *parity(n)* in all possible pre-vocabulary-state parity combinations. Rule 2.27 creates an abstract integer from a variable by performing a variable "lookup" operation in a. Rules 2.28 and 2.29 delegate computation to the corresponding abstract-integer operations.

Rules 2.30 and 2.31 are abstract-multiplication and abstract-addition operations respectively. The operators $*^\sharp$ and $+^\sharp$ for $\{e, o\}$ are defined in Tab. 2.2.

Rules 2.32-2.35 provide abstract interpretation of conditionals. The result of these operations is an abstract transformer that overapproximates the set of values satisfied by the conditional for the given input abstract transformer a. Rule 2.32 handles the true and false conditionals. Rules 2.33 and 2.34 handles the conditionals which check for the parity of an expression "*e*xp". Rule 2.33 filters the tuples in the abstract integers for "*e*xp", where the parity for "*e*xp" is even. Similarly, rule 2.34 only considers tuples where the parity for "*e*xp" is odd. Rule 2.35 safely handles all the other expression to return $\top$. For brevity, we do not explicitly give rules for the Boolean operators AND, OR and NOT, but they can be implemented

as set intersection, set union, and set complementation, respectively.

Rules 2.36 and 2.37 are variable lookup and update operations. Lookup takes an abstract transformer a and a variable v_i, and returns the abstract integer ai such that the relationship of t with $\overrightarrow{v}$ in ai is the same as the relationship of v'_i with $\overrightarrow{v}$ in a. The variable-update operation works in the opposite direction. Update takes an abstract transformer a, a variable v'_i, and ai, and returns a' such that the relationship of v'_i with $\overrightarrow{v}$ in a' is the same as the relationship of t with $\overrightarrow{v}$ in ai, and all the other relationships that do not involve v' remain the same.

Example 2.4. *For instance, consider the abstract transformer for the concrete operation* $v = v + 1$*, denoted by* $\tau^{\sharp}_{L0\to L1}$*, Remember that the desired answer is* $\{(e,o),(o,e)\}$*. Semantic reinterpretation can arrive at this answer by performing* $[\![[v = v + 1; \mathtt{jmp}\ L1]]\!]^{\sharp}_{Seq}(Id, L1)$*, where* $Id = \{(e,e),(o,o)\}$*. First, it will perform* $[\![v = v + 1]\!]^{\sharp}_{Stmt}$*, which in turn will obtain* $[\![v]\!]^{\sharp}_{Expr} = \{(e,e),(o,o)\}$ *(through rule 2.36) and* $[\![1]\!]^{\sharp}_{Expr} = \{(o,e),(o,o)\}$ *(through rule 2.26). Using these values,* $[\![v+1]\!]^{\sharp}_{Expr} = [\![v]\!]^{\sharp}_{Expr} +^{\sharp} [\![1]\!]^{\sharp}_{Expr} = \{(e +^{\sharp} o, e), (o +^{\sharp} o, o)\} = \{(o,e),(e,o)\}$*. Then,* $update(Id, v', \{(o,e),(e,o)\})$ *is performed to obtain* $[\![v = v + 1]\!]^{\sharp}_{Stmt} = \{(e,o),(o,e)\}$*. Finally, this value is left unchanged by the unconditional jump to* $L1$ *to obtain* $\tau^{\sharp}_{L0\to L1} = \{(e,o),(o,e)\}$*.*

2.3.2 Symbolic Abstraction

Symbolic abstraction (denoted by $\hat{\alpha}$) [84] provides an automatic way of obtaining the best abstract transformer, given the concrete semantics of the transformer as a formula (for abstract domains and logics that meet certain properties [84]).

Best Abstract Transformer For every concrete operation (concrete transformer) in the actual program, the corresponding over-approximating abstract operation (abstract transformer) needs to be provided to perform

abstract interpretation (§2.2). Cousot and Cousot [21] give the notion of a *best* abstract transformer, but do not provide an algorithm to either (i) apply the best transformer, or (ii) create a representation of the best transformer. Their notion is as follows: If τ is a concrete transformer, let τ^s be τ lifted to operate on a set pointwise (i.e., $\tau^s(S) = \{\tau(s) | s \in S\}$). Then the best abstract transformer $\tau^\sharp$ equals $\alpha \circ \tau^s \circ \gamma$.

An abstract transformer can be viewed as an element in an abstract domain in which each element represents an input-output relation. The advantage of adopting this viewpoint is that it allows symbolic abstraction — and in particular *algorithms* for symbolic abstraction — to be applied to the problem of creating best abstract transformers. That is, if φ_τ is a formula in $\mathcal{L}$ that captures the semantics of concrete transformer τ, $\mathcal{A}$ is the abstract domain of abstract transformers, and $\widehat{\alpha}_{\mathcal{A}}$ is a symbolic-abstraction algorithm for mapping an $\mathcal{L}$ formula to the smallest over-approximating value in $\mathcal{A}$, then $\widehat{\alpha}_{\mathcal{A}}(\varphi_\tau)$ yields the best abstract transformer for τ. $\widehat{\alpha}_{\mathcal{A}}(\varphi_\tau)$ is the symbolic version of the α function, referred as the symbolic-abstraction operation. Similarly, the symbolic-concretization operation, denoted by $\widehat{\gamma}_{\mathcal{A}}(a)$, is the symbolic version of the γ function; it takes an abstract value $a \in \mathcal{A}$ and returns a formula $\varphi \in \mathcal{L}$ describing the abstract value symbolically as a logical formula. Thus, a and $\widehat{\gamma}_{\mathcal{A}}(a)$ represent the same set of concrete states (i.e., $\gamma_{\mathcal{A}}(a) = [\![\widehat{\gamma}_{\mathcal{A}}(a)]\!]$). Usually, symbolic concretization is straightforward to implement for abstract domains. For instance,

$$\widehat{\gamma}_{\mathcal{A}_{\mathrm{Par}}}(a) = \bigvee_{(p_1,p_2,\cdots,p_k,p'_1,p'_2,\cdots,p'_k)\in a} \Big(\bigwedge_{i=0}^{k} (v_i \% 2 = (p_i = e)?0 : 1) \wedge (v'_i \% 2 = (p'_i = e)?0 : 1) \Big)$$

I now discuss two approaches to symbolic abstraction, namely, *symbolic abstraction from below* [84] and the *bilateral approach to symbolic abstraction* [100].

Symbolic Abstraction From Below

Symbolic Abstraction From Below ($\hat{\alpha}^{be}$) [84] starts from $\bot$ (the abstract value that denotes the empty set), and climbs up the abstract-domain lattice using models obtained from a succession of calls to a Satisfiability Modulo Theory (SMT) solver (Alg. 1). A satisfiability modulo theories (SMT) solver [74, 26] is a decision procedure for logical formulas with respect to combinations of background theories expressed in classical logic. For instance, the logic can be Quantifier Free Bit-Vector logic (QFBV). The SMT solver takes a logical formula and returns SAT (or UNSAT) if the formula is satisfiable (or unsatisfiable). The beta (β) function, also referred as the representation function, obtains an abstract-domain element given a concrete state or model (M). The algorithm refines the search space after each SMT call to ensure progress by using $\neg\hat{\gamma}(ans)$ as a blocking clause. Once the SMT call returns unsatisfiable, ans contains an over-approximation of $[\![\varphi]\!]$.

This algorithm is not guaranteed to terminate when the abstract-domain lattice has infinite ascending chains. Moreover, because *ans* is initialized to $\bot$ and always has a value that is $\sqsubseteq$ the desired final answer, the algorithm is unable to give back a non-trivial abstract value, i.e., a value other than $\top$, in case the SMT solver times out.

Algorithm 1 Symbolic Abstraction From Below $\hat{\alpha}^{be}(\varphi)$

```
1: ans ← ⊥
2: while r = SMTCall(φ ∧ ¬γ̂(ans)) is Sat do
3:     M ← GetModel(r)
4:     ans ← ans ⊔ β(M)
5: return ans
```

Example 2.5. *Consider the abstraction of the concrete transformer* $\tau_{L3\to L4}$*. The symbolic formula for this transformer is* $\varphi_{\tau_{L3\to L4}} = (v' = v) \wedge (\mathrm{mod}(v', 4) = 2)$*, where* mod *is the modulus operator. First,* $\mathtt{ans} = \bot$ *and SMTCall returns true.*

Suppose the model M *is* $(v = 2, v' = 2)$. $\beta(v = 2, v' = 2) = (e, e)$, *because that is the best value that overapproximates model* M. *Thus,* $ans = \{(e, e)\}$ *and* $\hat{\gamma}(ans) = (mod(v', 2) = 0 \wedge mod(v, 2) = 0)$ *and consequently, the next SMT query is given as* $(v' = v) \wedge (mod(v', 4) = 2) \wedge \neg(mod(v', 2) = 0 \wedge mod(v, 2) = 0)$. *This time the query is unsatisfiable and* $ans = \{(e, e)\}$ *is returned.*

Note that the value obtained via semantic reinterpretation is $[\![L3 \rightarrow L4]\!]^{\sharp}_{Block} = \top$. *This answer is returned because the reinterpretation semantics is myopic and greedy, and is incapable of obtaining the best abstract transformer in the general case.*

Bilateral Approach to Symbolic Abstraction

The bilateral algorithm for symbolic abstraction ($\hat{\alpha}^{bi}$) [100] starts from both below ($\bot$) and above ($\top$). (See Alg. 2.) At any point in the computation, it maintains two values, *lower* (an under-approximation of the best abstract transformer) and *upper* (an over-approximation of the best abstract transformer). When *lower* is equal to *upper*, the algorithm stops and returns *upper*. For *lower* and *upper* such that *lower* $\sqsubsetneq$ *upper*, an abstract consequence is any value p, such that $p \sqsupseteq$ *lower* and $p \not\sqsubseteq$ *upper*. These restrictions on p guarantee that the algorithm makes progress on each step. For conjunctive domains, p can be chosen as any conjunct in *lower* not implied by *upper*.

Like $\hat{\alpha}^{be}$, this algorithm is not guaranteed to terminate when the abstract-domain lattice has infinite ascending chains. The key advantage of this algorithm over $\hat{\alpha}^{be}$ is that $\hat{\alpha}^{bi}$ can return *upper* as a safe over-approximation in case of a timeout. This safe over-approximation of the symbolic abstraction ($\hat{\alpha}$) is denoted by $\tilde{\alpha}$. Thus, for any given φ, $\hat{\alpha}(\varphi) \sqsubseteq \tilde{\alpha}(\varphi)$.

Example 2.6. *Consider the abstraction of the concrete transformer* $\tau_{L3 \rightarrow L4}$ *again. First,* $lower = \bot$, $upper = \top$ *and SMTCall returns true. Suppose that the model*

Algorithm 2 Bilateral Symbolic Abstraction $\widehat{\alpha}^{\mathtt{bi}}(\varphi)$

```
1: lower ← ⊥
2: upper ← ⊤
3: while lower ≠ upper do
4:     p ← AbstractConsequence(lower, upper)
5:     r ← SMTCall(φ ∧ ¬γ̂(p))
6:     if r is Sat then
7:         M ← GetModel(r)
8:         lower ← lower ⊔ β(M)
9:     else
10:        upper ← upper ⊓ p
11: return upper
```

M *is* $(v = 2, v' = 2)$*. Thus,* $\beta(v = 2, v' = 2) = (e, e)$*,* $lower = \{(e, e)\}$ *and* $\widehat{\gamma}(lower) = (\mathtt{mod}(v', 2) = 0 \wedge \mathtt{mod}(v, 2) = 0)$*. The bilateral algorithm calls the AbstractConsequence operation. Note that the abstract consequence* p *can be created automatically by adding 0 or more tuples to lower such that* p *is a subset of upper. Suppose that AbstractConsequence adds the tuple* (e, o)*, so that* $p = \{(e, o), (e, e)\}$*. Consequently, the next SMT query is given as* $(v' = v) \wedge (\mathtt{mod}(v', 4) = 2) \wedge \neg(\mathtt{mod}(v, 2) = 0)$*. This time the query is unsatisfiable and upper is set to* $\top \sqcap p = \{(e, o), (e, e)\}$*. Because upper is still not equal to lower, AbstractConsequence* p *is called again. This time, there is only one option for* p*; namely* $\{(e, e)\}$*. The next SMT query is given as* $(v' = v) \wedge (\mathtt{mod}(v', 4) = 2) \wedge \neg(\mathtt{mod}(v', 2) = 0 \wedge \mathtt{mod}(v, 2) = 0)$*. The query is unsatisfiable and upper is set to* $\{(e, o), (e, e)\} \sqcap p = \{(e, e)\}$*. upper is now equal to lower, and upper is returned.*

Note that the advantage of this approach is that a sound non-trivial value of $upper = \{(e, o), (e, e)\}$ *could be returned if the third SMT call timed out.*

2.4 Fixed-point computation

To obtain program summaries, an iterative fixed-point computation needs to be performed. Tab. 2.3 provides the fixpoint analysis for Ex. 2.3. Each row in the table shows the value of path summaries $PS_n(L0 \rightarrow L*)$ starting from the beginning of the program L0 to different program points in the program, where n is the iteration number of the path summary. The intermediate path summary at the first iteration (i), denoted by $PS^\sharp_{(i)}(L0 \rightarrow L1)$, is obtained from $\tau^\sharp_{L0\rightarrow L1}$. The intermediate path summary at the first iteration (ii), denoted by $PS^\sharp_{(ii)}(L0 \rightarrow L1)$ is calculated as the join of the path summary at the previous iteration $PS^\sharp_{(i)}(L0 \rightarrow L1)$ with the abstract composition of $\tau^\sharp_{L2\rightarrow L1}$ and $PS^\sharp_{(i)}(L0 \rightarrow L1)$. Quiescence is discovered during the second iteration, because $PS^\sharp_{(ii)}(L0 \rightarrow L1) \sqsubseteq PS^\sharp_{(i)}(L0 \rightarrow L1)$. Therefore, $PS^\sharp_{(ii)}(L0 \rightarrow L1)$ is the path summary from L0 to L1. Using this path summary for program point L1, path summary at L3, denoted by $PS^\sharp_{(i)}(L0 \rightarrow L3)$ can be calculated by abstract composition of $\tau^\sharp_{L1\rightarrow L3}$ and $PS^\sharp_{(ii)}(L0 \rightarrow L1)$. In a similar manner, $PS^\sharp_{(i)}(L0 \rightarrow L4)$ is calculated as $\tau^\sharp_{L3\rightarrow L4} \circ PS^\sharp_{(i)}(L0 \rightarrow L3)$. Because node L5 is the confluence point of edges L3 $\rightarrow$ L5 and L4 $\rightarrow$ L5, the fixpoint iteration needs to perform a join operation of the path summaries obtained from abstract composition along these two edges. Finally, $PS^\sharp_{(i)}(L0 \rightarrow END) = \{(e, o), (o, e), (o, o)\}$ is obtained by composing $PS^\sharp_{(i)}(L0 \rightarrow L5)$ with the abstract semantics $\tau_{L5\rightarrow END}$ of the edge from L5 to END, which has the identity transformation Id.

The abstract interpretation with the parity abstract domain $\mathcal{A}_{Par}$ is able to show that if the input value of v is even, then the output value of v will always be odd; if the input value is odd, then nothing is known about the parity of the output value of v.

Table 2.3: Snapshots in the fixed-point analysis for Ex. 2.3.

Iteration	Equation	Value
$PS^{\sharp}_{(i)}(L0 \to L1)$ $PS^{\sharp}_{(ii)}(L0 \to L1)$	$\tau^{\sharp}_{L0\to L1}$ $PS^{\sharp}_{(i)}(L0 \to L1) \sqcup$ $(\tau^{\sharp}_{L1\to L2} \circ PS^{\sharp}_{(i)}(L0 \to L1))$	$\{(e,o),(o,e)\}$ $\{(e,o),(o,e)\} \sqcup \{(e,o),(o,e)\}$ $= \{(e,o),(o,e)\}$
$PS^{\sharp}_{(i)}(L0 \to L3)$	$\tau^{\sharp}_{L1\to L3} \circ PS^{\sharp}_{(ii)}(L0 \to L1)$	$\{(e,e),(o,o)\} \circ \{(e,o),(o,e)\}$ $= \{(e,o),(o,e)\}$
$PS^{\sharp}_{(i)}(L0 \to L4)$	$\tau^{\sharp}_{L3\to L4} \circ PS^{\sharp}_{(i)}(L0 \to L3)$	$\{(e,e)\} \circ \{(e,o),(o,e)\}$ $= \{(o,e)\}$
$PS^{\sharp}_{(i)}(L0 \to L5)$	$(\tau^{\sharp}_{L4\to L5} \circ PS^{\sharp}_{(i)}(L0 \to L4)) \sqcup$ $(\tau^{\sharp}_{L3\to L5} \circ PS^{\sharp}_{(i)}(L0 \to L3))$	$\{((e,o),(o,e)\} \circ \{(o,e)\}) \sqcup$ $\{(e,e),(o,o)\} \circ \{(e,o),(o,e)\}$ $= \{(o,o)\} \sqcup \{(e,o),(o,e)\}$ $= \{(e,o),(o,e),(o,o)\}$
$PS^{\sharp}_{(i)}(L0 \to END)$	$(\tau^{\sharp}_{L5\to END} \circ PS^{\sharp}_{(i)}(L0 \to L5))$	$Id \circ \{(e,o),(o,e),(o,o)\}$ $= \{(e,o),(o,e),(o,o)\}$

2.5 Weighted Pushdown Systems

Weighted PushDown Systems (WPDSs) are a modern formalism for solving flow-sensitive, context-sensitive, interprocedural dataflow-analysis problems [10, 85].

A weighted pushdown system is a pushdown system whose rules are given values from some domain of weights. The weight domains of interest are bounded idempotent semirings.

2.5.1 Bounded Idempotent Semiring

Definition 2.7. *A **bounded idempotent semiring** is a quintuple* $(D, \oplus, \otimes, 0, 1)$*, where D is a set, 0 and 1 are elements of D, and* $\oplus$ *(the join operation) and* $\otimes$ *(the extend operation) are binary operators on D such that*

1. *(D,* $\oplus$*) is a commutative monoid with 0 as its neutral element, and where* $\oplus$ *is idempotent (i.e., for all* $a \in D$*,* $a \oplus a = a$*).*
2. *(D,* $\otimes$*) is a monoid with the neutral element 1.*

3. $\otimes$ *distributes over* $\oplus$*, i.e., for all* $a, b, c \in D$ *we have*

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and}$$
$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c).$$

4. *0 is an annihilator with respect to* $\otimes$*, i.e., for all* $a \in D$, $a \otimes 0 = 0 = 0 \otimes a$.
5. *In the partial order* $\sqsubseteq$ *defined by:* $\forall a, b \in D, a \sqsubseteq b$ *iff* $a \oplus b = b$*, there are no infinite ascending chains.*

Defn. 2.7(1) and Defn. 2.7(5) mean that $(D, \oplus)$ is a **join semilattice with no infinite ascending chains**.

Note that the distributivity property (Property 3 in Defn. 2.7) holds only if the extend operation distributes over the join operation. The analysis is still sound if the semiring is not distributive. If the semiring is distributive, then the WPDS-based dataflow analysis gives the join-over-all-paths solution.

2.5.2 Application of WPDS to Interprocedural Dataflow Analysis

For our interprocedural-analysis experiments, we use the WALi system [50] for WPDSs to perform abstract interpretation (§2.2). To construct the WPDS from a given control flow graph (CFG), each edge in the CFG (which represents a concrete transformer) is converted to a rule in WPDS. The semiring weight on an edge in the CFG is calculated by abstracting the concrete transformer for the edge to an abstract transformer. Hence, the WPDS weights are abstract transformers, represented as semiring weights. The asymptotic cost of weight generation is linear in the size of the program: to generate the weights, each basic block in the program is visited once, and a weight is generated by the relevant method.

Table 2.4: Semiring operators in terms of abstract-domain operations.

Semiring Operator	Abstract-domain Operation	Description
0	$\bot$	*bottom element*
1	Id	*identity element*
$(a_1 \oplus a_2)$	$(a_1 \sqcup a_2)$	*join*
$(a_1 \oplus a_2)$	$(a_1 \nabla a_2)$	*widen at loop headers*
$(a_1 \otimes a_2)$	$(a_2 \circ a_1)$	*composition*

Tab. 2.4 provides the relationship between the semiring operators and the corresponding abstract-domain operations. 0 is the bottom element. 1 is the identity operation. The join operation $(a_1 \oplus a_2)$ is implemented as the join of the abstract transformers, except at the loop headers where it is implemented as widening operation to ensure termination. The extend operation $(a_1 \otimes a_2)$ is implemented as abstract-composition operation $(a_2 \circ a_1)$. Notice that the order of the arguments in abstract composition has changed.

Semiring EWPDS merge functions [56] are used to preserve caller-save and callee-save registers across call sites. Running a WPDS-based analysis to find the least fixed-point value for a given set of program points involves calling two operations, "post*" and "path summary", as detailed in [85]. The post* queries use the FWPDS algorithm [55].

3 ABSTRACT DOMAINS OF AFFINE RELATIONS

This chapter considers some known abstract domains for affine-relation analysis, along with several variants, and studies how they relate to each other. An affine relation is a *linear-equality* constraint over a given set of variables that hold machine integers. An abstract-domain element represents the set of states that satisfies a conjunction of affine relations. ARA finds, for each point in the program, a domain element that over-approximates the set of states that can arise at that point. ARA generalizes such analyses as linear-constant propagation [87] and induction-variable analysis.

We show that the abstract domains for ARA of Müller-Olm/Seidl (MOS) and King/Søndergaard (KS) are, in general, incomparable. However, we give sound interconversion methods. That is, we give an algorithm to convert a KS element v_{KS} to an over-approximating MOS element v_{MOS}—i.e., $\gamma(v_{KS}) \subseteq \gamma(v_{MOS})$—as well as an algorithm to convert an MOS element w_{MOS} to an over-approximating KS element w_{KS}—i.e., $\gamma(w_{MOS}) \subseteq \gamma(w_{KS})$.

We describe our own variant of the KS domain, which is inspired by—but different from, and arguably easier to use than—the version of KS developed by King and Søndergaard. Our version is presented in §3.4.

For MOS, it was not previously known how to perform $\tilde{\alpha}_{MOS}(\varphi)$ in a non-trivial fashion (i.e., other than defining $\tilde{\alpha}_{MOS}$ to be $\lambda f.\top$). In contrast, [53, Fig. 2] gave an algorithm for $\hat{\alpha}_{KS}$, which led us to examine more closely how MOS and KS are related.

Contributions This chapters contributions include the following:

- We introduce a third domain for representing affine relations, called AG, which stands for *affine generators* (§3.1.3). Whereas an element in the KS domain consists of a set of *constraints* on the values of variables, AG represents a collection of allowed values of variables

via a set of *generators*. We show that AG is the generator counterpart of KS: a KS element can be converted to an AG element, and vice versa, with no loss of precision (§3.2).

- We show that MOS and KS/AG are, in general, *incomparable* (§3.3.1). In particular, we show that KS and AG can express transformers with affine guards, which MOS cannot express.
- We give sound interconversion methods between MOS and KS/AG (§3.3.2–§3.3.4):
  - We show that an AG element v_{AG} can be converted to an over-approximating MOS element v_{MOS}—i.e., $\gamma(v_{AG}) \subseteq \gamma(v_{MOS})$.
  - We show that an MOS element w_{MOS} can be converted to an over-approximating AG element w_{AG}—i.e., $\gamma(w_{MOS}) \subseteq \gamma(w_{AG})$.
- Consequently, by means of the conversion path $\varphi \rightarrow$ KS $\rightarrow$ AG $\rightarrow$ MOS, we obtain a method for performing $\tilde{\alpha}_{MOS}(\varphi)$ (§3.3.5).

Experiments §3.5 presents an experimental study with the Intel IA32 (x86) instruction set in which the $\hat{\alpha}_{KS}$ method and two greedy, operator-by-operator reinterpretation methods—KS-reinterpretation (§2.3.1) and MOS-reinterpretation [62, §4.1.2]—are compared in terms of their performance and precision. The precision comparison is done by comparing the affine invariants obtained at branch points, as well as the affine procedure summaries obtained for procedures. For KS-reinterpretation and MOS-reinterpretation, we also compare the abstract transformers generated for individual x86 instructions. The experiments were designed to answer the following questions:

- Which method of obtaining abstract transformers is fastest: $\hat{\alpha}_{KS}$, KS-reinterpretation, or MOS-reinterpretation?
- Does MOS-reinterpretation or KS-reinterpretation yield more precise abstract transformers for machine instructions?
- For what percentage of branch points and procedures does $\hat{\alpha}_{KS}$ pro-

duce more precise answers than KS-reinterpretation?

Organization The chapter is organized as follows: §3.1 summarizes relevant features of the various ARA domains. §3.2 presents the AG domain, and shows how an AG element can be converted to a KS element, and vice versa. §3.3 presents our results on the incomparability of the MOS and KS domains, but gives sound methods to convert a KS element into an over-approximating MOS element, and vice versa. §3.4 explains how to use the KS domain for interprocedural analysis. §3.5 presents experimental results. §3.6 discusses related work. Proofs can be found in the appendices.

3.1 Abstract Domains for affine-relation analysis

The two existing ARA domains—one defined by [76, 78] (MOS) and one defined by [52, 53] (KS) are introduced in this section. Both MOS and KS are based on an extension of linear algebra to modules over a ring [44, 43, 1, 99, 76, 78]. §3.1.1 introduces terminology used in describing the ARA domains. §3.1.2 introduces Howell form, which can be used as a normal form in the MOS and KS domains. §3.1.3 defines the affine-generator (AG) domain, which is a generator representation of the KS domain. §3.1.4 and §3.1.5 define the KS domain and MOS domain, respectively. Finally, we discuss the domain heights of AG, KS, and MOS domains in §3.1.6.

3.1.1 Terminology

All numeric values are integers in $\mathbb{Z}_{2^w}$ for some bit width w. That is, values are w-bit machine integers with the standard operations for machine addition and multiplication. Addition and multiplication in $\mathbb{Z}_{2^w}$ form a

ring, not a field, so some facets of standard linear algebra do not apply, and thus we must be cautious about carrying over intuition from standard linear algebra. In particular, each odd element in $\mathbb{Z}_{2^w}$ has a multiplicative inverse (which may be found in time $O(\log w)$ [102, Fig. 10-5]), but no even element has a multiplicative inverse. The *rank* of a value $x \in \mathbb{Z}_{2^w}$ is the maximum integer $p \leqslant w$ such that 2^p divides evenly into x [76, 78]. For example, $\text{rank}(1) = 0$, $\text{rank}(24) = 3$, and $\text{rank}(0) = w$.

Throughout the paper, k is the size of the *vocabulary*, the variable-set under analysis. A *two-vocabulary* relation is a transition relation between values of variables in the *pre-state* vocabulary and values of variables in the *post-state* vocabulary.

Matrix addition and multiplication are defined as usual, forming a matrix ring. We denote the transpose of a matrix M by M^t. A *one-vocabulary matrix* is a matrix with $k+1$ columns. A *two-vocabulary matrix* is a matrix with $2k+1$ columns. In each case, the "$+1$" is for technical reasons (which vary according to what kind of matrix we are dealing with). I denotes the (square) identity matrix (whose size can be inferred from context). The rows of a matrix M are numbered from 1 to $\texttt{rows}(M)$; the columns of M are numbered starting from 1.

Because the MOS domain inherently involves pre-state-vocabulary to post-state-vocabulary transformers (see §3.1.5), our definitions of the AG and KS domains (§3.1.3 and §3.1.4, respectively) are also two-vocabulary domains. Technically, AG and KS can have an arbitrary number of vocabularies, including just a single vocabulary. To be able to give simpler examples, some of the AG and KS examples use one-vocabulary domain elements.

States in the various abstract domains are represented by row vectors of length $k+1$. The *row space* of a matrix M is defined by $\text{row}\, M \stackrel{\text{def}}{=} \{\vec{v} \mid \exists w\colon wM = \vec{v}\}$. When we speak of the "null space" of a matrix, we actually mean the set of row vectors whose transposes are in the traditional

null space of the matrix. Thus, we define $\text{null}^{\text{t}}\, M \stackrel{\text{def}}{=} \left\{ \vec{v} \;\middle|\; M\vec{v}^{\text{t}} = 0 \right\}$.

3.1.2 Matrices in Howell Form

One way to appreciate how linear algebra in rings differs from linear algebra in fields is to see how certain issues are finessed when converting a matrix to *Howell form* [44]. The Howell form of a matrix is an extension of reduced row-echelon form [67] suitable for matrices over $\mathbb{Z}_n$. Because Howell form is *canonical* for matrices over principal ideal rings [44, 99], it provides a way to test whether two abstract-domain elements are equal—i.e., whether they represent the same set of concrete values. Such an equality test is needed during program analysis to determine whether a fixed point has been reached.

Definition 3.1. *The leftmost nonzero value in a row vector is its* **leading value**. *The leading value's index is the* **leading index**. *A matrix* M *is in* **row-echelon form** *if*

- *All all-zero rows are at the bottom.*
- *Each row's leading index is greater than that of the row above it.*

If M *is in row-echelon form, let* $[M]_i$ *denote the matrix that consists of all rows of* M *whose leading index is* i *or greater.*

A matrix M *is in* **Howell form** *if*

1. M *is in row-echelon form and has no all-zero rows,*
2. *the leading value of every row is a power of two,*
3. *each leading value is the largest value in its column, and*
4. *for every row* r *of* M*, for any* $p \in \mathbb{Z}$*, if* i *is the leading index of* $2^p r$*, then* $2^p r \in \text{row}([M]_i)$.

□

In Defn. 3.1, item (4) may be confusing, and thus warrants an example.

Example 3.2. *Suppose that* $w = 4$*, so that we are working in* $\mathbb{Z}_{16}$*. Consider the following two matrices and their Howellizations:*

$$M_1 \stackrel{\text{def}}{=} \begin{bmatrix} 4 & 2 & 4 \end{bmatrix} \qquad \textsc{Howellize}(M_1) = \begin{bmatrix} 4 & 2 & 4 \\ 0 & 8 & 0 \end{bmatrix}$$

$$M_2 \stackrel{\text{def}}{=} \begin{bmatrix} 4 & 2 & 4 \\ 0 & 4 & 0 \end{bmatrix} \qquad \textsc{Howellize}(M_2) = \begin{bmatrix} 4 & 2 & 4 \\ 0 & 4 & 0 \end{bmatrix}$$

First, notice that M_1 *does not satisfy item (4).* M_1 *has only one row,* $[4\ 2\ 4]$*, and consider what happens when this row is multiplied by powers of* 2*:*

$$\begin{aligned} 2^1 \cdot [4\ 2\ 4] &= [8\ 4\ 8] \\ 2^2 \cdot [4\ 2\ 4] &= [0\ 8\ 0] \\ 2^3 \cdot [4\ 2\ 4] &= [0\ 0\ 0] \end{aligned}$$

In particular, the leading index of $2^2 \cdot [4\ 2\ 4] = [0\ 8\ 0]$ *is 2; however, because* $\text{row}([M]_2) = \varnothing$*,* $[0\ 8\ 0] \notin \text{row}([M]_2)$*. Consequently,* $[0\ 8\ 0]$ *must be included in* $\textsc{Howellize}(M_1)$*. We say that a row like* $[0\ 8\ 0]$ *is a* logical consequence of $[4\ 2\ 4]$ that is added to satisfy item (4) of Defn. 3.1.

In contrast, matrix M_2 *satisfies item (4) (and, in fact,* M_2 *is already in Howell form). For matrix* M_2 *to fail to satisfy item (4), there would have to be some row* r *and power* p *for which (a) the leading index* i *of* $2^p r$ *is strictly greater than the leading index of* r*, (b)* $2^p r \neq 0$*, and (c)* $2^p r \notin \text{row}([M]_i)$*. In this example, the only interesting quantity of the form* $2^p r$ *is* $2^2 \cdot [4\ 2\ 4] = [0\ 8\ 0]$*. The leading index of* $[0\ 8\ 0]$ *is* 2*, but* $[0\ 8\ 0] = 2 \cdot [0\ 4\ 0]$*, and so* $[0\ 8\ 0] \in \text{row}([M]_2)$*. Consequently,* M_2 *satisfies item (4).* □

The Howell form of a matrix is unique among all matrices with the same row space (or null space) [44]. As mentioned earlier, this property of Howell form provides a way to test two MOS elements, two KS elements, or two AG elements for equality.

Algorithm 3 HOWELLIZE: Put the matrix G in Howell form.

```
 1: procedure HOWELLIZE(G)
 2:     Let j = 0                          ▷ j is the number of already-Howellized rows
 3:     for all i from 1 to 2k + 1 do
 4:         Let R = {all rows of G with leading index i}
 5:         if R ≠ ∅ then
 6:             Pick an r ∈ R that minimizes rank r_i
 7:             Pick the odd u and rank p so that u2^p = r_i
 8:             r ← u^{-1} r                          ▷ Adjust r, leaving r_i = 2^p
 9:             for all s in R \ {r} do
10:                 Pick the odd v and rank t so that v2^t = s_i
11:                 s ← s − (v2^{t−p}) r                          ▷ Zero out s_i
12:                 if row s contains only zeros then
13:                     Remove s from G
14:             In G, swap r with G_{j+1}          ▷ Place r for row-echelon form
15:             for all h from 1 to j do ▷ Set values above r_i to be 0 ⩽ · < r_i
16:                 d ← G_{h,i} ≫ p          ▷ Pick d so that 0 ⩽ G_{h,i} − dr_i < r_i
17:                 G_h ← G_h − dr  ▷ Adjust row G_h, leaving 0 ⩽ G_{h,i} < r_i
18:             if r_i ≠ 1 then          ▷ Add logical consequences of r to G
19:                 Add 2^{w−p} r as last row of G       ▷ New row has leading
    index > i
20:             j ← j + 1
```

The notion of a *saturated set of generators* used by [78] is closely related to Howell form, but is defined for an unordered set of matrices rather than row-vectors arranged in a matrix, and has no analogue of item (3). The algorithms of Müller-Olm and Seidl do not compute multiplicative inverses (see §3.6.2), so a saturated set has no analogue of item (2). Consequently, a saturated set is not canonical among generators of the same space.

Our technique for putting a matrix in Howell form is the procedure HOWELLIZE (Alg. 3). Much of HOWELLIZE is similar to a standard Gaussian-elimination algorithm, and it has the same overall cubic-time complexity as Gaussian elimination. In particular, HOWELLIZE minus lines 15–19 puts G in row-echelon form (item (1) of Defn. 3.1) with the leading value of every row

a power of two. (Line 8 enforces item (2) of Defn. 3.1.) Howellize differs from standard Gaussian elimination in how the pivot is picked (line 6) and in how the pivot is used to zero out other elements in its column (lines 7–13). Lines 15–17 of Howellize enforce item (3) of Defn. 3.1, and lines 18–19 enforce item (4). Lines 12–13 remove all-zero rows, which is needed for Howell form to be canonical.

Alg. 3 is simple and easy to implement. For analyses over large vocabularies, one should replace Alg. 3, which has cubic-time complexity with, say, the algorithm of [99], which has the same asymptotic complexity as matrix multiplication.

3.1.3 The Affine Generator Domain

An element in the Affine Generator domain (AG) is a two-vocabulary matrix whose rows are the affine generators of a two-vocabulary relation.

An AG element is an r-by-$(2k+1)$ matrix G, with $0 < r \leqslant 2k+1$. The concretization of an AG element is

$$\gamma_{\text{AG}}(G) \stackrel{\text{def}}{=} \left\{ (\vec{v}, \vec{v}') \mid \vec{v}, \vec{v}' \in \mathbb{Z}_{2^w}^k \wedge \left[1 \middle| \vec{v}\ \vec{v}'\right] \in \text{row } G \right\}.$$

The AG domain captures all two-vocabulary affine spaces, and treats them as relations between pre-states and post-states.

The bottom element of the AG domain is the empty matrix, and the AG element that represents the identity relation is the matrix $\left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & I & I \end{array}\right]$ (with columns 1, $\vec{v}$, $\vec{v}'$). The AG element

$$\left[\begin{array}{c|cccc} 1 & v_1 & v_2 & v_1' & v_2' \\ \hline 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{array}\right] \tag{3.1}$$

represents the transition relation in which $v_1' = v_1$, v_2 can have any value, and v_2' can have any even value.

To compute the join of two AG elements, stack the two matrices vertically and Howellize the result.

3.1.4 The King/Søndergaard Domain

An element in the King/Søndergaard domain (KS) is a two-vocabulary matrix whose rows represent constraints on a two-vocabulary relation. A KS element is an r-by-$(2k+1)$ matrix M, with $0 \leqslant r \leqslant 2k+1$. The concretization of a KS element M is

$$\gamma_{KS}(M) \stackrel{\text{def}}{=} \left\{ (\vec{v}, \vec{v}') \mid \vec{v}, \vec{v}' \in \mathbb{Z}_{2^w}^k \wedge \left[\vec{v} \ \vec{v}' | 1 \right] \in \text{null}^t G \right\}.$$

Like the AG domain, the KS domain captures all two-vocabulary affine spaces, and treats them as relations between pre-states and post-states.

It is easy to read out affine equalities from a KS element M (regardless of whether M is in Howell form): if

$$\begin{array}{c} \begin{array}{cccccc} v_1 & \ldots & v_k & v_1' & \ldots & v_k' \end{array} \quad 1 \\ \left[\begin{array}{cccccc|c} a_1 & \ldots & a_k & a_1' & \ldots & a_k' & b \end{array}\right] \end{array}$$

is a row of M, then $\sum_i a_i v_i + \sum_i a_i' v_i' = -b$ is a constraint on $\gamma_{KS}(M)$. The conjunction of these constraints describes $\gamma_{KS}(M)$ exactly.

The bottom element of the KS domain is the matrix $\begin{array}{c} \begin{array}{cc} \vec{v} & \vec{v}' \end{array} \quad 1 \\ \left[\begin{array}{cc|c} 0 & 0 & 1 \end{array}\right] \end{array}$, and the KS element that represents the identity relation is the matrix $\begin{array}{c} \begin{array}{cc} \vec{v} & \vec{v}' \end{array} \quad 1 \\ \left[\begin{array}{cc|c} I & -I & 0 \end{array}\right] \end{array}$. Suppose that $w = 4$, so that we are working in $\mathbb{Z}_{16}$. The

KS element

$$\left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ \hline 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 \end{array}\right] \tag{3.2}$$

represents the transition relation in which $\nu_1' = \nu_1$, ν_2 can have any value, and ν_2' can have any even value. Thus, Eqns. (3.1) and (3.2) represent the same transition relation in AG and KS, respectively.

A Howell-form KS element can easily be checked for emptiness: it is empty if and only if it contains a row whose leading entry is in its last column. In that sense, an implementation of the KS domain in which all elements are kept in Howell form has redundant representations of bottom (whose concretization is $\varnothing$). However, such KS elements can always be detected during Howellize and replaced by the canonical representation of bottom, namely, $\left[\begin{array}{cc|c} \overrightarrow{v} & \overrightarrow{v}' & 1 \\ \hline 0 & 0 & 1 \end{array}\right]$.

The original King and Søndergaard paper 2008 gives polynomial-time algorithms for join and projection; projection can be used to implement composition (see §3.4.3).

3.1.5 The Müller-Olm/Seidl Domain

An element in the Müller-Olm/Seidl domain (MOS) is an affine-closed set of affine transformers, as detailed in [78]. An MOS element is represented by a set of $(k+1)$-by-$(k+1)$ matrices. Each matrix T is a one-vocabulary transformer of the form $T = \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right]$, which represents the state transformation $\overrightarrow{v}' := \overrightarrow{v} \cdot M + b$, or, equivalently, $\left[1|\overrightarrow{v}'\right] := [1|\overrightarrow{v}]\, T$.

An MOS element B consists of a set of $(k+1)$-by-$(k+1)$ matrices, and represents the affine span of the set, denoted by $\langle B \rangle$ and defined as follows:

$$\langle B \rangle \stackrel{\text{def}}{=} \left\{ T \;\middle|\; \exists w \in \mathbb{Z}_{2^w}^{|B|} \colon T = \sum_{B \in B} w_B B \wedge T_{1,1} = 1 \right\}.$$

The meaning of B is the union of the graphs of the affine transformers in $\langle B \rangle$

$$\gamma_{\text{MOS}}(B) \stackrel{\text{def}}{=} \left\{ (\vec{v}, \vec{v}') \;\middle|\; \vec{v}, \vec{v}' \in \mathbb{Z}_{2^w}^k \wedge \exists T \in \langle B \rangle : [1|\vec{v}]\, T = \left[1|\vec{v}'\right] \right\}.$$

The bottom element of the MOS domain is $\varnothing$, and the MOS element that represents the identity relation is the singleton set $\{I\}$. If $w = 4$, the MOS element $B = \left\{ \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 0 & 0 & 2 \\ \hline 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right] \right\}$ represents the affine span

$$\langle B \rangle = \left\{ \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 2 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 4 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right], \ldots, \left[\begin{array}{c|cc} 1 & 0 & 14 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right] \right\},$$

which corresponds to the transition relation in which $v_1' = v_1$, v_2 can have any value, and v_2' can have any even value—i.e., B represents the same transition relation as Eqns. (3.1) and (3.2).

The operations join and compose can be performed in polynomial time. If B and C are MOS elements, $B \sqcup C = \textsc{Howellize}(B \cup C)$ and $B\,;C = \textsc{Howellize}\{bc \mid b \in B \wedge c \in C\}$. In this setting, Howellize of a set of $(k+1)$-by-$(k+1)$ matrices $\{M_1, \ldots, M_n\}$ means "Apply Alg. 3 to a larger, n-by-$(k+1)^2$ matrix, each of whose rows is the linearization (e.g., in row-major order) of one of the M_i."

3.1.6 Domain Heights

In all three domains, an element can be represented via an appropriate matrix in Howell form (where in the case of the MOS domain, we mean a matrix in the extended sense discussed in §3.1.5). For a fixed bit width and a fixed number of columns, there are only a constant number of Howell-form matrices. Consequently, the KS, AG, and MOS domains are all finite domains, and hence of finite height.

Domain elements need not necessarily be maintained in Howell form; instead, they could be Howellized on demand when it is necessary to check containment (see §3.4.6). Our implementation maintains domain elements in Howell form using essentially the "list of lists" sparse-matrix representation: each matrix is represented via a C++ vector of rows; each row is a vector of (column-index, nonzero-value) pairs.

3.2 Relating AG and KS Elements

AG and KS are equivalent domains. One can convert an AG element to an equivalent KS element with no loss of precision, and vice versa. In essence, these are a single abstract domain with two representations: constraint form (KS) and generator form (AG).

We use an operation similar to singular value decomposition, called diagonal decomposition:

Definition 3.3. *The* **diagonal decomposition** *of a square matrix* M *is a triple of matrices,* L, D, R, *such that* $M = LDR$; L *and* R *are invertible matrices; and* D *is a diagonal matrix in which all entries are either 0 or a power of 2.* □

[78, Lemma 2.9] give a decomposition algorithm that nearly performs diagonal decomposition, except that the entries in their D might not be powers of 2. We can easily adapt that algorithm. Suppose that their method yields LDR (where L and R are invertible). Pick u and r so that $u_i 2^{r_i} = D_{i,i}$ with each u_i odd, and define U as the diagonal matrix where $U_{i,i} = u_i$. (If $D_{i,i} = 0$, then $u_i = 1$.) It is easy to show that U is invertible. Let $L' = LU$ and $D' = U^{-1}D$. Consequently, $L'D'R = LDR = M$, and $L'D'R$ is a diagonal decomposition.

From diagonal decomposition we derive the dualization operation, denoted by $\cdot^{\perp}$, such that the rows of $M^{\perp}$ generate the null space of M, and vice versa.

Definition 3.4. *The* **dualization** *of* M*, denoted by* $M^{\perp}$*, is defined as follows:*

- $\textsc{Pad}(M)$ *is the* $(2k+1)$*-by-*$(2k+1)$ *matrix* $\left[\begin{array}{c} M \\ \hline \overrightarrow{0} \end{array}\right]$,
- L, D, R *is the diagonal decomposition of* $\textsc{Pad}(M)$,
- T *is the diagonal matrix with* $T_{i,i} \stackrel{def}{=} 2^{w-\operatorname{rank}(D_{i,i})}$*, and*
- $M^{\perp} \stackrel{def}{=} \left(L^{-1}\right)^{t} T \left(R^{-1}\right)^{t}$

□

This definition of dualization has the following useful property:

Theorem 3.5. *For any matrix* M*,* $\operatorname{null}^{t} M = \operatorname{row} M^{\perp}$ *and* $\operatorname{row} M = \operatorname{null}^{t} M^{\perp}$.

Proof. For any matrix M, it is a common lemma that $\left(M^{-1}\right)^{t} = (M^{t})^{-1}$. Thus, the notation M^{-t} denotes $\left(M^{-1}\right)^{t}$.

Lemma 3.6. *Let* D *and* T *be square, diagonal matrices, where* $D_{ii} = 2^{p_i}$ *and* $T_{ii} = 2^{w-p_i}$ *for all* i*. Then,* $\operatorname{null}^{t} T = \operatorname{row} D$ *and* $\operatorname{null}^{t} D = \operatorname{row} T$.

Proof. Let z be any row vector. To see that $\operatorname{null}^{t} T = \operatorname{row} D$:

$$\begin{aligned} z \in \operatorname{null}^{t} T &\Leftrightarrow Tz^{t} = 0 \Leftrightarrow \forall i\colon z_i 2^{w-p_i} = 0 \\ &\Leftrightarrow \forall i\colon 2^{p_i} | z_i \Leftrightarrow \exists v\colon \forall i\colon v_i 2^{p_i} = z_i \\ &\Leftrightarrow \exists v\colon vD = z \Leftrightarrow z \in \operatorname{row} D. \end{aligned}$$

One can show that $\operatorname{null}^{t} D = \operatorname{row} T$ by essentially the same reasoning. □

Again, let L, D, and R be the diagonal decomposition of M (see Defn. 3.3, and construct T from D as in Lem. 3.6. Recall that L is invertible. To see that $\operatorname{row} M = \operatorname{null}^{t} M^{\perp}$,

$$\begin{aligned} \operatorname{row} M = \operatorname{row} LDR = \operatorname{row} DR,\ \text{so } x \in \operatorname{row} DR &\Leftrightarrow xR^{-1} \in \operatorname{row} D \\ \Leftrightarrow xR^{-1} \in \operatorname{null}^{t} T &\Leftrightarrow TR^{-t}x^{t} = 0 \Leftrightarrow x \in \operatorname{null}^{t} TR^{-t}. \end{aligned}$$

We know that L^{-t} is also invertible, so

$$\text{null}^t\, TR^{-t} = \text{null}^t\, L^{-t}TR^{-t} = \text{null}^t\, M^{\perp}.$$

Thus, row $M = \text{null}^t\, M^{\perp}$. One can show that $\text{null}^t\, M = \text{row}\, M^{\perp}$ by essentially the same reasoning. □

We can therefore use dualization to convert between equivalent KS and AG elements. For a given (padded, square) AG matrix $G = [c|Y\ Y']$, we seek a KS matrix Z of the form $[X\ X'|b]$ such that $\gamma_{KS}(Z) = \gamma_{AG}(G)$. We construct Z by letting $[b|X\ X'] = G^{\perp}$ and permuting those columns to $Z \stackrel{\text{def}}{=} [X\ X'|b]$. This works by Thm. 3.5, and because

$$\begin{aligned}
\gamma_{AG}(G) &= \{(x, x') \mid [1|x\ x'] \in \text{row}\, G\} \\
&= \left\{(x, x') \,\middle|\, [1|x\ x'] \in \text{null}^t\, G^{\perp}\right\} \\
&= \left\{(x, x') \,\middle|\, [x\ x'|1] \in \text{null}^t\, Z\right\} = \gamma_{KS}(Z)\,.
\end{aligned}$$

Furthermore, to convert from any KS matrix to an equivalent AG matrix, we reverse the process. Reversal is possible because dualization is an involution: for any matrix M, $\left(M^{\perp}\right)^{\perp} = M$.

3.3 Relating KS and MOS

3.3.1 MOS and KS are Incomparable

The MOS and KS domains are incomparable: some relations are expressible in each domain that are not expressible in the other. Intuitively, the central difference is that MOS is a domain of sets of *functions*, while KS is a domain of *relations*.

KS can capture restrictions on both the pre-state and post-state vocabularies, while MOS can capture restrictions only on its post-state vocab-

ulary. For example, when $k = 1$, the KS element for "assume $v = 2$" (in un-Howellized form) is $\left[\begin{array}{cc|c} v & v' & 1 \\ 1 & 0 & -2 \\ 1 & -1 & 0 \end{array}\right]$, i.e., "$v = 2 \wedge v = v'$".

In contrast, an MOS element cannot encode an assume statement. The smallest MOS element that over-approximates "assume $v = 2$" is the identity transformer $\left\{\left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 1 \end{array}\right]\right\}$. In general, an MOS element cannot encode a non-trivial condition on the pre-state. If an MOS element contains a single transition, it encodes that transition for every possible pre-state. Therefore, KS can encode relations that MOS cannot encode.

On the other hand, an MOS element can encode two-vocabulary relations that are not affine. One example is the matrix basis $\mathcal{B} = \left\{\left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 1 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \end{array}\right]\right\}$. The set that $\mathcal{B}$ encodes is

$$\begin{aligned}
\gamma_{\text{MOS}}(\mathcal{B}) &= \left\{ \begin{bmatrix} v_1 & v_2 & v_1' & v_2' \end{bmatrix} \;\middle|\; \begin{array}{c} \exists w_0, w_1 : \left[\begin{array}{c|cc} 1 & v_1 & v_2 \end{array}\right] \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & w_0 & w_0 \\ 0 & w_1 & w_1 \end{array}\right] = \left[\begin{array}{c|cc} 1 & v_1' & v_2' \end{array}\right] \\ \wedge\ w_0 + w_1 = 1 \end{array} \right\} \\
&= \left\{ \begin{bmatrix} v_1 & v_2 & v_1' & v_2' \end{bmatrix} \;\middle|\; \exists w_0 : v_1' = v_2' = w_0 v_1 + (1 - w_0) v_2 \right\} \\
&= \left\{ \begin{bmatrix} v_1 & v_2 & v_1' & v_2' \end{bmatrix} \;\middle|\; \exists w_0 : v_1' = v_2' = v_1 + (1 - w_0)(v_2 - v_1) \right\} \\
&= \left\{ \begin{bmatrix} v_1 & v_2 & v_1' & v_2' \end{bmatrix} \;\middle|\; \exists p : v_1' = v_2' = v_1 + p(v_2 - v_1) \right\} \qquad (3.3)
\end{aligned}$$

Affine spaces are closed under affine combinations of their elements. Thus, $\gamma_{\text{MOS}}(\mathcal{B})$ is not an affine space because some affine combinations of its elements are not in $\gamma_{\text{MOS}}(\mathcal{B})$. For instance, let $a = \begin{bmatrix} 1 & -1 & 1 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 2 & -2 & 6 & 6 \end{bmatrix}$, and $c = \begin{bmatrix} 0 & 0 & -4 & -4 \end{bmatrix}$. By Eqn. (3.3), we have $a \in \gamma_{\text{MOS}}(\mathcal{B})$ when $p = 0$ in Eqn. (3.3), $b \in \gamma_{\text{MOS}}(\mathcal{B})$ when $p = -1$, and $c \notin \gamma_{\text{MOS}}(\mathcal{B})$ (the equation "$-4 = 0 + p(0 - 0)$" has no solution for p). Moreover, $2a - b = c$, so c is an affine combination of a and b. Thus, $\gamma_{\text{MOS}}(\mathcal{B})$ is not closed under affine combinations of its elements, and so $\gamma_{\text{MOS}}(\mathcal{B})$ is not an affine space. Because every KS element encodes a two-

vocabulary affine space, MOS can represent $\gamma_{\text{MOS}}(\mathcal{B})$ but KS cannot.

3.3.2 Converting MOS Elements to KS

Soundly converting an MOS element $\mathcal{B}$ to a KS element is equivalent to stating two-vocabulary affine constraints satisfied by $\mathcal{B}$.

To convert an MOS element $\mathcal{B}$ to a KS element, we

1. rewrite $\mathcal{B}$ so that every matrix it contains has a 1 in its top-left corner,
2. build a two-vocabulary AG matrix from each one-vocabulary matrix in $\mathcal{B}$,
3. join the resulting AG matrices, and
4. convert the joined AG matrix to a KS element.

For Step (1), we rewrite $\mathcal{B}$ so that

$$\mathcal{B} = \left\{ \left[\begin{array}{c|c} 1 & c_i \\ \hline 0 & N_i \end{array}\right] \right\}, \text{ where } c_i \in \mathbb{Z}_{2^w}^{1\times k} \text{ and } N_i \in \mathbb{Z}_{2^w}^{k\times k}.$$

If our original MOS element $\mathcal{B}_0$ fails to satisfy this property, we can construct an equivalent $\mathcal{B}$ that does. Let $\mathcal{C} = \textsc{Howellize}(\mathcal{B}_0)$; pick the unique $B \in \mathcal{C}$ such that $B_{1,1} = 1$, and let $\mathcal{B} = \{B\} \cup \{B + C \mid C \in (\mathcal{C} \setminus \{B\})\}$. $\mathcal{B}$ now satisfies the property, and $\langle \mathcal{B} \rangle = \langle \mathcal{B}_0 \rangle$.

In Step (2), we construct the matrices

$$G_i = \left[\begin{array}{c|cc} 1 & 0 & c_i \\ \hline 0 & I & N_i \end{array}\right].$$

Note that, for each matrix $B_i \in \mathcal{B}$, $\gamma_{\text{MOS}}(\{B_i\}) = \gamma_{\text{AG}}(G_i)$. In Step (3), we join the G_i matrices in the AG domain to yield one matrix G. Thm. 3.7 states the soundness of this transformation from MOS to AG, i.e., $\gamma_{\text{MOS}}(\mathcal{B}) \subseteq \gamma_{\text{AG}}(G)$. Finally, G is converted in Step (4) to an equivalent KS element by the method given in §3.2.

Thm. 3.7 states that the transformation from MOS to AG is sound.

Theorem 3.7. *Suppose that $\mathcal{B}$ is an MOS element such that, for every $B \in \mathcal{B}$, $B = \left[\begin{array}{c|c} 1 & c_B \\ \hline 0 & M_B \end{array}\right]$ for some $c_B \in \mathbb{Z}_{2^w}^{1\times k}$ and $M_B \in \mathbb{Z}_{2^w}^{k\times k}$. Define $G_B = \left[\begin{array}{c|cc} 1 & 0 & c_B \\ \hline 0 & I & M_B \end{array}\right]$ and $G = \bigsqcup_{AG} \{G_B \mid B \in \mathcal{B}\}$. Then, $\gamma_{MOS}(\mathcal{B}) \subseteq \gamma_{AG}(G)$.*

Proof. See App. A. □

Because we can easily read affine relations from KS elements (§3.1.4), this conversion method also gives an easy way to create a quantifier-free formula that over-approximates the meaning of an MOS element. In particular, the formula read out of the KS element obtained from MOS-to-KS conversion captures affine relations implied by the MOS element.

3.3.3 Converting KS Without Pre-State Guards to MOS

If a KS element is total with respect to pre-state inputs, then we can convert it to an equivalent MOS element. First, convert the KS element to an AG element G. When G expresses no restrictions on its pre-state, it has the form

$$G = \left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \\ 0 & 0 & R \end{array}\right], \tag{3.4}$$

where $b \in \mathbb{Z}_{2^w}^{1\times k}$; $I, M \in \mathbb{Z}_{2^w}^{k\times k}$; and $R \in \mathbb{Z}_{2^w}^{k\times r}$ with $0 \leqslant r \leqslant k$.

Definition 3.8. *An AG matrix of the form*

$$\left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \end{array}\right],$$

such as the G_i matrices discussed in §3.3.2, is said to be in **explicit form**. *An AG matrix in this form represents the transition relation $x' = x \cdot M + b$.* □

Explicit form is desirable because we can immediately convert the AG matrix of Defn. 3.8 into the MOS element

$$\left\{ \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] \right\}.$$

The matrix G in Eqn. (3.4) is not in explicit form because of the rows $[0|0\ R]$; however, G is quite close to being in explicit form, and we can read off a *set* of matrices to create an appropriate MOS element. We produce this set of matrices via the SHATTER operation, where

$$\textsc{Shatter}(G) \stackrel{\text{def}}{=} \left\{ \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] \right\} \cup \left\{ \left[\begin{array}{c|c} 0 & R_{j,*} \\ \hline 0 & 0 \end{array}\right] \middle| \, 1 \leqslant j \leqslant r \right\},$$

where $R_{j,*}$ is row j of R.

As shown in Thm. 3.9, $\gamma_{\text{AG}}(G) = \gamma_{\text{MOS}}(\textsc{Shatter}(G))$. Intuitively, this property holds because the coefficients of the $[0|0\ R_{j,*}]$ rows in an affine combination of the rows of G correspond to coefficients of the $\left\{ \left[\begin{array}{c|c} 0 & R_{j,*} \\ \hline 0 & 0 \end{array}\right] \right\}$ matrices in an affine combination of the matrices in $\textsc{Shatter}(G)$.

Theorem 3.9. *When* $G = \left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \\ 0 & 0 & R \end{array}\right]$, *then* $\gamma_{AG}(G) = \gamma_{MOS}(\textsc{Shatter}(G))$.

Proof. See App. A. □

3.3.4 Converting KS With Pre-State Guards to MOS

If a KS element is not total with respect to pre-state inputs, then there is no MOS element with the same concretization. However, we can find sound over-approximations within MOS for such KS elements.

We convert the KS element into an AG matrix G as in §3.3.3 and put G in Howell form. There are two ways that G can enforce guards on the

Algorithm 4 MakeExplicit: Transform an AG matrix G in Howell form to near-explicit form.

Require: G is an AG matrix in Howell form

```
1:  procedure MakeExplicit(G)
2:      for all i from 2 to k + 1 do    ▷ Consider each col. of the pre-state voc.
3:          if there is a row r of G with leading index i then
4:              if rank r_i > 0 then
5:                  for all j from 1 to 2k + 1 do ▷ Build s from r, with s_i = 1
6:                      s_j ← r_j ≫ rank r_i
7:                  Append s to G
8:                  G ← Howellize(G)
9:      for all i from 2 to k + 1 do
10:         if there is no row r of G with leading index i then
11:             Insert, as the i^th row of G, a new row of all zeroes
```

pre-state vocabulary: it might contain one or more rows whose leading value is even, or it might skip some leading indexes in row-echelon form.

While we cannot put G in explicit form, we can run MakeExplicit to coarsen G so that it is close enough to the form that arose in §3.3.3. Adding extra rows to an AG element can only enlarge its concretization. Thus, to handle a leading value $2^p, p > 0$ in the pre-state vocabulary, MakeExplicit introduces an extra, over-approximating row constructed by copying the row with leading value 2^p and right-shifting each value in the copied row by p bits (lines 4–8). After the loop on lines 2–8 finishes, every leading value in a row that generates pre-state-vocabulary values is 1. MakeExplicit then introduces all-zero rows so that each leading element from the pre-state vocabulary lies on the diagonal (lines 9–11).

Example 3.10. *Suppose that* $k = 3$, $w = 4$, *and* $G = \left[\begin{array}{c|cccccc} 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ \hline & 4 & 0 & 12 & 2 & 4 & 0 \\ & & & & 4 & 0 & 8 \end{array}\right]$. *After line 11 of* MakeExplicit, *all pre-state vocabulary leading values of* G *have been made ones, and the resulting* G′ *has* row G′ ⊇ row G. *In our case,* G′ =

$\left[\begin{array}{c|cccccc} 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ \hline & 1 & 0 & 3 & 0 & 1 & 0 \\ & & & & 2 & 0 & 0 \\ & & & & & & 8 \end{array}\right]$. *To handle "skipped" indexes, lines 9–11 insert all-zero rows into* G' *so that each leading element from the pre-state vocabulary lies on the diagonal. The resulting matrix is* $\left[\begin{array}{c|cccccc} 1 & 0 & 2 & 0 & 0 & 0 & 0 \\ \hline & 1 & 0 & 3 & 0 & 1 & 0 \\ & & 0 & 0 & 0 & 0 & 0 \\ & & & 0 & 0 & 0 & 0 \\ & & & & 2 & 0 & 0 \\ & & & & & & 8 \end{array}\right]$. □

Theorem 3.11. *For* $\mathsf{G} \in AG$, $\gamma_{AG}(\mathsf{G}) \subseteq \gamma_{MOS}(\textsc{Shatter}(\textsc{MakeExplicit}(\mathsf{G})))$.

Proof. See App. A. □

Thus, we can use the KS–to–AG conversion method of §3.2, MakeExplicit, and Shatter to obtain an over-approximation of a KS element in MOS.

Example 3.12. *The final MOS element for Ex. 3.10 is*

$$\left\{ \left[\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 2 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 0 & 0 & 8 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \right\}.$$

□

3.3.5 Symbolic Abstraction for the MOS Domain

As mentioned in the introduction, it was not previously known how to perform symbolic abstraction for MOS. Using $\hat{\alpha}_{\mathrm{KS}}$ (see §2.3.2 and §3.4.8) in conjunction with the algorithms from §3.2 and §3.3.4, we can soundly define $\tilde{\alpha}_{\mathrm{MOS}}(\varphi)$ as

$$\textbf{let } \mathsf{G} = \textsc{ConvertKStoAG}(\hat{\alpha}_{\mathrm{KS}}(\varphi)) \textbf{ in } \textsc{Shatter}(\textsc{MakeExplicit}(\mathsf{G})).$$

3.4 Using KS for Interprocedural Analysis

This section presents a two-vocabulary version of the KS abstract domain, focusing on the operations that are useful in a program analyzer. Unlike previous work by [52, 53], it is not necessary to perform bit-blasting to use the version of KS presented here. §3.4.1–§3.4.7 describe the suite of operations needed to use the KS domain in an interprocedural-analysis algorithm in the style of [92] or [54], or to use the KS domain as a weight domain in a weighted pushdown system (WPDS) [10, 85, 56, 50].

§3.4.8 discusses symbolic abstraction for the KS domain, which provides one way to create two-vocabulary KS elements that represent abstract transformers needed by a program analyzer. (§2.3.1 provides an alternative method for creating KS abstract transformers, based on the operator-reinterpretation paradigm.)

§3.4.9 shows how to compute the number of tuples that satisfy a KS element.

3.4.1 Meet

As discussed in §3.1.4, the meaning of a KS matrix X can be expressed as a formula by forming a conjunction that consists of one equality for each row of X. We can obtain a KS element that precisely represents the conjunction of any number of such formulas by stacking the rows that represent the equalities, and putting the resulting matrix in Howell form. Consequently, we can compute the meet $X \sqcap Y$ of any two KS elements X and Y by putting the block matrix $\left[\begin{smallmatrix} X \\ Y \end{smallmatrix}\right]$ into Howell form. The resulting matrix exactly represents the intersection of the meanings of X and Y:

$$\gamma_{KS}(X \sqcap Y) = \gamma_{KS}(X) \cap \gamma_{KS}(Y).$$

3.4.2 Project and Havoc

[52, §3] describe a way to project a KS element X onto a suffix $x_i, \ldots, x_k$ of its vocabulary: (i) put X in row-echelon form to create X′; (ii) create X″ by removing from X′ every row a in which any of $a_1, \ldots, a_{i-1}$ is nonzero (i.e., $X'' = [X']_i$); and (iii) remove columns $1, \ldots, i-1$. (Note that the resulting matrix has only a portion of the original vocabulary; we have projected away $\{x_1, \ldots, x_{i-1}\}$.) However, although their method works for Boolean-valued KS elements (i.e., KS elements over $\mathbb{Z}_2^k$), when the leading values of X are not all 1, as can occur in KS elements over $\mathbb{Z}_{2^w}^k$ for $w > 1$, step (ii) is not guaranteed to produce the most-precise projection of X onto $x_i, \ldots, x_k$, although the KS element obtained is always sound.

Example 3.13. *Suppose that* $X = \begin{array}{c} \begin{matrix} \nu_1 & \nu_2 & 1 \end{matrix} \\ \left[\begin{array}{cc|c} 4 & 2 & 6 \end{array}\right] \end{array}$*, with* $w = 4$*, and the goal is to project away the first column (for* ν_1*). When the King/Søndergaard projection algorithm is applied to* X*, we obtain the empty matrix, which represents no constraints on* ν_2*—i.e.,* $\nu_2 \in \{0, 1, \ldots, 15\}$*. However, closer inspection reveals that* ν_2 *cannot be even; if* ν_2 *were even, then both of the terms* $4\nu_1$ *and* $2\nu_2$ *would both be divisible by 4, and hence both values would have at least two zeros as their least-significant bits. Such a pair of values could not sum to a value congruent to 6 because the binary representation of 6 ends with* $\ldots 10$*.* □

Instead, we put X in Howell form before removing rows. By Thm. 3.14, step (ii) above returns the exact projection of the original KS element onto the smaller vocabulary.

Theorem 3.14. *Suppose that* M *has* c *columns. If matrix* M *is in Howell form,* $\vec{v} \in \text{null}^t M$ *if and only if* $\forall i : \forall y_1, \ldots y_{i-1} : \begin{bmatrix} y_1 & \cdots y_{i-1} & \nu_i & \cdots & \nu_c \end{bmatrix} \in \text{null}^t([M]_i)$.

Proof. See App. B. □

Example 3.15. *The Howell form of* X *from Ex. 3.13 is* $\begin{array}{c} \begin{matrix} \nu_1 & \nu_2 & 1 \end{matrix} \\ \left[\begin{array}{cc|c} 4 & 2 & 6 \\ 0 & 8 & 8 \end{array}\right] \end{array}$*, and thus we obtain the following answer for the projection of* X *onto* ν_2*:* $\begin{array}{c} \begin{matrix} \nu_2 & 1 \end{matrix} \\ \left[\begin{array}{c|c} 8 & 8 \end{array}\right] \end{array}$*, which represents* $\nu_2 \in \{1, 3, \ldots, 15\}$*.*

This example illustrates that while the answer produced in Ex. 3.13 by the King/Søndergaard projection algorithm is a sound over-approximation, it is not as precise as the most-precise answer that can be represented in the KS domain. □

Given KS element M, it is also possible to project away a set of variables V that does not constitute a prefix of the vocabulary: create M′ by permuting the columns of M so that the columns for the variables in V come first—the order chosen for the V columns themselves is unimportant—and then project away V from M′ as described earlier.

The *havoc* operation removes all constraints on a set of variables V. To havoc V from KS element M, project away V and then (i) add back an all-0 column for each variable in V, and (ii) permute columns to restore the original variable order. Because of the all-0 columns, the resulting KS element has no constraints on the values of the variables in V.

Example 3.16. *Suppose that we wish to havoc* ν_2 *from the KS value* $\begin{array}{c} \begin{matrix} \nu_1 & \nu_2 & 1 \end{matrix} \\ \left[\begin{array}{cc|c} 2 & 4 & 6 \end{array}\right] \end{array}$*. We permute columns and Howellize to create* $\begin{array}{c} \begin{matrix} \nu_2 & \nu_1 & 1 \end{matrix} \\ \left[\begin{array}{cc|c} 4 & 2 & 6 \\ 0 & 8 & 8 \end{array}\right] \end{array}$*, project onto the vocabulary suffix* ν_1*, obtaining* $\begin{array}{c} \begin{matrix} \nu_1 & 1 \end{matrix} \\ \left[\begin{array}{c|c} 8 & 8 \end{array}\right] \end{array}$*, add back an all-0 column for* ν_2*,* $\begin{array}{c} \begin{matrix} \nu_2 & \nu_1 & 1 \end{matrix} \\ \left[\begin{array}{cc|c} 0 & 8 & 8 \end{array}\right] \end{array}$*, and permute columns back to the original order to*

obtain $\left[\begin{array}{cc|c} \overset{v_1}{8} & \overset{v_2}{0} & \overset{1}{8} \end{array}\right]$. □

3.4.3 Compose

[53, §5.2] present a technique to compose two-vocabulary affine relations. For completeness, that algorithm follows. Suppose that we have KS elements $Y = \left[\begin{array}{cc|c} Y_{pre} & Y_{post} & y \end{array}\right]$ and $Z = \left[\begin{array}{cc|c} Z_{pre} & Z_{post} & z \end{array}\right]$, where Y_{pre}, Y_{post}, Z_{pre}, and Z_{post} are k-column matrices, and y and z are column vectors. We want to compute the relational composition "$Y ; Z$"; i.e., find some X such that $(\vec{v}, \vec{v}'') \in \gamma_{KS}(X)$ if and only if $\exists \vec{v}' : (\vec{v}, \vec{v}') \in \gamma_{KS}(Y) \wedge (\vec{v}', \vec{v}'') \in \gamma_{KS}(Z)$.

Because the KS domain has a projection operation, we can create $Y ; Z$ by first constructing the three-vocabulary matrix W,

$$W = \left[\begin{array}{ccc|c} Y_{post} & Y_{pre} & 0 & y \\ Z_{pre} & 0 & Z_{post} & z \end{array}\right],$$

and then projecting away the first vocabulary of W. Any element $(\vec{v}', \vec{v}, \vec{v}'') \in \gamma_{KS}(W)$ has $(\vec{v}, \vec{v}') \in \gamma_{KS}(Y)$ and $(\vec{v}', \vec{v}'') \in \gamma_{KS}(Z)$; consequently, the projection yields a matrix X such that $\gamma_{KS}(X) = \gamma_{KS}(Y) ; \gamma_{KS}(Z)$, as required.

Alternatively, we can think of abstract composition as happening in three steps: (i) adding 0-columns and reordering vocabularies in Y and Z; (ii) computing the meet W of the resulting matrices; and (iii) projecting onto the initial and final vocabulary of W. Thus, because reordering, meet, and projection are all exact operations, abstract composition is also an exact operation:

$$\gamma_{KS}(Y ; Z) = \gamma_{KS}(Y) ; \gamma_{KS}(Z).$$

Note that the steps of the abstract-composition algorithm mimic a standard

way to express the composition of concrete relations, i.e.,

$$Y\,;Z = \exists U\colon Y[\mathrm{pre},\mathrm{post}] \wedge Z[\mathrm{pre},\mathrm{post}] \wedge U = Y.\mathrm{post} \wedge U = Z.\mathrm{pre}.$$

3.4.4 Join

To join two KS elements Y and Z, we first construct the matrix $\begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix}$ and then project onto the last $2k+1$ columns.

[52, §3] give a method to compute the join of two KS elements by building a $(6k+3)$-column matrix and projecting onto its last $2k+1$ variables. We improve their approach slightly, building a $(4k+2)$-column matrix and then projecting onto its last $2k+1$ variables.

If Y and Z are considered as representing *linear* spaces, rather than affine spaces, this approach works because $\begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix}\begin{bmatrix} u \\ v \end{bmatrix} = 0$ is true just if $(Y(v-u) = 0) \wedge (Zu = 0)$. Because $(v-u) \in \operatorname{null} Y$, and $u \in \operatorname{null} Z$, we know that v is the sum of values in $\operatorname{null} Y$ and $\operatorname{null} Z$, and so v is in their linear closure. Thm. 3.17 demonstrates the correctness of the same algorithm in affine spaces; that proof is driven by roughly the same intuition.

Theorem 3.17. *If Y and Z are both $N+1$-column KS matrices, and $\gamma_{KS}(Y)$ and $\gamma_{KS}(Z)$ are both non-empty sets, then $Y \sqcup Z$ is the projection of $\begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix}$ onto its right-most $N+1$ columns.*

Proof. See App. C. □

Join is not exact in the same sense that meet, project, and compose are above: affine spaces are not closed under union. However, this algorithm does return the least upper bound of Y and Z in the space of KS elements.

Neither meet nor compose distribute over join, as illustrated in the following examples:

Meet over join [1]

$$
\begin{aligned}
\text{“}(v_1 = v_1')\text{”} = \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 0 \end{array}\right] &= \top \sqcap \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 0 \end{array}\right] \\
&= \left(\left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & 0 & 0 \end{array}\right] \sqcup \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 0 & 1 & 0 \end{array}\right] \right) \sqcap \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 0 \end{array}\right] \\
&\neq \left(\left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & 0 & 0 \end{array}\right] \sqcap \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 0 \end{array}\right] \right) \\
&\quad \sqcup \left(\left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 0 & 1 & 0 \end{array}\right] \sqcap \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 0 \end{array}\right] \right) \\
&= \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right] \sqcup \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right] = \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right] = \text{“}(v_1 = 0) \wedge (v_1' = 0)\text{”}
\end{aligned}
$$

Compose over join

(i) $\text{“}(v_1 = v_2)\text{”} = \left[\begin{array}{cccc|c} v_1 & v_2 & v_1' & v_2' & 1 \\ 1 & -1 & 0 & 0 & 0 \end{array}\right] = \left[\begin{array}{cccc|c} v_1 & v_2 & v_1' & v_2' & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{array}\right]$;

$$
\left[\begin{array}{cccc|c} v_1 & v_2 & v_1' & v_2' & 1 \\ 1 & -1 & 0 & 0 & 0 \end{array}\right]
$$

$$
= \left(\left[\begin{array}{cccc|c} v_1 & v_2 & v_1' & v_2' & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right] \sqcup \left[\begin{array}{cccc|c} v_1 & v_2 & v_1' & v_2' & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right] \right);
$$

---

[1] In this example, we use the fact that $\top = \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & 0 & 0 \end{array}\right] \sqcup \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 0 & 1 & 0 \end{array}\right]$. Although technically we are not working with a vector space over a field, the intuition is that the KS element $\left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & 0 & 0 \end{array}\right]$ represents the “line” $v_1 = 0$, the KS element $\left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 0 & 1 & 0 \end{array}\right]$ represents the “line” $v_1' = 0$, and their affine closure is the whole “plane” (i.e., $\top$).

$$\begin{array}{rl}
 & \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 1 & -1 & 0 & 0 & 0 \end{array}\right] \\
\neq & \left\{\begin{array}{l} \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right]; \quad \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 1 & -1 & 0 & 0 & 0 \end{array}\right] \\ \\ \sqcup \quad \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right]; \quad \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 1 & -1 & 0 & 0 & 0 \end{array}\right] \end{array}\right\} \\
= & \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right] \sqcup \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right] \\
= & \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right] = \text{“}(\nu_1 = 0) \wedge (\nu_2 = 0)\text{”}
\end{array}$$

$$\begin{array}{rl}
\text{(ii) “}(\nu_1' = \nu_2')\text{”} = & \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 0 & 0 & 1 & -1 & 0 \end{array}\right] \\
= & \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 0 & 0 & 1 & -1 & 0 \end{array}\right]; \quad \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{array}\right] \\
= & \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 0 & 0 & 1 & -1 & 0 \end{array}\right]; \\
 & \left(\begin{array}{l} \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{array}\right] \sqcup \left[\begin{array}{cccc|c} \nu_1 & \nu_2 & \nu_1' & \nu_2' & 1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right] \\ \\ \end{array}\right)
\end{array}$$

$$
\begin{aligned}
&\begin{array}{c} \overset{\nu_1\ \ \nu_2\ \ \nu_1'\ \ \nu_2'\ \ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & -1 & 0 \end{array}\right]} \end{array} ; \left(\begin{array}{c} \overset{\nu_1\ \ \nu_2\ \ \nu_1'\ \ \nu_2'\ \ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{array}\right]} \end{array} \right) \\
\neq & \\
&\begin{array}{c} \overset{\nu_1\ \ \nu_2\ \ \nu_1'\ \ \nu_2'\ \ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & -1 & 0 \end{array}\right]} \end{array} ; \left(\begin{array}{c} \overset{\nu_1\ \ \nu_2\ \ \nu_1'\ \ \nu_2'\ \ 1}{\left[\begin{array}{cccc|c} -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right]} \end{array} \right)
\end{aligned}
$$

$$
= \quad \overset{\nu_1\ \ \nu_2\ \ \nu_1'\ \ \nu_2'\ \ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right]} \sqcup \overset{\nu_1\ \ \nu_2\ \ \nu_1'\ \ \nu_2'\ \ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right]}
$$

$$
= \quad \overset{\nu_1\ \ \nu_2\ \ \nu_1'\ \ \nu_2'\ \ 1}{\left[\begin{array}{cccc|c} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{array}\right]} = \text{“}(\nu_1' = 0) \wedge (\nu_2' = 0)\text{”}
$$

Similarly, join does not distribute over either meet or compose, as illustrated in the following examples:

Join over meet

$$
\text{“}(\nu_1 = \nu_1')\text{”} = \overset{\nu_1\ \ \nu_1'\ \ 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]} = \overset{\nu_1\ \ \nu_1'\ \ 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right]} \sqcup \overset{\nu_1\ \ \nu_1'\ \ 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]}
$$

$$
= \left(\overset{\nu_1\ \ \nu_1'\ \ 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \end{array}\right]} \sqcap \overset{\nu_1\ \ \nu_1'\ \ 1}{\left[\begin{array}{cc|c} 0 & 1 & 0 \end{array}\right]} \right) \sqcup \overset{\nu_1\ \ \nu_1'\ \ 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]}
$$

$$
\neq \left(\overset{\nu_1\ \ \nu_1'\ \ 1}{\left[\begin{array}{cc|c} 1 & 0 & 0 \end{array}\right]} \sqcup \overset{\nu_1\ \ \nu_1'\ \ 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]} \right) \sqcap \left(\overset{\nu_1\ \ \nu_1'\ \ 1}{\left[\begin{array}{cc|c} 0 & 1 & 0 \end{array}\right]} \sqcup \overset{\nu_1\ \ \nu_1'\ \ 1}{\left[\begin{array}{cc|c} 1 & -1 & 0 \end{array}\right]} \right)
$$

$$
= \top \sqcap \top = \text{“true”}
$$

Join over compose

$$
\begin{aligned}
\text{“}(v_1' = v_1 + 1)\text{”} = \quad & \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right] = \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right] \sqcup \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right] \\
= & \left(\mathrm{Id}\,;\ \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right]\right) \sqcup \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right] \\
= & \left(\left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 0 \end{array}\right]\,;\ \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right]\right) \sqcup \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right] \\
\neq & \left(\left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 0 \end{array}\right] \sqcup \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right]\right)\,; \\
& \left(\left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right] \sqcup \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right]\right) \\
= \top\,; \quad & \left[\begin{array}{cc|c} v_1 & v_1' & 1 \\ 1 & -1 & 1 \end{array}\right] = \top = \text{“true”}
\end{aligned}
$$

3.4.5 Assuming Conditions

By “assuming” a condition φ on a KS element X, we mean to compute a minimal KS element Y such that

$$\gamma_{\mathrm{KS}}(Y) \supseteq \gamma_{\mathrm{KS}}(X) \cap \{v \mid \varphi(v)\}\,.$$

This operation is needed to compute the transformer for an assume edge in a program graph (i.e., the true-branch or false-branch of an if-then-else statement). It can also be used to create transformers for assignments; for instance, the transformer for the assignment $x \leftarrow 3u + 2v$ can be created by starting with the KS element for the identity relation on vocabulary

$\begin{matrix} \vec{V} & \vec{V}' & 1 \end{matrix}$
$\vec{V}$, $\left[\begin{array}{cc|c} I & -I & 0 \end{array}\right]$, havocking $x' \in \vec{V}'$, and assuming the equality $x' = 3u + 2v$.

Assuming a w-bit affine constraint is straightforward: rewrite the constraint to isolate 0 on one side; form a matrix row from resulting constraint's coefficients; append the row to the KS element X; and Howellize. In other words, when φ is an affine constraint, we create a one-row KS element that represents φ exactly, and take the meet with X.

It is also possible to perform an assume with respect to an affine congruence of the form "lhs $=$ rhs $(\mathrm{mod}\ 2^h)$", with $h < w$. We rewrite the congruence as an equivalent congruence modulo 2^w, by multiplying the modulus 2^h and all of the coefficients by 2^{w-h}, to obtain the w-bit affine constraint "2^{w-h}lhs $= 2^{w-h}$rhs". We then proceed as before.

3.4.6 Containment

Two KS elements X and Y are equal if their concretizations are equal: $\gamma(X) = \gamma(Y)$. However, when each KS element is in Howell form, equality checking is trivial because Howell form is unique among all matrices with the same row space (or null space) [44]. Consequently, containment can be checked using meet and equality: $X \sqsubseteq Y$ iff $X = X \sqcap Y$.

3.4.7 Merge Functions

[54] extended the Sharir and Pnueli algorithm 1981 for interprocedural dataflow analysis to handle local variables. At a site where procedure P calls procedure Q, the local variables of P are modeled as if the current incarnations of P's locals are stored in locations that are inaccessible to Q and to procedures transitively called by Q. Because the contents of P's locals cannot be affected by the call to Q, a *merge function* is used to combine them with the element returned by Q to create the state in P after

the call to Q has finished. Other work using merge functions includes [75] and [56].

To simplify the discussion, assume that all scopes have the same number of locals L. Each merge function is of the form

$$\textsc{Merge}(\text{CallSiteVal}, \text{CalleeExitVal}) \stackrel{\text{def}}{=}$$
$$\text{CallSiteVal} \,;\, \textsc{ReplaceLocals}(\text{CalleeExitVal}).$$

Suppose that the i^{th} vocabulary consists of sub-vocabularies $\overrightarrow{g_i}$ and $\overrightarrow{l_i}$. The operation ReplaceLocals(CalleeExitVal) is defined as follows:

1. Project away vocabulary $\overrightarrow{l_2}$ from CalleeExitVal.
2. Insert L all-0 columns for vocabulary $\overrightarrow{l_2}$. The KS element now has no constraints on the variables in $\overrightarrow{l_2}$.
3. Append L rows, $\begin{array}{c} \overrightarrow{g_1} \quad \overrightarrow{l_1} \quad \overrightarrow{g_2} \quad \overrightarrow{l_2} \quad 1 \\ \left[\begin{array}{cccc|c} 0 & I & 0 & -I & 0 \end{array}\right] \end{array}$, so that in ReplaceLocals(CalleeExitVal) each variable in vocabulary $\overrightarrow{l_2}$ is constrained to have the value of the corresponding variable in vocabulary $\overrightarrow{l_1}$.

3.4.8 Symbolic Abstraction ($\widehat{\alpha}(\varphi)$)

[53, Fig. 2] gave an algorithm for the problem of *symbolic abstraction*(§2.3.2) with respect to the KS domain: given a quantifier-free bit-vector (QFBV) formula φ, the algorithm returns the best element in KS that over-approximates $[\![\varphi]\!]$. The algorithm given by King and Søndergaard, which we will denote by $\widehat{\alpha}^{\uparrow}_{KS}(\varphi)$, needs the minor correction of using Howell form instead of row-echelon form for the projections that take place in its join operations, as discussed in §3.4.4.

Pseudo-code for $\widehat{\alpha}^{\uparrow}_{KS}(\varphi)$ is shown in Fig. 3.1(a). The matrix *lower* is maintained in Howell form throughout. In line 6, $\widehat{\alpha}^{\uparrow}_{KS}(\varphi)$ uses the *symbolic concretization* of p, denoted by $\widehat{\gamma}(p)$. For most abstract domains, including

Require: φ: a QFBV formula
Ensure: $\hat{\alpha}(\varphi)$ for the KS domain

1: *lower* $\leftarrow \bot$
2: $i \leftarrow 1$
3:
4: **while** $i \leqslant$ `rows`(*lower*) **do**
5: $p \leftarrow$ *lower*[`rows`(*lower*) $- i + 1$]
$\{p \sqsupseteq$ *lower*$\}$
6: $S \leftarrow$ `Model`$(\varphi \wedge \neg\hat{\gamma}(p))$
7: **if** S is `TimeOut` **then return** $\top$
8: **else if** S is `None` **then** $\{\varphi \Rightarrow \hat{\gamma}(p)\}$
9: $i \leftarrow i + 1$
10:
11: **else** $\{S \not\models \hat{\gamma}(p)\}$
12: *lower* $\leftarrow$ *lower* $\sqcup\, \beta(S)$
13: *ans* $\leftarrow$ *lower*
14: **return** *ans*

(a)

Require: φ: a QFBV formula
Ensure: $\hat{\alpha}(\varphi)$ for the KS domain

1: *lower* $\leftarrow \bot$
2: $i \leftarrow 1$
3: *upper* $\leftarrow \top$
4: **while** $i \leqslant$ `rows`(*lower*) **do**
5: $p \leftarrow$ *lower*[`rows`(*lower*) $- i + 1$]
$\{p \sqsupseteq$ *lower*, $p \not\sqsupseteq$ *upper*$\}$
6: $S \leftarrow$ `Model`$(\varphi \wedge \neg\hat{\gamma}(p))$
7: **if** S is `TimeOut` **then return** *upper*
8: **else if** S is `None` **then** $\{\varphi \Rightarrow \hat{\gamma}(p)\}$
9: $i \leftarrow i + 1$
10: *upper* $\leftarrow$ *upper* $\sqcap\, p$
11: **else** $\{S \not\models \hat{\gamma}(p)\}$
12: *lower* $\leftarrow$ *lower* $\sqcup\, \beta(S)$
13: *ans* $\leftarrow$ *lower*
14: **return** *ans*

(b)

Figure 3.1: (a) The King-Søndergaard algorithm for symbolic abstraction ($\hat{\alpha}^{\uparrow}_{\mathrm{KS}}(\varphi)$). (b) The Thakur-Elder-Reps bilateral algorithm for symbolic abstraction, instantiated for the KS domain: $\hat{\alpha}^{\updownarrow}_{\mathrm{TER[KS]}}(\varphi)$. In both algorithms, *lower* is maintained in Howell form throughout.

KS, it is easy to write a $\hat{\gamma}$ function. As mentioned in §3.1.4, affine equalities can be read out from a KS element M (regardless of whether M is in Howell form) as follows:

If $\begin{array}{c} \begin{matrix} v_1 & \dots & v_k & v'_1 & \dots & v'_k & 1 \end{matrix} \\ \left[\begin{array}{cccccc|c} a_1 & \dots & a_k & a'_1 & \dots & a'_k & b \end{array}\right] \end{array}$ is a row of M, then $\sum_i a_i v_i + \sum_i a'_i v'_i = -b$ is a constraint on $\gamma_{\mathrm{KS}}(M)$.

The conjunction of these constraints describes $\gamma_{\mathrm{KS}}(M)$ exactly. Conse-

quently, $\hat{\gamma}(M)$ can be defined as follows:

$$\hat{\gamma}(M) \stackrel{\text{def}}{=} \bigwedge_{\substack{[a_1 \cdots a_k\ a'_1 \cdots a'_k\ |b] \\ \text{is a row of } M}} \sum_i a_i v_i + \sum_i a'_i v'_i = -b$$

The algorithm $\hat{\alpha}^{\uparrow}_{\text{KS}}(\varphi)$ is a successive-approximation algorithm: it computes a sequence of successively larger approximations to $[\![\varphi]\!]$. It maintains an under-approximation of the final answer in the variable "*lower*", which is initialized to $\bot$ on line 1. On each iteration, the algorithm selects p, a single row (constraint) of *lower* (line 5), and calls a decision procedure to determine whether there is a model that satisfies the formula "$\varphi \wedge \neg\hat{\gamma}(p)$" (line 6). When $\varphi \wedge \neg\hat{\gamma}(p)$ is unsatisfiable, φ implies $\hat{\gamma}(p)$. In this case, p cannot be used to figure out how to make *lower* larger, so variable i is incremented (line 9), which means that on the next iteration of the loop, the algorithm selects the row immediately above p (line 5).

On the other hand, if the decision procedure returns a model S, the under-approximation *lower* is updated to make it larger via the join performed on the right-hand side of the assignment in line 12

$$\mathit{lower} \leftarrow \mathit{lower} \sqcup \beta(S). \tag{3.5}$$

Because KS elements represent two-vocabulary relations, S is an assignment of concrete values to both the pre-state and post-state variables:

$$S = [\ldots, v_i \mapsto v_i, \ldots, v'_i \mapsto v'_i, \ldots],$$

or, equivalently,

$$S = [\overrightarrow{X} \mapsto \overrightarrow{v}, \overrightarrow{X}' \mapsto \overrightarrow{v}']. \tag{3.6}$$

The notation $\beta(S)$ in line 12 denotes the abstraction of the singleton state-set $\{S\}$ to a KS element. $\{S\}$ can always be represented exactly in the KS

domain as follows (where the superscript t denotes the operation of vector transpose):

$$\beta(S) \stackrel{\text{def}}{=} \left[\begin{array}{cc|c} \overrightarrow{x} & \overrightarrow{x}' & 1 \\ \hline I & 0 & (-\overrightarrow{v})^t \\ 0 & I & (-\overrightarrow{v}')^t \end{array}\right] \tag{3.7}$$

An Improvement to $\widehat{\alpha}^{\uparrow}_{\mathbf{KS}}$

Algorithm $\widehat{\alpha}^{\uparrow}_{\text{KS}}$ from Fig. 3.1(a) is related to, but distinct from, an earlier $\widehat{\alpha}$ algorithm, due to [84] (RSY), which applies not just to the KS domain, but to all abstract domains that meet a certain interface. (In other words, $\widehat{\alpha}_{\text{RSY}}$ is the cornerstone of a *framework* for symbolic abstraction.) The two algorithms resemble one another in that they both find $\widehat{\alpha}(\varphi)$ via successive approximation from below. However, there is a key difference in the nature of the satisfiability queries that are passed to the decision procedure by the two algorithms. Compared to $\widehat{\alpha}_{\text{RSY}}$, $\widehat{\alpha}_{\text{KS}}$ issues comparatively inexpensive satisfiability queries in which only a single affine equality is negated[2]—i.e., line 6 of Fig. 3.1(a) calls `Model`$(\varphi \wedge \neg\widehat{\gamma}(p))$, where p is a single constraint from *lower*.

This difference—together with the observation that in practice $\widehat{\alpha}_{\text{KS}}$ was about ten times faster than $\widehat{\alpha}_{\text{RSY}}$ when the latter was instantiated for the KS domain—led Thakur, Elder, and Reps 2012 (TER) to investigate the fundamental principles underlying $\widehat{\alpha}_{\text{RSY}}$ and $\widehat{\alpha}_{\text{KS}}$. They developed a new framework, $\widehat{\alpha}^{\updownarrow}_{\text{TER}}$, that transfers $\widehat{\alpha}_{\text{KS}}$'s advantages from the KS domain to other abstract domains [100].

In addition to generating less expensive satisfiability queries, the second benefit of $\widehat{\alpha}^{\updownarrow}_{\text{TER}}$ is that $\widehat{\alpha}^{\updownarrow}_{\text{TER}}$ generally returns a more precise answer than $\widehat{\alpha}_{\text{RSY}}$ and $\widehat{\alpha}_{\text{KS}}$ when a timeout occurs. Because $\widehat{\alpha}_{\text{RSY}}$ and $\widehat{\alpha}_{\text{KS}}$ maintain only under-approximations of the desired answer, if the successive-

[2]See [100, §3] for a more extensive explanation of the differences between $\widehat{\alpha}_{\text{KS}}$ and $\widehat{\alpha}_{\text{RSY}}$.

approximation process takes too much time and needs to be stopped, they must return $\top$ to be sound. In contrast, $\widehat{\alpha}^{\updownarrow}_{\text{TER}}$ is *bilateral*, and can generally return a nontrivial (non-$\top$) element in case of a timeout. That is, $\widehat{\alpha}^{\updownarrow}_{\text{TER}}$ maintains both an under-approximation and a (nontrivial) over-approximation of the desired answer, and hence is resilient to timeouts: $\widehat{\alpha}^{\updownarrow}_{\text{TER}}$ returns the over-approximation if it is stopped at any point.

Fig. 3.1(b) shows the $\widehat{\alpha}^{\updownarrow}_{\text{TER}}$ algorithm instantiated for the KS domain, which we call $\widehat{\alpha}^{\updownarrow}_{\text{TER[KS]}}$. The differences between Fig. 3.1(a) and (b) are highlighted in gray. $\widehat{\alpha}^{\updownarrow}_{\text{TER[KS]}}$ initializes *upper* to $\top$ on line 3. At any stage in the algorithm $\mathit{upper} \sqsupseteq \widehat{\alpha}(\varphi)$. It is sound to update *upper* on line 10 by performing a meet with the row $\mathtt{p}$ that was selected in line 5. Because $\mathtt{p}$'s leading index, $\mathit{LI}(\mathtt{p})$, is less than the leading index of every row in *upper*, $\mathtt{p}$ constrains the value of variable $v_{\mathit{LI}(\mathtt{p})}$, whereas *upper* places no constraints on variable $v_{\mathit{LI}(\mathtt{p})}$. Therefore, $\mathtt{p} \not\sqsupseteq \mathit{upper}$, which guarantees progress because $\mathtt{p} \sqcap \mathit{upper} \sqsubset \mathit{upper}$. Termination is guaranteed.

In case of a decision-procedure timeout (line 7), $\widehat{\alpha}^{\updownarrow}_{\text{TER[KS]}}$ returns *upper* as the answer (line 7). If the algorithm finishes without a timeout, then $\widehat{\alpha}^{\updownarrow}_{\text{TER[KS]}}$ computes $\widehat{\alpha}(\varphi)$; on the other hand, if a timeout occurs, the element returned is generally an over-approximation of $\widehat{\alpha}(\varphi)$—i.e., $\widehat{\alpha}^{\updownarrow}_{\text{TER[KS]}}$ computes $\widetilde{\alpha}(\varphi)$.

In the KS instantiation of $\widehat{\alpha}^{\updownarrow}_{\text{TER}}$, *upper* can actually be represented implicitly by $\mathit{lower}[(\mathtt{rows}(\mathit{lower}) - \mathtt{i} + 2) \ldots \mathtt{rows}(\mathit{lower})]$. Consequently, the assignment $\mathit{upper} \leftarrow \mathit{lower}[(\mathtt{rows}(\mathit{lower}) - \mathtt{i} + 2) \ldots \mathtt{rows}(\mathit{lower})]$ need only be performed if line 7 is reached, and neither of the assignments on lines 3 and 10 need to be performed explicitly.

3.4.9 Number of Satisfying Solutions

The *size* of a KS element X with $\mathtt{k}$ variables over $\mathbb{Z}_{2^w}$ is the number of $\mathtt{k}$-tuples that satisfy X. The size computation is inexpensive; the size of X

depends on the leading values in X, and the number of rows in X. (X is assumed to be in Howell form.)

- If X is bottom, then $\textsc{Size}(X) = 0$.
- Otherwise, we can derive how to compute $\textsc{Size}(X)$ by imagining that we are building up a partial assignment for the variables, from right to left. (In what follows, for simplicity we assume that we have a one-vocabulary KS element and "right to left" means from higher-indexed variables to lower-indexed variables.) In this case, each variable v_i is constrained by the current partial assignment to the variables $\{v_j \mid i < j\}$, and by the row with leading index i:
  - If the leading value of that row is 1, then for every partial assignment to the variables $\{v_j \mid i < j\}$, there is exactly one consistent value for v_i, namely, whatever value for v_i satisfies the equation for the row when the values in the partial assignment are used for the higher-indexed variables.
  - If the leading value of a row is 2^m for some value m, then for every partial assignment to the variables $\{v_j \mid i < j\}$, there is exactly one consistent value for $2^m v_i$, namely, whatever value y for $2^m v_i$ satisfies the equation for the row when the values in the partial assignment are used for the higher-indexed variables.

    However, there are 2^m different ways to choose v_i to obtain the needed value y. That is, if $\overline{v}$ is a value such that $2^m\overline{v} = y$, then so are all 2^m values in the set

    $$\{(\overline{v} + 2^{w-m}p) \pmod{2^w} \mid 0 \leqslant p \leqslant 2^m - 1\}.$$

  - Finally, if there is no row with leading index i, then v_i is fully unconstrained, and can take on any of the 2^w available values.

  Altogether, the product of these counts is the number of satisfying solutions of KS element X. In particular, let u be the number of indices that are not the leading index of any row of X. Then $\textsc{Size}(X)$

Instruction Characteristics		
Kind	# instruction instances	# different opcodes
ordinary	12,734	164
lock prefix	2,048	147
rep prefix	2,143	158
repne prefix	2,141	154
full corpus	19,066	179

Figure 3.2: Some of the characteristics of the corpus of 19,066 (non-privileged, non-floating point, non-mmx) instructions.

is the product of the leading values in X, times $(2^w)^u$.

Example 3.18. *Consider again the KS element from Eqn. (3.2)*

$$X_0 = \left[\begin{array}{cccc|c} v_1 & v_2 & v_1' & v_2' & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 \end{array}\right],$$

where $w = 4$*, so that we are working in* $\mathbb{Z}_{16}$*. Then* $\textsc{Size}(X_0)$ *equals* $1 \times 8 \times (2^4)^2 = 2,048$*.*

3.5 Experiments

In this section, we present the results of experiments to evaluate the costs and benefits—in terms of time and precision—of the methods described in earlier sections. The experiments were designed to shed light on the following questions:

1. Which method of obtaining abstract transformers is fastest: $\hat{\alpha}_{KS}$ (§2.3.2 and §3.4.8), KS-reinterpretation (§2.3.1), or MOS-reinterpretation (§2.3.1, [62, §4.1.2])?

2. Does MOS-reinterpretation or KS-reinterpretation yield more precise abstract transformers for machine instructions?
3. For what percentage of program points does $\hat{\alpha}_{\mathrm{KS}}$ produce more precise answers than KS-reinterpretation and MOS-reinterpretation? This question actually has two versions, depending on whether we are interested in
   - one-vocabulary affine relations that hold at branch points
   - two-vocabulary procedure summaries obtained at procedure-exit points.

As shown in §3.3.1, the MOS and KS domains are incomparable. To compare the final results obtained using the two domains, we converted each MOS element to a KS element, using the algorithm from §3.3.2, and then checked for equality, containment (§3.4.6), or incomparability. It might be argued that this approach biases the results in favor of KS. However, if we have run an MOS-based analysis and are interested in using affine relations in a client application, we must extract an affine relation from each computed MOS element. In §3.3.1, we showed that, in general, an MOS element $\mathcal{B}$ does not represent an affine relation; thus, a client application needs to obtain an affine relation that over-approximates $\gamma_{\mathrm{MOS}}(\mathcal{B})$. Consequently, the comparison method that we used *is* sensible, because it compares the precision of the affine relations that would be seen by a client application.

3.5.1 Experimental Setup

To address these questions, we performed two experiments. Both experiments were run on a single core of a single-processor 16-core 2.27 GHz Xeon computer running 64-bit Windows 7 Enterprise (Service Pack 1), configured so that a user process has 4 GB of memory.

Program Name	Measures of Size						
					WPDS Rules		
	Instrs	Procs	BBs	Branches	Δ_0	Δ_1	Δ_2
write	232	10	134	26	10	151	5
finger	532	18	298	48	18	353	20
subst	1093	16	609	74	16	728	13
chkdsk	1468	18	787	119	18	887	32
convert	1927	38	1013	161	38	1266	22
route	1982	40	931	243	40	1368	63
comp	2377	35	1261	224	35	1528	30
logoff	2470	46	1145	306	46	1648	72
setup	4751	67	1862	589	67	2847	121

Figure 3.3: Program information. All nine utilities are from Microsoft Windows version 5.1.2600.0, except setup, which is from version 5.1.2600.5512. The columns show the number of instructions (Instrs); the number of procedures (Procs); the number of basic blocks (BBs); the number of branch instructions (Branches); and the number of Δ_0, Δ_1, and Δ_2 rules in the WPDS encoding (WPDS Rules).

Per-Instruction Experiment On a corpus of 19,066 instances of x86 instructions, we measured (i) the time taken to create MOS and KS transformers via the operator-by-operator reinterpretation method supported by TSL [61, 62], and (ii) the relative precision of the abstract transformers obtained by the two methods.

This corpus was created using the ISAL instruction-decoder generator [62, §2.1] in a mode in which the input specification of the concrete syntax of the x86 instruction set was used to create a randomized instruction *generator*—instead of the standard mode in which ISAL creates an instruction *recognizer*. By this means, we are assured that the corpus has substantial coverage of the syntactic features of the x86 instruction set (including opcodes, addressing modes, and prefixes, such as “lock”, “rep”, and “repne”); see Fig. 3.2.

Interprocedural-Analysis Experiment We performed flow-sensitive, context-sensitive, interprocedural affine-relation analysis on the executables of nine Windows utilities, using four different sets of abstract transformers:

1. MOS transformers for basic blocks, created by performing operator-by-operator MOS-reinterpretation.
2. KS transformers for basic blocks, created by performing operator-by-operator KS-reinterpretation.
3. KS transformers for basic blocks, created by symbolic abstraction of quantifier-free bit-vector (QFBV) formulas that capture the precise bit-level semantics of register-access/update operations in the different basic blocks. We denote this symbolic-abstraction method by $\widehat{\alpha}_{\mathrm{KS}}$.
4. KS transformers for basic blocks, created by symbolic abstraction of quantifier-free bit-vector (QFBV) formulas that, in addition to register-access/update operations, also capture the precise bit-level semantics of all *memory-access/update* and *flag-access/update* operations. We denote this symbolic-abstraction method by $\widehat{\alpha}^{+}_{\mathrm{KS}}$.

For these programs, the generated abstract transformers were used as "weights" in a weighted pushdown system (WPDS).

Fig. 3.3 lists several size parameters of the executables (number of instructions, procedures, basic blocks, branches, and number of WPDS rules). WPDS rules can be divided into three categories, called Δ_0, Δ_1, and Δ_2 rules [10, 85]. The number of Δ_1 rules corresponds roughly to the total number of edges in a program's intraprocedural control-flow graphs; the number of Δ_2 rules corresponds to the number of call sites in the program; the number of Δ_0 rules corresponds to the number of procedure-exit sites.

$\widehat{\alpha}^{+}_{\mathrm{KS}}$ has the potential to create more precise KS weights than $\widehat{\alpha}_{\mathrm{KS}}$ because $\widehat{\alpha}^{+}_{\mathrm{KS}}$ can account for transformations of register values that involve a sequence of memory-access/update and/or flag-access/update opera-

tions within a basic block $\mathcal{B}$. For example, suppose that $\mathcal{B}$ contains a store to memory of register `eax`'s value, and a subsequent load from memory of that value into `ebx`. Because $\widehat{\alpha}^{+}_{KS}$ uses a formula that captures the two memory operations, it can find a weight that captures the transformation `ebx' = eax`. A second type of example involving a store to memory followed by a load from memory within a basic block involves a sequence of the form "`push` *constant*; ... `pop edi`", and thus represents the transformation `edi' = constant`. Such sequences occur in several of the programs listed in Fig. 3.3.

Fig. 3.4 shows a TSL specification for the `MOV` and `ADD` instructions of the Intel `IA32` instruction set. As illustrated in line 13 of Fig. 3.4, the top-level function that is reinterpreted in TSL is interpInstr, which is of type

$$\text{interpInstr} : \text{instruction} \times \text{state} \rightarrow \text{state}.$$

To use semantic reinterpretation to implement CreateAbsTrans for the KS domain, interpInstr is reinterpreted as a KS[$V; V'$] transformer; that is, interpInstr_{KS} has the type

$$\text{interpInstr}_{KS} : \text{instruction} \times \text{KS}[V; V'] \rightarrow \text{KS}[V; V'].$$

Let Id denote the KS[$V; V'$] identity relation, $\begin{array}{c} \vec{V} \quad \vec{V}' \quad 1 \\ \left[\begin{array}{cc|c} I & -I & 0 \end{array}\right] \end{array}$. To reinterpret an individual instruction ι, one invokes $\text{interpInstr}_{KS}(\iota, \text{Id})$.

For a basic block $B = [\iota_1, \ldots, \iota_m]$, there are two approaches to performing KS[$V; V'$] reinterpretation:

- *Composed reinterpretation*:

$$w^{B}_{KS} \leftarrow \text{interpInstr}_{KS}(\iota_1, \text{Id}) ; \text{interpInstr}_{KS}(\iota_2, \text{Id}) ; \ldots ; \text{interpInstr}_{KS}(\iota_m, \text{Id}).$$

```
[1]   // Abstract-syntax declarations
[2]   reg: EAX() | EBX() | . . . ;
[3]   flag: ZF() | SF() | . . . ;
[4]   operand: Indirect(reg reg INT8 INT32) | DirectReg(reg) | Immediate(INT32) | ...;
[5]   instruction: MOV(operand operand) | ADD(operand operand) | . . . ;
[6]   state: State(MAP[INT32,INT8]   // memory-map
[7]                MAP[reg,INT32]  // register-map
[8]                MAP[flag,BOOL]);   // flag-map
[9]   // Interpretation functions
[10]  INT32 interpOp(state S, operand op) { . . . };
[11]  state updateFlag(state S, INT32 v1, INT32 v2, INT32 v3) { . . . };
[12]  state updateState(state S, operand op, INT32 val) { . . . };
[13]  state interpInstr(instruction I, state S) {
[14]    with(I) (
[15]      MOV(dstOp, srcOp):
[16]        let srcVal = interpOp(S, srcOp);
[17]        in ( updateState(S, dstOp, srcVal) ),
[18]      ADD(dstOp, srcOp):
[19]        let dstVal = interpOp(S, dstOp);
[20]            srcVal = interpOp(S, srcOp);
[21]            result = dstVal + srcVal;
[22]            S2 = updateFlag(S, dstVal, srcVal, result);
[23]        in ( updateState(S2, dstOp, result) ),
[24]      . . .
[25]    );
[26]  };
```

Figure 3.4: A fragment of the TSL specification of the concrete semantics of the Intel IA32 instruction set.

- *Chained reinterpretation*:

$$w^{\mathrm{B}}_{\mathrm{KS}} \leftarrow \mathrm{interpInstr}_{\mathrm{KS}}(\iota_m, \ldots \mathrm{interpInstr}_{\mathrm{KS}}(\iota_2, \mathrm{interpInstr}_{\mathrm{KS}}(\iota_1, \mathrm{Id})) \ldots).$$

Our experiments use chained reinterpretation for two reasons:

1. There are cases in which chained reinterpretation creates a more precise KS[$V; V'$] element. For instance, consider the following code fragment, which zeros the two low-order bytes of register `eax` and does a bitwise-or of `eax` into `ebx` (`ax` denotes the two low-order bytes of register `eax`):

$$\begin{aligned} \iota_1 &: \texttt{xor}\ \ \texttt{ax},\texttt{ax} \\ \iota_2 &: \texttt{or}\ \ \texttt{ebx},\texttt{eax} \end{aligned}$$

The semantics of this code fragment can be expressed as follows:

$$\texttt{ebx'} = (\texttt{ebx} \mid (\texttt{eax} \,\&\, \texttt{xFFFF0000})) \wedge \texttt{eax'} = (\texttt{eax} \,\&\, \texttt{xFFFF0000}),$$

where "&" and "|" denote bitwise-and and bitwise-or, respectively. $\text{interpInstr}_{\text{KS}}(\iota_1, \text{Id})$ creates the KS element $\left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right]$, which captures $(\texttt{ebx'} = \texttt{ebx}) \wedge (2^{16}\texttt{eax'} = 0)$. The two approaches to reinterpretation produce the following answers:

- *Composed reinterpretation*:

$$\begin{aligned} &\text{interpInstr}_{\text{KS}}(\iota_1, \text{Id}) \,;\, \text{interpInstr}_{\text{KS}}(\iota_2, \text{Id}) \\ &\quad = \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right] ;\ \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 1 & 0 & -1 & 0 & 0 \end{array}\right] \\ &\quad = \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 0 & 0 & 2^{16} & 0 & 0 \end{array}\right] \\ &\quad = (2^{16}\texttt{eax}' = 0). \end{aligned}$$

- *Chained reinterpretation*:

$$\begin{aligned}
&\text{interpInstr}_{\text{KS}}(\iota_2, \text{interpInstr}_{\text{KS}}(\iota_1, \text{Id})) \\
&\quad = \text{interpInstr}_{\text{KS}}(\iota_2, \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right]) \\
&\quad = \left[\begin{array}{cccc|c} \texttt{eax} & \texttt{ebx} & \texttt{eax}' & \texttt{ebx}' & 1 \\ \hline 0 & 2^{16} & 0 & -2^{16} & 0 \\ 0 & 0 & 2^{16} & 0 & 0 \end{array}\right] \\
&\quad = (2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx}) \wedge (2^{16}\texttt{eax}' = 0).
\end{aligned}$$

  In particular, the reinterpretation of "ι_2: `or ebx, eax`" takes place in a context in which the two low-order bytes of `eax` are partially constant ($2^{16}\texttt{eax} = 0$). Because of this additional piece of information, the reinterpretation technique recovers the additional conjunct "$2^{16}\texttt{ebx}' = 2^{16}\texttt{ebx}$".

2. In the case of $\hat{\alpha}_{\text{KS}}$, a formula φ_B is created that captures the concrete semantics of B (via symbolic execution), and then the KS weight for B is obtained by performing $w_{\text{KS}}^{B} \leftarrow \hat{\alpha}_{\text{KS}}(\varphi_{\mathcal{B}})$. Letting QFBV denote the type of quantifier-free bit-vector formulas, the QFBV reinterpretation of interpInstr has the type

$$\text{interpInstr}_{\text{QFBV}} : \text{instruction} \times \text{QFBV} \to \text{QFBV}.$$

   Symbolic execution is performed by chained reinterpretation:

$$\varphi_B \leftarrow \text{interpInstr}_{\text{QFBV}}(\iota_m, \ldots \text{interpInstr}_{\text{QFBV}}(\iota_2, \text{interpInstr}_{\text{QFBV}}(\iota_1, \text{Id})) \ldots).$$

   The $\hat{\alpha}_{\text{KS}}^{+}$ weight for B is created similarly, except that we also arrange for $\varphi_{\mathcal{B}}$ to encode all memory-access/update and flag-access/update operations.

Total instructions	MOS-reinterpretation time (seconds)	KS-reinterpretation time (seconds)
19,066	23.3	348.2

Figure 3.5: Comparison of the performance of MOS-reinterpretation and KS-reinterpretation for x86 instructions.

Identical precision	MOS-reinterpretation more precise	KS-reinterpretation more precise	Total
18,158	0	908	19,066

Figure 3.6: Comparison of the precision of MOS-reinterpretation and KS-reinterpretation for x86 instructions.

For our experiments, we wanted to control for any precision improvements that might be due solely to the use of chained reinterpretation; thus, we use chained reinterpretation for all of the weight-generation methods.[3]

Due to the high cost in §3.5.3 of constructing WPDS weights via $\hat{\alpha}_{KS}$ and $\hat{\alpha}^{+}_{KS}$, we ran all WPDS analyses without the code for libraries. Values are returned from x86 procedure calls in register `eax`, and thus in our experiments library functions were modeled approximately (albeit unsoundly, in general) by "`havoc(eax')`".

To implement $\hat{\alpha}_{KS}$ and $\hat{\alpha}^{+}_{KS}$, we used the Yices solver [26], version 1.0.19, with the timeout for each invocation set to three seconds.

We compared the precision of the one-vocabulary affine relations at branch points, as well as two-vocabulary affine relations at procedure exits, which can be used as procedure summaries.

3.5.2 Reinterpretation of Individual Instructions

Figs. 3.5 and 3.6 summarize the results of the per-instruction experiment. They answer questions (1) and (2) posed at the beginning of §3.5.

- KS-reinterpretation created an abstract transformer that was more precise than the one created by MOS-reinterpretation for about 4.76% of the instructions. MOS-reinterpretation never created an abstract transformer that was more precise than the one created by KS-reinterpretation.
- However, MOS-reinterpretation is much faster: to generate abstract transformers for the entire corpus of instructions, MOS-reinterpretation is about 14.9 times faster than KS-reinterpretation.

Example 3.19. *One instruction for which the abstract transformer created by KS-reinterpretation is more precise than the transformer created by MOS-reinterpretation is* $\iota \stackrel{def}{=}$ "`add bh,al`". *This instruction adds the value of* `al`*, the low-order byte of register* `eax`*, to the value of* `bh`*, the second-to-lowest byte of register* `ebx`*, and stores the result in* `bh`*. The semantics of this instruction can be expressed as a QFBV formula as follows:*

$$\varphi_\iota \stackrel{def}{=} \texttt{ebx'} = \begin{pmatrix} (\texttt{ebx} \,\&\, \texttt{0xFFFF00FF}) \\ |\, ((\texttt{ebx} + 256 * (\texttt{eax} \,\&\, \texttt{0xFF})) \,\&\, \texttt{0xFF00}) \end{pmatrix} \wedge (\texttt{eax'} = \texttt{eax}). \tag{3.8}$$

Eqn. (3.8) shows that the semantics of the instruction involves non-linear bit-masking operations.

The abstract transformer created via MOS-reinterpretation corresponds to `havoc(ebx')`*; all other registers are unchanged. That is, if we only had the three registers* `eax`*,* `ebx`*, and* `ecx`*, the abstract transformer created via MOS-*

[3] MOS-reinterpretation of a basic block is performed by chained reinterpretation, using $\text{interpInstr}_{\text{MOS}} : \text{instruction} \times \text{MOS} \to \text{MOS}$.

reinterpretation would be

$$\left\{ \left[\begin{array}{c|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right], \left[\begin{array}{c|ccc} 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right] \right\},$$

which captures the affine transformation "$(\texttt{eax}' = \texttt{eax}) \wedge (\texttt{ecx}' = \texttt{ecx})$*". In contrast, the transformer created via KS-reinterpretation is*

$$\begin{array}{c} \texttt{eax} \quad \texttt{ebx} \quad \texttt{ecx} \quad \texttt{eax}' \quad \texttt{ebx}' \quad \texttt{ecx}' \quad 1 \\ \left[\begin{array}{cccccc|c} 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 2^{24} & 0 & 0 & -2^{24} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{array}\right], \end{array}$$

which corresponds to "$(\texttt{eax}' = \texttt{eax}) \wedge (2^{24}\texttt{ebx}' = 2^{24}\texttt{ebx}) \wedge (\texttt{ecx}' = \texttt{ecx})$*". Both transformers are over-approximations of the instruction's semantics, but the extra conjunct* $(2^{24}\texttt{ebx}' = 2^{24}\texttt{ebx})$ *in the KS element captures the fact that the low-order byte of* `ebx` *is not changed by executing "*`add bh,al`*".*

In contrast, $\hat{\alpha}_{KS}(\varphi_\iota)$*, the most-precise over-approximation of* φ_ι *that can be expressed as a KS element is (the Howellization of)*

$$\begin{array}{c} \texttt{eax} \quad \texttt{ebx} \quad \texttt{ecx} \quad \texttt{eax}' \quad \texttt{ebx}' \quad \texttt{ecx}' \quad 1 \\ \left[\begin{array}{cccccc|c} 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 2^{24} & 2^{16} & 0 & 0 & -2^{16} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{array}\right], \end{array}$$

which corresponds to "$(\texttt{eax'} = \texttt{eax}) \wedge (2^{16}\texttt{ebx'} = 2^{16}\texttt{ebx} + 2^{24}\texttt{eax}) \wedge (\texttt{ecx'} = \texttt{ecx})$*". Multiplying by a power of 2 serves to shift bits to the left; because it is performed in arithmetic mod* 2^{32}*, bits shifted off the left end are unconstrained. Thus, the second conjunct captures the relationship between the low-order two bytes of* `ebx'`*, the low-order two bytes of* `ebx`*, and the low-order byte of* `eax`*.* □

3.5.3 Interprocedural Analysis

Fig. 3.7 shows the times for WPDS construction (including constructing the weights that serve as abstract transformers) and performing interprocedural dataflow analysis by performing post* and path summary. Columns 11 and 15 of Fig. 3.7 show the number of $\hat{\alpha}$ calls for which weight generation timed out during $\hat{\alpha}_{\text{KS}}$ and $\hat{\alpha}^{+}_{\text{KS}}$, respectively. During WPDS construction, if Yices times out during $\hat{\alpha}_{\text{KS}}$ or $\hat{\alpha}^{+}_{\text{KS}}$, the implementation uses a weight that is less precise than the best transformer, but it always uses a weight that is at least as precise as the weight obtained using KS-reinterpretation.[4] The number of WPDS rules is given in Fig. 3.3; a timeout occurred for about 1.0% of the $\hat{\alpha}_{\text{KS}}$ calls (computed as a geometric mean[5]), and for about 1.65%

[4]This footnote explains more precisely how weights were constructed in the $\hat{\alpha}_{\text{KS}}$ runs. We used the following "chained" method for generating weights:

1. KS-reinterpretation is the method of §2.3.1.
2. "Stålmarck" is the generalized-Stålmarck algorithm of [101], *starting with the element obtained via KS-reinterpretation*. The generalized-Stålmarck algorithm successively over-approximates the best transformer from above. By starting the algorithm with the element obtained via KS-reinterpretation, the generalized-Stålmarck algorithm does not have to work its way down from $\top$; it merely continues to work its way down from the over-approximation already obtained via KS-reinterpretation.
3. $\hat{\alpha}_{\text{KS}}$ is actually $\hat{\alpha}^{\updownarrow}_{\text{TER[KS]}}$, from Fig. 3.1(b), which maintains both an under-approximation and a (nontrivial) over-approximation of the desired answer, and hence is resilient to timeouts—i.e., it returns the over-approximation if a timeout occurs. In the chained method for generating weights, $\hat{\alpha}^{\updownarrow}_{\text{TER[KS]}}$ is *started with the element obtained via the Stålmarck method as an over-approximation* as a way to accelerate its performance.

The generalized-Stålmarck algorithm is a faster algorithm than $\hat{\alpha}^{\updownarrow}_{\text{TER[KS]}}$, but is not guaranteed to find the best abstract transformer [101]. $\hat{\alpha}^{\updownarrow}_{\text{TER[KS]}}$ *is* guaranteed to obtain the best abstract transformer, except for cases in which an SMT solver timeout is reported. The use of KS-reinterpretation accelerates Stålmarck, and the use of Stålmarck accelerates $\hat{\alpha}^{\updownarrow}_{\text{TER[KS]}}$. Moreover, $\hat{\alpha}^{\updownarrow}_{\text{TER[KS]}} \sqsubseteq$ KS-reinterpretation is always guaranteed to hold for the weights that are computed.

[5]We use "computed as a geometric mean" as a shorthand for "computed by converting the data to ratios; finding the geometric mean of the ratios; and converting the result back to a percentage". For instance, suppose that you have improvements of 3%, 17%, 29% (i.e., .03, .17, and .29). The geometric mean of the values .03, .17, and .29 is .113.

Prog. name	Performance of Interprocedural Analysis													
	MOS-reinterp			KS-reinterp			$\widehat{\alpha}_{KS}$				$\widehat{\alpha}^{+}_{KS}$			
	WPDS	post*	query at branch points	WPDS	post*	query at branch points	WPDS	post*	query at branch points	t/o	WPDS	post*	query at branch points	t/o
write	0.088	0.073	0.091	0.58	0.063	0.257	59.167	0.067	2.681	2(1.17%)	97.732	0.066	6.047	2(1.17%)
finger	0.25	0.889	0.275	1.405	0.192	0.366	80.294	0.208	3.61	3(0.78%)	272.939	0.178	6.217	7(1.82%)
subst	0.38	0.895	0.257	2.045	0.412	0.535	164.033	0.427	5.266	2(0.29%)	261.865	0.433	8.775	4(0.57%)
chkdsk	0.472	0.187	0.314	2.577	0.261	0.831	359.949	0.267	61.907	8(0.81%)	349.519	0.275	26.828	11(1.11%)
convert	0.672	1.643	0.7	3.481	0.647	1.091	206.25	0.629	22.409	17(1.43%)	317.142	0.627	30.959	19(1.60%)
route	1.144	3.581	1.413	6.393	0.82	2.073	396.732	0.959	87.864	4(0.32%)	704.816	0.902	75.348	28(2.27%)
comp	0.8	2.916	1.112	4.55	0.738	1.606	599.272	0.835	41.013	6(0.39%)	912.319	0.806	54.349	9(0.59%)
logoff	1.111	4.858	1.936	7.843	1.01	2.908	449.851	1.061	57.276	24(1.58%)	915.493	1.033	141.981	32(2.10%)
setup	2.054	3.259	1.907	13.598	0.521	5.731	1191.37	0.591	277.8	58(2.28%)	1852.79	0.599	464.781	93(3.65%)

Figure 3.7: Performance of WPDS-based interprocedural analysis. The times, in seconds, for WPDS construction, performing interprocedural dataflow analysis (i.e., running post* and performing path-summary) and finding one-vocabulary affine relations at branch instructions, using MOS-reinterpretation, KS-reinterpretation, $\widehat{\alpha}_{KS}$, and $\widehat{\alpha}^{+}_{KS}$ to generate weights. The columns labeled "t/o" report the number of WPDS rules for which weight generation timed out during symbolic abstraction.

of the $\hat{\alpha}^{+}_{\mathrm{KS}}$ calls.

The experiment showed that the cost of constructing weights via $\hat{\alpha}_{\mathrm{KS}}$ is high, which was to be expected because $\hat{\alpha}_{\mathrm{KS}}$ repeatedly calls an SMT solver. Creating KS weights via $\hat{\alpha}_{\mathrm{KS}}$ is about 81.5 times slower than creating them via KS-reinterpretation (computed as the geometric mean of the construction-time ratios).

Moreover, creating KS weights via KS-reinterpretation is itself 5.9 times slower than creating MOS weights using MOS-reinterpretation. The latter number is different from the 14.9-fold slowdown reported in §3.5.2 for two reasons: (i) §3.5.2 reported the cost of creating KS and MOS abstract transformers for individual instructions, whereas in Fig. 3.7 the transformers are for basic blocks, and (ii) the WPDS construction times in Fig. 3.7 include the cost of creating merge functions for use at procedure-exit sites, which was about the same for KS-reinterpretation and MOS-reinterpretation.

A comparison of the $\hat{\alpha}_{\mathrm{KS}}$ columns of Fig. 3.7 against the $\hat{\alpha}^{+}_{\mathrm{KS}}$ columns reveals that

- Creating KS weights via $\hat{\alpha}^{+}_{\mathrm{KS}}$ is about 1.7 times slower than creating weights via $\hat{\alpha}_{\mathrm{KS}}$ (computed as the geometric mean of the construction-time ratios). The slowdown occurs because the formula created for use by $\hat{\alpha}^{+}_{\mathrm{KS}}$ is more complicated than the one created for use by $\hat{\alpha}_{\mathrm{KS}}$: the former contains additional conjuncts that capture the effects of memory-access/update and flag-access/update operations.
- The times for performing "post*" and "path summary" are almost the same for both methods, because these phases do not involve any calls to the respective $\hat{\alpha}$ procedures.
- Answering queries at branch points was 1.4 times slower for $\hat{\alpha}^{+}_{\mathrm{KS}}$ compared to $\hat{\alpha}_{\mathrm{KS}}$. The reason for the slowdown is that this phase must

Instead, we express the original improvements as ratios and take the geometric mean of 1.03, 1.17, and 1.29, obtaining 1.158. We subtract 1, convert to a percentage, and report "15.8% improvement (computed as a geometric mean)".

The advantage of this approach is that it handles datasets that include one or more instances of 0% improvement, as well as negative percentage improvements.

Prog. name	Δ_1 rules Rules	WPDS Weights MOS-reinterp < KS-reinterp	KS-reinterp < MOS-reinterp	$\widehat{\alpha}_{KS}$ < KS-reinterp	$\widehat{\alpha}^{+}_{KS}$ < $\widehat{\alpha}_{KS}$
write	151	0(0.00%)	0(0.00%)	11(7.28%)	0(0.00%)
finger	353	0(0.00%)	0(0.00%)	29(8.22%)	1(0.28%)
subst	728	0(0.00%)	0(0.00%)	59(8.10%)	0(0.00%)
chkdsk	887	0(0.00%)	0(0.00%)	86(9.70%)	1(0.11%)
convert	1266	0(0.00%)	2(0.16%)	131(10.35%)	0(0.00%)
route	1368	0(0.00%)	3(0.22%)	142(10.38%)	0(0.00%)
comp	1528	0(0.00%)	1(0.07%)	163(10.67%)	0(0.00%)
logoff	1648	0(0.00%)	4(0.24%)	191(11.59%)	1(0.06%)
setup	2847	0(0.00%)	20(0.70%)	432(15.17%)	8(0.28%)

Figure 3.8: Comparison of the precision of the WPDS weights computed using MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{KS}$. (E.g., KS-reinterp < MOS-reinterp reports the number of rules for which the KS-reinterp weight was more precise than the MOS-reinterp weight.)

call the respective $\widehat{\alpha}$ procedures once for each branch point: "post*" and "path summary" return the weight that holds at the *beginning* of a basic block $B = [\iota_1, \ldots, \iota_m]$. To obtain the one-vocabulary affine relation that hold just before branch point ι_m at the end of B, we need to perform an additional $\widehat{\alpha}$ computation for $[\iota_1, \ldots, \iota_{m-1}]$ (i.e., for B, but *without* the branch instruction at the end of B).

Figs. 3.8, 3.9, and 3.10 present three studies that compare the precision obtained via MOS-reinterpretation, KS-reinterpretation, $\widehat{\alpha}_{KS}$, and $\widehat{\alpha}^{+}_{KS}$.

Fig. 3.8 compares the precision of the WPDS weights computed by the different methods for each of the example programs. It shows that $\widehat{\alpha}_{KS}$ creates strictly more precise weights than KS-reinterpretation for about 10.14% of the WPDS rules (computed as a geometric mean). The "$\widehat{\alpha}_{KS}$ < KS-reinterp" column of Fig. 3.8 is particularly interesting in light of the fact that a study of relative precision of abstract transformers created for *individual* instructions via KS-reinterpretation and $\widehat{\alpha}_{KS}$ [62, §5.4.1], reported that $\widehat{\alpha}_{KS}$ creates strictly more precise transformers than KS-reinterpretation for only about 3.2% of the instructions that occur in the corpus of 19,066

Prog. name	Branches	1-Vocabulary Affine Relations at Branch Points			
		MOS-reinterp < KS-reinterp	KS-reinterp < MOS-reinterp	$\widehat{\alpha}_{KS}$ < KS-reinterp	$\widehat{\alpha}^{+}_{KS}$ < $\widehat{\alpha}_{KS}$
write	26	0(0.00%)	0(0.00%)	4(15.38%)	0(0.00%)
finger	48	0(0.00%)	0(0.00%)	14(29.17%)	32(66.67%)
subst	74	1(1.35%)	0(0.00%)	15(20.27%)	0(0.00%)
chkdsk	119	0(0.00%)	0(0.00%)	13(10.92%)	0(0.00%)
convert	161	1(0.62%)	0(0.00%)	49(30.43%)	0(0.00%)
route	243	0(0.00%)	4(1.65%)	63(25.93%)	0(0.00%)
comp	224	0(0.00%)	0(0.00%)	7(3.12%)	0(0.00%)
logoff	306	0(0.00%)	0(0.00%)	91(29.74%)	20(6.54%)
setup	589	0(0.00%)	0(0.00%)	39(6.62%)	0(0.00%)

Figure 3.9: Comparison of the precision of the one-vocabulary affine relations identified to hold at branch points via interprocedural analysis, using weights created using MOS-reinterpretation, KS-reinterpretation, and $\widehat{\alpha}_{KS}$. (E.g., KS-reinterp < MOS-reinterp reports the number of branch points at which the KS-reinterp results were more precise than the MOS-reinterp results.)

instructions from §3.5.2. The numbers in Fig. 3.8 differ from that study in two ways: (i) Fig. 3.8 compares the precision of abstract transformers for basic blocks rather than for individual instructions; and (ii) Fig. 3.8 is a comparison for the instructions that appear in specific programs, whereas the corpus of 19,066 instructions used in the per-instruction study from [62, §5.4.1] was created using a randomized instruction generator.

$\widehat{\alpha}^{+}_{KS}$ creates strictly more precise weights than $\widehat{\alpha}_{KS}$ for only about 0.1% of the WPDS rules (computed as a geometric mean). Improvements are obtained in only four of the nine programs (`finger`, `chkdsk`, `logoff`, and `setup`).

Figs. 3.9 and 3.10 answer question (3) posed at the beginning of this section:

> For what percentage of program points does $\widehat{\alpha}_{KS}$ produce more precise answers than KS-reinterpretation and MOS-reinterpretation?

Prog. name	Procs	2-Vocabulary Procedure Summaries			
		MOS-reinterp < KS-reinterp	KS-reinterp < MOS-reinterp	$\hat{\alpha}_{KS}$ < KS-reinterp	$\hat{\alpha}^{+}_{KS}$ < $\hat{\alpha}_{KS}$
write	10	0(0.00%)	0(0.00%)	5(50.00%)	0(0.00%)
finger	18	0(0.00%)	0(0.00%)	10(55.56%)	2(11.11%)
subst	16	0(0.00%)	0(0.00%)	6(37.50%)	0(0.00%)
chkdsk	18	0(0.00%)	0(0.00%)	9(50.00%)	0(0.00%)
convert	38	0(0.00%)	0(0.00%)	8(21.05%)	0(0.00%)
route	40	0(0.00%)	0(0.00%)	18(45.00%)	0(0.00%)
comp	35	1(2.86%)	0(0.00%)	13(37.14%)	0(0.00%)
logoff	46	0(0.00%)	0(0.00%)	14(30.43%)	1(2.17%)
setup	67	0(0.00%)	1(1.49%)	40(59.70%)	0(0.00%)

Figure 3.10: Comparison of the precision of the two-vocabulary affine relations identified to hold at procedure-exit points via interprocedural analysis, using weights created using MOS-reinterpretation, KS-reinterpretation, and $\hat{\alpha}_{KS}$. (E.g., KS-reinterp < MOS-reinterp reports the number of procedure-exit points at which the KS-reinterp results were more precise than the MOS-reinterp results.)

Figs. 3.9 and 3.10 summarize the results obtained from comparing the precision of the affine relations identified via interprocedural analysis using the different weights.[6]

Compared to runs based on either KS-reinterpretation or MOS-reinterpretation, the analysis runs based on $\hat{\alpha}_{KS}$ weights identified more precise affine relations at a substantial number of points (for both one-vocabulary affine relations that hold at branch points—Fig. 3.9, col. 5—and two-vocabulary affine relations that hold at procedure-exit points—Fig. 3.10, col. 5). For one-vocabulary affine relations, the $\hat{\alpha}_{KS}$ analysis results are strictly better than the KS-reinterpretation results at 18.6% of all branch points (computed as a geometric mean). For two-vocabulary

[6]Register `eip` is the x86 instruction pointer. There are some situations that cause the MOS-reinterpretation weights and KS-reinterpretation weights to fail to capture the value of the post-state `eip` value. Therefore, before comparing affine relations, we performed `havoc(eip')`. This adjustment avoids biasing the results merely because of trivial affine relations of the form "`eip'` = *constant*".

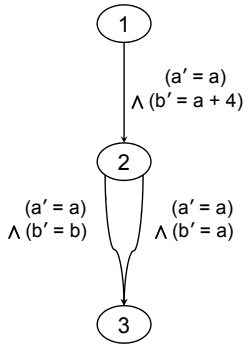


Figure 3.11: Simplified version of an example that caused KS results to be less precise than MOS results, due to compose not distributing over join in the KS domain.

affine relations describing procedure summaries, the $\hat{\alpha}_{\text{KS}}$ analysis results are strictly better than the KS-reinterpretation results at 42% of all procedures (computed as a geometric mean).

For one-vocabulary affine relations, the $\hat{\alpha}^{+}_{\text{KS}}$ analysis results are strictly better than the $\hat{\alpha}_{\text{KS}}$ results at 7.3% of all branch points (computed as a geometric mean). For two-vocabulary affine relations describing procedure summaries, the $\hat{\alpha}^{+}_{\text{KS}}$ analysis results are strictly better than the $\hat{\alpha}_{\text{KS}}$ results at 1.4% of all procedures (computed as a geometric mean). However, in both cases improvements are obtained in only two of the nine programs (`finger` and `logoff`).

3.5.4 Imprecision Due to Non-Distributivity of KS

Fig. 3.9 shows that in a couple of cases, one in `subst` and the other in `convert`, the MOS-reinterpretation results were better than the KS-reinterpretation results. (Although not shown in Fig. 3.9, the MOS-reinterpretation results were also better than the $\hat{\alpha}_{\text{KS}}$ results in the two cases.) We examined these cases, and found that they were an artifact of (i) the evaluation order chosen, and (ii) compose failing to distribute over

join in the KS domain (see §3.4.4).

Fig. 3.11 is a simplified version of the actual transformers in `subst` and `convert` that caused the KS-based analyses to return a less-precise element than the MOS-based analysis. In particular, if the join of the transformers on the two edges from node 2 to node 3 is performed before the composition of the individual $2 \to 3$ transformers with the $1 \to 2$ transformer, the combined $2 \to 3$ KS transformer is "$a' = a$" (i.e., b and b' are unconstrained). The loss of information about b and b' cascades to the $1 \to 3$ KS transformer, which is also "$a' = a$". In particular, it fails to contain the conjunct $2^{30}b' = 2^{30}a$, which expresses that the two low-order bits of b' at node 3 are the same as the two low-order bits of a at node 1.

In contrast, in the MOS domain, the combined $2 \to 3$ transformer is the affine closure of the transformers "$a' = a \wedge b' = b$" and "$a' = a \wedge b' = a$", which avoids the complete loss of information about b and b', and hence the $1 \to 3$ MOS transformer is able to capture the relation "$a' = a \wedge 2^{30}b' = 2^{30}a$".

3.6 Related Work

3.6.1 Abstract Domains for Affine-Relation Analysis

The original work on affine-relation analysis (ARA) was an intraprocedural ARA algorithm due to [49]. In Karr's work, a domain element represents a set of points that satisfy affine relations over variables that hold elements of a *field*. Karr's algorithms are based on linear algebra (i.e., vector spaces).

[75] gave an algorithm for interprocedural ARA, again for vector spaces over a field. Later 2005, 2007, they generalized their techniques to work for modular arithmetic: they introduced the MOS domain, in which an element represents an affine-closed set of affine transformers over variables that hold machine integers, and gave an algorithm for interprocedural

ARA. The algorithms for operations of the MOS domain are based on an extension of linear algebra to modules over a ring.

The version of the KS domain presented in this paper was inspired by, but is somewhat different from, the techniques described in two papers by [52, 53]. Our goals and theirs are similar, namely, to be able to create abstract transformers automatically that are bit-precise, modulo the inherent limitation on precision that stems from having to work with affine-closed sets of values. Compared with their work, we avoid the use of bit-blasting, and work directly with representations of w-bit affine-closed sets. The greatly reduced number of variables that comes from working at word level opened up the possibility of applying our methods to much larger problems, and as discussed in §3.4 and §3.5, we were able to apply our methods to interprocedural program analysis.

As shown in §3.4.2, the algorithm for projection given by [52, §3] does not always find answers that are as precise as the domain is capable of representing. One consequence is that their join algorithm does not always find the least upper bound of its two arguments. In this paper, these issues have been corrected by employing the Howell form of matrices to normalize KS elements (§3.1.2, §3.4.2, and §3.4.4; see also §3.6.2 below).

King and Søndergaard introduced another interesting technique that we did not explore, which is to use affine relations over m-bit numbers, for $m > 1$, to represent sets of 1-bit numbers. To make this approach sensible, their concretization function intersects the "natural" concretization, which yields an affine-closed set of tuples of m-bit numbers, with the set $\{\langle v_1, \ldots, v_k \rangle \mid v_1, \ldots, v_k \in \{0, 1\}\}$. In essence, this approach restricts the concretization to tuples of 1-bit numbers [53, Defn. 2]. The advantage of the approach is that KS elements over $\mathbb{Z}_{2^m}^k$ can then represent sets of 1-bit numbers that can only be over-approximated using KS elements over $\mathbb{Z}_2^k$.

Example 3.20. *Suppose that we have three variables* $\{v_1, v_2, v_3\}$*, and want to represent the set of assignments* $\{\langle 001 \rangle, \langle 010 \rangle, \langle 100 \rangle\}$*. The best KS element over*

$\mathbb{Z}_2^3$ *is*

$$\left[\begin{array}{ccc|c} \overset{v_1}{1} & \overset{v_2}{1} & \overset{v_3}{1} & \overset{1}{1} \end{array}\right], \tag{3.9}$$

which corresponds to the affine relation $v_1 + v_2 + v_3 + 1 = 0$*. The set of satisfying assignments is* $\{\langle 001\rangle, \langle 010\rangle, \langle 100\rangle, \langle 111\rangle\}$*, which includes the extra tuple* $\langle 111\rangle$*.*

Now consider what sets can be represented when $m = 2$*, so that arithmetic is performed mod 4. In particular, instead of the matrix in Eqn. (3.9) we use the following KS element over* $\mathbb{Z}_4^3$*:*

$$\left[\begin{array}{ccc|c} \overset{v_1}{1} & \overset{v_2}{1} & \overset{v_3}{1} & \overset{1}{3} \end{array}\right], \tag{3.10}$$

which corresponds to the affine relation $v_1 + v_2 + v_3 + 3 = 0$*. The matrix in Eqn. (3.10) has sixteen satisfying assignments:*

$$\begin{array}{cccccccc} \langle 001\rangle & \langle 010\rangle & \langle 100\rangle & \langle 122\rangle & \langle 212\rangle & \langle 221\rangle & \langle 032\rangle & \langle 023\rangle \\ \langle 203\rangle & \langle 230\rangle & \langle 302\rangle & \langle 320\rangle & \langle 113\rangle & \langle 131\rangle & \langle 311\rangle & \langle 333\rangle \end{array}$$

However, only three of the assignments are in the restricted concretization, namely, the desired set $\{\langle 001\rangle, \langle 010\rangle, \langle 100\rangle\}$*.* □

To represent sets of tuples of w-bit numbers, as considered in this paper, the analogous technique would use a y-bit KS domain, $y > w$, in a similar fashion. That is, the "natural" concretization would be intersected with the set $\{\langle v_1, \ldots, v_k\rangle \mid v_1, \ldots, v_k \in \{0, 1, \ldots, 2^w - 1\}\}$. We leave the exploration of these issues for possible future research.

Like some other relational domains, including polyhedra [23, 3, 45] and grids [2], KS/AG fits the dual-representation paradigm of having both a constraint representation (KS) and a generator representation (AG). MOS is based on a generator representation. Whereas many implemen-

tations of domains with a dual representation perform some operations in one representation and other operations in the other representation, converting between representations as necessary, one of the clever aspects of both MOS and KS is that they avoid the need to convert between representations.

[39] describes congruence lattices, where the lattice elements are cosets in the group $\mathbb{Z}^k$. The generator form for a congruence-lattice element is defined by a point and a basis. The basis is used to describe the coset. The corresponding constraint form for a domain element is a system of Diophantine linear congruence equations. Conversion from the "normalized representation" (generator form) to an equation system is done by an elimination algorithm. The reverse direction is carried out by solving a set of equations. Granger's normalized representation can be used as a domain for representing affine relations over machine integers. However, Granger's approach does not have unique normalized representations, because a coset space can have multiple bases. As a result, his method for checking that two domain elements are equal is to check containment in both directions (i.e., to perform two containment checks). Moreover, checking containment is costly because a representation conversion is required: one has to compare the cosets, which involves converting one coset into equational form and then checking if the other coset (in generator form) satisfies the constraints of equational form. In contrast, for the KS domain, one can easily check containment using the KS meet and equality operations:

$$\gamma(X) \subseteq \gamma(Y) \quad \text{iff} \quad X \sqsubseteq Y \quad \text{iff} \quad X = (X \sqcap Y).$$

Similarly, containment checking in AG can be performed using the AG join and equality operations. Thus, in both KS and AG, the costly step of converting between generator form and constraint form (or vice versa) is avoided.

Gulwani and Necula introduced the technique of random interpretation (for vector spaces over a field), and applied it to identifying both intraprocedural 2003 and interprocedural 2005 affine relations. The fact that random interpretation involves collecting samples—which are similar to rows of AG elements—suggests that the AG domain might be used as an efficient abstract datatype for storing and manipulating data during random interpretation. Because the AG domain is equivalent to the KS domain (see §3.2), the KS domain would be an alternative abstract datatype for storing and manipulating data during random interpretation.

3.6.2 Howell Form

In contrast with both the Müller-Olm/Seidl work and the King/Søndergaard work, our work takes advantage of the Howell form of matrices. Howell form can be used with each of the domains KS, AG, and MOS defined in §3.1. Because Howell form is canonical for non-empty sets of basis vectors, it provides a way to test pairs of elements for equality of their concretizations—an operation needed by analysis algorithms to determine when a fixed point is reached. In contrast, [78, §2] and [52, Fig. 1], [53, Fig. 2] use "echelon form" (called "triangular form" by King and Søndergaard), which is not canonical.

The algorithms given by Müller-Olm and Seidl avoid computing multiplicative inverses, which are needed to put a matrix in Howell form (line 8 of Alg. 3). However, their preference for algorithms that avoid inverses was originally motivated by the fact that at the time of their original 2005 work they were unaware [77] of Warren's $O(\log w)$ algorithms [102, §10-15] for computing the inverse of an odd element, and only knew of an $O(w)$ algorithm [76, Lemma 1].

3.6.3 Symbolic Abstraction for Affine-Relation Analysis

[52], 2010 defined the KS domain, and used it to create implementations of best KS transformers for the individual bits of a bit-blasted concrete semantics. They used bit-blasting to express a bit-precise concrete semantics for a statement or basic block. The use of bit-blasting let them track the effect of non-linear bit-twiddling operations, such as shift and xor.

In this paper, we also work with a bit-precise concrete semantics; however, we avoid the need for bit-blasting by working with QFBV formulas expressed in terms of word-level operations; such formulas also capture the precise bit-level semantics of each instruction or basic block. We take advantage of the ability of an SMT solver to decide the satisfiability of such formulas, and use $\widehat{\alpha}_{KS}$ to create best word-level transformers.

Prior to our SAS 2011 paper [29], it was not known how to perform $\widetilde{\alpha}_{MOS}(\varphi)$ in a non-trivial fashion (other than defining $\widetilde{\alpha}_{MOS}$ to be $\lambda f.\top$). The fact that [53, Fig. 2] had been able to devise an algorithm for $\widehat{\alpha}_{KS}$ caused us to look more closely at the relationship between MOS and KS. The results presented in §3.3.1 establish that MOS and KS are different, incomparable abstract domains. We were able to give sound interconversion methods (§3.3.2–§3.3.4), and thereby obtained a method for performing $\widetilde{\alpha}_{MOS}(\varphi)$ (§3.3.5).

3.7 Chapter Notes

A more detailed discussion of this project can be found in our journal paper [30]. Matt Elder and Thomas Reps developed the improvement to the symbolic abstraction algorithm (§3.4.8). They also designed and implemented the KS reinterpretation. A detailed discussion of KS reinterpretation and its associated implementation can be found in [30, §6]. Junghee Lim and Thomas Reps developed the TSL infrastructure [62] on which our implementation is based on. They also implemented an earlier

version of the MOS domain, which I modified so that Howell matrices are now used as the underlying representation. Junghee Lim, Tycho Andersen, and Thomas Reps helped me in the setup and implementation of the analysis infrastructure used in the experiments.

4 A NEW ABSTRACTION FRAMEWORK FOR AFFINE TRANSFORMERS

This chapter addresses the problem of abstracting a set of affine transformers $\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}$, where $\overrightarrow{v}$ and $\overrightarrow{v}'$ represent the pre-state and post-state, respectively. C is the linear component of the transformation and $\overrightarrow{d}$ is a constant vector. For example, $\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 2 & 0 \end{bmatrix} + \begin{bmatrix} 10 & 0 \end{bmatrix}$ denotes the affine transformation $(x' = x + 2y + 10 \wedge y' = 0)$ over variables $\{x, y\}$. We denote an affine transformation by $C : \overrightarrow{d}$.

We introduce a framework to harness any base abstract domain $\mathcal{B}$ in an abstract domain of affine transformations. Abstract domains are usually used to define constraints on the variables of a program. In this work, however, abstract domain $\mathcal{B}$ is re-purposed to constrain the elements of C and $\overrightarrow{d}$—thereby defining a set of affine transformers on program states. This framework facilitates intra- and interprocedural analyses to obtain function and loop summaries, as well as to prove program assertions.

We provide analysis techniques to abstract the behavior of the program as a set of affine transformations over bit-vectors. This work is based on the following observation:

Observation 1. *Abstract domains are usually used to define constraints on the variables of a program. However, they can be re-purposed to constrain the elements of* $C : \overrightarrow{d}$*—thereby defining a set of affine transformers on program states.*

Chapter 3 compared two abstract domains for affine-relation analysis over bitvectors: (i) an affine-closed abstraction of relations over program variables (AG/KS), and (ii) an affine-closed abstraction of affine transformers over program variables (MOS). We observed that the MOS domain can encode two-vocabulary relations that are not affine-closed even though the affine transformers themselves are affine closed. (See §3.3.1 for an

example.) Thus, moving the abstraction from affine relations over program variables to affine relations over affine transformations possibly offers some advantages because it allows some non-affine-closed sets to be representable.

Problem Statement. Our goal is to generalize the ideas used in the MOS domain—in particular, to have an abstraction of *sets of affine transformers*—but to provide a way for a client of the abstract domain to have some control over the performance/precision trade-off. Toward this end, we define a new family of numerical abstract domains, denoted by ATA[$\mathcal{B}$]. (ATA stands for Affine-Transformers Abstraction.) Following Obs. 1, ATA[$\mathcal{B}$] is parameterized by a base numerical abstract domain $\mathcal{B}$, and allows one to represent a set of affine transformers (or, alternatively, certain disjunctions of transition formulas).

Summary of the Approach. Let the $(k + k^2)$-tuple $(d_1, d_2, \ldots, d_k, c_{11}, c_{12}, \ldots, c_{1k}, c_{21}, c_{22}, ..., c_{kk})$ denote the affine transformation $\bigwedge_{j=1}^{k} \left(v'_j = \sum_{i=1}^{k} (c_{ij} v_i) + d_j \right)$, also written as "$C : \overrightarrow{d}$." The key idea is that we will use $(k + k^2)$ symbolic constants to represent the $(k + k^2)$ coefficients in a transformation of the form $C : \overrightarrow{d}$, and use a base abstract domain $\mathcal{B}$—provided by a client of the framework—to represent *sets* of possible values for these symbolic constants. In particular, $\mathcal{B}$ is an abstract domain for which, for all $b \in \mathcal{B}$, $\gamma(b)$ is a set of $(k + k^2)$-tuples—each tuple of which provides values for $\{d_i\} \cup \{c_{ij}\}$, and can thus be interpreted as an affine transformation $C : \overrightarrow{d}$.

With this approach, a given $b \in \mathcal{B}$ represents the disjunction $\bigvee\{(C : \overrightarrow{d}) \in \gamma(b)\}$. When $\mathcal{B}$ is a non-relational domain, each $b \in \mathcal{B}$ constrains the values of $\{d_i\} \cup \{c_{ij}\}$ *independently*. When $\mathcal{B}$ is a relational domain, each $b \in \mathcal{B}$ can impose *intra-component* constraints on the allowed tuples $(d_1, d_2, \ldots, d_k, c_{11}, c_{12}, \ldots, c_{1k}, c_{21}, c_{22}, \ldots, c_{kk})$.

ATA[$\mathcal{B}$] generalizes the MOS domain, in the sense that the MOS domain is exactly ATA[AG/KS], where AG/KS is a relational abstract domain that captures affine equalities of the form $\sum_i a_i k_i = b$, where $a_i, b \in Z_{2^w}$ and $\mathbb{Z}_{2^w}$ is the set of w-bit bitvectors (see §3.1.3 and §3.1.4). For instance, an element in ATA[AG] can capture the set of affine transformers "$x' = k_1 * x + k_1 * y + k_2$, where k_1 is odd, k_2 is even, and k_1 is the coefficient of both x and y." On the other hand, an element in the abstract domain ATA[$\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+k^2)}$], where $\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+k^2)}$ is the abstract domain of $(k + k^2)$-tuples of intervals over bitvectors, can capture a set of affine transformers such as $x' = k_3 * x + k_4 * y + k_5$, where $k_3 \in [0, 1]$, $k_4 \in [2, 2]$, and $k_5 \in [0, 10]$.

This work addresses a wide variety of issues that arise in defining the ATA[$\mathcal{B}$] framework, including describing the abstract-domain operations of ATA[$\mathcal{B}$] in terms of the abstract-domain operations available in the base domain $\mathcal{B}$.

Contributions. The overall contribution of our work is the framework ATA[$\mathcal{B}$], for which we present

- methods to perform basic abstract-domain operations, such as equality and join.
- a method to perform abstract composition, which is needed to perform abstract interpretation.
- a faster method to perform abstract composition when the base domain is non-relational.

Organization This chapter is organized as follows: §4.1 introduces the terminology used in the chapter; and presents some needed background material. §4.2 demonstrates the framework with the help of an example. §4.3 formally introduces the parameterized abstract domain ATA[$\mathcal{B}$]. §4.4 provides discussion and related work. Proofs are given in

4.1 Preliminaries

Matrix addition and multiplication are defined as usual, forming a matrix ring. We denote the transpose of a matrix M by M^t. A *one-vocabulary matrix* is a matrix with $k+1$ columns. A *two-vocabulary matrix* is a matrix with $2k+1$ columns. In each case, the "+1" is related to the fact that we capture affine rather than linear relations. I_n denotes the $n \times n$ identity matrix. Given a matrix C, we use $C[i,j]$ to refer to the entry at the i-th column and j-th row of C. Given a vector $\overrightarrow{d}$, we use $\overrightarrow{d}[j]$ to refer to the j-th entry in $\overrightarrow{d}$.

4.1.1 Affine Programs

$$
\begin{array}{rcl}
\langle \mathit{Block} \rangle & :: & ł : (\langle \mathit{Stmt} \rangle \,\mathbf{;})^* \, \langle \mathit{Next} \rangle \\
\langle \mathit{Next} \rangle & :: & \mathbf{jump}\ ł\mathbf{;} \\
 & | & \mathbf{jump}\ \langle \mathit{Cond} \rangle\ \mathbf{?}\ ł_1\ \mathbf{:}\ ł_2 \\
\langle \mathit{Cond} \rangle & :: & ? \mid \langle \mathit{Expr} \rangle\ \mathit{Op}\ \langle \mathit{Expr} \rangle \\
\langle \mathit{Op} \rangle & :: & = \mid \neq \mid \geqslant \mid \leqslant \\
\langle \mathit{Expr} \rangle & :: & c_0 + \sum_{i=1}^{k} c_i * v_i \\
\langle \mathit{Stmt} \rangle & :: & v_j := \langle \mathit{Expr} \rangle \\
 & | & v_j := ?
\end{array}
$$

We borrow the notion of affine programs from [78]. We restrict our affine programs to consist of a single procedure. The statements are restricted to either affine assignments or non-deterministic assignments. The control-flow instruction consists of either an unconditional jump statement, or a conditional jump with an affine equality, an affine disequality, an affine inequality, or unknown guard condition.

4.1.2 Abstract-Domain Operations

Tab. 4.1 lists the abstract-domain operations needed to generate the program abstraction and perform fixpoint analysis for an affine program. Bottom, equality, and join are standard abstract-domain operations. The *widen* operation is needed for domains with infinite ascending chains to ensure termination. The two operations of the form $\alpha(\mathit{Stmt})$ perform

Table 4.1: Abstract-domain operations.

Type	Operation	Description
$\mathcal{A}$	$\bot$	*bottom element*
bool	$(a_1 == a_2)$	*equality*
$\mathcal{A}$	$(a_1 \sqcup a_2)$	*join*
$\mathcal{A}$	$(a_1 \nabla a_2)$	*widen*
$\mathcal{A}$	Id	*identity element*

Type	Operation	Description
$\mathcal{A}$	$\alpha(v_j :=?)$	*abstraction for nondeterministic assignments*
$\mathcal{A}$	$\alpha(v_j := c_0 + \sum_{i=1}^{k} c_{ij} * v_i)$	*abstraction for affine assignments*
$\mathcal{A}$	$(a_1 \circ a_2)$	*composition*

abstraction on an assignment statement $Stmt$ to generate an abstract transformer. Id is the identity element; which represents the identity transformation $(\bigwedge_{i=1}^{k} v_i' = v_i)$. Finally, the abstract-composition operation $a_1 \circ a_2$ returns a sound overapproximation of the composition of the abstract transformation a_1 with the abstract transformation a_2.

4.1.3 Relating MOS and AG

Since KS and AG are the same domain, we will use AG to refere to KS/AG domain (See §3.2). There are two ways to relate the MOS and AG domains. One way is to use them as abstractions of two-vocabulary relations and provide (approximate) inter-conversion methods. The other is to use a variant of the AG domain to represent the elements of the MOS domain exactly.

Comparison of MOS and AG elements as abstraction of two-vocabulary relations.

As shown in §3.3.1, the MOS and AG domains are incomparable: some relations are expressible in each domain that are not expressible in the other. Intuitively, the central difference is that MOS is a domain of sets of *functions*, while AG is a domain of *relations*.

Soundly converting an MOS element M to an overapproximating AG element is equivalent to stating two-vocabulary affine constraints satisfied by M §3.3.2.

Table 4.2: Example demonstrating two ways of relating MOS and AG.

MOS element (M)	Overapproximating AG element (A_1)	Reformulation as abstraction over affine transformers (A_2)
$\left\{ \left[\begin{array}{c\|cc} 1 & 0 & 0 \\ \hline 0 & 1 & 1 \\ 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c\|cc} 1 & 0 & 0 \\ \hline 0 & 0 & 0 \\ 0 & 1 & 1 \end{array}\right] \right\}$	$\left[\begin{array}{c\|cccc} 1 & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 & 1 \end{array}\right]$	$\left[\begin{array}{c\|cccccccc} 1 & a_{01} & a_{02} & a_{10} & a_{11} & a_{12} & a_{20} & a_{21} & a_{22} \\ \hline 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right]$

Reformulation of MOS elements as AG elements.

An MOS element $M = \{M_1, M_2, ..., M_n\}$ represents the set of $(k+1)\times(k+1)$ matrices in the affine closure of the matrices in M. Each matrix can be thought of as a $(k+1) \times (k+1)$ vector, and hence M can be represented by an AG element of size $n \times ((k+1) \times (k+1))$.

Example 4.1. *Tab. 4.2 shows the two ways MOS and AG elements can be related. Column 1 shows the MOS element* M*, which represents the set of matrices in the affine closure of the two* $(k+1) \times (k+1)$ *matrices, with* $k = 2$*. The second column gives the AG element* A_1 *(a matrix with* $2k+1$ *columns) representing the affine-closed space over* $\{x, y, x', y'\}$ *satisfied by* M*. Consequently,* $\gamma_{AG}(A_1) \supseteq \gamma_{MOS}(M)$*. Column 3 shows the two matrices of* M *as the* $2\times((k+1)\times(k+1))$ *AG element* A_2*. Because* A_2 *is just a reformulation of* M*,* $\gamma_{AG}(A_2) = \gamma_{MOS}(M)$*.* □

4.2 Overview

In this section, we motivate and illustrate the ATA[$\mathcal{B}$] framework, with the help of several examples. The first two examples illustrate the following principle, which restates Obs. 1 more formally:

Observation 2. *Each affine transformation* $C : \overrightarrow{d}$ *in a set of affine transformations involves* $(k+1)^2$ *coefficients* $\in \mathbb{Z}_{2^w} : (1, d_1, d_2, \ldots, d_k, 0, c_{11}, c_{12}, \ldots, 0, c_{21}, ...c_{kk})$.[1] *Thus, we may use any abstract domain whose elements concretize*

[1] k of the coefficients are always 0, and one coefficient is always 1 (i.e., the first column is always $(1|\ 0\ 0\ ...\ 0)^t$). For this reason, we really need only $k + k^2$ elements, but we will sometimes refer to $(k+1)^2$ elements for brevity.

to subsets of $\mathbb{Z}_{2^w}^{(k+1)^2}$ *as a method for representing a set of affine transformers.* □

Example 4.2. *The AG element* A_2 *in column 3 of Tab. 4.2 illustrates how an AG element with* $(k+1)^2$ *columns represents the same set of affine transformers as the MOS element* M *shown in column 1. For instance, the first row of* A_2 *represents the first matrix in* M. □

Example 4.3. *Consider the element* $E = ([1,1],[0,10],[0,0],[0,0],[1,1],[2,3],[0,0],[0,0],[1,1])$ *of* $\mathcal{I}^9_{\mathbb{Z}_{2^w}}$. E *can be depicted more mnemonically as the following matrix:*

$$\left[\begin{array}{c|cc} {\scriptstyle 1} & {\scriptstyle x} & {\scriptstyle y} \\ \hline [1,1] & [0,10] & [0,0] \\ \hline [0,0] & [1,1] & [2,3] \\ [0,0] & [0,0] & [1,1] \end{array}\right]$$

, where every element in E *is an interval* $(\mathcal{I}_{\mathbb{Z}_{2^w}})$. E *represents the point set* $\{(x', y', x, y) : \exists i_1, i_2 \in \mathbb{Z}_{2^w} : x' = x + i_1 \wedge y' = i_2 x + y \wedge 0 \leqslant i_1 \leqslant 10 \wedge 2 \leqslant i_2 \leqslant 3\}$. □

Examples 4.2 and 4.3 both exploit Observation 2, but use different abstract domains. Ex. 4.2 uses the AG domain with $(k+1)^2$ columns, whereas Ex. 4.3 uses the domain $\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$. In particular, an abstract-domain element in our framework ATA[$\mathcal{B}$] is a set of affine transformations $\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}$, such that the allowed coefficients in the matrix C and the vector $\overrightarrow{d}$ are abstracted by a base abstract domain $\mathcal{B}$.

The remainder of this section shows how different instantiations of Observation 2 allow different properties of a program to be recovered.

Example 4.4. *In this example, the variable* r *of function* f *is initialized to* 0 *and conditionally incremented by* 2x *inside a loop with 10 iterations.*

```
ENT: int f(int x) {
L0:    int i = 0, r = 0;
L1:    while(i <= 10) {
L2:      if(*)
L3:        r = r + 2*x;
L4:      i = i + 1;
       }
L5:    return r;
     }
```

The exact function summary for function f*, denoted by* S_f*, is* $(\exists k.r' = 2kx \wedge 0 \leqslant k \leqslant 10)$. *Note that* S_f *expresses two important properties of the function: (i) the return value* r' *is an even multiple of* x*, and (ii) the multiplicative factor is contained in an interval.*

□

$\mathcal{B}$ = **AG with** $(k+1)^2$ **columns:** Fig. 4.1(a) shows the abstract transformers generated with the MOS domain.[2] Each matrix of the form $\left[\begin{array}{c|ccc} 1 & d_1 & d_2 & d_3 \\ \hline 0 & c_{11} & c_{12} & c_{13} \\ 0 & c_{21} & c_{22} & c_{23} \\ 0 & c_{31} & c_{32} & c_{33} \end{array}\right]$ represents the state transformation $(x' = d_1 + c_{11}x + c_{21}i + c_{31}r) \wedge (i' = d_2 + c_{12}x + c_{22}i + c_{32}r) \wedge (r' = d_3 + c_{13}x + c_{23}i + c_{33}r)$.

For instance, the abstract transformer for L3 → L4 is an MOS-domain element with a single matrix that represents the affine transformation: $(x' = x) \wedge (i' = i) \wedge (r' = 2x + r)$. The edges absent from Fig. 4.1(a), e.g., L1 → L2, have the identity MOS-domain element.

Edge	Transformer
L0 → L1	$\left\{\left[\begin{array}{c\|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]\right\}$
L3 → L4	$\left\{\left[\begin{array}{c\|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]\right\}$
L4 → L1	$\left\{\left[\begin{array}{c\|ccc} 1 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]\right\}$

(a)

Iteration	Node L1
(i)	$\left\{\left[\begin{array}{c\|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]\right\}$
(ii)	$\left\{\left[\begin{array}{c\|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c\|ccc} 1 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]\right\}$
(iii)	$\left\{\left[\begin{array}{c\|ccc} 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c\|ccc} 1 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right], \left[\begin{array}{c\|ccc} 1 & 0 & 2 & 0 \\ \hline 0 & 1 & 0 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right]\right\}$

(b)

Figure 4.1: Abstract transformers and snapshots in the fixpoint analysis with the MOS domain for Ex. 4.4.

To obtain function summaries, an iterative fixed-point computation needs to be performed. An abstract-domain element a is a *summary* at some program point L, if it describes a two-vocabulary transition relation that overapproximates the (compound) transition relation from the beginning of the function to program point L. Fig. 4.1(b) provides the iteration results for the summary at the program point L1. After iteration (i), the result represents $(x' = x) \wedge (i' = 0) \wedge (r' = 0)$. After iteration (ii), it adds the affine transformer $(x' = x) \wedge (i' = 1) \wedge (r' = 2x)$ to the summary. Quiescence is discovered on the third iteration because the affine-closure

[2] We will continue to refer to the MOS domain directly, rather than "the instantiation of Observation 2 with an AG element containing $(k+1)^2$ columns" (à la Ex. 4.2).

of the three matrices is the same as the affine-closure of the two matrices after the second iteration. As a result, the function summary that MOS learns, denoted by S_{MOS}, is $\exists k.r' = 2kx$, which is an overapproximation of the exact function summary S_f. Imprecision occurs because the MOS-domain is not able to represent inequality guards. Hence, the summary captures the evenness property, but not the bounds property.

$\underline{\mathcal{B} = \mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}}$**:** By using different $\mathcal{B}$s, an analyzer will be able to recover different properties of a program. Now consider what happens when the program above is analyzed with ATA[$\mathcal{B}$] instantiated with the non-relational base domain of environments of intervals ($\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$). The identity transformation for the abstract domain ATA[$\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$] is $\left[\begin{array}{c|ccc} 1 & [0,0] & [0,0] & [0,0] \\ \hline 0 & [1,1] & [0,0] & [0,0] \\ 0 & [0,0] & [1,1] & [0,0] \\ 0 & [0,0] & [0,0] & [1,1] \end{array}\right]$. The bottom element for the abstract domain ATA[$\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$], denoted by $\bot_{\text{ATA}[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]}$ is $\left[\begin{array}{c|ccc} 1 & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \\ \hline 0 & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \\ 0 & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \\ 0 & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} & \bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \end{array}\right]$.[3]

Fig. 4.2 shows the abstract transformers and the fixpoint analysis for the node L1 with the ATA[$\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$] domain. One advantage of using intervals as the base domain is that they can express inequalities. For instance, the abstract transformer for the edge L1 $\rightarrow$ L2 specifies the transformation $(x' = x) \wedge (0 \leqslant i' \leqslant 10) \wedge (r' = r)$. Consequently, the function summary that ATA[$\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$] learns, denoted by $S_{\text{ATA}[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]}$, is $r' = [0, 20]x$. This summary captures the bounds property, but not the evenness property. Notice that, $S_f = S_{\text{ATA}[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]} \wedge S_{MOS}$.

[3]The abstract domain $\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$ is the product domain of $(k+1)^2$ interval domains, that is, $\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}} = \mathcal{I}_{\mathbb{Z}_{2^w}} \times \mathcal{I}_{\mathbb{Z}_{2^w}} \times \ldots \times \mathcal{I}_{\mathbb{Z}_{2^w}}$. $\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$ uses smash product to maintain a canonical representation for $\bot_{\text{ATA}[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]}$. Thus, if any of the coefficients in an abstract-domain element $b \in \text{ATA}[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ is $\bot_{\mathcal{I}_{\mathbb{Z}_{2^w}}}$, then b is smashed to $\bot_{\text{ATA}[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]}$.

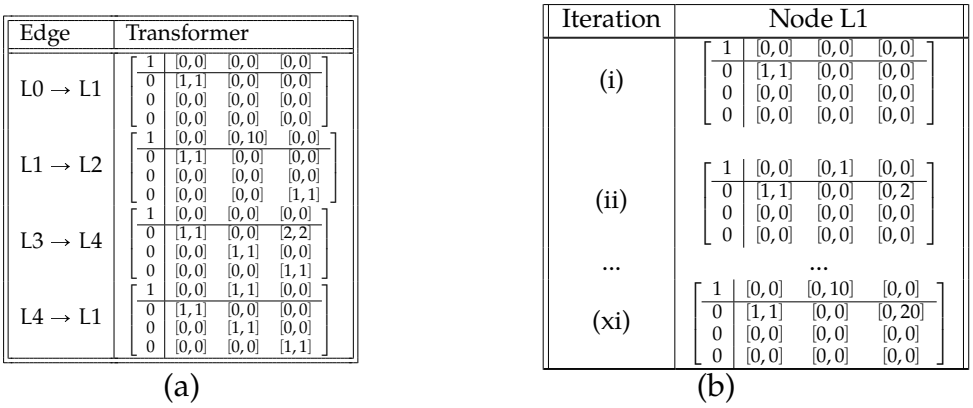


Figure 4.2: Abstract transformers and fixpoint analysis with the ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ domain for Ex. 4.4.

Consider the instantiation of the ATA framework with strided-intervals over bitvectors [83], denoted by $\mathcal{SI}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$. A strided interval represents a set of the form $\{l, l+s, l+2s, ..., l+(n-1)s\}$. Here, l is the beginning of the interval, s is the stride, and n is the interval size. Consequently, ATA$[\mathcal{SI}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ learns the function summary $\exists k.r' = kx \wedge k = 2[0, 10]$, which captures both the evenness property and the bounds property. Note that a traditional (non-ATA-framework) analysis based on the strided-interval domain alone would not be able to capture the desired summary because the strided-interval domain is non-relational.

Widening concerns. In principle, abstract domains $\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$ and $\mathcal{SI}^{(k+1)^2}_{\mathbb{Z}_{2^w}}$ do not need widening operations because the lattice height is finite. However, the height is exponential in the bitwidth w of the program variables, and thus in practice we need widening operations to speed-up the fixpoint iteration. In the presence of widening, neither ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ nor ATA$[\mathcal{SI}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ will be able to capture the bounds property for Ex. 4.4, because they are missing relational information between the loop counter i and the variable r. However, the reduced product of ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ (or

$\mathrm{ATA}[\mathcal{SI}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$) and MOS can learn the exact function summary.

4.3 Affine-Transformer-Abstraction Framework

In this section, we formally introduce the Affine-Transformer-Abstraction framework (ATA) and describe abstract-domain operations for the framework. We also discuss some specific instantiations.

ATA$[\mathcal{B}]$ **Definition.** Let C be a k-by-k matrix: $[c_{ij}]$, where each c_{ij} is a symbolic constant for the entry at i-th row and j-th column. Let $\overrightarrow{d}$ be a k-vector, $[d_i]$, where each d_i is a symbolic constant for the i-th entry in the vector. As mentioned in Chapter 1, an affine transformer, denoted by $C : \overrightarrow{d}$, describes the relation $\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}$, where $\overrightarrow{v}'$ and $\overrightarrow{v}$ are row vectors of size k that represent the post-transformation state and the pre-transformation state, respectively, on program variables.

Given a base abstract domain $\mathcal{B}$, the ATA framework generates a corresponding abstract domain $\mathrm{ATA}[\mathcal{B}]$ whose elements represent a transition relation between the pre-state and the post-state program vocabulary. Each element $a \in \mathrm{ATA}[\mathcal{B}]$ is represented using an element $base(a) \in \mathcal{B}$, such that:

$$\gamma(a) = \{(\overrightarrow{v}, \overrightarrow{v}') \mid \exists (C : \overrightarrow{d}) \in \gamma(base(a)) : \overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d}\}.$$

4.3.1 Abstract-Domain Operations for ATA$[\mathcal{B}]$

In this subsection, we provide all the abstract-domain operations for $\mathrm{ATA}[\mathcal{B}]$, with the exception of abstract composition, which is discussed in §4.3.2.

In the $\mathrm{ATA}[\mathcal{B}]$ framework, the symbolic constants in the base domain $\mathcal{B}$ are denoted by $symbols(C{:}\overrightarrow{d})$, where $symbols(C{:}\overrightarrow{d}) = (d_1, d_2, \ldots, d_n, c_{11}, c_{12}, \ldots, c_{1k}, c_{21}, c_{22}, \ldots, c_{2k}, \ldots, c_{kk})$ is the tuple of $k + k^2$

Table 4.3: Base abstract-domain operations.

Type	Operation	Description
$\mathcal{B}$	$\bot$	*bottom element*
$\mathcal{B}$	$\top$	*top element*
bool	$(b_1 == b_2)$	*equality*
$\mathcal{B}$	$(b_1 \sqcup b_2)$	*join*
$\mathcal{B}$	$(b_1 \nabla b_2)$	*widen*
$\mathcal{B}$	$\mathtt{havoc}(b_1, S)$	*remove all constraints on symbolic constants in* S
$\mathcal{B}$	$\alpha(\mathtt{ct})$	*abstraction for the concrete affine transformer* ct, *where* $\mathtt{ct} \in \mathtt{symbols}(C : \overrightarrow{d}) \to \mathbb{Z}_{2^w}$

symbolic constants in the affine transformation. Tab. 4.3 lists the abstract-domain interface for the base abstract domain $\mathcal{B}$ needed to implement these operations for ATA[$\mathcal{B}$]. The first five operations in the interface are standard abstract-domain operations. $\mathtt{havoc}(b_1, S)$ takes an element b_1 and a subset $S \subseteq \mathtt{symbols}(C{:}\,\overrightarrow{d})$ of symbolic constants, and returns an element without any constraints on the symbolic constants in S. The last operation in Tab. 4.3 defines an abstraction for a concrete affine transformer ct. A concrete affine transformer is a mapping from the symbolic constants in the affine transformer to bitvectors of size w. We represent concrete state ct with the $(k+1) \times (k+1)$ matrix: $\left[\begin{array}{c|cccc} 1 & \mathtt{ct}(d_1) & \mathtt{ct}(d_2) & \ldots & \mathtt{ct}(d_k) \\ \hline 0 & \mathtt{ct}(c_{11}) & \mathtt{ct}(c_{12}) & \ldots & \mathtt{ct}(c_{1k}) \\ 0 & \mathtt{ct}(c_{21}) & \mathtt{ct}(c_{22}) & \ldots & \mathtt{ct}(c_{2k}) \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \mathtt{ct}(c_{k1}) & \mathtt{ct}(c_{k2}) & \ldots & \mathtt{ct}(c_{kk}) \end{array}\right]$, where $\mathtt{ct}(s)$ denotes the concrete value in $\mathbb{Z}_{2^w}$ of symbol s in the concrete state ct.

Tab. 4.4 gives the abstract-domain operations for ATA[$\mathcal{B}$] in terms of the base abstract-domain operations in $\mathcal{B}$. The first operation is the $\bot$ element, which is simply defined as $\bot_{\mathcal{B}}$, the bottom element in the base domain. Similarly, equality, join, and widen operations are defined as the equality, join, and widen operations in the base domain. The equality operation is not the exact equality operation; that is, $(a_1 \widetilde{==} a_2)$ can return false, even if $\gamma(a_1) = \gamma(a_2)$. However, the equality operation is sound; that

Table 4.4: Abstract-domain operations for the ATA[$\mathcal{B}$]-domain.

Type	Operation	Description
$\mathcal{A}$	$\bot$	$\bot_{\mathcal{B}}$
bool	$(a_1 \widetilde{==} a_2)$	$(base(a_1) == base(a_2))$
$\mathcal{A}$	$(a_1 \tilde{\sqcup} a_2)$	$(base(a_1) \sqcup base(a_2))$
$\mathcal{A}$	$(a_1 \nabla a_2)$	$(base(a_1) \nabla base(a_2))$
$\mathcal{A}$	$\alpha(v_j := d_j + \sum_{i=1}^{k} c_{ij} * v_i)$	$\alpha\left(\left[\begin{array}{c\|ccc} 1 & 0 & d_j & 0 \\ \hline 0 & I_{j-1} & [c_{1j}, c_{2j}, ...c_{(j-1)j}]^t & 0 \\ 0 & 0 & c_{jj} & 0 \\ 0 & 0 & [c_{(j+1)j}, c_{(j+2)j}, ...c_{kj}]^t & I_{k-j} \end{array}\right]\right)$
$\mathcal{A}$	$\alpha(v_j :=?)$	$havoc(\alpha(I_{k+1}), \{d_j, c_{1j}, c_{2j}, ..., c_{kj}\})$
$\mathcal{A}$	Id	$\alpha(I_{k+1})$

is, when $(a_1 \widetilde{==} a_2)$ returns true, then $\gamma(a_1) = \gamma(a_2)$. The $\tilde{\sqcup}$ operation for the ATA[$\mathcal{B}$] is a quasi-join operation [34]. In other words, the least upper bound does not necessarily exist for ATA[$\mathcal{B}$], but a sound upper-bound operation $\tilde{\sqcup}$ is available.

The abstraction operation for the affine-assignment statement $\alpha(v_j := d_0 + \sum_{i=1}^{k} c_{ij} * v_i)$ gives back an ATA[$\mathcal{B}$]-element with a single transformer where every variable $v \in V - \{v_j\}$ is left unchanged and the variable v_j is transformed to reflect the assignment by updating the coefficients of the corresponding column. The abstraction operation for the non-deterministic assignment statement $\alpha(v_j :=?)$ gives back an ATA[$\mathcal{B}$]-element, such that every variable $v \in V - \{v_j\}$ is left unchanged but the symbolic constant corresponding to the coefficients in the column j of the affine transformation can be any value. This operation is carried out by performing *havoc* on the identity transformation with respect to the set $\{d_j, c_{1j}, c_{2j}, ..., c_{kj}\}$ of symbolic constants. The identity transformation *Id* is obtained by abstracting the concrete affine transformer ct that represents the identity transformer. We provide proofs of soundness for these abstract-domain operations in App. D.

4.3.2 Abstract Composition

We have shown that all the abstract-domain operations for ATA[$\mathcal{B}$] can be implemented in terms of abstract-domain operations in $\mathcal{B}$, with the exception of abstract composition. Let us consider the composition of two abstract values $a, a' \in \text{ATA}[\mathcal{B}]$, representing the two-vocabulary relations $R[\overrightarrow{v}; \overrightarrow{v}'] = \gamma(a)$ and $R'[\overrightarrow{v}'; \overrightarrow{v}''] = \gamma(a')$. An abstract operation $\circ^\sharp$ is a sound abstract-composition operation if, for all $a'' = a' \circ^\sharp a$, $\gamma(a'') \supseteq \{ (\overrightarrow{v}; \overrightarrow{v}'') \mid \exists \overrightarrow{v}'.R[\overrightarrow{v}; \overrightarrow{v}'] \wedge R'[\overrightarrow{v}'; \overrightarrow{v}''] \}$. This condition translates to:

$$\gamma(base(a'')) \supseteq \{(\overrightarrow{v}, \overrightarrow{v}'') \mid \exists (C : \overrightarrow{d}) \in \gamma(base(a)), (C' : \overrightarrow{d}') \in \gamma(base(a')), \\ (C'' : \overrightarrow{d}'') : (\overrightarrow{v}'' = \overrightarrow{v} \cdot C'' + \overrightarrow{d}'') \wedge (C'' = C \cdot C') \\ \wedge (\overrightarrow{d}'' = \overrightarrow{d} \cdot C' + \overrightarrow{d}')\} \tag{4.1}$$

The presence of the quadratic components $C \cdot C'$ and $\overrightarrow{d} \cdot C'$ makes the implementation of abstract composition non-trivial. One extremely expensive method to implement abstract composition is to enumerate the set of all concrete transformers $(C : \overrightarrow{d}) \in \gamma(base(a))$ and $(C' : \overrightarrow{d}') \in \gamma(base(a'))$, perform matrix multiplication for each pair of concrete transformers, and perform join over all pairs of them. This approach is impractical because the set of all concrete transformers in an abstract value can be very large.

First, we provide a general method to implement abstract composition. Then, we provide methods for abstract composition when the base domain $\mathcal{B}$ has certain properties, like non-relationality and weak convexity. The latter methods are faster, but are only applicable to certain classes of base abstract domains.

General Case.

We present a general method to perform abstract composition by reducing it to the symbolic-abstraction problem. The *symbolic abstraction* of a formula φ in logic $\mathbb{L}$, denoted by $\hat{\alpha}(\varphi)$, is the best value in $\mathcal{B}$ that over-approximates the set of all concrete affine transformers $(C : \overrightarrow{d})$ that satisfy φ [84, 100]. For all $b \in \mathcal{B}$, the *symbolic concretization* of $\mathcal{B}$, denoted by $\hat{\gamma}(b)$, maps b to a formula $\hat{\gamma}(b) \in \mathbb{L}$ such that b and $\hat{\gamma}(b)$ represent the same set of concrete affine transformers (i.e., $\gamma(b) = [\![\hat{\gamma}(b)]\!]$). We expect the base domain $\mathcal{B}$ to provide the $\hat{\gamma}$ operation. In our framework, there are slightly different variants of $\hat{\alpha}$ and $\hat{\gamma}$ according to which vocabulary of symbolic constants are involved. For instance, we use $\hat{\gamma}'$ to denote symbolic concretization in terms of the primed symbolic constants $\mathtt{symbols}(C' : \overrightarrow{d}')$. Similarly, $\hat{\alpha}''$ denotes symbolic abstraction in terms of the double-primed symbolic constants $\mathtt{symbols}(C'' : \overrightarrow{d}'')$. The function *dropPrimes* shifts the vocabulary of symbolic constants by removing the primes from the symbolic constants that an abstract value represents.

We use $\mathbb{L} = \mathrm{QF\_BV}$, i.e., quantifier-free bit-vector logic, to express abstract composition symbolically as follows:

$$\begin{aligned} \mathtt{base}(a'') = \mathit{dropPrimes}\big(\hat{\alpha}''(\varphi)\big), \text{ where} \\ \varphi = (C'' = C \cdot C') \wedge (\overrightarrow{d}'' = \overrightarrow{d} \cdot C' + \overrightarrow{d}') \\ \wedge\, \hat{\gamma}(\mathtt{base}(a)) \wedge \hat{\gamma}'(\mathtt{base}(a')). \end{aligned} \tag{4.2}$$

(Note that $\hat{\gamma}(\mathtt{base}(a))$ and $\hat{\gamma}'(\mathtt{base}(a'))$ are formulas over $\mathtt{symbols}(C : \overrightarrow{d})$ and $\mathtt{symbols}(C' : \overrightarrow{d}')$ respectively.) Past literature [84, 100, 30] provides various algorithms to implement symbolic abstraction. Symbolic-abstraction methods are usually slow because they make repeated calls to an SMT solver. Specifically, the symbolic-abstraction algorithms in [84, 100] require $\mathcal{O}(h'')$ calls to SMT, where h'' is the height of the abstract-domain element — i.e., $\mathtt{base}(a'')$ in the lattice $\mathcal{B}$.

Alg. 5 is a variant of the symbolic-abstraction algorithm from [84]. Alg. 5 needs a method to enumerate a generator set gs for each $b \in \mathcal{B}$. Such a set can easily be obtained from the generator representation of $\mathcal{B}$. For instance, each row in an AG element is an affine transformer, and a generator set for the AG element is the set of all rows in the AG matrix: the affine combination of the rows generate the concrete affine transformers that the AG element represents. Note that the generator set for an abstract value b is usually much smaller than the set of all affine transformers in b. For the AG domain, the generating set is worst-case polynomial size, whereas the set of all affine transformers is worst-case exponential in the number of variables k.

In Alg. 5, line 3 initializes the value *lower* to the product of each pair of abstract transformers. The product $t \times t'$, where $t = \left[\begin{array}{c|c} 1 & \vec{d} \\ \hline 0 & C \end{array}\right]$ and $t' = \left[\begin{array}{c|c} 1 & \vec{d}' \\ \hline 0 & C' \end{array}\right]$ is $\left[\begin{array}{c|c} 1 & \vec{d} \cdot C' + \vec{d}' \\ \hline 0 & C \cdot C' \end{array}\right]$. Because *lower* is initialized to $\{t \times t' \mid t \in gs_1, t' \in gs_2\}$ rather than $\bot$, the number of SMT calls in the symbolic abstraction is significantly reduced, compared to the algorithm from [84]. The function `GetModel`, used at line 5, returns the model $M \in \mathtt{symbols}(C'' : \vec{d}'') \to \mathbb{Z}_{2^w}$ satisfying the formula $(\varphi \wedge \neg\hat{\gamma}(\mathit{lower}))$ given to the SMT solver at line 4. Thus, the model M is a concrete affine transformer in a''. The representation function β, used at line 6, maps a singleton model M to the least value in $\mathcal{B}$ that overapproximates $\{M\}$ [84]. While the SMT call at line 4 is satisfiable, the loop keeps improving the value of *lower* by adding the satisfying model M to *lower* via the representation function β and the join operation. When line 4 is unsatisfiable, the loop terminates and returns *lower*. This method is sound because the unsatisfiable call proves that $\varphi \Rightarrow \hat{\gamma}(\mathit{lower})$. The loop terminates when the height of the base domain $\mathcal{B}$ is finite.

Algorithm 5 Abstract Composition via Symbolic Abstraction

1: $gs_1 \leftarrow \{t_1, t_2, ..., t_{l_1}\}$ ▷ where $base(a) = \bigsqcup_{i=0}^{l_1} t_i$
2: $gs_2 \leftarrow \{t'_1, t'_2, ..., t'_{l_2}\}$ ▷ where $base(a') = \bigsqcup_{i=0}^{l_2} t'_i$
3: $lower \leftarrow \{t \times t' \mid t \in gs_1, t' \in gs_2\}$
4: **while** $r \leftarrow SMTCall(\varphi \wedge \neg\hat{\gamma}(lower))$ is Sat **do**
5: $\quad M \leftarrow GetModel(r)$
6: $\quad lower \leftarrow lower \sqcup \beta(M)$
7: **return** *lower*

Non-relational base domains.

In this section, we present a method to implement abstract composition for ATA[$\mathcal{B}$], when $\mathcal{B}$ is non-relational. We focus on the non-relational case separately because it allows us to implement a sound abstract-composition operation efficiently.

Foundation domain. Each element in the non-relational domain $\mathcal{B}$ is a mapping from symbols S to a subset of Z_{2^w}. We introduce the concept of a *foundation domain*, denoted by $\mathcal{F}_{\mathcal{B}}$, to represent the abstractions of subsets of Z_{2^w} in the base abstract-domain elements. We can define a non-relational base domain in terms of the foundation domain as follows: $\mathcal{B} \stackrel{def}{=} S \to \mathcal{F}_{\mathcal{B}}$. For instance, the non-relational domain of intervals $\mathcal{I}_{\mathbb{Z}_{2^w}}^{(k+1)^2}$ can be represented by $S \to \mathcal{I}_{\mathbb{Z}_{2^w}}$, where $\mathcal{I}_{\mathbb{Z}_{2^w}}$ represents the interval lattice over $\mathbb{Z}_{2^w}$, and S is a set of $(k+1)^2$ symbolic constants that represent the coefficients of an affine transformer.

A foundation domain $\mathcal{F}$ is a lattice whose elements concretize to subsets of Z_{2^w}. Tab. 4.5 present the foundation-domain operations for $\mathcal{F}$. Bottom, equality, join, widen, and $\alpha(bv)$ are standard abstract-domain operations. The abstract addition and multiplication operations provide a sound reinterpretation of the collecting semantics of concrete addition and multiplication. For instance, with the interval foundation domain, $[0,7] +^{\sharp} [-3,17] = [-3,24]$ and $[0,6] \times^{\sharp} [-3,3] = [-18,18]$.

Table 4.5: Foundation-domain operations.

Type	Operation	Description
$\mathcal{F}$	$\bot$	*empty set*
bool	$(f_1 == f_2)$	*equality*
$\mathcal{F}$	$(f_1 \sqcup f_2)$	*join*
$\mathcal{F}$	$(f_1 \nabla f_2)$	*widen*

Type	Operation	Description
$\mathcal{F}$	$\alpha(bv)$	*abstraction for the bitvector value* $bv \in Z_{2^w}$
$\mathcal{F}$	$(f_1 +^\sharp f_2)$	*abstract addition*
$\mathcal{F}$	$(f_1 \times^\sharp f_2)$	*abstract multiplication*

Abstract composition for a non-relational domain is defined as follows:

$$\begin{aligned} a' \circ_{NR} a = \Big\{ (\vec{v}, \vec{v}') \mid \exists (C : \vec{d}) : (\vec{v}' = \vec{v} \cdot C + \vec{d}) \wedge b \in (symbols(C : \vec{d}) \to \mathcal{F}) \\ \wedge \Big(\bigwedge_{1 \leqslant i,j \leqslant k} (b[c_{ij}] = \sum^\sharp_{1 \leqslant l \leqslant k} (base(a)[c_{il}] \times^\sharp base(a')[c_{lj}]) \Big) \\ \wedge \Big(\bigwedge_{1 \leqslant j \leqslant k} b[d_j] = \sum^\sharp_{1 \leqslant l \leqslant k} (base(a)[d_l] \times^\sharp base(a')[c_{lj}]) +^\sharp base(a')[d_j] \Big) \Big\}. \end{aligned} \tag{4.3}$$

The term $b[s]$, where $b \in \mathcal{B}$ and $s \in symbols(C : \vec{d})$, refers to the element in the foundation domain $f \in \mathcal{F}_\mathcal{B}$, that corresponds to the symbol s. $\Sigma^\sharp_{1 \leqslant l \leqslant k}$ is calculated by abstractly adding the k terms indexed by l. Abstract composition for a non-relational domain uses abstract addition and abstract multiplication to soundly overapproximate the quadratic terms occurring in Eqn. (4.1). We provide a proof of the soundness for $a' \circ_{NR} a$ in App. E.1. The abstract-composition operation requires $\mathcal{O}(k^3)$ abstract-addition operations and $\mathcal{O}(k^3)$ abstract-multiplication operations.

Examples of foundation domains. We now present a few foundation domains that allow to construct the non-relational small-set, interval [19], and strided-interval [83] base domains.

Small sets. $\mathcal{F}_{SS_n} \stackrel{def}{=} \{\top\} \cup \{S | S \subseteq \mathbb{Z}_{2^w} \wedge |S| \leqslant n\}$. The join operation is defined by: $(f_1 \sqcup f_2) = \begin{cases} f_1 \cup f_2 & \text{if } |f_1 \cup f_s| \leqslant n \\ \top & \text{otherwise} \end{cases}$

n denotes the maximum cardinality allowed in the non-top elements of $\mathcal{F}_{SS_n}$. Other abstract operators, including abstract addition and multiplication, are implemented in a similar manner.

Intervals. $\mathcal{F}_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \stackrel{def}{=} \{\bot\} \cup \{[a,b] |\ a,b \in \mathbb{Z}_{2^w}, a \leqslant b\}$. Most abstract operations are straightforward (See [19] for details). The abstract-addition and abstract-multiplication operations need to be careful about overflows to preserve soundness. For instance,

$$[a_1,b_1] +^\sharp [a_2,b_2] = \begin{cases} [a_1+a_2, b_1+b_2] & \textit{if neither } a_1+a_2 \textit{ nor } b_1+b_2 \\ & \textit{overflows} \\ [min, max] & \textit{otherwise} \end{cases}$$

Strided Interval. $\mathcal{F}_{\mathcal{SI}_{\mathbb{Z}_{2^w}}} \stackrel{def}{=} \{\bot\} \cup \{s[a,b] \mid a,b,s \in \mathbb{Z}_{2^w}, a \leqslant b\}$, where $\gamma(s[a,b]) = \{i \mid a \leqslant i \leqslant b, i \equiv a (mod\ s)\}$. (See [83, 91] for the details of the abstract-domain operations.)

Affine-Closed Base Domain.

We discuss the special case when the base domain $\mathcal{B}$ is affine-closed, i.e., $\mathcal{B} = \text{AG}$. The abstract composition is defined as:

$$a' \circ_{\text{AG}} a = a'', \textit{ where } base(a'') = \left\langle \left\{ t_i \times t'_j |\ 1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l' \right\} \right\rangle \wedge \quad (4.4)$$
$$base(a) = \langle \{t_1, t_2, ... t_l\} \rangle \wedge base(a') = \langle \{t'_1, t'_2, ... t'_{l'}\} \rangle$$

Lemma 5.1 in [78] asserts that the above abstract composition method is sound by linearity of affine-closed abstractions. The abstract composition has time complexity $\mathcal{O}(hh'k^3)$, h (respectively h') is the height of the abstract-domain element $base(a)$ (or $base(a')$) in the AG lattice. Because the height of the AG lattice with $(k+1)^2$ columns is $\mathcal{O}(k^2)$, the time complexity for the abstract composition operation translates to $\mathcal{O}(k^7)$. Alg. 5 essentially implements Eqn. (4.4), but makes an extra SMT call to ensure that the result is sound. Because Eqn. (4.4) is sound by linearity for the AG domain, the very first SMT call in the while-loop condition at line 4 in Alg. 5 will be unsatisfiable.

Weakly-Convex Base Domain

We present methods to perform abstract composition when the base domain $\mathcal{B}$ satisfies a property we call *weak convexity*. Base domain $\mathcal{B}$ is *weakly convex* iff

- The abstraction of a single concrete affine transformer is exact: $\gamma(\alpha(t_i)) = \{t_i\}$.
- All abstract-domain elements $b \in \mathcal{B}$ are contained in a convex space over rationals: For any set of concrete affine transformers $\{t_0, t_1, ..., t_l\}$, such that $b = \bigsqcup_{i=0}^{l} t_i$, and any $t \in \gamma(b)$:
  $\exists \lambda_1, \lambda_2, \ldots, \lambda_l \in \mathbb{Q}.(0 \leqslant \lambda_1, \lambda_2, ..., \lambda_l \leqslant 1) \wedge \sum_{i=0}^{l} \lambda_i = 1 \wedge \mathtt{cast}_{\mathbb{Q}}(t) = \sum_{i=0}^{l} \lambda_i \, \mathtt{cast}_{\mathbb{Q}}(t_i)$.
  The $\mathtt{cast}_{\mathbb{Q}}$ function is used to specify the convexity property by moving the point space from bitvectors to rationals. For instance, the expression $\Sigma_{i=0}^{l} \lambda_i . \mathtt{cast}_{\mathbb{Q}}(t_i)$ specifies the convex combination of the concrete affine transformers m_i in the rational space $\mathtt{cast}_{\mathbb{Q}}^{(k+k^2)}$.

Any convex abstract domain over rationals, such as polyhedra [23] or octagons [68], can be used to create a weakly-convex domain over bitvectors [96, 94]. Abstract composition for weakly-convex base domains is defined as follows:

$$a' \circ_{WC} a = a'', \textit{ where } \mathtt{base}(a'') = \tag{4.5}$$
$$\begin{cases} \{t_i \times t'_j \mid 1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l'\} & \textit{if there are no overflows in any matrix multiplication } t_i \times t'_j \\ \top_{\mathcal{B}} & \textit{otherwise} \end{cases}$$
$$\text{where } \mathtt{base}(a) = \{t_1, t_2, ..., t_l\} \textit{ and } \mathtt{base}(a') = \{t'_1, t'_2, ..., t'_{l'}\}.$$

The intuition is that the weak-convexity properties are preserved under matrix multiplication in the absence of overflows. This principle is similar to the linearity argument used to show that abstract composition is sound

when the base domain is affine-closed. (See above for more details.) We provide a proof of the soundness for $a' \circ_{WC} a$ in App. E.2. Similar to the affine-closed case, abstract composition has time complexity $\mathcal{O}(H^2k^3)$, where H is the height of the $\mathcal{B}$ lattice.

Practical concerns. With the exception of the non-relational base domain, the complexity of the abstract-composition algorithms is dependent on the height of the abstract-domain elements involved in the composition, i.e., h and h'. Practical implementations of abstract composition might decide to return $\top$ for abstract composition if the number of matrices to multiply is beyond some threshold, say t, so that the complexity of the abstract composition is $\mathcal{O}(tk^3)$.

4.3.3 Merge Function

Knoop and Steffen [54] extended the Sharir and Pnueli [92] algorithm for interprocedural dataflow analysis to handle local variables. Suppose at a call site CS, procedure P calls procedure Q. The global variables, denoted by $\overrightarrow{g}$, are accessible to Q, but the local variables, denoted by $\overrightarrow{l}$, in P are inaccessible to Q. Thus, the values of local variables after the call site CS come from the values before the call point, and the values of global variables after the call site CS come from the values at the return site in procedure Q. A *merge function* is used to combine the abstract-domain element before the call to Q with the abstract-domain element returned by Q to create the abstract-domain element to use in P after the call to Q has finished.

We assume that in each function, the local variables are initialized to 0. To simplify the discussion, assume that all scopes have the same number of locals, and that each vocabulary $\overrightarrow{v}$ consists of subvocabularies $\overrightarrow{g}$ and $\overrightarrow{l}$—that is, $\overrightarrow{v} = (\overrightarrow{g}, \overrightarrow{l})$ Suppose that we have two relations, $R[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']$ and $R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']$, each of which is a subset of $\mathbb{Z}_{2^w}^k \times \mathbb{Z}_{2^w}^k$, where R is the

transition relation from the start state of the calling procedure P to the call site CS, and R' is the transition relation from the start state to the return site of the called procedure Q. Operationally, after completing the call at the call site CS, we want $\textsc{Merge}(R[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}'], R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}'])$ to act as a modified relational composition in which R' acts like the identity function on locals, so that $\overrightarrow{l}'$ values from R are passed through R' unchanged to become the $\overrightarrow{l}'$ values of the result. This semantics can be specified as follows:

$$\begin{aligned} &\textsc{Merge}(R[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}'], R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']) \\ &\quad = \quad \textsc{RevertLocals}(R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']) \circ R[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}'] \end{aligned} \tag{4.6}$$

We define $\textsc{RevertLocals}(R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}'])$ as follows:

$$\textsc{RevertLocals}(R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']) \stackrel{\text{def}}{=} \{(\overrightarrow{g}, \overrightarrow{q}, \overrightarrow{g}', \overrightarrow{q}) \mid R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']\} \tag{4.7}$$

Recall that the $k + k^2$ symbolic constants in an affine transformation, $symbols(C{:}\,\overrightarrow{d})$, can be padded with a one and k zeroes and arranged as follows: $\left[\begin{array}{c|c} 1 & \overrightarrow{d} \\ \hline 0 & C \end{array}\right]$. We can partition $symbols(C{:}\,\overrightarrow{d})$ into globals and locals to write the matrix as $\left[\begin{array}{c|c|c} 1 & \overrightarrow{d}_g & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{lg} & C_{ll} \end{array}\right]$, which represents the affine transformation

$$(\overrightarrow{g}' = \overrightarrow{g} \cdot C_{gg} + \overrightarrow{l} \cdot C_{lg} + \overrightarrow{d_g}) \wedge (\overrightarrow{l}' = \overrightarrow{g} \cdot C_{gl} + \overrightarrow{l} \cdot C_{ll} + \overrightarrow{d_l})$$

Let $a, a' \in \text{ATA}[\mathcal{B}]$ be the abstract transformers that represents the relations $R[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']$ and $R'[\overrightarrow{g}, \overrightarrow{l}; \overrightarrow{g}', \overrightarrow{l}']$, respectively. Then the merge function for a_1 and a_2 is defined as follows:

$$Merge(a, a') = a'', \text{ where} \tag{4.8}$$

$$
\begin{aligned}
base(a'') \;= (b_g \sqcap havoc(base(Id), gsyms)) \;\circ\; a \\
b_g = havoc(base(a'), lsyms), \\
lsyms = symbols(C_{gl}) \cup symbols(C_{ll}) \cup \\
\qquad symbols(d_l) \cup symbols(C_{lg}) \\
gsyms = symbols(C_{gg}) \cup symbols(d_g)
\end{aligned}
$$

$lsyms$ are the symbols in the affine transformation that involve local variables. $gsyms$ are the symbols in the affine transformation that are not in $lsyms$. The expression $b_{gi} = (b_g \sqcap havoc(base(Id), gsyms))$ transforms each affine transformer $\left[\begin{array}{c|c|c} 1 & \vec{d}_g & \vec{d}_l \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{lg} & C_{ll} \end{array}\right] \in \gamma(base(a))$ to $\left[\begin{array}{c|c|c} 1 & \vec{d}_g & 0 \\ \hline 0 & C_{gg} & 0 \\ 0 & 0 & I \end{array}\right]$. In this way, b_{gi} ensures that the modifications of the globals at the return point of Q are accounted for, while the locals for P pass through Q unmodified. We provide the proof of soundness for the merge-function definition (Eqn. (4.8)) in App. F.

4.4 Discussion and Related Work

The abstract-domain elements in our framework abstract two-vocabulary relationships arising between the pre-transformation state and post-transformation state. For the sake of simplicity, we assumed that the variable sets in the pre-transformation and post-transformation state are the same, and an affine transformer is represented by a $(k+1) \times (k+1)$ matrix, where k is the number of variables in the pre-transformation state. However, this requirement is not mandatory. We can easily adapt our abstract-domain operations to work on $(k+1) \times (k'+1)$ matrices where k' is the number of variables in the post-transformation state.

The abstract-domain elements in our framework are not necessarily closed under intersection. Consider the two abstract values a_1 and a_2

for the vocabulary $V = \{v_1\}$. Let a_1 represent the affine transformation $v_1' = 0$ and a_2 represent the identity affine transformation $v_1' = v_1$. Thus, $a_1 = \alpha(\left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 0 \end{array}\right])$, and $a_2 = \alpha(\left[\begin{array}{c|c} 1 & 0 \\ \hline 0 & 1 \end{array}\right])$. The intersection of $\gamma(a_1)$ and $\gamma(a_2)$ is the point $p = (v_1' = 0, v_1 = 0)$. There does not exist an abstract value in ATA$[\mathcal{B}]$, that can exactly represent the point p, because any abstract value containing p must contain at least one affine transformer of the form $v_1' = v_1 \cdot c$, and thus must contain all points of the form $(v_1' = t \cdot c, v_1 = t)$, where $t \in \mathbb{Z}_{2^w}$. As a consequence, there does not exist a Galois connection between ATA$[\mathcal{B}]$ and the concrete domain $\mathcal{C}$ of all two-vocabulary relations $R[V; V']$, which implies that there does not exist a best abstraction for a set of concrete points. For instance, consider the abstraction of the guard statement $S_G = \{v_1 \leqslant 10\}$, with the ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ domain. Consider $a_3 = \left[\begin{array}{c|c} 1 & [0,10] \\ \hline 0 & [0,0] \end{array}\right]$ and $a_4 = \left[\begin{array}{c|c} 1 & [0,0] \\ \hline 0 & [1,1] \end{array}\right]$. a_3 specifies the guard constraint $0 \leqslant v_1' \leqslant 10$, while a_4 is the identity transformation $v_1' = v_1$. Note that these abstract values are incomparable and can be used to represent the abstract transformer for S_G. Furthermore, $a_3 \sqcap a_4$ does not exist. Thus, an analysis has to settle for either a_3 or a_4. (In §4.2, we used an abstract transformer similar to a_3 for the guard in the while statement in Ex. 4.4. Using an identity transfer for the guard statement would not have been useful to capture the desired bounds constraint.)

The ATA constructor preserves finiteness; that is, if the base domain $\mathcal{B}$ is finite, then the domain ATA$[\mathcal{B}]$ is finite as well.

It is also possible to use the ATA constructor to infer affine transformations over rationals or reals. In these cases, the symbolic-composition methods for weakly-convex base domains (see §4.3.2) will carry over to affine transformations over rationals or reals for convex base domains (e.g., polyhedra) with only slight modifications. For instance. abstract composition for convex base domains over rationals or reals is defined as

follows:

$$a' \circ a = a'', \textit{ where } base(a'') = \left\{ t_i \times t'_j \middle| \, 1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l' \right\}$$
$$\text{where } base(a) = \{t_1, t_2, ..., t_l\} \textit{ and } base(a') = \{t'_1, t'_2, ..., t'_{l'}\}.$$

Chen et al. [18] devised the *interval-polyhedra* domain which can express constraints of the form $\Sigma_k [a_k, b_k] x_k \leqslant c$ over rationals. Interval polyhedra are more expressive than classic convex polyhedra, and thus can express certain non-convex properties. Abstract-domain operations for interval polyhedra are constructed by linear programming and interval Fourier-Motzkin elimination. The domain has similarities to the ATA$[\mathcal{I}^{(k+1)^2}_{\mathbb{Z}_{2^w}}]$ domain because the coefficients in the abstract values are intervals.

Miné [69] introduced *weakly relational domains*, which are a parameterized family of relational domains, parameterized by a non-relational base abstract domain. They can express constraints of the form $(v_j - v_i) \in \mathcal{F}$, where $\mathcal{F}$ is an abstraction over $\mathcal{P}(\mathbb{Z})$. Similar to ATA$[\mathcal{B}]$, Miné's framework requires the base non-relational domain to provide abstract-addition and abstract-unary-minus operations. These operations are used to propagate information between constraints via a closure operation that is similar to finding shortest paths.

Sankaranarayanan et al. [89] introduced a domain based on template constraint matrices (TCMs) that is less powerful than polyhedra, but more general than intervals and octagons. Their analysis discovers linear-inequality invariants using polyhedra with a predefined fixed shape. The predefined shape is given by the client in the form of a template matrix. Our approach is similar because an affine transformer with symbolic constants can be seen as a template. However, the approaches differ because Sankaranarayanan et al. use an LP solver to find values for template parameters, whereas we use operations and values from an abstract domain to find and represent a set of allowed values for template parameters.

An abstract-domain element in ATA[$\mathcal{B}$] can be seen as an abstraction over sets of functions: $\mathbb{Z}_{2^w}^k \to \mathbb{Z}_{2^w}^k$. Jeannet et al. [46] provide a theoretical treatment of the relational abstraction of functions. They describing existing and new methods of abstracting functions of signature: $D_1 \to D_2$, resulting in a family of relational abstract domains. ATA[$\mathcal{B}$] is not captured by their framework of functional abstractions.

4.5 Chapter Notes

Thomas Reps supervised me in the writing of the paper, and provided some key insights for this work. Jason Breck provided useful comments about this work.

5 AN ABSTRACT DOMAIN FOR BIT-VECTOR INEQUALITIES

The chapter describes the design and implementation of a new abstract domain, called the *Bit-Vector Inequality* ($\mathcal{BVI}$) domain, that is capable of capturing certain inequalities over bit-vector-valued variables. We also consider some variants of the $\mathcal{BVI}$ domain.

Key Insight: The View-Product Combinator. The key insight used to design $\mathcal{BVI}$ domain (and its variants) involves a new domain combinator (denoted by $\mathcal{V}$), called the *view-product combinator*. $\mathcal{V}$ constructs a reduced product of two domains [21], using shared *view-variables* to communicate information between the domains. The following example illustrates the concept of a view-variable:

Example 5.1. *Consider the equality constraint* $H := x+2\mathtt{mm}[y+2] = 4$, *where* x *and* y *are bit-vector variables and* $\mathtt{mm}$ *is the memory map. The term* $\mathtt{mm}[e]$ *denotes the contents of* $\mathtt{mm}$ *at address* e. *Let* $\mathcal{E}_{\mathbb{Z}_{2^w}}$ *denote any of the abstract domains of relational affine equalities over bit-vector variables (See Chapters 3 and 4).* H *cannot be expressed using* $\mathcal{E}_{\mathbb{Z}_{2^w}}$ *alone. However, the formulas* $x+2\mathtt{mm}[y+2] = 4$ *and* $u = \mathtt{mm}[y+2] \wedge x + 2u = 4$ *are equisatisfiable. The variable* u *is called a* view-variable*; the constraint* $u = \mathtt{mm}[y+2]$ *is the* view-constraint *associated with* u.

The equality $x + 2u = 4$ *can be expressed using* $\mathcal{E}_{\mathbb{Z}_{2^w}}$*; therefore, what we require is a second abstract domain capable of expressing invariants that involve memory accesses, such as* $u = \mathtt{mm}[y+2]$. *This need motivates the* bit-vector memory domain $\mathcal{M}$ *that we introduce in §5.3. The constraint* H *can then be expressed by (i) introducing view-variable* u, *(ii) representing* u*'s view-constraint in* $\mathcal{M}$, *(iii) extending* $\mathcal{E}_{\mathbb{Z}_{2^w}}$ *with* u, *and (iv) using* $\mathcal{M}$ *and* $\mathcal{E}_{\mathbb{Z}_{2^w}}$ *together.* □

The *Bit-Vector Memory-Equality Domain* $\mathcal{BVME}$, a domain of bit-vector affine-equalities over variables and memory-values, is created by applying the view-product combinator $\mathcal{V}$ to the bit-vector memory domain (§5.3) and the bit-vector equality domain. The *Bit-Vector Inequality Domain* $\mathcal{BVI}$, a domain of bit-vector affine-inequalities over variables, is created by applying $\mathcal{V}$ to the bit-vector equality domain and a bit-vector interval domain. The *Bit-Vector Memory-Inequality Domain* $\mathcal{BVMI}$, a domain of relational bit-vector affine-inequalities over variables and memory, is then created by applying $\mathcal{V}$ to the $\mathcal{BVME}$ domain and the bit-vector interval domain. The latter construction illustrates that $\mathcal{V}$ composes: the $\mathcal{BVMI}$ domain is created via two applications of $\mathcal{V}$.

The design of the view-product combinator $\mathcal{V}$ was inspired by the Subpolyhedra domain [58] (SubPoly), a domain for inferring relational linear inequalities over rationals. SubPoly is constructed as a reduced product of an affine-equality domain $\mathcal{K}$ over rationals [49], and an interval domain $\mathcal{J}$ over rationals [19] in which *slack variables* are used to communicate between the two domains. Such a design enables SubPoly to be as expressive as Polyhedra, but more scalable. $\mathcal{V}$ provides a generalization of the construction used in SubPoly. In fact, SubPoly can be constructed by applying $\mathcal{V}$ to $\mathcal{K}$ and $\mathcal{J}$ (see Eqn. (5.1) in §5.7).

Enabling Technology: Automatic Synthesis of Best Abstract Operations. Using a symbolic abstraction method §2.3.2, we give a procedure for *synthesizing best abstract operations* for general reduced-product domains. In particular, we use this framework to ensure that the transformers for the reduced-product domains constructed via $\mathcal{V}$ are sound and precise, thereby guaranteeing that analysis results will be a conservative over-approximation of the concrete semantics.

Contributions.

The contributions of the chapter are:

- The bit-vector memory domain, a non-relational memory domain capable of expressing invariants involving memory accesses (§5.3).
- The view-product combinator $\mathcal{V}$, a general procedure to construct more expressive domains (§5.4).
- Three domains for machine-code analysis constructed using $\mathcal{V}$:
  - The bit-vector memory-equality domain $\mathcal{BVME}$, which captures equality relations among bit-vector variables and memory (Defn. 5.6).
  - The bit-vector inequality domain $\mathcal{BVI}$, which captures inequality relations among bit-vector variables (Defn. 5.7).
  - The bit-vector memory-inequality domain $\mathcal{BVMI}$, which captures inequality relations among bit-vector variables and memory (Defn. 5.8).
- A procedure for synthesizing best abstract operations for reduced products of domains that meet certain requirements (§5.5).
- Experimental results that illustrate the effectiveness of the $\mathcal{BVI}$ domain applied to machine-code analysis (§5.6). On average (geometric mean), our $\mathcal{BVI}$-based analysis is about 3.5 times slower than an affine-equality-based analysis, while finding improved (more-precise) invariants at 29% of the branch points.

§5.1 provides an overview of our solution. §5.2 defines terminology. §5.7 describes related work.

5.1 Overview

In this section, we illustrate the design of the bit-vector memory-inequality domain. Consider the Intel x86 machine-code snippet shown in Fig. 5.1. Pseudo-code for each instruction is shown at the right-hand side of each

```
    mov   ecx, [ebp-4]        //ecx = mm[ebp-4]
    add   ecx, eax            //ecx = ecx + eax
    cmp   ecx, 10d            //if ecx >u 10
    ja    L                   //     goto L
    xor   ecx, ecx            //  ecx = 0
  0 ⩽ mm[ebp − 4] + eax ⩽ 10
L:
```

Figure 5.1: Example snippet of Intel x86 machine code.

instruction. Note that the **ja** instruction, "jump if above, unsigned", treats **ecx** as unsigned. Thus, control reaches the **xor** instruction only if the value in **ecx** is greater than or equal to 0, and less than or equal to 10. Let $R := \{eax, ecx, ebp\}$ denote the set of register variables, and $\mathtt{mm}$ denote the memory map.

The highlighted text in Fig. 5.1 states the invariant $H := 0 \leqslant \mathtt{mm}[ebp - 4] + eax \leqslant 10$ that holds after the **xor** instruction. Again, let $\mathcal{E}_{\mathbb{Z}_{2^w}}$ denote any of the abstract domains of relational affine equalities over bit-vector variables. The invariant H cannot be represented using $\mathcal{E}_{\mathbb{Z}_{2^w}}$, because (i) H involves an invariant about a value in memory $\mathtt{mm}$, and (ii) H is an inequality. To handle memory, we introduce the bit-vector memory domain $\mathcal{M}$, which is capable of expressing certain constraints on memory accesses. In particular, we can use $\mathcal{M}$ to express the constraint $u = \mathtt{mm}[ebp - 4]$, where u is a fresh variable. We call u a *view-variable* and $u = \mathtt{mm}[ebp - 4]$ a *view-constraint* for u. H can be written as the equisatisfiable formula $H_m := u = \mathtt{mm}[ebp - 4] \wedge 0 \leqslant u + eax \leqslant 10$.

Notice there are no memory accesses in the constraint $0 \leqslant u + eax \leqslant 10$. The inequality $0 \leqslant u + eax \leqslant 10$ and $u + eax = s \wedge 0 \leqslant s \leqslant 10$ are equisatisfiable, where s is a fresh variable. Similar to u, s is a view-variable with $u + eax = s$ is a view-constraint for s. Thus, H_m can be rewritten as $H_{mi} := u = \mathtt{mm}[ebp - 4] \wedge u + eax = s \wedge 0 \leqslant s \leqslant 10$. Furthermore, $u + eax = s$ can be expressed in $\mathcal{E}_{\mathbb{Z}_{2^w}}$, and $0 \leqslant s \leqslant 10$ can be expressed

using a bit-vector interval domain $\mathcal{I}_{\mathbb{Z}_{2^w}}$. Thus, by introducing the view-variables u and s, the invariant H can be expressed using $\mathcal{M}$, $\mathcal{E}_{\mathbb{Z}_{2^w}}$, and $\mathcal{I}_{\mathbb{Z}_{2^w}}$ together. The following derivation illustrates the above decomposition of the invariant H:

$$\textsc{Memory}\ \frac{u = \texttt{mm}[ebp-4] \qquad \textsc{Inequality}\ \dfrac{u + eax = s \qquad s \in [0,10]}{0 \leqslant u + eax \leqslant 10}}{0 \leqslant \texttt{mm}[ebp-4] + eax \leqslant 10}$$

Note that the view-constraints $u = \texttt{mm}[ebp-4]$ and $s = u + eax$ do not directly constrain the values of R and $\texttt{mm}$; they only constrain the view-variables u and s. The shared view-variables are used to exchange information among the various domains. In particular, the view-variable u is used to exchange information between memory domain $\mathcal{M}$ and equality domain $\mathcal{E}_{\mathbb{Z}_{2^w}}$, and the view-variable s is used to exchange information between $\mathcal{E}_{\mathbb{Z}_{2^w}}$ and interval domain $\mathcal{I}_{\mathbb{Z}_{2^w}}$.

5.2 Terminology

For a somewhat technical reason, we introduce the device of an *abstract-domain constructor*. Given an abstract-domain family $\mathcal{A}$ and vocabularies V_1 and V_2, an abstract-domain constructor for $\mathcal{A}$, denoted by $\mathfrak{C}_{\mathcal{A}}(V_1, V_2)$, constructs $\mathcal{A}[V_3]$, where $V_1 \subseteq V_3 \subseteq V_1 \uplus V_2$. In particular, $\mathfrak{C}_{\mathcal{A}}$ is free to decide what subset of V_2 to use when constructing $\mathcal{A}[V_3]$. (In our applications, the abstract-domain constructors either use all of V_2 or none of V_2.)

Let $A \in \mathcal{A}[V]$; we denote by $A{\downarrow_{V_1}}$ the value obtained by projecting A onto the vocabulary $V_1 \subseteq V$. We use $\oplus$ to denote a *vocabulary-extension operator* over domains; in particular, given domain $\mathcal{A}[V_1]$ and vocabulary V_2, $\mathcal{A}[V_1] \oplus V_2 = \mathcal{A}[V_1 \uplus V_2]$.

Let $\mathcal{G}_1 = \mathcal{C} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \mathcal{A}_1$ and $\mathcal{G}_2 = \mathcal{C} \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \mathcal{A}_2$ be two Galois connections.

We use $\mathcal{A}_1[V_1] \star \mathcal{A}_2[V_2]$ to denote the reduced product of the domains [21, §10.1], and $\langle A_1; A_2 \rangle$ to denote an element of $\mathcal{A}_1[V_1] \star \mathcal{A}_2[V_2]$.

5.3 Base Domains

Let $\mathtt{mm}$ denote a memory map, and $\mathtt{mm}[t]$ denote the w-bit value at memory address t. For instance, for Intel x86 machine code, $\mathtt{mm}[t]$ denotes (the little-endian interpretation of) the four bytes in memory pointed to by term t (cf. Ex. 1.3).

Bit-Vector Memory Domain ($\mathcal{M}$). Domain $\mathcal{M}$ can capture a limited class of invariants involving memory accesses. In particular, the domain is capable of capturing the constraint that the value of a variable v equals the value of the memory $\mathtt{mm}$ at address e: $v = \mathtt{mm}[e]$.

Definition 5.2. *Given variables* P *and memory map* $\mathtt{mm}$*, an element* M *of the* ***bit-vector memory domain*** $\mathcal{M}[(\mathtt{mm}, P)]$ *is either (i)* $\bot$*, or (ii) a set of constraints, where each constraint* C_i *is of the form* $v_i = \mathtt{mm}[\Sigma_j a_{ij} v_j + b_i]$, $a_{ij}, b_i \in \mathbb{Z}_{2^w}$, $v_i, v_j \in P$*. The concretization of* M *is*

$$\gamma(M) = \{(\mathtt{mm}, \overrightarrow{v}) \mid \textstyle\bigwedge_{C_i \in M} (\mathtt{mm}, \overrightarrow{v}) \models C_i\}.$$

□

The join and meet operation are defined as intersection and union of constraints, respectively. The join operation $\sqcup_{\mathcal{M}}$ is: $M_1 \sqcup_{\mathcal{M}} M_2 \stackrel{\text{def}}{=} \{C \mid C \in M_1 \text{ and } C \in M_2\}$. The meet operation $\sqcap_{\mathcal{M}}$ is: $M_1 \sqcap_{\mathcal{M}} M_2 \stackrel{\text{def}}{=} \{C \mid C \in M_1 \text{ or } C \in M_2\}$. The domain constructor for the bit-vector memory domain is defined as $\mathfrak{C}_{\mathcal{M}}(V_1, V_2) \stackrel{\text{def}}{=} \mathcal{M}[V_1 \uplus V_2]$.

Bit-Vector Equality Domain ($\mathcal{E}_{\mathbb{Z}_{2^w}}$). Domain $\mathcal{E}_{\mathbb{Z}_{2^w}} = \text{KS}$ can capture relational affine equalities over bit-vectors.

Bit-Vector Interval Domain ($\mathcal{I}_{\mathbb{Z}_{2^w}}$). Domain $\mathcal{I}_{\mathbb{Z}_{2^w}}$ can capture non-relational bit-vector interval constraints.

Definition 5.3. *Given variables* V*, an element* I *of the* ***bit-vector interval domain*** $\mathcal{I}_{\mathbb{Z}_{2^w}}[V]$ *is either (i)* $\bot$*, or (ii) a set of interval constraints, where each constraint* C_i *is of the form* $l_i \leqslant v_i \leqslant u_i, l_i, u_i \in \mathbb{Z}_{2^w}, v_i \in V$*. The concretization of* I *is*

$$\gamma(I) = \{\overrightarrow{v} \mid \bigwedge_{C_i \in I} \overrightarrow{v} \models C_i\}.$$

□

The domain constructor for the bit-vector interval domain is defined as $\mathfrak{C}_{\mathcal{I}}(V_1, V_2) \stackrel{\text{def}}{=} \mathcal{I}_{\mathbb{Z}_{2^w}}[V_1]$.

5.4 The View-Product Combinator

In this section, we define the view-product combinator $\mathcal{V}$, and construct three domains using different applications of $\mathcal{V}$. $\mathcal{V}$ constructs a reduced-product of domains $\mathcal{A}_1$ and $\mathcal{A}_2$ so that a set of shared view-variables communicates information between the domains. $\mathcal{V}$ makes use of two principles:

Principle 1: View-variables are constrained by view-constraints. Consider abstract domain $\mathcal{A}_1[V_1]$, and let $V_1 \cap V_2 = \varnothing$. The variables in V_2 are the *view-variables*. $\mathcal{V}$ will use the domain $\mathcal{A}_1[V_1] \oplus V_2 = \mathcal{A}_1[V_1 \uplus V_2]$ (cf. §5.2). A *view-constraint* C for V_2 is an element of $\mathcal{A}[V_1 \uplus V_2]$ that serves to constrain the variable set V_2.

Principle 2: A view-constraint does not constrain the values of variables in V_1. Given abstract domain $\mathcal{A}[V_1 \uplus V_2]$, an *acceptable view-constraint* C for V_2 is an abstract value $C \in \mathcal{A}[V_1 \uplus V_2]$ such that $C{\downarrow_{V_1}} = \top$.

We are now in a position to describe how $\mathcal{V}$ works.

Arguments. $\mathcal{V}$ takes four arguments:

- $\mathcal{A}_1[V_1]$, an abstract domain defined over vocabulary V_1,
- V_2, a vocabulary such that $V_1 \cap V_2 = \varnothing$,
- $C \in \mathcal{A}_1[V_1 \uplus V_2]$, a view-constraint for V_2, and
- $\mathfrak{C}_{\mathcal{A}2}$, an abstract-domain constructor for abstract domain $\mathcal{A}_2$,

Enforcement of view-constraint C**.** The view-constraint C constrains the variables in V_2. Therefore, $\mathcal{V}$ should only consider those elements $\mathcal{A}[V_1 \uplus V_2]$ that satisfy C; that is, only elements $A \in \mathcal{A}[V_1 \uplus V_2]$ such that $A \sqsubseteq_{\mathcal{A}[V_1 \uplus V_2]} C$ holds. Put another way, the view-constraint C can be seen as an *integrity constraint*. The next definition formalizes this notion (for a general integrity constraint $D \in \mathcal{A}[V]$):

Definition 5.4. *Given an abstract domain* $\mathcal{A}[V]$ *and an element* $D \in \mathcal{A}[V]$*, the* ***abstract domain*** $\mathcal{A}[V]$ ***modulo*** D*, denoted by* $\mathcal{A}[V]\mid_D$*, is an abstract domain* $\mathcal{A}'[V]$ *that contains exactly the elements* $D \sqcap A$*,* $A \in \mathcal{A}$*; that is,*

$$\mathcal{A}[V]\mid_D \stackrel{def}{=} \{D \sqcap A \mid D \in \mathcal{A}[V]\}$$

□

Using the notation from Defn. 5.4, the domain that only contains elements that satisfy view-constraint C is $\mathcal{A}_1[V_1 \uplus V_2]\mid_C$.

Reduced product. Let $\mathcal{A}_1'[V_1 \uplus V_2]$ be $(\mathcal{A}_1[V_1] \oplus V_2)\mid_C$, where C is the view-constraint for view-variables V_2. All that is left is to perform a reduced product of $\mathcal{A}_1'$ and $\mathcal{A}_2$. However, for $\mathcal{A}_1'$ and $\mathcal{A}_2$ to be able to exchange information, the vocabulary of $\mathcal{A}_2$ should include at least the view-variables V_2. This condition is satisfied by the domain $\mathfrak{C}_{\mathcal{A}2}(V_2, V_1)$ constructed using the abstract-domain constructor for $\mathcal{A}_2$ supplied as the fourth argument to $\mathcal{V}$ (cf. §5.2).

Summing up, the reduced product performed by $\mathcal{V}$ creates $\mathcal{A}_3[V_1 \uplus V_2] := \mathcal{A}'_1[V_1 \uplus V_2] \star \mathfrak{C}_{\mathcal{A}2}(V_2, V_1)$.

Definition 5.5 (View-Product Combinator ($\mathcal{V}$))**.** *Given*

- $\mathcal{A}_1[V_1]$*, an abstract domain defined over vocabulary* V_1*,*
- V_2*, a vocabulary such that* $V_1 \cap V_2 = \varnothing$*,*
- $C \in \mathcal{A}_1[V_1 \uplus V_2]$*, a view-constraint for* V_2*, and*
- $\mathfrak{C}_{\mathcal{A}2}$*, an abstract-domain constructor for abstract domain* $\mathcal{A}_2$*,*

the ***view-product combinator*** $\mathcal{V}[\mathcal{A}_1[V_1], V_2, C, \mathfrak{C}_{\mathcal{A}2}]$ *constructs a domain* $\mathcal{A}_3[V_1 \uplus V_2]$ *such that*

$$\mathcal{A}_3[V_1 \uplus V_2] \stackrel{def}{=} (\mathcal{A}_1[V_1] \oplus V_2) \mid_C \star \mathfrak{C}_{\mathcal{A}2}(V_2, V_1)$$

□

Instantiations. We now describe three applications of $\mathcal{V}$ that use the bit-vector domains defined in §5.3.

$\mathcal{V}$ applied to the bit-vector memory domain $\mathcal{M}$ and the bit-vector equality domain $\mathcal{E}_{\mathbb{Z}_{2^w}}$ constructs the *Bit-Vector Memory-Equality Domain* $\mathcal{BVME}$, a domain of bit-vector affine-equalities over variables and memory-values.

Definition 5.6 (Bit-Vector Memory-Equality Domain ($\mathcal{BVME}$))**.**

$$\mathcal{BVME}[(\mathtt{mm}, P \uplus U)] \stackrel{def}{=} \mathcal{V}[\mathcal{M}[(\mathtt{mm}, P)], U, C_{\mathtt{m}}, \mathfrak{C}_{\mathcal{E}_{\mathbb{Z}_{2^w}}}],$$

where

- $\mathcal{M}[(\mathtt{mm}, P)]$*, the bit-vector memory domain over memory map* $\mathtt{mm}$ *and variables* P *(cf. Defn. 5.2),*
- U*, a vocabulary of variables such that* $U \cap P = \varnothing$*,*
- $C_{\mathtt{m}} \in \mathcal{M}[(\mathtt{mm}, P \uplus U)]$*, a view-constraint for* U*, and*
- $\mathfrak{C}_{\mathcal{E}_{\mathbb{Z}_{2^w}}}(V_1, V_2) \stackrel{def}{=} \mathcal{E}_{\mathbb{Z}_{2^w}}[V_1 \uplus V_2]$.

□

$\mathcal{V}$ applied to the bit-vector equality domain $\mathcal{E}_{\mathbb{Z}_{2^w}}$ and the bit-vector interval domain $\mathcal{I}_{\mathbb{Z}_{2^w}}$ constructs the *Bit-Vector Inequality Domain* $\mathcal{BVI}$, a domain of bit-vector affine-equalities over variables.

Definition 5.7 (The Bit-Vector Inequality Domain ($\mathcal{BVI}$))**.**

$$\mathcal{BVI}[\mathsf{P} \uplus \mathsf{S}] \stackrel{def}{=} \mathcal{V}[\mathcal{E}_{\mathbb{Z}_{2^w}}[\mathsf{P}], \mathsf{S}, \mathsf{C}_s, \mathfrak{C}_{\mathcal{I}}],$$

where

- $\mathcal{E}_{\mathbb{Z}_{2^w}}[\mathsf{P}]$*, the bit-vector equality domain over variables* P*,*
- S*, a vocabulary of variables such that* $\mathsf{S} \cap \mathsf{P} = \varnothing$*,*
- $\mathsf{C}_s \in \mathcal{E}_{\mathbb{Z}_{2^w}}[\mathsf{P} \uplus \mathsf{S})]$*, a view-constraint for* S*, and*
- $\mathfrak{C}_{\mathcal{I}}(\mathsf{V}_1, \mathsf{V}_2) \stackrel{def}{=} \mathcal{I}_{\mathbb{Z}_{2^w}}[\mathsf{V}_1]$.

□

The combinator $\mathcal{V}$ applied to the bit-vector memory-equality domain $\mathcal{BVME}$, and a bit-vector interval domain $\mathcal{I}_{\mathbb{Z}_{2^w}}$ constructs the *Bit-Vector Memory-Inequality Domain* $\mathcal{BVMI}$, a domain of relational bit-vector affine-inequalities over variables and memory.

Definition 5.8 (Bit-Vector Memory-Inequality Domain ($\mathcal{BVMI}$))**.**

$$\mathcal{BVMI}[(\mathtt{mm}, \mathsf{P} \uplus \mathsf{U} \uplus \mathsf{S})] \stackrel{def}{=} \mathcal{V}[\mathcal{BVME}[(\mathtt{mm}, \mathsf{P} \uplus \mathsf{U})], \mathsf{S}, \mathsf{C}_s, \mathfrak{C}_{\mathcal{I}}]$$

where

- $\mathcal{BVME}[(\mathtt{mm}, \mathsf{P} \uplus \mathsf{U})]$*, the bit-vector memory-equality domain over memory map* $\mathtt{mm}$ *and variables* $\mathsf{P} \uplus \mathsf{S}$ *(cf. Defn. 5.6),*
- S*, a vocabulary of variables such that* $\mathsf{S} \cap (\mathsf{P} \uplus \mathsf{U}) = \varnothing$*,*
- $\mathsf{C}_s \in \mathcal{BVME}[(\mathtt{mm}, \mathsf{P} \uplus \mathsf{U})]$*, a view-constraint for* S*, and*
- $\mathfrak{C}_{\mathcal{I}}(\mathsf{V}_1, \mathsf{V}_2) \stackrel{def}{=} \mathcal{I}_{\mathbb{Z}_{2^w}}[\mathsf{V}_1]$.

□

5.5 Synthesizing Abstract Operations for Reduced-Product Domains

In this section, we first discuss a method for automatically synthesizing abstract operations for a general reduced-product domain $\mathcal{A}_3[V_1 \uplus V_2] := \mathcal{A}_1[V_1] \star \mathcal{A}_2[V_2]$ using the abstract operations of $\mathcal{A}_1$ and $\mathcal{A}_2$, which themselves may be automatically synthesized. We then discuss some pragmatic choices that we made for the reduced-product domains that the view-product combinator $\mathcal{V}$ creates.

Semantic Reduction. The semantic reduction of $\langle A_1; A_2\rangle \in \mathcal{A}_1 \star \mathcal{A}_2 = \mathcal{A}_3$ can be computed as $\hat{\alpha}_3(\psi)$, where ψ is $\hat{\gamma}_1(A_1) \wedge \hat{\gamma}_2(A_2)$. Computing the semantic reduction in this way can be computationally expensive. Instead we assume there exists a *weak semantic-reduction* operator Reduce. Using this weak semantic-reduction operator we can define the other abstract operations for the reduced-product domain. Furthermore, because it can be expensive to determine whether $\langle A_1; A_2\rangle \sqsubseteq \langle A_1'; A_2'\rangle$ holds, we define a weaker approximation order $\mathbb{E}$:

Definition 5.9. *The **weak approximation order** $\mathbb{E}$ is defined as follows:* $\langle A_1'; A_2'\rangle \mathbb{E} \langle A_1''; A_2''\rangle$ *if and only if* $A_1' \sqsubseteq_{\mathcal{A}_1} A_1''$ *and* $A_2' \sqsubseteq_{\mathcal{A}_2} A_2''$. □

It is easy to show that if $\langle A_1'; A_2'\rangle \mathbb{E} \langle A_1''; A_2''\rangle$, then $\langle A_1'; A_2'\rangle \sqsubseteq \langle A_1''; A_2''\rangle$, though the converse may not always hold.

We define *quasi-join*, an approximation to the join operator for reduced-product domain, which is not guaranteed to return the least-upper bound, but is sound and simple.

Definition 5.10. *The **quasi-join** operator, denoted by* $\mathbb{m}$*, is defined as follows:* $\langle A_1'; A_2'\rangle \mathbb{m} \langle A_1''; A_2''\rangle \stackrel{def}{=} \mathtt{Reduce}(\langle A_1' \sqcup_{\mathcal{A}_1} A_1''; A_2' \sqcup_{\mathcal{A}_2} A_2''\rangle)$. □

Theorem 5.11. ***[Soundness of Quasi-Join]*** *Let* $\langle A_1; A_2\rangle = \langle A_1'; A_2'\rangle \mathbb{m} \langle A_1''; A_2''\rangle$. *Then* $\gamma(\langle A_1; A_2\rangle) \supseteq \gamma(\langle A_1'; A_2'\rangle) \cup \gamma(\langle A_1''; A_2''\rangle)$. □

The *quasi-meet* operation is similar in flavor to quasi-join:

Definition 5.12. *The **quasi-meet** operator, denoted by* $\pitchfork$*, is defined as follows:* $\langle A_1'; A_2'\rangle \pitchfork \langle A_1''; A_2''\rangle \stackrel{def}{=} \mathtt{Reduce}(\langle A_1' \sqcap_{\mathcal{A}_1} A_1''; A_2' \sqcap_{\mathcal{A}_2} A_2''\rangle)$. □

Theorem 5.13. ***[Soundness of Quasi-Meet]*** *Let* $\langle A_1; A_2\rangle = \langle A_1'; A_2'\rangle \pitchfork \langle A_1''; A_2''\rangle$*. Then* $\gamma(\langle A_1; A_2\rangle) \supseteq \gamma(\langle A_1'; A_2'\rangle) \cap \gamma(\langle A_1''; A_2''\rangle)$. □

We now define weak semantic-reduction operators for each of the pairs of domains used in our constructions. The algorithms for these specific domains have not been stated explicitly in the literature, and are stated here for completeness. (Previous work tackled the rational-arithmetic variants of these domains.)

Algorithm 6 Algorithm for weak semantic-reduction for $\mathcal{M}[V] \star \mathcal{E}_{\mathbb{Z}_{2^w}}[V]$.

```
1: for constraint ν1 = mm[e1] ∈ M do
2:     for constraint ν2 = mm[e2] ∈ M do
3:         if E ⊑_{E_{Z_2w}} {e1 = e2} then
4:             E ← E ⊓_{E_{Z_2w}} {ν1 = ν2}
5: return ⟨M; E⟩
```

Weak Reduce for $\mathcal{M}\star\mathcal{E}_{\mathbb{Z}_{2^w}}$. Alg. 6 computes the weak semantic-reduction for the bit-vector memory domain and bit-vector equality domain. Given $\langle M; E\rangle \in \mathcal{M}[(\mathtt{mm}, V)] * \mathcal{E}_{\mathbb{Z}_{2^w}}[V]$, $\mathtt{Reduce}(\langle M; E\rangle)$ infers further equalities among V. The key insight is to model the memory map $\mathtt{mm}$ as an uninterpreted function; that is, $t_1 = t_2$ implies $\mathtt{mm}[t_1] = \mathtt{mm}[t_2]$. We are effectively approximating the theory of arrays using the theory of uninterpreted functions with equality (EUF) [14], thereby ensuring that the reduction operation is efficient. As shown in lines 3 and 4, if $e_1 = e_2$ then the algorithm infers that $\nu_1 = \nu_2$. The following example illustrates the working of Alg. 6.

Example 5.14. *Let* V *be* $\{x, y, z, u_1, u_2\}$. *Consider* $\langle M; E\rangle := \langle u_1 = \mathtt{mm}[x], u_2 = \mathtt{mm}[y+8]; x = y + z - 2, z = 10\rangle$. *From* E *we can infer that* $x = y + 8$. *Thus,* $\mathtt{mm}[x] = \mathtt{mm}[y+8]$, *and we can infer that* $u_1 = u_2$. *Thus,* E *can be updated to* $E \sqcap_{\mathcal{E}_{\mathbb{Z}_{2^w}}} \{u_1 = u_2\}$.

No further reduction is possible; thus, $\mathtt{Reduce}_{\mathcal{M}\star\mathcal{E}_{\mathbb{Z}_{2^w}}}(\langle M; E\rangle = \langle u_1 = \mathtt{mm}[x], u_2 = \mathtt{mm}[y+8]; u_1 = u_2, x = y + z - 2, z = 10\rangle$. □

The following example shows a case when $\mathtt{Reduce}_{\mathcal{M}\star\mathcal{E}_{\mathbb{Z}_{2^w}}}$ fails to find the most precise answer.

Example 5.15. *Consider the situation when analyzing Intel x86 machine code. The bit-width* w *is* 32. *The memory map* $\mathtt{mm}$ *is a map from 32-bit bit-vector to 8-bit bit-vectors, and the addressing mode is little-endian (cf. Ex. 1.3). Let* V *be* $\{r, u_1, u_2, u_3\}$. *Consider* $\langle M; E\rangle := \langle u_1 = \mathtt{mm}[r], u_2 = \mathtt{mm}[r+2], u_3 = \mathtt{mm}[r+4]; u_1 = 0, u_3 = 0; \top\rangle$. *Because neither* $r = r + 2$ *nor* $r = r + 4$ *hold, Alg. 6 cannot infer any further equalities among* u_1, u_2, *and* u_3. *Thus,* $\mathtt{Reduce}_{\mathcal{M}\star\mathcal{E}_{\mathbb{Z}_{2^w}}}(\langle M; E\rangle) = \langle M; E\rangle$.

However, $\langle M; E\rangle$ *is not the best reduction possible. Because* $u_1 = 0$ *and* $u_3 = 0$, *we can infer that* $\mathtt{mm}[r]$ *and* $\mathtt{mm}[r+4]$ *are 0. Thus, the bytes at addresses* r *through* $r + 7$ *are all 0, because we assumed little-endian addressing. Consequently,* $\mathtt{mm}[r+2] = 0$, *and* $u_2 = 0$. *Thus, the best reduction for* $\langle M; E\rangle$ *is* $\langle u_1 = \mathtt{mm}[r], u_2 = \mathtt{mm}[r+2], u_3 = \mathtt{mm}[r+4]; u_1 = 0, u_2 = 0, u_3 = 0\rangle$. *The reason the* $\mathtt{Reduce}_{\mathcal{M}\star\mathcal{E}_{\mathbb{Z}_{2^w}}}$ *algorithm was unable to deduce this was because the algorithm treats the memory map as an uninterpreted function. Thus, though sound, Alg. 6 is not always able to deduce the best possible reduced value.* □

Theorem 5.16. ***[Soundness of Alg. 6]*** *Let* $\langle M'; E'\rangle = \mathtt{Reduce}_{\mathcal{M}\star\mathcal{E}_{\mathbb{Z}_{2^w}}}(\langle M; E\rangle)$. *Then* $\gamma(\langle M'; E'\rangle) = \gamma(\langle M; E\rangle)$, *and* $\langle M'; E'\rangle \sqsubseteq \langle M; E\rangle$). □

Weak Reduce for $\mathcal{E}_{\mathbb{Z}_{2^w}} \star \mathcal{I}_{\mathbb{Z}_{2^w}}$**.** Alg. 7 computes a weak semantic-reduction for the bit-vector equality domain and the bit-vector interval domain.

Algorithm 7 Algorithm for weak semantic-reduction for $\mathcal{E}_{\mathbb{Z}_{2^w}}[P \uplus S] \star \mathcal{I}_{\mathbb{Z}_{2^w}}[S]$.

1: $E' \leftarrow E\downarrow_S$
2: **for** $s \in S$ **do**
3: **for** constraint $(s = c + \sum_{t \in S \wedge t \neq s} a_t t) \in E'$ **do**
4: $\iota \leftarrow [\![c + \sum_{t \in S \wedge t \neq s} a_t t]\!](I)$
5: $I \leftarrow I \sqcap_{\mathcal{I}_{\mathbb{Z}_{2^w}}} \top[s \in \iota]$
6: **return** $\langle E; I \rangle$

Given $\langle E; I \rangle \in \mathcal{E}_{\mathbb{Z}_{2^w}}[P \uplus S] \star \mathcal{I}_{\mathbb{Z}_{2^w}}[S]$, $\texttt{Reduce}_{\mathcal{E}_{\mathbb{Z}_{2^w}} \star \mathcal{I}_{\mathbb{Z}_{2^w}}}(\langle E; I \rangle)$ infers tighter interval bounds on the variables in S. Reduction is performed by first projecting E onto S to determine the affine relations E' that hold among the variables in S (line 1). These affine relations are used in lines 2–5 to infer tighter intervals for the variables in S. "$[\![expr]\!]$" denotes the evaluation of expression *expr* over interval domain $\mathcal{I}_{\mathbb{Z}_{2^w}}$ via interval arithmetic. New interval constraints are identified by evaluating expressions of the form $c + \sum_{t \in S \wedge t \neq s} a_t t$ over $\mathcal{I}_{\mathbb{Z}_{2^w}}$ using the current interval value I (line 4). These constraints are then incorporated into I via meet (line 5).

Example 5.17. *Let* $P := \{x, y\}$, *and* $S := \{s_1, s_2\}$, *where the bit-width of the bit-vector is 4. Consider* $\langle E; I \rangle := \langle s_1 = 2x + 2y, s_2 = x + y; s_1 \in [4, 9], s_2 \in [3, 5] \rangle$. *By projecting* E *onto the variables* $S := \{s_1, s_2\}$, *we obtain* $E' = \{s_1 = 2s_2\}$ *on line 1. On line 4, we have* $\iota = [\![2s_2]\!](I) = [6, 10]$. *Using this equation,* I *is updated on line 5; that is,* $I = \{s_1 \in [4, 9], s_2 \in [3, 5]\} \sqcap \{s_1 \in [6, 10], s_2 \in \top\} = \{s_1 \in [6, 9], s_2 \in [3, 5]\}$. *No further reduction is possible.*

Thus, $\texttt{Reduce}_{\mathcal{E}_{\mathbb{Z}_{2^w}} \star \mathcal{I}_{\mathbb{Z}_{2^w}}}(\langle E; I \rangle) = \langle s_1 = 2x + 2y, s_2 = x + y; s_1 \in [6, 9], s_2 \in [3, 5] \rangle$. *Note that Alg. 7 is not guaranteed to deduce the best possible interval constraints. The best possible reduction of* $\langle E; I \rangle$ *is* $\langle s_1 = 2x + 2y, s_2 = x + y; s_1 \in [6, 8], s_2 \in [3, 4] \rangle$. □

Theorem 5.18. ***[Soundness of Alg. 7]*** *Let* $\langle E'; I' \rangle = \texttt{Reduce}_{\mathcal{E}_{\mathbb{Z}_{2^w}} \star \mathcal{I}_{\mathbb{Z}_{2^w}}}(\langle E; I \rangle)$. *Then* $\gamma(\langle E'; I' \rangle) = \gamma(\langle E; I \rangle)$, *and* $\langle E'; I' \rangle \sqsubseteq \langle E; I \rangle$). □

Program Name	Measures of Size				Performance (sec.)		Precision ($\mathcal{BVI} \sqsubset \mathcal{E}_{\mathbb{Z}_{2^w}}$)	
	Instrs	CFGs	BBs	Branches	$\mathcal{E}_{\mathbb{Z}_{2^w}}$	$\mathcal{BVI}$	Control-Point Invariants	Procedure Summaries
finger	532	18	298	48	185	639	24/48	3/18
subst	1093	16	609	74	303	1151	11/74	3/16
label	1167	16	573	103	236	986	24/103	2/16
chkdsk	1468	18	787	119	631	1675	12/119	3/18
convert	1927	38	1013	161	441	1744	101/161	0/38
route	1982	40	931	243	749	2497	78/243	2/39
comp	2377	35	1261	224	849	2740	22/224	0/35
logoff	2470	46	1145	306	861	3253	124/306	13/46

Table 5.1: Machine-code analysis using $\mathcal{BVI}$. Columns 6–9 show the times (in seconds) for the $\mathcal{E}_{\mathbb{Z}_{2^w}}$-based analysis, and for the $\mathcal{BVI}$-based analysis; and the degree of improvement in precision measured as the number of control points at which $\mathcal{BVI}$-based analysis gave more precise invariants compared to $\mathcal{E}_{\mathbb{Z}_{2^w}}$-based analysis, and the number of procedures for which $\mathcal{BVI}$-based analysis gave more precise summaries compared to $\mathcal{E}_{\mathbb{Z}_{2^w}}$-based analysis.

5.6 Experimental Evaluation

In this section, we compare the performance and precision of the bit-vector equality domain $\mathcal{E}_{\mathbb{Z}_{2^w}}$ with that of the bit-vector inequality domain $\mathcal{BVI}$. The abstract transformers for the $\mathcal{BVI}$ domain were synthesized using the approach given in Section 6. The weak semantic-reduction operator described in Alg. 7 was used in the implementation of the compose operation needed for interprocedural analysis.

Experimental Setup. We analyzed a corpus of Windows utilities using the WALi [50] system for weighted pushdown systems (WPDSs). Tab. 5.1 lists several size parameters of the examples (number of instructions, procedures, basic blocks, and branches).[1] The weight on each WPDS rule encodes the abstract transformer for a basic block B of the program, including a jump or branch to a successor block. A formula φ_B is created

[1] Due to the high cost of the WPDS construction, all analyses excluded the code for libraries. Because register `eax` holds the return value from a call, library functions were modeled approximately (albeit unsoundly, in general) by "`havoc(eax)`".

that captures the concrete semantics of B, and then the weight for B is obtained by performing $\hat{\alpha}(\varphi_B)$. We used EWPDS merge functions [56] to preserve caller-save and callee-save registers across call sites. The post* query used the FWPDS algorithm [55].

View-selection Heuristic. Given the set of machine registers P, the view-constraints C_s were computed as $C_s := \bigcup_{r_i \in P} \{s_{i1} = r_i, s_{i2} = r_i + 2^{31}\}$. In particular, the view-variable s_{i2} allows us to keep track of whether r_i, when treated as a signed value, is less-than or equal 0.

Performance. Columns 6 and 7 of Tab. 5.1 list the time taken, in seconds, for $\mathcal{E}_{\mathbb{Z}_{2^w}}$-based analysis, and the $\mathcal{BVI}$-based analysis. On average (geometric mean), $\mathcal{BVI}$-based analysis is about 3.5 times slower than $\mathcal{E}_{\mathbb{Z}_{2^w}}$-based analysis.

Precision. We compare the procedure summaries, and the invariants for each control point—i.e., the point just before a branch instruction. Column 8 lists the number of control points at which $\mathcal{BVI}$-based analysis gave more precise invariants compared to $\mathcal{E}_{\mathbb{Z}_{2^w}}$-based analysis. $\mathcal{BVI}$-based analysis gives more precise invariants at up to 63% of control points, and, on average, $\mathcal{BVI}$-based analysis improves precision for 29% of control points. Column 9 lists the number of procedures for which $\mathcal{BVI}$-based analysis gave more precise summaries compared to $\mathcal{E}_{\mathbb{Z}_{2^w}}$-based analysis. $\mathcal{BVI}$-based analysis gives more precise summaries for up to 17% of procedures, and, on average $\mathcal{BVI}$-based analysis gave better summaries for 9.3% of procedures.

5.7 Related Work

Other work on identifying bit-vector-inequality invariants includes Brauer and King [11, 12] and Masdupuy [66]. Masdupuy proposed a relational

abstract domain of interval congruences on rationals. One limitation of his machinery is that the domain represents diagonal grids of parallelepipeds, where the dimension of each parallelepiped equals the number of variables tracked (say n). In our work, we can have any number of view-variables, which means that the point-spaces represented can be constrained by more than n constraints.

Brauer and King employ bit-blasting to synthesize abstract transformers for the interval and octagon [68] domains. One of their papers uses universal-quantifier elimination on Boolean formulas [11]; the other avoids quantifier elimination [12]. Compared with their work, we avoid the use of bit-blasting and work directly with representations of sets of w-bit bit-vectors. The greatly reduced number of variables that comes from working at word level opens up the possibility of applying our methods to much larger problems; as discussed in §5.6, we were able to apply our methods to interprocedural program analysis. The equality domain $\mathcal{E}_{\mathbb{Z}_{2^w}}$ that we work with can capture relations on an arbitrary number of variables, and thus so can the domains that we construct using $\mathcal{V}$. Compared with octagons, which are limited to two variables and coefficients of ± 1, the advantage is that our domains can express more interesting invariants and procedure summaries. In particular, Octagon-based summaries would be limited to one pre-state variable and one post-state variable.

The view-product combinator $\mathcal{V}$ is a compositional generalization of the construction used in SubPoly. SubPoly is constructed as a reduced product of an affine-equality domain $\mathcal{K}$ over rationals [49], and an interval domain $\mathcal{J}$ over rationals [19] with slack view-variables S to communicate between the two domains. SubPoly can be constructed by applying $\mathcal{V}$ to $\mathcal{K}$ and $\mathcal{J}$, as follows:

$$\text{SubPoly}[P \uplus S] \stackrel{\text{def}}{=} \mathcal{V}[\mathcal{K}[P], S, C_s, \mathfrak{J}], \tag{5.1}$$

where C_s is the view-constraint for S and $\mathfrak{J}(V_1, V_2) \stackrel{\text{def}}{=} \mathcal{J}[V_1]$.

When an analysis system works with two or more reasoning techniques, there is often an opportunity to share information to improve the precision of both. The principle is found in the classic papers of Cousot and Cousot [21] and Nelson and Oppen [80]. In practice, there are a range of choices as to what might be shared, and our work represents one point in that design space. The algorithms for weak semantic-reduction (Alg. 6 and 7) adapt techniques for theory combination that have been used in Satisfiability Modulo Theory (SMT) solvers [27, 33].

The work of Chang and Leino [16] is similar in spirit to ours. They developed a technique for extending the properties representable by a given abstract domain from schemas over variables to schemas over terms. To orchestrate the communication of information between domains, they designed the congruence-closure abstract domain, which introduces variables to stand for subexpressions that are alien to a base domain; to the base domain, these expressions are just variables. Their scheme for propagating information between domains is mediated by the e-graph of the congruence-closure domain. In contrast, our method can make use of past work on synthesizing best abstract operations [84, 100] to propagate information between domains. As discussed in §5.5, we also employ less precise, more pragmatic procedures that use information in one domain to iteratively refine information in another domain. Cousot et al. [22] have recently studied the iterated pairwise exchange of observations between components as a way to compute an overapproximation of a reduced product.

5.8 Chapter Notes

Aditya Thakur and Thomas Reps supervised me in this work. Aditya Thakur was instrumental in providing the key insights and helped me in the implementation of the view-combinator and reduced-product framework.

6 SOUND BIT-PRECISE NUMERICAL DOMAINS FRAMEWORK FOR INEQUALITIES

This chapter tackles the challenges of implementing a bit-precise relational domain capable of expressing program invariants. The chapter describes the design and implementation of a new framework for abstract domains, called the *Bit-Vector-Sound Finite-Disjunctive* (*BVSFD*) domains, which are capable of capturing useful program invariants such as inequalities over bit-vector-valued variables.

The $\mathcal{BVI}$ domain in Chapter 5 can handle certain kind of bit-vector inequalities, but it needs the client to provide a template for inequalities. Moreover, the domain is incapable of expressing simple inequalities of form $x \leqslant y$, because they have variables on both side of the inequality.

Simon et al. [96] introduced sound wrap-around to ensure that the polyhedral domain is sound over bit-vectors. The wrap-around operation is called selectively on the abstract-domain elements while calculating the fixpoint. The operation is called selectively to preserve precision while not compromising on soundness with respect to the concrete semantics. The Verasco static analyzer [48] provides a bit-precise parameterized framework for abstract domains using the wrap-around operation. This approach has two disadvantages:

- The wrap-around operation almost always loses information due to calls on join: the convex hull of the elements that did not overflow with those that did usually does not satisfy many inequality constraints.
- They do not show how to create abstract transformers automatically.

We introduce a class of abstract domains, called $BVS(\mathcal{A})$, that is sound with respect to bitvectors whenever $\mathcal{A}$ is sound with respect to mathematical integers. The $\mathcal{A}$ domain can be any numerical abstract domain. For example, it can be the polyhedral domain, which can represent useful

program invariants as inequalities. We also describe how to create abstract transformers for $BVS(\mathcal{A})$ that are sound with respect to bitvectors. For $v \subseteq Var$ and $av \in \mathcal{A}$, we denote the result by $WRAP_v(av)$; the operation performs wraparound on av for variables in v. We give an algorithm for $WRAP_v(av)$ that works for any relational abstract domain (see §6.3.1). We use a finite number of disjunction of $\mathcal{A}$ elements—captured in the domain $\mathcal{FD}_d(\mathcal{A})$—to help retain precision. The finite disjunctive domain is parameterized by the maximum number of disjunctions allowed in the domain (referred to as d). Note that d=1 is the same as convex polyhedra with wrap-around [96].

Problem statement.

> *Given a relational numeric domain over integers, capable of expressing inequalities, (i) provide an automatic method to create a relational abstract domain that can capture inequalities over bit-vector-valued variables; (ii) create sound bit-precise abstract transformers; and (iii) use them to identify inequality invariants over a set of program variables.*

Related work and contributions. Our work incorporates a number of ideas known from the literature, including

- the use of relational abstract domains [23, 71, 58, 69, 97, 82] that are sound over mathematical integers and capable of expressing inequalities.
- the use of a wrap-around operation [96, 13, 3] to ensure that the abstraction is sound with respect to the concrete semantics of the bitvector operations.
- the use of finite disjunctions [88, 4, 37] over abstract domains to obtain more precision.

- the use of instruction reinterpretation [47, 79, 81, 64, 60, 30] to obtain an abstract transformer automatically for an edge from a basic block to its successor.

Our contribution is that we put all of these to work together in a parameterized framework, along with a mechanism to increase precision by performing wrap-around on abstract values lazily.

- We propose a framework for abstract domains, called $BVSFD_{\mathtt{d}}(\mathcal{A})$, to express bit-precise relational invariants by performing wrap-around over abstract domain $\mathcal{A}$ and using disjunctions to retain precision. This abstract domain is parameterized by a positive value $\mathtt{d}$, which provides the maximum number of disjunctions that the abstract domain can make use of.
- We provide a generic technique via reinterpretation to create the abstract transformer for the path through a basic block to a given successor, such that the transformer incorporates lazy wrap-around.
- We present experiments to show how the performance and precision of $BVSFD_{\mathtt{d}}$ analysis changes with the tunable parameter $\mathtt{d}$.

§6.1 introduces the terminology and notation used in the rest of the chapter. §6.2 demonstrates our framework with the help of an example. §6.3 introduces the $BVSFD_{\mathtt{d}}$ abstract-domain framework, and formalizes abstract-transformer generation for the framework. §6.4 presents experimental results.

6.1 Terminology

$\mathcal{A}[V]$ denotes the specific instance of $\mathcal{A}$ that is defined over vocabulary V, where V is a tuple of variables $(v_1, v_2, ..., v_k)$. Each variable v_i also has an associated size in bits, denoted by $s(v_i)$. The domain $\mathcal{C}$ is the powerset of the set of concrete states.

6.1.1 Concretization.

Given an abstract value $A \in \mathcal{A}[V]$, where V consists of n variables $(v_1, v_2, ..., v_k)$, the concretization of A, denoted by $\gamma_{\mathcal{A}[V]}(A)$, is the set of concrete states covered by A.

A concrete state σ is a mapping from variables to their concrete values, $\sigma : V \rightarrow \Pi_{v \in V} BV^{s(v)}$, where $s(v)$ is the size of variable v in bits and BV^b is a bitvector with b bits.

$$\gamma_{\mathcal{A}[V]}(A) = \bigcup_{(a_1, a_2, .., a_k) \in A} \mu_V(bv_1, bv_2, ..., bv_k), \textbf{where } bv_i = a_i \% 2^{s(v_i)} \textbf{ for } i \in 0..n$$

$\mu_V(bv_1, bv_2, ..., bv_k)$ takes a tuple of bitvectors corresponding to vocabulary V and returns all concrete stores where each variable v_i in V has the value bv_i.

6.1.2 Wrap-around operation.

The wrap-around operation, denoted by $WRAP^{ty}_{V'}(A)$, takes an abstract-domain value $A \in \mathcal{A}[V]$, a subset V' of the vocabulary V, and the desired type ty for vocabulary V'. It returns an abstract value A' such that the wrap-around behavior of the points in A is soundly captured. This operation is performed by displacing the concrete values in $\gamma_{\mathcal{A}[V]}(A)$ that are outside the bitvector range for ty in vocabulary V' to the correct bitvector range by appropriate linear transformations.

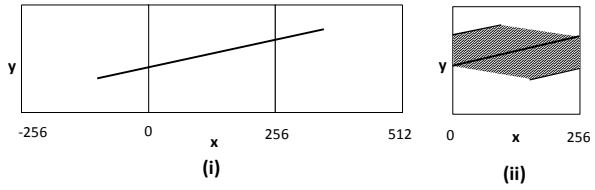


Figure 6.1: Wrap-around on variable x, treated as an unsigned char.

For example, in Fig. 6.1, the result of calling wrap-around on the line

$$
\begin{array}{rcl}
\langle \mathtt{ELang} \rangle & :: & (\mathtt{Block})* \\
\langle \mathtt{Block} \rangle & :: & \mathtt{l} : (\langle \mathtt{Stmt} \rangle \ \mathbf{;})^* \ \langle \mathtt{Next} \rangle \\
\langle \mathtt{Next} \rangle & :: & \mathbf{jump}\ \mathtt{l}\mathbf{;} \\
 & | & \mathbf{if}\ v\ \langle \mathtt{Op} \rangle_{\langle \mathtt{Type} \rangle}\ \langle \mathtt{Expr} \rangle\ \mathbf{then\ jump}\ \mathtt{l}\ \mathbf{;}\ \langle \mathtt{Next} \rangle \\
\langle \mathtt{Op} \rangle & :: & < \mid \leqslant \mid = \mid \neq \mid \geqslant \mid > \\
\langle \mathtt{Expr} \rangle & :: & n \mid n * v + \langle \mathtt{Expr} \rangle \\
\langle \mathtt{Stmt} \rangle & :: & v = \langle \mathtt{Expr} \rangle \\
 & | & v{:}\langle \mathtt{Type} \rangle = v{:}\langle \mathtt{Type} \rangle \\
\langle \mathtt{Type} \rangle & :: & (\mathbf{uint} \mid \mathbf{int})\ \langle \mathtt{Size} \rangle \\
\langle \mathtt{Size} \rangle & :: & \mathbf{1} \mid \mathbf{2} \mid \mathbf{4} \mid \mathbf{8}
\end{array}
$$

in (i) for variable x leads to an abstract value that is the abstraction of the points in the three line segments in (ii). For the abstract domain of polyhedra, that abstraction is the shaded area in (ii).

6.1.3 Soundness.

An abstract value $A \in \mathcal{A}[V]$ is sound with respect to a set of concrete values $C \in \mathcal{C}$, if $\gamma_{\mathcal{A}[V]} \supseteq C$.

6.1.4 $\mathcal{L}(\mathtt{ELang})$: A Concrete language featuring finite integer arithmetic.

We borrow the simple language featuring finite-integer arithmetic from § 2.1 of [96] (with minor syntactic changes). An ELang program is a sequence of basic blocks with execution starting from the first block. Each basic block consists of a sequence of statements and a list of control-flow instructions.

The statements are restricted to an assignment of a linear expression or a cast operation. The control-flow instruction consists of either a jump statement, or a conditional that is followed by more control-flow instructions. The assignment and condition instructions expect the variable and the expression involved to have the same type.

6.2 Overview

In this section, we motivate and illustrate the design of our analysis using the *BVSFD* domain.

Example 6.1. *This example illustrates a function* f *that takes two 32-bit integers* x *and* y *at different rates, and resets their values to zero in case* y *is negative or overflows to a negative value. The function summary that we would like to obtain states that the relationship* $x' \leqslant y'$ *holds. Here, the unprimed and primed variables denote the pre-state vocabulary variables and the post-state vocabulary variables, respectively.*

```
L0:  f(int x, int y) {
L1:    assume(x<=y)
L2:    while(*) {
L3:      if(*)
L4:        x=x+1, y=y+1
L5:      y=y+1
L6:      if (y<=0)
L7:        x=0, y=0
L8:    }
END: }
```

This example illustrates that merely detecting overflow would not be useful to assert the $x <= y$ *relationship at the end of the function.* □

6.2.1 Creation of Abstract Transformers

Consider the analysis for Ex. 6.1 with the abstract domain $BVSFD_2(\mathcal{OCT})$. The first step involves constructing the abstraction of the concrete operations in the program as abstract transformers.

For instance, the abstract transformer for the concrete operations starting from node L0 and ending at node L2, denoted by $\tau^{\sharp}_{L0\to L2}$, is defined as

$$\{m \leqslant x, y \leqslant M \wedge x' = x \wedge y' = y \wedge x' \leqslant y'\},$$

where m and M represent the minimum and maximum values for a signed 32-bit integer, respectively. The constraints $\{m \leqslant x, y \leqslant M\}$ are the bounding constraints on the pre-state vocabulary that are added because L0 is the entry point of the function, and the function expects three 32-bit signed values x and y as input. The equality constraints $\{x' = x, y' = y\}$ specify that the variables x and y are unchanged. Finally, the constraint $\{x' \leqslant y'\}$ is added as a consequence of the $assume$ call.

Now consider other concrete transformations, such as L4 → L5 and L5 → L6. For the transformation L4 → L5, the values for x' and y' might overflow because of the increment operations at L4. Consequently, the value of the incoming variable y in the transformation L5 → L6 might have overflowed as well. There are two ways to design the abstract transformer to deal with these kind of scenarios: 1) a naive eager approach, 2) a lazy approach.

Eager Abstract Transformers.

In the naive eager approach, the abstract transformers are created such that the pre-state vocabulary is always bounded as per the type requirements. For this example, that would mean that the pre-state vocabulary variables x and y are bounded in the range $[m, M]$. Consequently, the abstract transformers for L4 → L5 and L5 → L6 are:

- $\tau^{\sharp E}_{L4\to L5} = \{m \leqslant x, y \leqslant M \wedge x' = x + 1 \wedge y' = y + 1\}$
- $\tau^{\sharp E}_{L5\to L6} = \{m \leqslant x, y \leqslant M \wedge x' = x \wedge y' = y + 1\}$.

Because the eager approach expects the pre-state vocabulary to be bounded, an abstract-composition operation $a_1 \circ a_2$, where a_1 and a_2 are abstract transformers, needs to call the *WRAP* operation (§6.1.2) for the entire post-state vocabulary of a_2, for correctness. For instance, let $a_1 = \{m \leqslant u \leqslant M \wedge u' = u\}$ and $a_2 = \{m \leqslant u \leqslant M \wedge u' = M + 1\}$. The abstract transformer a_1 preserves u and a_2 changes the value of u' to $M+1$. The composition of these operations matches the pre-state vocabulary of a_1

with the post-state vocabulary of a_2, by renaming them to the same temporary variables and performing a meet. For this particular example, it will perform $\{m \leqslant u'' \leqslant M \wedge u' = u''\} \sqcap WRAP_{u''}(\{m \leqslant u \leqslant M \wedge u'' = M{+}1\})$, where it has matched the pre-state vocabulary variable u of a_1 with the post-state vocabulary variable u' of a_2, by renaming them both to a temporary variable u''. Note that in the absence of the *WRAP* operation on the post-state vocabulary of a_2, the meet operation above will return the empty element $\bot$. This result would be unsound because the value of u' in a_2 should have overflowed to m.

Now consider the composition $\tau^{\sharp E}_{L5\to L6} \circ \tau^{\sharp E}_{L4\to L5}$. After matching, composition will perform the meet of:

- $\{m \leqslant x'', y'' \leqslant M \wedge x' = x'' \wedge y' = y'' + 1\}$
- $WRAP_{\{x'',y''\}}\{m \leqslant x, y \leqslant M \wedge x'' = x + 1 \wedge y'' = y + 1\}$

The result of *WRAP* will be a join of four values. The four values are the combinations of cases where x'' and y'' might or might not overflow. As a result, the final composition will give an abstract transformer that overapproximate those four values. For $BVSFD_2(\mathcal{OCT})$, it will result in a loss of precision because it cannot express the disjunction of these four values precisely.

Lazy Abstract Transformers.

The eager approach to creating abstract transformers forces a call to *WRAP* at each compose operation. The lazy approach can avoid unnecessary calls to *WRAP* by not adding any bounding constraints to the pre-state vocabulary. The abstract transformer $\tau^{\sharp L}_{L4\to L5}$ is defined as $\{x' = x{+}1 \wedge y' = y{+}1\}$ and $\tau^{\sharp L}_{L5\to L6}$ is defined as $\{x' = x \wedge y' = y + 1\}$. The abstract transformer $\tau^{\sharp L}_{L4\to L5}$ is sound, because the concretization of the abstract transformer, denoted by $\gamma(\tau^{\sharp L}_{L4\to L5})$, overapproximates the collecting concrete semantics for L4 $\to$ L5 (see proposition 1 in [96]). A similar argument can be made for the abstract transformer $\tau^{\sharp L}_{L5\to L6}$. The composition $\tau^{\sharp L}_{L5\to L6} \circ \tau^{\sharp L}_{L4\to L5}$

gives $\{x' = x + 1 \wedge y' = y + 2\}$. Thus, the lazy abstract transformer can retain precision by avoiding unnecessary calls to *WRAP*. However, one cannot avoid calling *WRAP* for every kind of abstract transformer and still maintain soundness. Consider the abstract transformer $\tau^{\sharp L}_{L5\to L7}$, which is similar to $\tau^{\sharp L}_{L5\to L6}$, but must additionally handle the branch condition for L6 → L7. Defining it in a similar vein as L4 → L5 will result in $\{x' = x \wedge y' = y + 1 \wedge y' \leqslant 0\}$. While the first three constraints are sound, the fourth constraint representing the branch condition, is unsound with respect to the concrete semantics. The reason is that y' might overflow to a negative value, in which case the condition evaluates to true and the branch to L7 is taken. However, the abstract transformer does not capture that behavior and is, therefore, unsound with respect to the concrete semantics. To achieve soundness in the presence of a branch condition, the following steps are performed for each variable v' involved in a branch condition:

- A backward dependency analysis is performed to find the subset V_b of the pre-state vocabulary on which v' depends. For the edge L5 → L6, only y' is involved in a branch condition. The backward-dependency analysis yields $V_b = \{y\}$, because the only pre-state vocabulary variable that y' depends on is y.
- The bounding constraints for the vocabulary V_b are added to the abstract transformer. For example, after adding bounding constraints, $\tau^{\sharp L}_{L5\to L6}$ becomes $\{m \leqslant y \leqslant M \wedge x' = x \wedge y' = y + 1 \wedge y' \leqslant 0\}$.
- The wrap operation is called on the abstract transformer for the variable v'. For the edge L5 → L6, this step will soundly set the abstract transformer to the disjunction of
  - $\{m \leqslant y \leqslant M - 1 \wedge x' = x \wedge y' = y + 1 \wedge y' \leqslant 0\}$
  - $\{y = M - 1 \wedge x' = x \wedge y' = m\}$

  Note that the abstract domain $BVSFD_2(\mathcal{OCT})$ can precisely express the above disjunction because the number of disjunctions is $\leqslant 2$.

Our analysis uses the lazy approach to create abstract transformers, because it can provide more precise function summaries. Fig. 6.2 illustrates the lazy abstract transformers generated for Ex. 6.1.

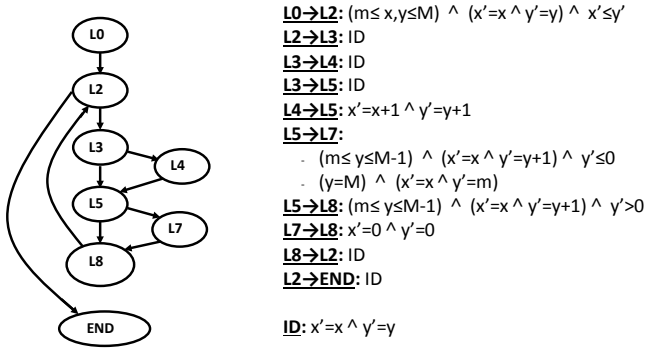


Figure 6.2: Lazy abstract transformers with the $BVSFD_2(\mathcal{POLY})$ domain for Ex. 6.1. ID refers to the identity transformation.

6.2.2 Fixed-point computation

To obtain function summaries, an iterative fixed-point computation needs to be performed. Tab. 6.1 provides some snapshots of the fixpoint analysis with the $BVSFD_2(\mathcal{OCT})$ domain for Ex. 6.1.

To simplify the discussion, we focus only on three program points: L2, L5, and L7. Each row in the table shows the intermediate value of path summaries from L0 to each of the three program points. Quiescence is discovered during the fifth iteration. The abstract value in row (i) and column L2 shows the intermediate path summary for L2 calculated after one iteration of the analysis. It states that the pre-state vocabulary variables x and y are bounded and neither of them has been modified, because at this iteration the analysis has not considered the paths that go through the loop. At row (i) and column L5, the domain precisely captures the disjunction of two paths arising at the conditional at L3. The abstract

Table 6.1: Snapshots in the fixed-point analysis for Ex. 6.1 using the $BVSFD_2(\mathcal{OCT})$ domain. $B_{v_1,v_2,..,v_k}$ are the bounding constraints for the variables v_1,v_2,..v_k.

Node	L2	L5	L7
(i)	• $B_{x,y} \wedge (x' = x) \wedge (y' = y)$	• $B_{x,y} \wedge (x' = x) \wedge (y' = y)$ • $B_{x,y} \wedge (x' = x+1) \wedge (y' = y+1)$	• $B_{x,y} \wedge (m \leqslant y' \leqslant 0) \wedge (x \leqslant x' \leqslant x+1) \wedge (y \leqslant y' \leqslant y+2) \wedge (x' \leqslant y')$ • $B_{x,y} \wedge (m \leqslant y' \leqslant m+1) \wedge (x \leqslant x' \leqslant x+1)$
(ii)	• $B_{x,y,y'} \wedge (x \leqslant x' \leqslant x+1) \wedge (y \leqslant y' \leqslant y+2) \wedge (x' \leqslant y')$ • $B_{x,y} \wedge (x' = 0) \wedge (y' = 0)$	• $B_{x,y} \wedge (x \leqslant x' \leqslant x+2) \wedge (y \leqslant y' \leqslant y+3) \wedge (x' \leqslant y')$ • $B_{x,y} \wedge (0 \leqslant x' \leqslant 1) \wedge (y' = x')$	• $B_{x,y} \wedge (m \leqslant y' \leqslant 0) \wedge (x \leqslant x' \leqslant x+2) \wedge (y \leqslant y' \leqslant y+4) \wedge (x' \leqslant y')$ • $B_{x,y} \wedge (m \leqslant y' \leqslant m+3) \wedge (x \leqslant x' \leqslant x+2)$
(iii)	• $B_{x,y,y'} \wedge (x \leqslant x' \leqslant x+2) \wedge (y \leqslant y' \leqslant y+4) \wedge (x' \leqslant y')$ • $B_{x,y} \wedge (0 \leqslant x' \leqslant 1) \wedge (0 \leqslant y' \leqslant 2) \wedge (x' \leqslant y')$	• $B_{x,y} \wedge (x \leqslant x' \leqslant x+3) \wedge (y \leqslant y' \leqslant y+5) \wedge (x' \leqslant y')$ • $B_{x,y} \wedge (0 \leqslant x' \leqslant 2) \wedge (0 \leqslant y' \leqslant 4) \wedge (x' \leqslant y')$	• $B_{x,y} \wedge (m \leqslant y' \leqslant 0) \wedge (x \leqslant x' \leqslant x+3) \wedge (y \leqslant y' \leqslant y+6) \wedge (x' \leqslant y')$ • $B_{x,y} \wedge (m \leqslant y' \leqslant m+5) \wedge (x \leqslant x' \leqslant x+3)$
(iv)	• $B_{x,y,y'} \wedge (x \leqslant x') \wedge (y \leqslant y') \wedge (x' \leqslant y')$ • $B_{x,y,y'} \wedge (0 \leqslant x') \wedge (x' \leqslant y')$	• $B_{x,y} \wedge (x \leqslant x') \wedge (y \leqslant y') \wedge (x' \leqslant y')$ • $B_{x,y} \wedge (0 \leqslant x') \wedge (x' \leqslant y')$	• $B_{x,y} \wedge (m \leqslant y' \leqslant 0) \wedge (x \leqslant x') \wedge (y \leqslant y') \wedge (x' \leqslant y')$ • $B_{x,y,y'} \wedge (m \leqslant y' \leqslant 0)$

value at row (i), column L7 is obtained as the composition of the abstract transformer L5 → L7 with the path summary at L5 in row (i). The abstract-composition operations for abstract values with disjunctions performs abstract composition for all pairs of abstract transformers in the arguments, and then does a join on that set of values. To obtain the abstract transformer for L7 in row (i), it computes the join of the following values:

1. $B_x \wedge (m \leqslant y \leqslant M-1) \wedge (x' = x) \wedge (y' = y+1) \wedge (y' \leqslant 0)$
2. $B_x \wedge (m \leqslant y \leqslant M-2) \wedge (x' = x+1) \wedge (y' = y+2) \wedge (y' \leqslant 0)$
3. $B_x \wedge (y = M) \wedge (x' = x) \wedge (y' = m)$

4. $B_x \wedge (M \leqslant y \leqslant M-1) \wedge (x' = x) \wedge (m \leqslant y' \leqslant m+1)$

Our abstract-domain framework uses a distance heuristic (see §6.3.1) to merge abstract values that are closest to each other. For this particular case, the abstract transformers (1) and (2) describe the scenarios where y' does not overflow, and are merged to give the first disjunct of row (i), column L7. Similarly, the abstract transformers (3) and (4) describe the scenarios where y' overflows, and are merged to give the second disjunct of row (i), column L7.

In the second iteration, shown in row (ii), the first disjunct for L2 is the join of the effect of first iteration of the loop, where x and y are incremented, with the old value, where x and y are unchanged. Additionally, the second disjunct in row (ii), column L2 captures the case where both x and y are set to 0 at program point L7. Iteration (iii) proceeds in a similar manner, and finally the value at L2 saturates due to widening. The value of L2 at iteration (iv) is propagated to the end of the function to give the following function summary:

- $B_{x,y,y'} \wedge (x \leqslant x') \wedge (y \leqslant y') \wedge (x' \leqslant y')$
- $B_{x,y,y'} \wedge (0 \leqslant x') \wedge (x' \leqslant y')$

Thus, the function summary enables us to establish that $x' \leqslant y'$ is true at the end of the function.

6.3 The *BVSFD* Abstract-Domain Framework

In this section, we present the intuition and formalism behind the design and implementation of the *BVSFD* abstract-domain framework.

6.3.1 Abstract-Domain Constructors

BVSFD uses of the following abstract-domain constructors:

- Bit-Vector-Sound Constructor: This constructor, denoted by $BVS[\mathcal{A}]$, takes an arbitrary abstract domain and constructs a bit-precise ver-

Algorithm 8 Wrap for a single variable

```
1: function WRAP(a, v, ty)
2:     if a is ⊥ then
3:         return ⊥
4:     (m, M) ← Range(ty)
5:     s ← (M − m) + 1
6:     [l, u] ← GetBounds(a, v)
7:     if l ≠ −∞ ∧ u ≠ ∞ then
8:         ⟨q_l, q_u⟩ ← ⟨⌊(l − m)/s⌋, ⌊(u − m)/s⌋⟩
9:     b ← C(m ≤ v) ⊓ C(v ≤ M)
10:    if l = −∞ ∨ u = ∞ ∨ (q_u − q_l) > t then
11:        return RM_{v}(a) ⊓ b
12:    else
13:        return ⋃_{q∈[q_l,q_u]} ((a ▷ v := v − qs) ⊓ b)
```

Type	Operation	Description
$\mathcal{A}$	$\top$	*top element*
$\mathcal{A}$	$\bot$	*bottom element*
bool	$(a_1 == a_2)$	*equality*
$\mathcal{A}$	$(a_1 \sqcap a_2)$	*meet*
$\mathcal{A}$	$(a_1 \sqcup a_2)$	*join*
$\mathcal{A}$	$(a_1 \nabla a_2)$	*widen*
$\mathcal{A}$	$\pi_W(a)$	*project on vocabulary W*
$\mathcal{A}$	$\mathrm{RM}_W(a)$	*remove vocabulary W*
$\mathcal{A}$	$\rho(a_1, v_1, v_2)$	*rename variable v_1 to v_2*
$\mathcal{A}$	$\mathcal{C}(le_1 \text{ op } le_2)$	*construct abstract value*
set$[\mathcal{A}]$	$WRAP^{ty}_W(a_1)$	*wrap vocabulary W*
D	$\mathcal{D}(a_1, a_2)$	*distance*

Figure 6.3: Abstract-domain interface for $\mathcal{A}$.

sion of the domain that is sound with respect to the concrete semantics. It needs the base domain $\mathcal{A}$ to provide a *WRAP* operator.

- Finite-Disjunctive Constructor: This constructor, denoted by $\mathcal{FD}_d[\mathcal{A}]$, takes an abstract domain $\mathcal{A}$ and a parameter d, and constructs a finite-disjunctive version of the domain, where the number of disjunctions in any abstract value should not exceed d. This constructor uses a distance measure, denoted by $\mathcal{D}$, to determine which disjuncts are combined when the number of disjunctions exceeds d.

The $BVSFD_d[\mathcal{A}]$ domain is constructed as $BVS[\mathcal{FD}_d[\mathcal{A}]]$. Fig. 6.3 shows the interface that the base abstract domain $\mathcal{A}$ needs to provide to instantiate the $BVSFD_d[\mathcal{A}]$ framework.

The first seven operations are standard abstract-domain operations. The remove-vocabulary operation $\mathrm{RM}_W(\mathcal{A})$, can be implemented as $\pi_{V-W}(\mathcal{A})$, where V is the full vocabulary. The rename operation $\rho(\mathcal{A}, v_1, v_2)$ can be easily implemented in most abstract-domain implementations through simple variable renaming and/or variable-order permutation. The construct operation, denoted by $\mathcal{C}$, constructs an abstract value from the linear constraint $le_1 = le_2$, where le_1 and le_2 are linear ex-

pressions, and operation $op \in \{=, \leqslant, \geqslant\}$. This operation is available for any numeric abstract domain that can capture linear constraints. If the domain cannot express a specific type of linear constraint (for instance, the octagon domain cannot express linear constraints with more than two variables), it can safely return $\top$. The *WRAP* operation is similar to the wrap operation in [96], except that it returns a set of abstract-domain values, whose disjunction correctly captures the wrap-around behavior. The *WRAP* operation from [96] is modified to return a *set* of abstract-domain values by placing values in a set instead of calling join. Alg. 6.3.1 shows how wrap is performed for a single variable. It takes the abstract value a and perform wrap-around on variable v, treated as type ty. Line 4 obtains the range for a type, and line 5 calculates the size of that range. Line 6 obtains the range of v in abstract value a. This operation can be implemented by projecting a on v and reading the resultant interval. Lines 7-8 calculate the range of the quadrants for the variable v. Line 9 computes the bounding constraints on v, treated as type ty. Line 10 compares the number of quadrants to a threshold t. If the number of quadrants exceeds t, the result is computed by removing constraints on v in a using the RM operation, and adding the bounding constraints to the final result. Otherwise, for each quadrant, the appropriate value is computed by displacing the quadrants to the correct range. The displacing of the abstract value a for the quadrant q, denoted by $a \rhd v := v - qs$, is implemented as $RM_{\{u\}}(\rho(a, v, u) \sqcap \mathcal{C}(v = u - qs))$. We used $t = 16$ in the experiments reported in §6.4.

We implement $\mathcal{D}(a_1, a_2)$ by converting a_1 and a_2 into the strongest boxes b_1 and b_2 that overapproximate a_1 and a_2, and computing the distance between b_1 and b_2. A box is essentially a conjunction of intervals on each variable in the vocabulary. We measure the distance between two boxes as a tuple (d_1, d_2), where d_1 is the number of incompatible intervals, and d_2 is the sum of the distances between *compatible* intervals. Two intervals are considered to be *incompatible* if one is unbounded in

a direction that the other one is not. For example, intervals $[0, \infty]$ and $[-7, \infty]$ are compatible, but $[0, 17]$ and $[-7, \infty]$, and $[-\infty, 12]$ and $[-7, \infty]$ are not. The *distance* between two compatible intervals is 0 if their intersection is non-empty; otherwise, it is the difference of the lower bound of the higher interval and the upper bound of the lower interval. For example, the distance between $[0, 11]$ and $[17, 21]$ is $(17 - 11) = 6$. Given two distances $d = (d_1, d_2)$ and $d' = (d'_1, d'_2)$, $d > d'$ iff either (i) $d_1 > d'_1$, or (ii) $d_1 = d'_1 \wedge d_2 > d'_2$. If the number of disjunctions in an abstract value exceeds parameter d, the abstract-domain constructor $\mathcal{FD}_d[\mathcal{A}]$ merges (using join) the pair of abstract-domain elements that are closest as measured by the distance measure.

6.3.2 Abstract Transformers

In this section, we describe how the abstract transformers are generated using reinterpretation (§2.3.1). The reinterpretation consists of a domain of abstract transformers $BVSFD_d[\mathcal{A}[V; V']]$, a domain of abstract integers $BVSFD_d^{INT}[\mathcal{A}[t; V]]$, and operations to lookup a variable's value in the post-state of an abstract transformer and to create an updated version of a given abstract transformer [30]. Here, V denotes the pre-state vocabulary variables, V' denotes the post-state vocabulary variables, and t denotes a temporary variable not in V or V'. Given blocks $B : [l : s_1; ...s_l; nxt]$ and $B' : [l' : t'; ...t_l; nxt]$ in an ELang program (see §6.1.4), where B' is a successor of B, reinterpretation of B can provide an abstract transformer for the transformation that starts from the first instruction in B and ends in the first instruction in B', denoted by $B \rightarrow B'$.

Rule 1 in Fig. 6.4 specifies how abstract-transformer evaluation for basic-block pairs feeds into abstract-transformer evaluation on a sequence of statements. The evaluation on a sequence of statements starts with the identity abstract transformer, denoted by id. Rule 2 states that the abstract transformer for a sequence of instruction can be broken down into an

Basic Block:

$$[\![B \to B']\!]^\sharp_{Block} = [\![[s_1;...;s_l;nxt]]\!]^\sharp_{Seq}(id, l') \tag{6.1}$$

$$[\![[s_1;...;s_l;nxt]]\!]^\sharp_{Seq}(a, l') = [\![[s_2;...;s_l;nxt]]\!]^\sharp_{Seq}(([\![s_1]\!]^\sharp_{Stmt}a), l') \tag{6.2}$$

Control Flow:

$$[\![[nxt]]\!]^\sharp_{Seq}(a, l') = [\![nxt]\!]^\sharp_{Next}(a, l') \tag{6.3}$$

$$[\![\textbf{jump } l'']\!]^\sharp_{Next}(a, l') = \textbf{ if } l'' \textit{ is } l' \textbf{ then } a \textbf{ else } \bot \tag{6.4}$$

$$[\![\textbf{if } v\ op_{type}\ exp \textbf{ then jump } l''; nxt]\!]^\sharp_{Next}(a, l') = \tag{6.5}$$

$$\textbf{if } l'' \textit{ is } l' \textbf{ then } a \sqcap p \textbf{ else } a \sqcap n, \textbf{where}$$

$$p = v_{Int}\ op_{type}\ exp_{Int}, n = v_{Int}\ !op\ exp_{Int},$$

$$v_{Int} = LazyWrap^{type}([\![v]\!]^\sharp_{Expr}a), exp_{Int} = LazyWrap^{type}([\![exp]\!]^\sharp_{Expr}a)$$

Assignments:

$$[\![v = exp]\!]^\sharp_{Stmt}a = update(a, v', [\![exp]\!]^\sharp_{Expr}a) \tag{6.6}$$

$$[\![v_1 : t_1 = v_2 : t_2]\!]^\sharp_{Stmt}a = \textbf{if } s(t_1) \leqslant s(t_2) \tag{6.7}$$

$$\textbf{then } update(a, v_1', [\![v_2]\!]^\sharp_{Expr}a)$$

$$\textbf{else } update(a, v_1', LazyWrap^{t_1}([\![v_2]\!]^\sharp_{Expr}a))$$

Expressions:

$$[\![n]\!]^\sharp_{Expr}a = const\_int(n) \tag{6.8}$$

$$[\![v]\!]^\sharp_{Expr}a = lookup(v', a) \tag{6.9}$$

$$[\![exp_1 * exp_2]\!]^\sharp_{Expr}a = mult([\![exp_1]\!]^\sharp_{Expr}a, [\![exp_2]\!]^\sharp_{Expr}a) \tag{6.10}$$

$$[\![exp_1 + exp_2]\!]^\sharp_{Expr}a = add([\![exp_1]\!]^\sharp_{Expr}a, [\![exp_2]\!]^\sharp_{Expr}a) \tag{6.11}$$

Abstract Integers:

$$const\_int(n) = \mathcal{C}(t = n) \tag{6.12}$$

$$get\_const(i) = \textbf{if } \pi_{\{t\}}(i) \textbf{ is } \{t = n\} \textbf{ then } (true, n) \textbf{ else } (false, 0), \tag{6.13}$$

$$mult(i_i, i_2) = \textbf{let } (b, n) = get\_const(i_i) \textbf{ in} \tag{6.14}$$

$$(\textbf{if } b \textbf{ then } RM_{\{t'\}}(\rho(i_1, t, t') \sqcap \mathcal{C}(t = n * t')) \textbf{ else } \top)$$

$$add(i_i, i_2) = RM_{\{t',t''\}}\big(\rho(i_1, t, t') \sqcap \rho(i_2, t, t'') \sqcap \mathcal{C}(t = t' + t'')\big) \tag{6.15}$$

$$i_i\ op\ i_2 = RM_{\{t',t''\}}\big(\rho(i_1, t, t') \sqcap \rho(i_2, t, t'') \sqcap \mathcal{C}(t'\ op\ t'')\big), \tag{6.16}$$

$$\textbf{where } op \in \{=, \leqslant, \geqslant\}$$

$$i_i > i_2 = RM_{\{t',t''\}}\big(\rho(i_1, t, t') \sqcap \rho(i_2, t, t'') \sqcap \mathcal{C}(t' \geqslant t'' + 1)\big) \tag{6.17}$$

$$i_i < i_2 = RM_{\{t',t''\}}\big(\rho(i_1, t, t') \sqcap \rho(i_2, t, t'') \sqcap \mathcal{C}(t' \leqslant t'' - 1)\big) \tag{6.18}$$

$$i_i \neq i_2 = (i_i < i_2) \sqcup (i_1 > i_2) \tag{6.19}$$

$$LazyWrap^{type}(i) = WRAP^{type}_{\{t\}}(i \sqcap \mathcal{C}(B_W)), \textbf{where } W = DependentVoc_t(i) \tag{6.20}$$

Variable lookup and update:

$$lookup(a, v') = \pi_{V \cup \{t\}}(a \sqcap \{t = v'\}) \tag{6.21}$$

$$update(a, v', i) = RM_{\{v'\}}(a) \sqcap RM_{\{t\}}(i \sqcap \{v' = t\}) \tag{6.22}$$

Figure 6.4: Reinterpretation semantics for $\mathcal{L}(\text{ELANG})$.

abstract transformer for a smaller sequence of instruction, by recursively performing statement-level abstract interpretation $[\![.]\!]^{\sharp}_{Stmt}$ on the first instruction in the sequence. In this rule and subsequent $[\![.]\!]^{\sharp}_{Next}$ and $[\![.]\!]^{\sharp}_{Stmt}$ rules, "a" denotes the intermediate abstract transformer value. It starts as id at the beginning of the instruction sequence, and gets updated or accessed by assignment and control-flow statements in the sequence.

Rules 3,4, and 5 handle control-flow statements. Rule 3 delegates the responsibility of executing the last instruction in the statement sequence to $[\![.]\!]^{\sharp}_{Next}$. Rule 4 deals with unconditional-jump instructions. The label is checked against a goal label and either $\perp$ or the current transformer a is returned accordingly. Rule 5 handles conditional branching. It conjoins the input transformer with p in the true case and n in the false case. p and n are calculated by performing abstract versions of op and $!op$, respectively, on the sound abstract integers corresponding to v_{Int} and v_{exp}. Here, $!op$ denotes the negation of the op symbol. For example, negation of $\leqslant$ is $>$. The sound version of an abstract integer is created by calling $LazyWrap^{type}$ (see rule 20). This function is the key component behind lazy abstract-transformer generation (see §6.2.1). In our implementation, we compute $DependentVoc_t(i)$ by looking at the constraints in i and returning the vocabulary subset $W \subseteq V$ that depend on t. B_W refers to the bounding constraints on the variables in vocabulary W.

Rules 6 and 7 handle assignment statements. Assignment to a linear expression merely performs a post-state-vocabulary update on the current abstract transformer "a." Note that this rule does not call *WRAP* even though the result of computing exp can go out of bounds. Rule 7 handles the cast operation. $s(t_1)$ and $s(t_2)$ gets the size for the types t_1 and t_2, respectively. For a downcast operation, it performs simple update. In the case of upcast, $LazyWrap^{type}$ is called to preserve soundness (see Section 6 of [96]).

Rules 8, 9, 10, and 11 handle reinterpretation of expressions. Rules

8, 10, and 11 delegate computation to the corresponding abstract-integer operations. Rule 9 performs a variable lookup in the current value of abstract transformer "a."

Rules 12 to 20 deal with the operations on abstract integers in $BVSFD_d^{INT}[\mathcal{A}[t;V]]$. Rule 12 constructs an abstract integer from a constant. Rule 13 finds out if a variable is a constant. This operation is used by abstract multiplication (Rule 14) to determine if the multiplication of two abstract integers is linear or not. Rules 14-18 use vocabulary-removal(RM) and variable-rename operations (ρ) to ensure that the vocabulary of the output is $\{t\} \cup V$.

Rules 21 and 22 are variable lookup and update operations. Lookup takes an abstract transformer $a \in BVSFD_d$ and a variable $v' \in V'$, and returns the abstract integer $i \in BVSFD_d^{INT}$ such that the relationship of t with V in i is the same as the relationship of v' with V in a. The variable-update operation works in the opposite direction. Update takes an abstract transformer $a \in BVSFD_d$, a variable $v' \in V'$, and $i \in BVSFD_d^{INT}$, and returns $a' \in BVSFD_d$ such that the relationship of v' with V in a' is the same as the relationship of t with V in i, and all the other relationships that do not involve v' remain the same.

6.4 Experimental Evaluation

In this section, we compare the performance and precision of the bit-precise disjunctive-inequality domain $BVSFD_d$ for different values of d. We perform this comparison for the base domains of octagons and polyhedra. The abstract transformers for the *BVSFD* domain were automatically synthesized for each path through a basic block to one of its successor by using reinterpretation (see §6.3.2). We also perform array-out-of-bounds checking to quantify the usefulness of the precision gain for different values of d. The experiments were designed to shed light on the following

questions:

1. How much does the performance of the analysis degrade as $\mathtt{d}$ is increased?
2. How much does the precision of the analysis increase as $\mathtt{d}$ is increased?
3. What is the value of $\mathtt{d}$ beyond which no further precision is gained?
4. What is the effect of adding sound bit-precise handling of variables on performance and precision?

6.4.1 Experimental Setup

Given a C file, we first create the corresponding LLVM bitcode [57, 63]. We then feed the bitcode to our solver, which uses the WALi [50] system to create a Weighted Pushdown System (WPDS) corresponding to the LLVM CFG. The transitions in the WPDS correspond to CFG edges from a basic block to one of its successors. The semiring weights on the edges are abstract transformers in the $BVSFD_{\mathtt{d}}$ abstract domain. We then perform interprocedural analysis by performing post* followed by the path-summary operation [85] to obtain overapproximating function summaries and an overapproximation of the reachable states at all branch points. We used EWPDS merge functions [56] to preserve local variables across call sites. We used the `Pointset_Powerset` [4] framework in the Parma Polyhedra Library [3, 82] to implement the $\mathcal{FD}_{\mathtt{d}}[\mathcal{A}]$ constructor. For each example used in the experiments, we use a timeout of 200 seconds.

6.4.2 Assertion Checking

For this set of experiments, we picked the subset of the SVCOMP [7] loop benchmarks for which all assertions hold. Because our analyzer is sound, we are interested in the percentage of true assertions that it can verify. Tab. 6.2 provides information about the benchmarks that we used. We

Table 6.2: Information about the loop benchmarks containing true assertions, a subset of the SVCOMP benchmarks.

Benchmark	examples	instructions	assertions
loop-invgen	18	2373	90
loop-lit	15	1173	16
loops	34	3974	32
loop-acceleration	19	1001	19
total	86	8521	158

performed the analysis on these examples and performed assertion checking by checking whether the program points corresponding to assertion failures had the bottom abstract state.

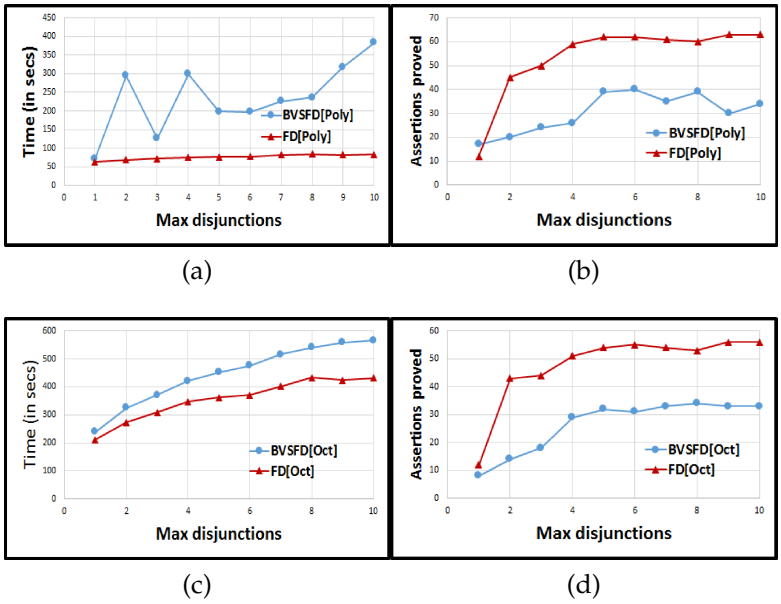


Figure 6.5: Precision and performance numbers for SV-COMP loop benchmarks.

Fig. 6.5a and Fig. 6.5b show the performance and precision numbers, respectively, for the loop SVCOMP benchmarks, with $\mathcal{POLY}$ as the base domain. The results answer the experimental questions as follows:

1. With two exceptions, at $d = 2$ and $d = 4$, the performance steadily decreases as the number of maximum allowed disjunctions d is

increased. The analysis times for $d = 2$ and $d = 4$ do not fit the trend because one example times out for $d = 2$ or $d = 4$, but does not time out for $d = 3$ or $d = 5$. This behavior can be attributed to the non-monotonic behaviors of the finite-disjunctive join and widening operations.

2. The precision, measured as the number of proved assertions, increases from $d = 1$ to $d = 6$. From $d = 7$ onwards the change in precision is haphazard.
3. The analysis achieves the best precision at $d = 6$, where it proves 40 out of 157 assertions.
4. The sound analysis using $BVSFD_d[\mathcal{POLY}]$ is 1.1-4.6 times slower than the unsound analysis using $\mathcal{FD}_d[\mathcal{POLY}]$, and is able to prove 44-142% of the assertions obtained with $\mathcal{FD}_d[\mathcal{POLY}]$.

Fig. 6.5c and Fig. 6.5d show the performance and precision numbers, respectively, for the loop SVCOMP benchmarks, with $\mathcal{OCT}$ as the base domain. The results answer the experimental questions as follows:

1. The performance steadily decreases with increase in d.
2. The precision, measured as the number of proved assertions, increases from $d = 1$ to $d = 5$. From $d = 5$ onwards the precision is essentially unchanged.
3. The analysis achieves the best precision at $d = 8$, where it proves 34 out of 157 assertions.
4. The sound analysis using $BVSFD_d[\mathcal{OCT}]$ is 1.1-1.3 times slower than the unsound analysis using $\mathcal{FD}_d[\mathcal{OCT}]$, and is able to prove 33-67% of the assertions obtained with $\mathcal{FD}_d[\mathcal{OCT}]$.

In our experiments, we found the $BVSFD_d[\mathcal{OCT}]$-based analysis to be 1-3.3x slower than $BVSFD_d[\mathcal{POLY}]$. This slowdown occurs because the maximum vocabulary size in abstract transformers is $\leqslant 12$, and abstract operations for octagons are slower than that of polyhedra for such a small vocabulary size.

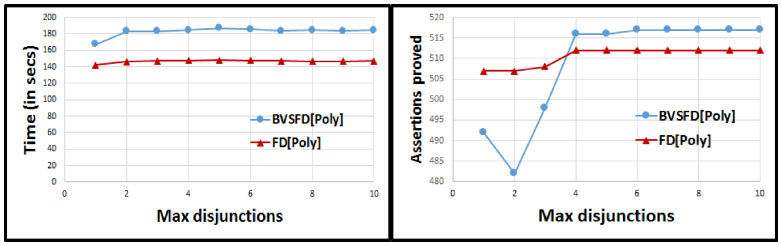


Figure 6.6: Precision and performance numbers for SV-COMP array benchmarks with $\mathcal{POLY}$ as the base domain.

6.4.3 Array-Bounds Checking

We perform array-bound checking using invariants from the $BVSFD_d$ analysis. For each array access and update we create an error state that is reached when an array bound is violated. These array-bounds checks are verified by checking if the path summaries at the error states are $\bot$. There are 88 examples in the benchmark, with a total of 14,742 instructions and 598 array-bounds checks. Fig. 6.6 lists the number of array-bound checks proven for each application, for different values of d, for the SVCOMP array benchmarks.

The results answer the experimental questions as follows:

1. The performance of the analysis increases by 9% from $d = 1$ to $d = 2$. After $d = 2$, the performance stabilizes.
2. With one exception at $d = 2$, the precision—measured as the number of array-bounds checks proven—increases from $d = 1$ to $d = 4$. From $d = 4$ onwards the precision is essentially unchanged.
3. The analysis achieves the best precision at $d = 4$, where it proves 515 out of the 598 array-bounds checks.
4. The sound analysis using $BVSFD_d[\mathcal{POLY}]$ is 1.18-1.26 times slower than the unsound analysis using $\mathcal{FD}_d[\mathcal{POLY}]$, and is able to prove 95-101% of the array-bound checks obtained with $\mathcal{FD}_d[\mathcal{POLY}]$.

6.5 Chapter Notes

Thomas Reps supervised me in the writing, and provided useful comments for this work.

7 CONCLUSION AND FUTURE WORK

This dissertation described novel abstract domains and abstract-domain frameworks that can soundly provide bit-vector-sound program invariants, that can be used to verify assertions and provide loop and function summaries. This dissertation i) compared two abstract domains for bit-vector-sound equalities, ii) extended one of the domains to provide a novel framework of bit-vector-sound equalities, iii) introduced a bit-vector-sound inequality domain capable of expressing memory variables, and iv) introduced a bit-vector-sound numerical inequality domain framework. The thesis makes the following contributions:

- **Bit-Vector Precise Equality Abstract Domains**
  - **Abstract Domains of Affine Relations:** An affine relation is a *linear-equality* constraint over a given set of variables that hold machine integers. In this work, we compare the MOS and KS abstract domains, along with several variants. We show that MOS and KS are, in general, *incomparable* and give sound interconversion methods for KS and MOS. We introduce a third domain for representing affine relations, called AG, which stands for *affine generators*. Furthermore, we present an experimental study comparing the precision and performance of analyses with the KS and MOS domains.
  - **Abstraction Framework for Affine Transformers:** In this work, we define the Affine-Transformers Abstraction Framework, which represents a new family of numerical abstract domains. This framework is parameterized by a base numerical abstract domain, and allows one to represent a set of affine transformers (or, alternatively, certain disjunctions of transition formulas). Specifically, this framework is a generalization of the MOS domain. The choice of the base abstract domain allows the client

to have some control over the performance/precision trade-off.

- **Bit-Vector Precise Inequality Abstract Domains**
  - **An Abstract Domain for Bit-vector Inequalities:** This work describes the design and implementation of a new abstract domain, called the *Bit-Vector Inequality* domain, which is capable of capturing certain inequalities over bit-vector-valued variables (which represent a program's registers and/or its memory variables). This domain tracks properties of the values of selected registers and portions of memory via *views*, and provides automatic heuristics to gather equality and inequality views from the program. Furthermore, experiments are provided to show the usefulness of the Bit-Vector Inequality domain.
  - **Sound Bit-Precise Numerical Domains Framework for Inequalities:** This work introduces a class of abstract domains, parameterized on a base domain, that is sound with respect to bitvectors whenever the base domain is sound with respect to mathematical integers. The base domain can be any numerical abstract domain. We also describe how to create abstract transformers for this framework that incorporate lazy wrap-around to achieve more precision, without sacrificing soundness with respect to bitvectors. We use a finite number of disjunctions of base-domain elements to help retain precision. Furthermore, we present experiments to empirically demonstrate the usefulness of the framework.

In the remainder of this chapter, we present conclusion for the thesis and future directions for the bit-vector-precise equality domains and bit-vector-precise inequality domains, respectively, described in this dissertation.

7.1 Bit-vector-precise Equality Domains

In Chapter 3, we considered MOS and KS abstract domains, along with several variants, and studied how they relate to each other. We showed that MOS and KS are, in general, *incomparable*, and introduced a third domain for representing affine relations, called AG, which stands for *affine generators*, and showed that it is the generator representation of the KS domain.

In Chapter 4, we provided analysis techniques to abstract the behavior of the program as a set of affine transformations over bit-vectors. While the relationship between MOS and KS/AG is interesting, we realized that there is a general framework that can allow one to build the MOS domain from the KS/AG domain. Thus, we generalized the ideas used in the MOS domain—in particular, to have an abstraction of *sets of affine transformers*—but to provide a way for a client of the abstract domain to have some control over the performance/precision trade-off. Toward this end, we defined a new family of numerical abstract domains, denoted by ATA[$\mathcal{B}$]. (ATA stands for Affine-Transformers Abstraction.) Following our observation, ATA[$\mathcal{B}$] is parameterized by a base numerical abstract domain $\mathcal{B}$, and allows one to represent a set of affine transformers (or, alternatively, certain disjunctions of transition formulas). Specifically, MOS can be defined as ATA[KS/AG].

Future Directions. One obvious opportunity for future work is to provide an implementation of the ATA framework, and compare the performance and precision of this framework for different base domains $\mathcal{B}$. The main challenge in this work would be to figure out the best way of generating abstract transformers for guard statements. The best method to use is not obvious because there are multiple choices for implementing guards and depending on the assertion, one choice might work better than the other. A second direction would be to extend this generic framework so

that it can be applied for sound program analysis for floating-point numbers. There has been a lot of work on tracking floating-point arithmetic soundly in an abstract domain [70, 72, 17]. Once the generic abstract-transformation-abstraction framework is implemented for floating-point numbers, these base domains can be used to provide new sound abstract domains capable of expressing a certain class of disjunctions of affine transformers over floating-point numbers. The biggest challenge is to provide floating-point-sound — yet precise — methods to perform abstract composition.

7.2 Bit-vector-precise Inequality Domains

In Chapter 5, we expanded the set of techniques available for abstract interpretation and model checking of machine code. This work described the design and implementation of a new abstract domain, called the *Bit-Vector Inequality* domain, which is capable of capturing certain inequalities over bit-vector-valued variables (which represent a program's registers and/or its memory variables). This domain tracks properties of the values of selected registers and portions of memory via *views*, and provides automatic heuristics to gather equality and inequality views from the program. Furthermore, experiments are provided to show the usefulness of the Bit-Vector Inequality domain.

Chapter 6 described a second approach to designing and implementing a bit-precise relational domain capable of expressing inequality invariants. This work presents the design and implementation of a new framework for abstract domains, called the *Bit-Vector-Sound Finite-Disjunctive* (*BVSFD*) domains, which are capable of capturing useful program invariants such as inequalities over bit-vector-valued variables. We introduced a class of abstract domains, called $BVS(\mathcal{A})$, that is sound with respect to bitvectors whenever $\mathcal{A}$ is sound with respect to mathematical integers. The $\mathcal{A}$ do-

main can be any numerical abstract domain. We use a finite number of disjunctions of $\mathcal{A}$ elements—captured in the domain $\mathcal{FD}_{\mathtt{d}}(\mathcal{A})$—to help retain precision, where $\mathtt{d}$ is the maximum number of disjunctions that the abstract domain can make use of. We also described a generic technique to create abstract transformers for $BVS(\mathcal{A})$ that are sound with respect to bitvectors. The abstract transformers incorporate a lazy wrap-around mechanism to achieve more precision. Finally, we presented experiments to show how the performance and precision of $BVSFD_{\mathtt{d}}$ analysis changes with the tunable parameter $\mathtt{d}$.

Future Directions One interesting future research direction is improving the lazy wrap-around method to improve precision. Intuitively, the wrap-around operation performs join operations that leads to loss of precision. Lazy wrap-around aims to improve the precision by only calling wrap-around procedure at guard and cast statements. While calling wrap-around at these points is essential for correctness, the result of a wrap-around can be modified to a value that requires fewer disjunctions to express by using a dual of the wrap-around operation, thus leading to improved precision. The challenge is in implementing the dual operation efficiently.

§2.3.2 introduced symbolic abstraction as one of the automatic methods to create abstract transformers. However, this method is not applicable to the polyhedral domain because the polyhedral domain has both infinite ascending and descending chains. The symbolic abstraction method is desirable because the concrete semantics of bit-twiddling operations can be precisely described in Quantifier-Free Bit-Vector (QFBV) logic. Recent improvements in SMT optimization techniques [59, 8] provides insights in designing a symbolic-abstraction operation for the polyhedral domain. Moreover, with the help of the dual of wrap-around operation, sound and precise abstract transformers can be created for *BVSFD* domain.

A.1 Soundness of MOS to AG transformation

Thm. 3.7 states that the transformation from MOS to AG given in §3.3.2 is sound. *Suppose that $\mathcal{B}$ is an MOS element such that, for every* $B \in \mathcal{B}$, $B = \left[\begin{array}{c|c} 1 & c_B \\ \hline 0 & M_B \end{array}\right]$ *for some* $c_B \in \mathbb{Z}_{2^w}^{1\times k}$ *and* $M_B \in \mathbb{Z}_{2^w}^{k\times k}$. *Define* $G_B = \left[\begin{array}{c|cc} 1 & 0 & c_B \\ \hline 0 & I & M_B \end{array}\right]$ *and* $G = \bigsqcup_{AG} \{G_B \mid B \in \mathcal{B}\}$. *Then,* $\gamma_{MOS}(\mathcal{B}) \subseteq \gamma_{AG}(G)$.

Proof. First, recall that for any two AG elements E and F, $E \sqcup_{AG} F$ equals $\textsc{Howellize}\left(\left[\begin{smallmatrix} E \\ F \end{smallmatrix}\right]\right)$. Because Howellize does not change the row space of a matrix, $\gamma_{AG}\left(E \sqcup_{AG} F\right)$ equals $\gamma_{AG}\left(\left[\begin{smallmatrix} E \\ F \end{smallmatrix}\right]\right)$. By the definition of G, we know that $\gamma_{AG}(G) = \gamma_{AG}(\mathbb{G})$, where $\mathbb{G}$ is all of the matrices G_B stacked vertically. Therefore, to show that $\gamma_{MOS}(\mathcal{B}) \subseteq \gamma_{AG}(G)$, we show that $\gamma_{MOS}(\mathcal{B}) \subseteq \gamma_{AG}(\mathbb{G})$.

Suppose that $(\vec{v}, \vec{v}\,') \in \gamma_{MOS}(\mathcal{B})$. Then, for some vector $\vec{w}$,

$$\left[\begin{array}{c|c} 1 & \vec{v} \end{array}\right] \left(\sum_{B\in\mathcal{B}} w_B B\right) = \left[\begin{array}{c|c} 1 & \vec{v}\,' \end{array}\right].$$

If we break this equation apart, we see that

$$\sum_{B\in\mathcal{B}} w_B = 1 \quad \text{and} \quad \sum_{B\in\mathcal{B}} w_B c_B + \vec{v} \left(\sum_{B\in\mathcal{B}} w_B M_B\right) = \vec{v}\,'.$$

Let $\otimes$ denote Kronecker product. Now consider the following product, which uses $(\vec{w} \otimes \left[\begin{array}{c|c} 1 & \vec{v} \end{array}\right])$ as a vector of coefficients for the rows of $\mathbb{G}$:

$$\begin{aligned} \left(\vec{w} \otimes \left[\begin{array}{c|c} 1 & \vec{v} \end{array}\right]\right)\mathbb{G} &= \sum_{B\in\mathcal{B}} \left[\begin{array}{c|cc} w_B & w_B \vec{v} I & w_B c_B + w_B \vec{v} M_B \end{array}\right] \\ &= \left[\begin{array}{c|cc} \sum_{B\in\mathcal{B}} w_B & \vec{v}\left(\sum_{B\in\mathcal{B}} w_B\right) & \sum_{B\in\mathcal{B}} w_B c_B + \vec{v}\left(\sum_{B\in\mathcal{B}} w_B M_B\right) \end{array}\right] \\ &= \left[\begin{array}{c|cc} 1 & \vec{v} & \vec{v}\,' \end{array}\right]. \end{aligned}$$

Thus, $\left[\begin{array}{c|cc} 1 & \vec{v} & \vec{v}' \end{array}\right]$ is a linear combination of the rows of $\mathbb{G}$, and so $(x, x') \in \gamma_{AG}(\mathbb{G})$. Therefore, $\gamma_{MOS}(\mathcal{B}) \subseteq \gamma_{AG}(G)$. □

A.2 Soundness of KS Without Pre-State Guards to MOS transformation

When $G = \left[\begin{array}{c|cc} 1 & 0 & b \\ \hline 0 & I & M \\ 0 & 0 & R \end{array}\right]$, *then* $\gamma_{AG}(G) = \gamma_{MOS}(\textsc{Shatter}(G))$.

Proof.

$$\begin{aligned}
(x, x') \in \gamma_{AG}(G) &\Leftrightarrow \exists v\colon \left[\begin{array}{c|cc} 1 & x & v \end{array}\right] G = \left[\begin{array}{c|cc} 1 & x & x' \end{array}\right] \\
&\Leftrightarrow \exists v\colon b + xM + vR = x' \\
&\Leftrightarrow \exists v\colon \left[\begin{array}{c|c} 1 & x \end{array}\right] \left(\left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] + \sum_i v_i \left[\begin{array}{c|c} 0 & R_i \\ \hline 0 & 0 \end{array}\right] \right) = \left[\begin{array}{c|c} 1 & x' \end{array}\right] \\
&\Leftrightarrow (x, x') \in \gamma_{MOS}(\textsc{Shatter}(G))
\end{aligned}$$

□

A.3 Soundness of KS Without Pre-State Guards to MOS transformation

Lemma A.1. *Suppose that* M *and* N *are square matrices of equal dimension such that*

1. M *has only ones and zeroes on its diagonal,*
2. *if* $M_{i,i} = 1$, *then* $M_{h,i} = 0$ *for all* $h \neq i$, *and*
3. *if* $M_{i,i} = 0$, *then* $N_{i,h} = 0$ *for all* h.

Then, $MN = N$.

Proof. We know $(MN)_{i,j} = \sum_h M_{i,h}N_{h,j}$. By Items 2 and 3, if $h \neq i$ then either $M_{i,h} = 0$ or $N_{h,j} = 0$, so $(MN)_{i,j} = M_{i,i}N_{i,j}$. If $M_{i,i} = 0$, then by Item 2, $N_{i,j} = 0$; otherwise, $M_{i,i} = 1$. In either case, $(MN)_{i,j} = N_{i,j}$, as we require. □

Lemma A.2. *When* $G = \left[\begin{array}{c|cc} 1 & a & b \\ \hline 0 & J & M \\ 0 & 0 & R \end{array}\right]$, *such that* $\left[\begin{array}{c|c} 1 & a \\ \hline 0 & J \end{array}\right]$ *and* $\left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right]$ *satisfy the conditions of Lem. A.1, then* $\gamma_{AG}(G) \subseteq \gamma_{MOS}(\textsc{Shatter}(G))$.

Proof.

$$\begin{aligned}(x, x') \in \gamma_{AG}(G) &\Rightarrow \exists v, v' \colon \left[\begin{array}{c|cc} 1 & v & v' \end{array}\right] G = \left[\begin{array}{c|cc} 1 & x & x' \end{array}\right] \\ &\Rightarrow \exists v, v' \colon \left[\begin{array}{c|c} 1 & v \end{array}\right] \left[\begin{array}{c|c} 1 & a \\ \hline 0 & J \end{array}\right] = \left[\begin{array}{c|c} 1 & x \end{array}\right] \\ &\qquad \wedge \left[\begin{array}{c|c} 1 & v \end{array}\right] \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] + v' \left[\begin{array}{c|c} 0 & R \end{array}\right] = \left[\begin{array}{c|c} 1 & x' \end{array}\right]\end{aligned}$$

By Lem. A.1, $\left[\begin{array}{c|c} 1 & v \end{array}\right] \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] = \left[\begin{array}{c|c} 1 & v \end{array}\right] \left[\begin{array}{c|c} 1 & a \\ \hline 0 & J \end{array}\right] \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] = \left[\begin{array}{c|c} 1 & x \end{array}\right] \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right]$, so

$$\begin{aligned}(x, x') \in \gamma_{AG}(G) &\Rightarrow \exists v' \colon \left[\begin{array}{c|c} 1 & x \end{array}\right] \left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] + v' \left[\begin{array}{c|c} 0 & R \end{array}\right] = \left[\begin{array}{c|c} 1 & x' \end{array}\right] \\ &\Rightarrow \exists v' \colon \left[\begin{array}{c|c} 1 & x \end{array}\right] \left(\left[\begin{array}{c|c} 1 & b \\ \hline 0 & M \end{array}\right] + \sum_i v'_i \left[\begin{array}{c|c} 0 & R_i \\ \hline 0 & 0 \end{array}\right] \right) = \left[\begin{array}{c|c} 1 & x' \end{array}\right] \\ &\Rightarrow (x, x') \in \gamma_{MOS}(\textsc{Shatter}(G))\end{aligned}$$

□

Now we prove Thm. 3.11. *For* $G \in AG$, $\gamma_{AG}(G) \subseteq$

γ_{MOS} (Shatter (MakeExplicit(G))).

Proof. Without loss of generality, assume that G has $2k+1$ columns and is in Howell form.

MakeExplicit(G) consists of two loops. In the first loop, every row r with leading index $i \leqslant k+1$ for which the rank of the leading value is greater than 0 is generalized by creating from r a row s, which is added to G, such that s's leading index is also i, but its leading value is 1. Consequently, after the call on Howellize(G) in line 8 of MakeExplicit, the leading value of the row with leading index i is 1.

In the second loop, the matrix is expanded by all-zero rows so that any row with leading index $i \leqslant k+1$ is placed in row i.

Thus, for any AG element G, we can decompose MakeExplicit(G) into the matrix $\left[\begin{array}{c|cc} 1 & c & b \\ \hline 0 & J & M \\ 0 & 0 & R \end{array}\right]$, where $c, b \in \mathbb{Z}_{2^w}^{1\times k}$; $J, M \in \mathbb{Z}_{2^w}^{k\times k}$; and $R \in \mathbb{Z}_{2^w}^{r\times k}$ for some $r \leqslant k$. Moreover, we know that

- J is upper-triangular,
- J has only ones and zeroes on its diagonal,
- if $J_{j,j} = 1$, then column j of J is zero everywhere else, and
- if $J_{j,j} = 0$, then row j of J and row j of M are all zeroes.

By these properties, Lem. A.2 holds, so we know that

$$\gamma_{AG}(G) \subseteq \gamma_{MOS}(\text{Shatter}(\text{MakeExplicit}(G))).$$

□

Definition B.1. *Two module spaces* R *and* S *are* **perpendicular** *(denoted by* $R \perp S$*) if*

1. $r \in R \wedge s \in S \Rightarrow rs^t = 0$,
2. $(\forall r \in R : rs^t = 0) \Rightarrow s \in S$, *and*
3. $(\forall s \in S : rs^t = 0) \Rightarrow r \in R$.

□

Lemma B.2. *If* $R \perp S$ *and* $R \perp S'$*, then* $S = S'$.

Lemma B.3. *For any matrix* M, $\operatorname{row} M \perp \operatorname{null}^t M$.

These are standard facts in linear algebra; their standard proofs essentially carry over for module spaces.

Lemma B.4. *If* $R \perp R'$ *and* $S \perp S'$*, then* $R + S \perp R' \cap S'$.

Proof. Pick G_R and G_S so that $\operatorname{row} G_R = R$ and $\operatorname{row} G_S = S$. Because the rows of a matrix are linear generators of its row space,

$$R + S = \operatorname{row} \left[\begin{smallmatrix} G_R \\ G_S \end{smallmatrix}\right], \qquad \text{so, by Lem. B.3,} \qquad R + S \perp \operatorname{null}^t \left[\begin{smallmatrix} G_R \\ G_S \end{smallmatrix}\right].$$

Because each row of a matrix acts as a constraint on its null space,

$$R + S \perp \operatorname{null}^t G_R \cap \operatorname{null}^t G_S.$$

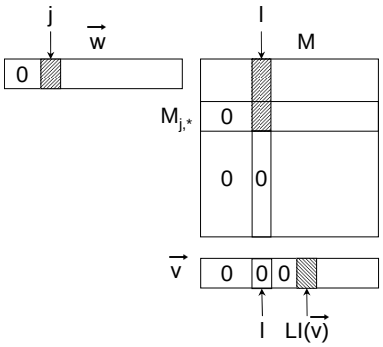
By Lem. B.3 again, we know that $\operatorname{row} G_R \perp \operatorname{null}^t G_R = R \perp R'$, so $\operatorname{null}^t G_R = R'$ by Lem. B.2. Similarly, $\operatorname{null}^t G_S = S'$. Thus, $R + S \perp R' \cap S'$. □

Note. Recall from §3.1 that $[M]_i$ is the matrix that consists of all rows of M whose leading index is i or greater. For any row r, define $LI(r)$ to be

the leading index of r. Define e_i to be the vector with 1 at index i and 0 everywhere else.

Theorem B.5. *If matrix M is in Howell form, and $x \in \operatorname{row} M$, then $x \in \operatorname{row}([M]_{LI(x)})$.*

Proof. Pick v so that $x = vM$, let $j \stackrel{\text{def}}{=} LI(v)$, and let $\ell \stackrel{\text{def}}{=} LI(M_{j,*})$. If $\ell \geqslant LI(x)$, then we already know that $x \in \operatorname{row}([M]_{LI(x)})$. Otherwise, assume $\ell < LI(x)$. Under these conditions, as depicted in the diagram below,



- $(vM)_\ell = 0$, because $LI(vM) = LI(x) > \ell$,
- $M_{h,\ell} = 0$ for any $h > j$, by Rule 1 of Defn. 3.1, and
- $v_h = 0$ for any $h < j$, because $j = LI(v)$.

Therefore, $0 = (vM)_\ell = \sum_h v_h M_{h,\ell} = v_j M_{j,\ell}$. Thus, because $j = LI(v)$, we know that $LI(v_j M_{j,*})$ is strictly greater than $\ell = LI(M_{j,*})$.

Because multiplication by invertible values can never change nonzero values to zero, we have $LI(v_j M_{j,*}) = LI\left(2^{\operatorname{rank}(v_j)} M_{j,*}\right)$. Thus, by Rule 4 of Defn. 3.1, we know that $v_j M_{j,*}$ can be stated as a linear combination of rows $j+1$ and greater. That is, $v_j M_{j,*} \in \operatorname{row}([M]_{j+1})$, or equivalently, $v_j M_{j,*} = uM$ with $LI(u) \geqslant j+1$. We can thus construct $v' = v - v_j e_j + u$ for which $x = v'M$ and $LI(v') \geqslant j+1$.

By employing this construction iteratively for increasing values of j, we can construct $x = yM$ with $LI\left(M_{LI(y),*}\right) \geqslant LI(x)$. Consequently, x can

be stated as a linear combination of rows with leading indexes $LI(x)$ or greater; i.e., $x \in \text{row}([M]_{LI(x)})$. □

Now we prove Thm. 3.14. *Suppose that* M *has* c *columns. If matrix* M *is in Howell form,* $x \in \text{null}^t M$ *if and only if* $\forall i : \forall y_1, \ldots y_{i-1} : \begin{bmatrix} y_1 & \cdots y_{i-1} & x_i & \cdots & x_c \end{bmatrix} \in \text{null}^t([M]_i)$.

Proof. We know that $\text{row}\, M \perp \text{null}^t M$, and that $\text{row}([M]_i) \perp \text{null}^t([M]_i)$. Let E_i be the module space generated by $\{e_j \mid j < i\}$, and let F_i be the module space generated by $\{e_j \mid j \geqslant i\}$. Clearly, $E_i \perp F_i$. By Thm. B.5, we have that $\text{row}([M]_i) = \text{row}\, M \cap F_i$. Thus,

$$\text{null}^t([M]_i) \perp \text{row}\, M \cap F_i.$$

By Lem. B.4, we therefore have

$$\text{null}^t([M]_i) = \text{null}^t M + E_i, \tag{B.1}$$

Because $(\text{null}^t M + E_i)$ is the set $\left\{x + y \,\middle|\, x \in \text{null}^t M \wedge \forall h \geqslant i : y_h = 0\right\}$, Eqn. (B.1) is an equivalent way of stating the property to be shown. □

C CORRECTNESS OF KS JOIN

We present the proof of Thm. 3.17 in this appendix. *If* Y *and* Z *are both* $N+1$*-column KS matrices, and* $\gamma_{KS}(Y)$ *and* $\gamma_{KS}(Z)$ *are both non-empty sets, then* $Y \sqcup Z$ *is the projection of* $\begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix}$ *onto its right-most* $N+1$ *columns.*

Proof. $\gamma_{KS}(Y \sqcup Z)$ is the affine closure of $\gamma_{KS}(Y) \cup \gamma_{KS}(Z)$. Thus, we need to show that, for all $\vec{v} \in \mathbb{Z}_{2^w}^N$,

$$\exists \vec{u} \in \mathbb{Z}_{2^w}^N, \sigma \in \mathbb{Z}_{2^w} : \begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix} \begin{bmatrix} u \\ \sigma \\ \vec{v} \\ 1 \end{bmatrix} = 0$$

$$\Leftrightarrow \vec{v} \text{ is an affine combination of values in } \gamma_{KS}(Y) \text{ and } \gamma_{KS}(Z).$$

Recall that an affine combination is a linear combination whose coefficients sum to 1.

Proof of the "if" direction: Fix a particular $\vec{v} \in \mathbb{Z}_{2^w}^N$, and suppose that we have specific values for $\lambda \in \mathbb{Z}_{2^w}$, $\vec{y} \in \gamma_{KS}(Y)$, and $\vec{z} \in \gamma_{KS}(Z)$, such that $\vec{v} = \lambda\vec{y} + \vec{z}(1-\lambda)$. Pick $\sigma = 1-\lambda$, and $\vec{u} = (1-\lambda)\vec{z}$. Then,

$$\begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix} \begin{bmatrix} (1-\lambda)\vec{z} \\ 1-\lambda \\ \vec{v} \\ 1 \end{bmatrix} = 0$$

$$\Leftrightarrow -Y \begin{bmatrix} (1-\lambda)\vec{z} \\ 1-\lambda \end{bmatrix} + Y \begin{bmatrix} \vec{v} \\ 1 \end{bmatrix} = 0 \text{ and } Z \begin{bmatrix} (1-\lambda)\vec{z} \\ 1-\lambda \end{bmatrix} = 0$$

$$\Leftrightarrow Y \begin{bmatrix} -(1-\lambda)\vec{z} + \vec{x} \\ \lambda \end{bmatrix} = 0 \text{ and } (1-\lambda) Z \begin{bmatrix} \vec{z} \\ 1 \end{bmatrix} = 0$$

$$\Leftrightarrow \lambda Y \begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = 0 \text{ and } (1-\lambda) Z \begin{bmatrix} \vec{z} \\ 1 \end{bmatrix} = 0.$$

These last equations are true because $\overrightarrow{y} \in \gamma_{KS}(Y)$ and $\overrightarrow{z} \in \gamma_{KS}(Z)$. Thus, if $\overrightarrow{v}$ is in the affine closure of $\gamma_{KS}(Y) \cup \gamma_{KS}(Z)$, then $\begin{bmatrix} \overrightarrow{v} \\ 1 \end{bmatrix}$ is in the null space of the projected matrix.

Proof of the "only if" direction: Suppose that x is in the null space of the projected matrix. Pick $\overrightarrow{u} \in \mathbb{Z}_{2^w}^N$ and $\sigma \in \mathbb{Z}_{2^w}$ such that

$$\begin{bmatrix} -Y & Y \\ Z & 0 \end{bmatrix} \begin{bmatrix} u \\ \sigma \\ \overrightarrow{v} \\ 1 \end{bmatrix} = 0.$$

We must show that $\overrightarrow{v}$ is in the affine closure of $\gamma_{KS}(Y) \cup \gamma_{KS}(Z)$.

Immediately, we know that $Z \begin{bmatrix} \overrightarrow{u} \\ \sigma \end{bmatrix} = 0$ and $Y \begin{bmatrix} \overrightarrow{v} - \overrightarrow{u} \\ 1 - \sigma \end{bmatrix} = 0$. Because $\gamma_{KS}(Y)$ and $\gamma_{KS}(Z)$ are nonempty, we can select an arbitrary $\overrightarrow{y_0} \in \gamma_{KS}(Y)$ and $\overrightarrow{z_0} \in \gamma_{KS}(Z)$. Thus,

$$0 = Y \begin{bmatrix} \overrightarrow{v} - \overrightarrow{u} \\ 1 - \sigma \end{bmatrix} + \sigma Y \begin{bmatrix} \overrightarrow{y_0} \\ 1 \end{bmatrix} = Y \begin{bmatrix} \overrightarrow{v} - \overrightarrow{u} + \sigma \overrightarrow{y_0} \\ 1 \end{bmatrix}, \text{ and}$$

$$0 = Z \begin{bmatrix} \overrightarrow{u} \\ \sigma \end{bmatrix} + (1 - \sigma) Z \begin{bmatrix} \overrightarrow{z_0} \\ 1 \end{bmatrix} = Z \begin{bmatrix} \overrightarrow{u} + (1 - \sigma) \overrightarrow{z_0} \\ 1 \end{bmatrix}.$$

Now define $\overrightarrow{y}$ and $\overrightarrow{z}$ to be the values that we have just shown to be in $\gamma_{KS}(Y)$ and $\gamma_{KS}(Z)$:

$$\overrightarrow{y} \stackrel{\text{def}}{=} \overrightarrow{v} - \overrightarrow{u} + \sigma \overrightarrow{y_0} \text{ and } \overrightarrow{z} \stackrel{\text{def}}{=} \overrightarrow{u} + (1 - \sigma) \overrightarrow{z_0}.$$

If we solve for $\overrightarrow{v}$ and eliminate $\overrightarrow{u}$ in these equations, we get:

$$\overrightarrow{v} = \overrightarrow{y} - \sigma \overrightarrow{y_0} + \overrightarrow{z} + (\sigma - 1) \overrightarrow{z_0}.$$

Because $\overrightarrow{y}, \overrightarrow{y_0} \in \gamma_{KS}(Y)$, $\overrightarrow{z}, \overrightarrow{z_0} \in \gamma_{KS}(Z)$, and $1 - \sigma + 1 + (\sigma - 1) = 1$, we

have now stated $\vec{v}$ as an affine combination of values in $\gamma_{KS}(Y)$ and $\gamma_{KS}(Z)$, as required. □

D SOUNDNESS OF THE ABSTRACT-DOMAIN OPERATIONS FOR AFFINE-TRANSFORMERS-ABSTRACTION FRAMEWORK

In this section, we show that the abstract-domain operations for the ATA[$\mathcal{B}$] framework are sound with respect to the concrete semantics of the programming language.

Lemma D.1. *The bottom element represents the empty set.*

$$\gamma(\bot) = \varnothing \tag{D.1}$$

Proof.

$$\begin{aligned}
&\gamma(\bot) = \{(\vec{v}, \vec{v}') : \vec{v}' = \vec{v} \cdot C + \vec{d} \wedge (C : \vec{d}) \in \gamma(\bot_{\mathcal{B}})\} \\
\Rightarrow &\gamma(\bot) = \{(\vec{v}, \vec{v}') : \vec{v}' = \vec{v} \cdot C + \vec{d} \wedge (C : \vec{d}) \in \varnothing)\} \\
\Rightarrow &\gamma(\bot) = \varnothing
\end{aligned}$$

□

Lemma D.2. *The equality operation is sound.*

$$(a_1 \widetilde{==} a_2) \Rightarrow (\gamma(a_1) == \gamma(a_2)) \tag{D.2}$$

Proof. We will prove Lemma D.2 by contradiction. Assume that $a_1 \widetilde{==} a_2$ but $\gamma(a_1) \neq \gamma(a_2)$. Without loss of generality, we can assume that there exists $(\vec{v}, \vec{v}')$ such that:

$$\begin{aligned}
&(\vec{v}, \vec{v}') \in \gamma(base(a_1)) \wedge (\vec{v}, \vec{v}') \notin \gamma(base(a_2)) \\
\Rightarrow\ &\vec{v}' = \vec{v} \cdot C + \vec{d} \wedge (C : \vec{d}) \in \gamma(base(a_1)) \wedge (C : \vec{d}) \notin \gamma(base(a_2)) \\
\Rightarrow\ &\exists b.\ b \in \gamma(base(a_1)) \wedge b \notin \gamma(base(a_2)) \\
\Rightarrow\ &\gamma(base(a_1)) \neq \gamma(base(a_2)
\end{aligned}$$

$\Rightarrow base(a_1) \neq base(a_2)$ (by soundness of equality on $\mathcal{B}$)

$\Rightarrow a_1 \neq a_2$ (Contradiction!)

□

Lemma D.3. *The join operation is sound.*

$$\gamma(a_1 \tilde{\sqcup} a_2) \supseteq \gamma(a_1) \cup \gamma(a_2) \tag{D.3}$$

Proof. Assume that Lemma D.3 is incorrect. Then there exists $(\overrightarrow{v}, \overrightarrow{v}') \notin \gamma(a_1 \tilde{\sqcup} a_2)$ such that:

$$\begin{aligned}
&(\overrightarrow{v}, \overrightarrow{v}') \in \gamma(a_1) \cup \gamma(a_2) \\
\Rightarrow\ &\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d} \wedge (C : \overrightarrow{d}) \in \gamma(base(a_1)) \\
&\text{(Without loss of generality.)} \\
\Rightarrow\ &\overrightarrow{v}' = \overrightarrow{v} \cdot C + \overrightarrow{d} \wedge (C : \overrightarrow{d}) \in \gamma(base(a_1) \sqcup base(a_2)) \\
\Rightarrow\ &(\overrightarrow{v}, \overrightarrow{v}') \in \gamma(a_1 \tilde{\sqcup} a_2) \text{ (Contradiction!)}
\end{aligned}$$

□

The soundness of widening, statement abstractions, and identity function are easy to prove, and follow similar reasoning.

E SOUNDNESS OF ABSTRACT COMPOSITION FOR AFFINE-TRANSFORMERS-ABSTRACTION FRAMEWORK

In this section, we show that the abstract-composition operations defined in §4.3.2 are sound. From Eqn. (4.1), an abstract composition $a''_e = a' \circ a$ is exact iff:

$$\gamma(a''_e) = \{(\vec{v}, \vec{v}') \mid \exists (C : \vec{d}) \in \gamma(base(a)), (C' : \vec{d}') \in \gamma(base(a')), (C'' : \vec{d}'') : (\vec{v}' = \vec{v} \cdot C'' + \vec{d}'') \wedge (C'' = C \cdot C') \wedge (\vec{d}'' = \vec{d} \cdot C' + \vec{d}')\}$$

E.1 Non-Relational Base Domain

In this section, we show that the fast abstract composition for ATA[$\mathcal{B}$] (Eqn. (4.3)), when $\mathcal{B}$ is non-relational, is sound. Remember that any non-relational domain can be formulated as follows: $\mathcal{B} \stackrel{def}{=} symbols(C : \vec{d}) \to \mathcal{F}_{\mathcal{B}}$. The term $b[s]$, where $b \in \mathcal{B}$ and $s \in symbols(C : \vec{d})$, refers to the element in the foundation domain $f \in \mathcal{F}_{\mathcal{B}}$ corresponding to the symbol s.

Axiom 1. *Abstract addition is sound for $\mathcal{F}_{\mathcal{B}}$.*

$$e_1 \in \gamma(f_1) \wedge e_2 \in \gamma(f_2) \Rightarrow e_1 + e_2 \in \gamma(f_1 +^\sharp f_2) \tag{E.1}$$

Axiom 2. *Abstract multiplication is sound for $\mathcal{F}_{\mathcal{B}}$.*

$$e_1 \in \gamma(f_1) \wedge e_2 \in \gamma(f_2) \Rightarrow e_1 \times e_2 \in \gamma(f_1 \times^\sharp f_2) \tag{E.2}$$

Theorem E.1.

$$\gamma(a''_e) \subseteq \gamma(a' \circ_{NR} a). \tag{E.3}$$

Proof. We will prove Thm. E.1 by contradiction. Consider a model $m = (\overrightarrow{v}, \overrightarrow{v}'')$, such that $m \in \gamma(a''_e)$ and $m \notin \gamma(a' \circ_{NR} a)$. We will show that such a model cannot exists.

$$
\begin{aligned}
& m \in \gamma(a''_e) \\
\Leftrightarrow\ & \exists (C : \overrightarrow{d}) \in \gamma(base(a)), (C' : \overrightarrow{d}') \in \gamma(base(a')), (C'' : \overrightarrow{d}'') : \\
& \qquad (\overrightarrow{v}' = \overrightarrow{v} \cdot C'' + \overrightarrow{d}'') \wedge (C'' = C \cdot C') \wedge (\overrightarrow{d}'' = \overrightarrow{d} \cdot C' + \overrightarrow{d}') \\
\Leftrightarrow\ & \exists (C : \overrightarrow{d}), (C' : \overrightarrow{d}'), (C'' : \overrightarrow{d}'') : (\overrightarrow{v}' = \overrightarrow{v} \cdot C'' + \overrightarrow{d}'') \\
& \wedge \left(\bigwedge_{1 \leqslant i,j \leqslant k} (C''[i,j] = \sum_{1 \leqslant l \leqslant k} (C[i,l] \times C'[l,j])) \right) \\
& \wedge \left(\bigwedge_{1 \leqslant j \leqslant k} (\overrightarrow{d}''[j] = (\sum_{1 \leqslant l \leqslant k} (\overrightarrow{d}[l] \times C'[l,j]')) + \overrightarrow{d}'[j]) \right) \\
& \wedge \left(\bigwedge_{1 \leqslant i,j \leqslant k} C[i,j] \in \gamma(base(a)[c_{ij}]) \right) \wedge \left(\bigwedge_{1 \leqslant j \leqslant k} \overrightarrow{d}[j] \in \gamma(base(a)[d_j]) \right) \\
& \wedge \left(\bigwedge_{1 \leqslant i,j \leqslant k} C'[i,j] \in \gamma(base(a')[c_{ij}]) \right) \wedge \left(\bigwedge_{1 \leqslant j \leqslant k} \overrightarrow{d}'[j] \in \gamma(base(a')[d_j]) \right) \\
\Rightarrow\ & \exists (C'' : \overrightarrow{d}'') : (\overrightarrow{v}' = \overrightarrow{v} \cdot C'' + \overrightarrow{d}'') \\
& \wedge \left(\bigwedge_{1 \leqslant i,j \leqslant k} (C''[i,j] \in \Sigma^{\sharp}_{1 \leqslant l \leqslant k} (base(a)[c_{il}] \times^{\sharp} base(a')[c_{l,j}])) \right) \\
& \wedge \left(\bigwedge_{1 \leqslant j \leqslant k} (\overrightarrow{d}''[j] \in \Sigma^{\sharp}_{1 \leqslant l \leqslant k} (base(a)[d_l] \times^{\sharp} base(a')[c_{l,j}]) +^{\sharp} base(a')[d_j]) \right)
\end{aligned}
$$

(by application of axioms 1 and 2 to the expressions

$\sum_{1 \leqslant l \leqslant k} (C[i,l] \times C'[l,j])$ and $\sum_{1 \leqslant l \leqslant k} (\overrightarrow{d}[l] \times C'[l,j]') + \overrightarrow{d}'[j]$)

$$
\begin{aligned}
\Leftrightarrow\ & \exists (C'' : \overrightarrow{d}'') : (\overrightarrow{v}' = \overrightarrow{v} \cdot C'' + \overrightarrow{d}'') \wedge b \in (symbols(C'' : \overrightarrow{d}'') \to \mathcal{F}) \\
& \wedge \left(\bigwedge_{1 \leqslant i,j \leqslant k} \left(b[c''_{ij}] = \Sigma^{\sharp}_{1 \leqslant l \leqslant k} (base(a)[c_{il}] \times^{\sharp} base(a')[c_{lj}]) \right) \right)
\end{aligned}
$$

$$\wedge \left(\bigwedge_{1 \leqslant j \leqslant k} b[d''_j] = \sum^{\sharp}_{1 \leqslant l \leqslant k} (base(a)[d_l] \times^{\sharp} base(a')[c_{lj}]) +^{\sharp} base(a')[d_j] \right)$$

$\Leftrightarrow m \in \gamma(a' \circ_{NR} a)$ (by Eqn. (4.3))

□

E.2 Weakly-Convex Base Domain

In this section, we present a proof of soundness of abstract composition for weakly-convex base domains, denoted by $a' \circ_{WC} a$ (Eqn. (4.5)).

We present some useful axioms and lemmas before presenting the soundness theorem and its proof. Let $min_{\mathbb{Z}_{2^w}}$ and $max_{\mathbb{Z}_{2^w}}$ be the minimum and maximum bitvector values in $\mathbb{Z}_{2^w}$. Let $min_{\mathbb{Q}} = min_{\mathbb{Z}_{2^w}}$ and $max_{\mathbb{Q}} = max_{\mathbb{Z}_{2^w}}$.

Axiom 3. $cast_{\mathbb{Q}}$ *is distributive over bitvector addition in the absence of overflows: that is, if* $min_{\mathbb{Q}} \leqslant cast_{\mathbb{Q}}(bv_1 + bv_2) \leqslant max_{\mathbb{Q}}$, *where* $bv_1, bv_2 \in \mathbb{Z}_{2^w}$, *then*

$$cast_{\mathbb{Q}}(bv_1) + cast_{\mathbb{Q}}(bv_2) = cast_{\mathbb{Q}}(bv_1 + bv_2) \tag{E.4}$$

Axiom 4. $cast_{\mathbb{Q}}$ *is distributive over bitvector multiplication in the absence of overflows: that is, if* $min_{\mathbb{Q}} \leqslant bv_1 \cdot bv_2 \leqslant max_{\mathbb{Q}}$, *where* $bv_1, bv_2 \in \mathbb{Z}_{2^w}$, *then*

$$cast_{\mathbb{Q}}(bv_1) \cdot cast_{\mathbb{Q}}(bv_2) = cast_{\mathbb{Q}}(bv_1 \cdot bv_2) \tag{E.5}$$

Lemma E.2. $cast_{\mathbb{Q}}$ *is distributive over matrix multiplication for bitvectors, if there are no overflows in the matrix multiplication. That is, for* $n \times n$ *matrices* M *and* M' *where* $\forall_{1 \leqslant i,j \leqslant n} M[i,j], M'[i,j] \in \mathbb{Z}_{2^w}$,

$$cast_{\mathbb{Q}}(M) \times cast_{\mathbb{Q}}(M') = cast_{\mathbb{Q}}(M \times M'). \tag{E.6}$$

Proof. Let $M'' = cast_{\mathbb{Q}}(M) \times cast_{\mathbb{Q}}(M')$. Then,

$$\begin{aligned}
&\forall_{1\leqslant i,j\leqslant n} : M''[i,j] = \Sigma_{1\leqslant l\leqslant n} cast_{\mathbb{Q}}(M[i,l]) \cdot cast_{\mathbb{Q}}(M[l,j]) \\
\Rightarrow &\forall_{1\leqslant i,j\leqslant n} : M''[i,j] = \Sigma_{1\leqslant l\leqslant n} cast_{\mathbb{Q}}(M[i,l] \cdot M[l,j]) \text{ (by Axiom 4)} \\
\Rightarrow &\forall_{1\leqslant i,j\leqslant n} : M''[i,j] = cast_{\mathbb{Q}}(\Sigma_{1\leqslant l\leqslant n} M[i,l] \cdot M[l,j]) \text{ (by Axiom 3)} \\
\Rightarrow &cast_{\mathbb{Q}}(M) \times cast_{\mathbb{Q}}(M') = cast_{\mathbb{Q}}(M \times M')
\end{aligned}$$

□

Lemma E.3. *A convex combination of a set of rationals is inside bitvector boundaries if each of the rational values in the set is inside bitvector boundaries. Given any* $1 \leqslant \lambda_1, \lambda_2, \ldots, \lambda_l \leqslant 1$*, such that* $(\sum_{i=1}^{l} \lambda_i = 1)$*.*

$$min_{\mathbb{Q}} \leqslant q_1, q_2, \cdots, q_l \leqslant max_{\mathbb{Q}} \Rightarrow min_{\mathbb{Q}} \leqslant \sum_{i=1}^{l} \lambda_i q_i \leqslant max_{\mathbb{Q}} \tag{E.7}$$

Proof. Suppose $min_{\mathbb{Q}} > \sum_{i=1}^{l} \lambda_i q_i$.

$$\begin{aligned}
&min_{\mathbb{Q}} > \sum_{i=1}^{l} \lambda_i q_i \Rightarrow (min_{\mathbb{Q}} > \sum_{i=1}^{l} \lambda_i min_{\mathbb{Q}}) \text{ (because } min_{\mathbb{Q}} \leqslant q_1, q_2, \cdots, q_l) \\
\Leftrightarrow\ &min_{\mathbb{Q}} > \sum_{i=1}^{l} \lambda_i min_{\mathbb{Q}} \Rightarrow (min_{\mathbb{Q}} > min_{\mathbb{Q}}) \text{ (because } (\sum_{i=1}^{l} \lambda_i = 1).) \\
\Leftrightarrow\ &\textit{false}
\end{aligned}$$

Consequently, $min_{\mathbb{Q}} \leqslant \sum_{i=1}^{l} \lambda_i q_i$. The other inequality $\sum_{i=1}^{l} \lambda_i q_i \leqslant max_{\mathbb{Q}}$ can be proved in a similar fashion. □

Lemma E.4. *There are no overflows in a matrix multiplication of a convex combination of matrices, if there are no overflows in the matrix multiplications of the underlying matrices involved in the convex combination.*

$\forall_{1\leqslant i\leqslant l, 1\leqslant j\leqslant l'}$: $(t_i \times t'_j)$ *does not overflow* $\Rightarrow (t \times t')$ *does not overflow, where*

$$\left(\mathrm{cast}_{\mathbb{Q}}(t) = \sum_{i=1}^{l} \lambda_i \, \mathrm{cast}_{\mathbb{Q}}(t_i)\right) \wedge \bigwedge_{i=1}^{l} (1 \leqslant \lambda_i \leqslant 1) \wedge \sum_{i=1}^{l} \lambda_i = 1, \tag{E.8}$$

$$\textit{and,} \left(\mathrm{cast}_{\mathbb{Q}}(t') = \sum_{j=1}^{l'} \lambda'_j \, \mathrm{cast}_{\mathbb{Q}}(t'_j)\right) \wedge \bigwedge_{j=1}^{l'} (1 \leqslant \lambda'_j \leqslant 1) \wedge \sum_{j=1}^{l'} \lambda'_j = 1. \tag{E.9}$$

Proof. Because $(t_i \times t'_j)$ does not overflow, we know that each entry in the computation of the matrix multiplication does not overflow:

$$\forall_{1 \leqslant p,q \leqslant o} : \mathrm{min}_{\mathbb{Q}} \leqslant \Sigma_{n=1}^{o} \mathrm{cast}_{\mathbb{Q}}(t_i[p,n]) \cdot \mathrm{cast}_{\mathbb{Q}}(t'_j[n,q]) \leqslant \mathrm{max}_{\mathbb{Q}} \tag{E.10}$$

where t_i and t'_j are $o \times o$ matrices.
Suppose that $l'' = l \cdot l'$ and $\bigwedge_{m=1}^{l''} (1 \leqslant (\lambda''_m = \lambda_{\lceil m/l' \rceil} \cdot \lambda'_{(m-1)\%l'+1}) \leqslant 1)$. Then,
$\sum_{m=1}^{l''} \lambda''_m = \sum_{i=1}^{l'} \lambda_i \cdot \sum_{j=1}^{l'} \lambda'_j = 1 \cdot 1 = 1$. Then, by applying Lem. E.3 to Eqn. (E.10), we get for all $1 \leqslant p, q \leqslant o$:

$$\begin{aligned}
& \mathrm{min}_{\mathbb{Q}} \leqslant \Sigma_{m=1}^{l''} \lambda''_m (\Sigma_{n=1}^{o} \mathrm{cast}_{\mathbb{Q}}(t_{\lceil m/l \rceil}[p,n]) \cdot \mathrm{cast}_{\mathbb{Q}}(t'_{m\%l+1}[n,q])) \leqslant \mathrm{max}_{\mathbb{Q}} \\
\Leftrightarrow\ & \mathrm{min}_{\mathbb{Q}} \leqslant \Sigma_{i=1}^{l} \Sigma_{j=1}^{l'} \lambda_i \lambda'_j (\Sigma_{n=1}^{o} \mathrm{cast}_{\mathbb{Q}}(t_i[p,n]) \cdot \mathrm{cast}_{\mathbb{Q}}(t'_j[n,q])) \leqslant \mathrm{max}_{\mathbb{Q}} \\
\Leftrightarrow\ & \mathrm{min}_{\mathbb{Q}} \leqslant (\Sigma_{n=1}^{o} (\Sigma_{i=1}^{l} \lambda_i \mathrm{cast}_{\mathbb{Q}}(t_i[p,n])) \cdot (\Sigma_{j=1}^{l'} \lambda'_j \mathrm{cast}_{\mathbb{Q}}(t'_j[n,q]))) \leqslant \mathrm{max}_{\mathbb{Q}} \\
& \text{(by distributivity of multiplication over addition for rationals)} \\
\Leftrightarrow\ & \mathrm{min}_{\mathbb{Q}} \leqslant (\Sigma_{n=1}^{o} \mathrm{cast}_{\mathbb{Q}}(t[p,n]) \cdot \mathrm{cast}_{\mathbb{Q}}(t'_j[n,q])) \leqslant \mathrm{max}_{\mathbb{Q}} \\
& \text{(by the definition of } \mathrm{cast}_{\mathbb{Q}}(t) \text{ and } \mathrm{cast}_{\mathbb{Q}}(t') \text{ in Eqn. (E.8) and Eqn. (E.9))}
\end{aligned}$$

Hence $(t \times t')$ does not overflow. □

Theorem E.5.

$$\gamma(a''_e) \subseteq \gamma(a' \circ_{WC} a). \tag{E.11}$$

Proof. Consider any model $m = (\overrightarrow{v}, \overrightarrow{v}'')$, such that $m \in \gamma(a''_e)$. To prove Thm. E.1, we need to show that $m \in \gamma(a' \circ_{WC} a)$. Eqn. (4.5) defines $a'' = a' \circ_{WC} a$ as follows

$$\mathtt{base}(a'') = \begin{cases} \{t_i \times t'_j \mid 1 \leqslant i \leqslant l, 1 \leqslant j \leqslant l'\} & \textit{if there are no overflows in any} \\ & \textit{matrix multiplication } t_i \times t'_j \\ \top_{\mathcal{B}} & \textit{otherwise} \end{cases}$$
$$\text{where } \mathtt{base}(a) = \{t_1, t_2, ..., t_l\} \textit{ and } \mathtt{base}(a') = \{t'_1, t'_2, ..., t'_{l'}\}. \tag{E.12}$$

We know that for $m = (\overrightarrow{v}, \overrightarrow{v}')$,

$$\exists (C : \overrightarrow{d}) \in \gamma(\mathtt{base}(a)), (C' : \overrightarrow{d}') \in \gamma(\mathtt{base}(a')), (C'' : \overrightarrow{d}'') : \tag{E.13}$$
$$(\overrightarrow{v}' = \overrightarrow{v} \cdot C'' + \overrightarrow{d}'') \wedge (C'' = C \cdot C') \wedge (\overrightarrow{d}'' = \overrightarrow{d} \cdot C' + \overrightarrow{d}').$$

By the properties of weakly-convex domains (see §4.3.2), we know that

$$\mathtt{cast}_{\mathbb{Q}}\left(\left[\begin{array}{c|c} 1 & \overrightarrow{d} \\ \hline 0 & C \end{array}\right]\right) = \sum_{i=1}^{l} \lambda_i \, \mathtt{cast}_{\mathbb{Q}}(t_i), \text{ for some } \lambda_1, \lambda_2, \ldots, \lambda_l \in \mathbb{Q} \text{ such that} \tag{E.14}$$
$$\bigwedge_{i=1}^{l} (0 \leqslant \lambda_i \leqslant 1) \wedge (\sum_{i=1}^{l} \lambda_i = 1) \wedge \bigwedge_{i=1}^{l} \left(t_i = \left[\begin{array}{c|c} 1 & \overrightarrow{d}_i \\ \hline 0 & C_i \end{array}\right]\right), \textit{ and}$$
$$\mathtt{cast}_{\mathbb{Q}}\left(\left[\begin{array}{c|c} 1 & \overrightarrow{d}' \\ \hline 0 & C' \end{array}\right]\right) = \sum_{i=1}^{l'} \lambda'_i \, \mathtt{cast}_{\mathbb{Q}}(t'_i), \text{ for some } \lambda'_1, \lambda'_2, \ldots, \lambda'_l \in \mathbb{Q} \text{ such that} \tag{E.15}$$
$$\bigwedge_{i=1}^{l'} (0 \leqslant \lambda'_i \leqslant 1) \wedge (\sum_{i=1}^{l'} \lambda'_i = 1) \wedge \bigwedge_{i=1}^{l'} \left(t'_i = \left[\begin{array}{c|c} 1 & \overrightarrow{d}'_i \\ \hline 0 & C'_i \end{array}\right]\right).$$

To show that $m \in \gamma(a' \circ_{WC} a)$, we consider two cases.

Overflows in matrix multiplication. If there is an overflow encountered in any matrix multiplication $t_i \times t'_j$, then $base(a'') = \top_{\mathcal{B}}$ and consequently, $m \in \gamma(a' \circ_{WC} a)$ is true trivially.

No overflows in matrix multiplication. If there is no overflow encountered in any of the matrix multiplications $t_i \times t'_j$, then it suffices to prove that

$$(C'' : \overrightarrow{d}'') \in \bigsqcup_{i=1}^{l} \bigsqcup_{j=1}^{l'} \{t_i \times t'_j\}. \tag{E.16}$$

Eqn. (E.16) translates to proving that for some $\{\lambda''_1, \lambda''_2, ..., \lambda''_{l''}\}$:

$$(cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c} 1 & \overrightarrow{d}'' \\ \hline 0 & C'' \end{array}\right]\right) = \sum_{i=1}^{l''} \lambda''_i \, cast_{\mathbb{Q}}(t''_i)), \text{ for some } \lambda''_1, \lambda''_2, \ldots, \lambda''_{l''} \in \mathbb{Q} \text{ such that} \tag{E.17}$$

$$\bigwedge_{i=1}^{l''} (0 \leqslant \lambda''_i \leqslant 1) \wedge (\sum_{i=1}^{l''} \lambda''_i = 1) \wedge \bigwedge_{i=1}^{l''} \left(t''_i = \left[\begin{array}{c|c} 1 & \overrightarrow{d}''_i \\ \hline 0 & C''_i \end{array}\right]\right).$$

$$\begin{aligned}
& cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c} 1 & \overrightarrow{d}'' \\ \hline 0 & C'' \end{array}\right]\right) \\
&= cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c} 1 & \overrightarrow{d} \\ \hline 0 & C \end{array}\right] \times \left[\begin{array}{c|c} 1 & \overrightarrow{d}' \\ \hline 0 & C' \end{array}\right]\right) \text{ (by Eqn. (E.13))} \\
&= cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c} 1 & \overrightarrow{d} \\ \hline 0 & C \end{array}\right]\right) \times cast_{\mathbb{Q}}\left(\left[\begin{array}{c|c} 1 & \overrightarrow{d}' \\ \hline 0 & C' \end{array}\right]\right) \\
&\quad \text{(by Lem. E.4, because } \left[\begin{array}{c|c} 1 & \overrightarrow{d} \\ \hline 0 & C \end{array}\right] \times \left[\begin{array}{c|c} 1 & \overrightarrow{d}' \\ \hline 0 & C' \end{array}\right] \text{ does not overflow)} \\
&= \sum_{i=1}^{l} \lambda_i \, cast_{\mathbb{Q}}(t_i) \times \sum_{j=1}^{l'} \lambda'_j \, cast_{\mathbb{Q}}(t'_j) \\
&\quad \text{(by Eqn. (E.14) and Eqn. (E.15).)} \\
&= \sum_{i=1}^{l} \sum_{j=1}^{l'} \lambda_i \lambda'_j \, cast_{\mathbb{Q}}(t_i) \times \; cast_{\mathbb{Q}}(t'_j) \\
&\quad \text{(by distributivity of matrix multiplication over addition)}
\end{aligned}$$

$$= \sum_{i=1}^{l} \sum_{j=1}^{l'} \lambda_i \lambda'_j \, \mathsf{cast}_{\mathbb{Q}}(t_i \times t'_j)$$

(by Lem. E.2)

$$= \sum_{m=1}^{l''} \lambda''_m \, \mathsf{cast}_{\mathbb{Q}}(t''_m)$$

where $l'' = l \cdot l'$, $\bigwedge_{m=1}^{l''} (1 \leqslant (\lambda''_m = \lambda_{\lceil m/l' \rceil} \cdot \lambda'_{(m-1)\%l'+1}) \leqslant 1)$, and $\sum_{m=1}^{l''} \lambda''_m = \sum_{i=1}^{l'} \lambda_i \cdot \sum_{j=1}^{l'} \lambda'_j = 1 \cdot 1 = 1$. □

F SOUNDNESS OF THE MERGE OPERATION

In this section, we show that the merge operation defined in §4.3.3 is sound. Recall that the merge function is defined as:

$$\begin{aligned} \mathit{Merge}(a, a') &= a'', \text{where} \\ \mathit{base}(a'') &= (b_g \sqcap \mathit{havoc}(\mathit{base}(\mathrm{Id}), \mathit{gsyms})) \circ a \\ & \quad b_g = \mathit{havoc}(\mathit{base}(a'), \mathit{lsyms}), \\ & \quad \mathit{lsyms} = \mathit{symbols}(C_{gl}) \cup \mathit{symbols}(C_{ll}) \cup \\ & \qquad \mathit{symbols}(d_l) \cup \mathit{symbols}(C_{lg}) \\ & \quad \mathit{gsyms} = \mathit{symbols}(C_{gg}) \cup \mathit{symbols}(d_g) \end{aligned} \tag{F.1}$$

As mentioned in Eqn. (4.6), the exact merge-function semantics are specified as follows:

$$\textsc{Merge}(\gamma(a), \gamma(a')) = \textsc{RevertLocals}(\gamma(a')) \circ \gamma(a) \tag{F.2}$$

Theorem F.1.

$$\textsc{Merge}(\gamma(a), \gamma(a')) \subseteq \gamma(\mathit{Merge}(a, a')). \tag{F.3}$$

Proof. We will prove Thm. F.1 by contradiction. Consider a model $m = (\overrightarrow{g_m}, \overrightarrow{l_m}; \overrightarrow{g_m}', \overrightarrow{l_m}')$, such that $m \in \textsc{Merge}(\gamma(a), \gamma(a'))$ and $m \notin \gamma(\mathit{Merge}(a, a'))$. Let $a_{\mathit{RevLocs}} \in \mathrm{ATA}[\mathcal{B}]$ be an abstract domain value such that

$$\mathit{base}(a_{\mathit{RevLocs}}) = \mathit{havoc}(\mathit{base}(a'), \mathit{lsyms}) \sqcap \mathit{havoc}(\mathit{base}(\mathrm{Id}), \mathit{gsyms}) \tag{F.4}$$

By the soundness of abstract composition, existence of m implies existence of $n = (\overrightarrow{g_n}, \overrightarrow{l_n}; \overrightarrow{g_n}', \overrightarrow{l_n}')$, such that $n \in \textsc{RevertLocals}(\gamma(a'))$ and $n \notin \gamma(a_{\mathit{RevLocs}})$. We will show that n cannot exist. Consequently, m

cannot exist, and thus merge is sound.

$n \in \textsc{RevertLocals}(\gamma(a'))$

$\Leftrightarrow (\overrightarrow{g_n}, \overrightarrow{l_n}; \overrightarrow{g_n}', \overrightarrow{l_n}') \in \{(\overrightarrow{g}, \overrightarrow{q}, \overrightarrow{g}', \overrightarrow{q}) \mid (\overrightarrow{g}, \overrightarrow{l}, \overrightarrow{g}', \overrightarrow{l}') \in \gamma(a')\}$ (by Eqn. (4.7))

$\Leftrightarrow \exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(base(a')), \overrightarrow{l}, \overrightarrow{l}', \overrightarrow{q} :$

$(\overrightarrow{g_n}' = \overrightarrow{g_n} \cdot C_{gg} + \overrightarrow{l} \cdot C_{lg} + \overrightarrow{d_g}) \wedge (\overrightarrow{l}' = \overrightarrow{g_n} \cdot C_{gl} + \overrightarrow{l} \cdot C_{ll} + \overrightarrow{d_l}),$

$\wedge (\overrightarrow{l_n} = \overrightarrow{q}) \wedge (\overrightarrow{l_n}' = \overrightarrow{q}) \wedge (\overrightarrow{l} = \overrightarrow{0})$

($\overrightarrow{l}$ are initialized to 0 in each function)

$\Leftrightarrow \exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(base(a')) :$

$(\overrightarrow{g_n}' = \overrightarrow{g_n} \cdot C_{gg} + \overrightarrow{d_g}) \wedge (\overrightarrow{l_n} = \overrightarrow{l_n}')$

(by removing the existential variables $\overrightarrow{l}$, $\overrightarrow{l}'$ and $\overrightarrow{q}$)

$\Leftrightarrow \exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(base(a')) :$

$$havoc(\left[\begin{array}{c|cc} 1 & \overrightarrow{g_n} & \overrightarrow{l_n} \end{array}\right] \left[\begin{array}{c|cc} 1 & \overrightarrow{d_g} & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & 0 & C_{ll} \end{array}\right] = \left[\begin{array}{c|cc} 1 & \overrightarrow{g_n}' & \overrightarrow{l_n}' \end{array}\right], lsyms) \wedge (\overrightarrow{l_n}' = \overrightarrow{l_n})$$

(because $(\overrightarrow{g_n}' = \overrightarrow{g_n} \cdot C_{gg} + \overrightarrow{d_g})$ is the result of havoc on *lsyms* for

$(\overrightarrow{g_n}' = \overrightarrow{g_n} \cdot C_{gg} + \overrightarrow{d_g}) \wedge (\overrightarrow{l_n}' = \overrightarrow{g_n} \cdot C_{gl} + \overrightarrow{l_n} \cdot C_{ll} + \overrightarrow{d_l})$)

$\Leftrightarrow \exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(base(a')) :$

$$havoc(\left[\begin{array}{c|cc} 1 & \overrightarrow{g_n} & \overrightarrow{l_n} \end{array}\right] \left[\begin{array}{c|cc} 1 & \overrightarrow{d_g} & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{gl} & C_{ll} \end{array}\right] = \left[\begin{array}{c|cc} 1 & \overrightarrow{g_n}' & \overrightarrow{l_n}' \end{array}\right], lsyms) \wedge (\overrightarrow{l_n}' = \overrightarrow{l_n})$$

(because $\overrightarrow{l_n}$ are initialized to zero)

$$\Leftrightarrow \exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(base(a')) :$$

$$havoc(\left[\begin{array}{c|cc} 1 & \overrightarrow{g_n} & \overrightarrow{l_n} \end{array}\right] \left[\begin{array}{c|cc} 1 & \overrightarrow{d_g} & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{gl} & C_{ll} \end{array}\right] = \left[\begin{array}{c|cc} 1 & \overrightarrow{g_n}' & \overrightarrow{l_n}' \end{array}\right], lsyms) \wedge$$

$$havoc(\left[\begin{array}{c|cc} 1 & \overrightarrow{g_n} & \overrightarrow{l_n} \end{array}\right] \left[\begin{array}{c|cc} 1 & 0 & 0 \\ \hline 0 & I & 0 \\ 0 & 0 & I \end{array}\right] = \left[\begin{array}{c|cc} 1 & \overrightarrow{g_n}' & \overrightarrow{l_n}' \end{array}\right], gsyms)$$

(because havoc of *gsyms* on the identity transformation yields $\overrightarrow{l_n}' = \overrightarrow{l_n}$)

$$\Leftrightarrow \left(\exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(havoc(base(a'), lsyms)) : \right.$$

$$\left. \left[\begin{array}{c|cc} 1 & \overrightarrow{g_n} & \overrightarrow{l_n} \end{array}\right] \left[\begin{array}{c|cc} 1 & \overrightarrow{d_g} & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{gl} & C_{ll} \end{array}\right] = \left[\begin{array}{c|cc} 1 & \overrightarrow{g_n}' & \overrightarrow{l_n}' \end{array}\right] \right) \bigwedge$$

$$\left(\exists \left(\begin{bmatrix} C_{gg} & C_{gl} \\ C_{lg} & C_{ll} \end{bmatrix} : (\overrightarrow{d_g}, \overrightarrow{d_l}) \right) \in \gamma(havoc(base(Id), gsyms)) : \right.$$

$$\left. \left[\begin{array}{c|cc} 1 & \overrightarrow{g_n} & \overrightarrow{l_n} \end{array}\right] \left[\begin{array}{c|cc} 1 & \overrightarrow{d_g} & \overrightarrow{d_l} \\ \hline 0 & C_{gg} & C_{gl} \\ 0 & C_{gl} & C_{ll} \end{array}\right] = \left[\begin{array}{c|cc} 1 & \overrightarrow{g_n}' & \overrightarrow{l_n}' \end{array}\right] \right)$$

$\Leftrightarrow (\overrightarrow{g_n}, \overrightarrow{l_n}, \overrightarrow{g_n}', \overrightarrow{l_n}') \in \gamma(a_{RevLocs})$

$\Leftrightarrow$ Model $(\overrightarrow{g_n}, \overrightarrow{l_n}, \overrightarrow{g_n}', \overrightarrow{l_n}')$ does not exist.

$\Rightarrow$ Model $(\overrightarrow{g_m}, \overrightarrow{l_m}, \overrightarrow{g_m}', \overrightarrow{l_m}')$ does not exist. (Contradiction.)

(by soundness of abstract composition)

If the abstract composition operation is exact, then the implication in the last step of the proof becomes biconditional. Thus, if abstract composition is exact then the merge operation is exact. □

REFERENCES

[1] Bach, E. 1992. Linear algebra modulo ɳ. Unpublished manuscript.

[2] Bagnara, R., K. Dobson, P.M. Hill, M. Mundell, and E. Zaffanella. 2006. Grids: A domain for analyzing the distribution of numerical values. In *Int. Workshop on Logic Based Prog. Dev. and Transformation*.

[3] Bagnara, R., P. M. Hill, and E. Zaffanella. 2008. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming* 72(1–2):3–21.

[4] Bagnara, R., P.M. Hill, and E. Zaffanella. 2006. Widening operators for powerset domains. *Int. Journal on Software Tools for Technology Transfer* 8(4/5):449–466.

[5] Balakrishnan, G., and T. Reps. 2010. WYSINWYX: What You See Is Not What You eXecute. *Trans. on Prog. Lang. and Syst.*

[6] BBC. 2014. Gangnam style music video 'broke' youtube view limit. `http://www.bbc.com/news/world-asia-30288542`.

[7] Beyer, D. 2015. Software verification and verifiable witnesses - (report on sv-comp 2015). In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*.

[8] Bjørner, N., A. Phan, and L. Fleckenstein. 2015. ν*z*-an optimizing smt solver. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*.

[9] Bloch, Joshua. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. "googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html".

[10] Bouajjani, A., J. Esparza, and T. Touili. 2003. A generic approach to the static analysis of concurrent programs with procedures. In *Symposium on Principles of Programming Languages*.

[11] Brauer, J., and A. King. 2010. Automatic abstraction for intervals using Boolean formulae. In *Static Analysis Symposium*.

[12] ———. 2011. Transfer function synthesis without quantifier elimination. In *European Symp. on Programming*.

[13] ———. 2012. Transfer function synthesis without quantifier elimination. *Logical Methods in Comp. Sci.* 8(3).

[14] Burch, Jerry, and David Dill. 1994. Automatic verification of pipelined microprocessor control. In *Int. Conf. on Computer Aided Verification*.

[15] Bygde, S., B. Lisper, and N. Holsti. 2011. Fully bounded polyhedral analysis of integers with wrapping. In *Int. Workshop on Numerical and Symbolic Abstract Domains*.

[16] Chang, B.-Y.E., and K.R.M. Leino. 2005. Abstract interpretation with alien expressions and heap structures. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*.

[17] Chen, L., A. Miné, and P. Cousot. 2008. A sound floating-point polyhedra abstract domain. In *Asian Symp. on Prog. Lang. and Systems*.

[18] Chen, L., A. Miné, J. Wang, and P. Cousot. 2009. Interval polyhedra: An abstract domain to infer interval linear relationships. In *Static Analysis Symposium*.

[19] Cousot, P., and R. Cousot. 1976. Static determination of dynamic properties of programs. In *Int. Symp on Programming*.

[20] ———. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symposium on Principles of Programming Languages*, 238–252.

[21] ———. 1979. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages*, 269–282.

[22] Cousot, P., R. Cousot, and L. Mauborgne. 2011. The reduced product of abstract domains and the combination of decision procedures. In *Int. Conf. on Foundations of Software Science and Computation Structures*.

[23] Cousot, P., and N. Halbwachs. 1978. Automatic discovery of linear constraints among variables of a program. In *Symposium on Principles of Programming Languages*, 84–96.

[24] Dietz, W., P. Li, J. Regehr, and V. Adve. 2012. Understanding integer overflow in C/C++. In *Int. Conf. on Software Engineering*.

[25] dSPACE. dSPACE TargetLink. `www.dspace.com/en/pub/home/products/sw/pcgs/targetli.cfm`.

[26] Dutertre, B., and L. de Moura. 2006. Yices: An SMT solver. `http://yices.csl.sri.com`.

[27] Dutertre, Bruno, and Leonardo de Moura. 2006. A fast linear-arithmetic solver for dpll(t). In *Int. Conf. on Computer Aided Verification*.

[28] EDN Network. 2013. Toyota's killer firmware: Bad design and its consequences. `http://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences`.

[29] Elder, M., J. Lim, T. Sharma, T. Andersen, and T. Reps. 2011. Abstract domains of affine relations. In *Static Analysis Symposium*.

[30] ———. 2014. Abstract domains of affine relations. *Trans. on Prog. Lang. and Syst.*

[31] Engadget. 2015. To keep a boeing dreamliner flying, reboot once every 248 days. `https://www.engadget.com/2015/05/01/boeing-787-dreamliner-software-bug`.

[32] FDA. 2014. Fda statement on radiation overexposures in panama. `https://www.fda.gov/radiation-emittingproducts/radiationsafety/alertsandnotices/ucm116533.htm`.

[33] Ganesh, V., and D.L. Dill. 2007. A decision procesure for bit-vectors and arrays. In *Int. Conf. on Computer Aided Verification*.

[34] Gange, G., J. Navas, P. Schachte, H. Søndergaard, and P. Stuckey. 2013. Abstract interpretation over non-lattice abstract domains. In *Static Analysis Symposium*.

[35] Garner, H. L. 1978. Theory of computer addition and overflow. *IEEE Trans. on Computers* C-27(4).

[36] GCN. 1998. Software glitches leave navy smart ship dead in the water. `https://gcn.com/Articles/1998/07/13/Software-glitches-leave-Navy-Smart-Ship-dead-in-the-water.aspx`.

[37] Ghorbal, K., F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta. 2012. Donut domains: Efficient non-convex domains for abstract interpretation. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*.

[38] Gleick, J. 1996. A bug and a crash. `https://around.com/ariane.html`.

[39] Granger, P. 1989. Static analysis of arithmetic congruences. *Int. J. of Comp. Math.*

[40] ———. 1991. Analyses semantiques de congruence. Ph.D. thesis, Ecole Polytechnique.

[41] Gulwani, S., and G.C. Necula. 2003. Discovering affine equalities using random interpretation. In *Symposium on Principles of Programming Languages*.

[42] ———. 2005. Precise interprocedural analysis using random interpretation. In *Symposium on Principles of Programming Languages*.

[43] Hafner, J., and K. McCurley. 1991. Asymptotically fast triangularization of matrices over rings. *SIAM J. Comput.* 20(6).

[44] Howell, J.A. 1986. Spans in the module $(\mathbb{Z}_m)^s$. *Linear and Multilinear Algebra* 19.

[45] Jeannet, B. New Polka. `http://pop-art.inrialpes.fr/~bjeannet/newpolka/index.html`.

[46] Jeannet, B., D. Gopan, and T. Reps. 2005. A relational abstraction for functions. In *Static Analysis Symposium*.

[47] Jones, N.D., and A. Mycroft. 1986. Data flow analysis of applicative programs using minimal function graphs. In *Symposium on Principles of Programming Languages*, 296–306.

[48] Jourdan, Jacques-Henri, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 2015. A formally-verified c static analyzer. In *Symposium on Principles of Programming Languages*.

[49] Karr, M. 1976. Affine relationship among variables of a program. *Acta Inf.* 6:133–151.

[50] Kidd, N., A. Lal, and T. Reps. 2007. WALi: The Weighted Automaton Library. `www.cs.wisc.edu/wpis/wpds/download.php`.

[51] Kildall, G.A. 1973. A unified approach to global program optimization. In *Symposium on Principles of Programming Languages*, 194–206.

[52] King, A., and H. Søndergaard. 2008. Inferring congruence equations with SAT. In *Int. Conf. on Computer Aided Verification*.

[53] ———. 2010. Automatic abstraction for congruences. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*.

[54] Knoop, J., and B. Steffen. 1992. The interprocedural coincidence theorem. In *Int. Conf. on Compiler Construction*.

[55] Lal, A., and T. Reps. 2006. Improving pushdown system model checking. In *Int. Conf. on Computer Aided Verification*.

[56] Lal, A., T. Reps, and G. Balakrishnan. 2005. Extended weighted pushdown systems. In *Int. Conf. on Computer Aided Verification*.

[57] Lattner, C., and V.S. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Int. Symp. on Code Generation and Optimization*.

[58] Laviron, V., and F. Logozzo. 2009. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*.

[59] Li, Y., A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik. 2014. Symbolic optimization with smt solvers. In *Symposium on Principles of Programming Languages*.

[60] Lim, J. 2011. Transformer Specification Language: A system for generating analyzers and its applications. Ph.D. thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Tech. Rep. 1689.

[61] Lim, J., and T. Reps. 2008. A system for generating static analyzers for machine instructions. In *Int. Conf. on Compiler Construction*.

[62] ———. 2013. TSL: A system for generating abstract interpreters and its application to machine-code analysis. *Trans. on Prog. Lang. and Syst.* 35(1).

[63] llvm. LLVM: Low level virtual machine. `llvm.org`.

[64] Malmkjær, K. 1993. Abstract interpretation of partial-evaluation algorithms. Ph.D. thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., Manhattan, Kansas.

[65] Masdupuy, F. 1992. Array abstractions using semantic analysis of trapezoid congruences. In *Int. Conf. Supercomputing*, 226–235.

[66] ———. 1992. Array abstractions using semantic analysis of trapezoid congruences. In *Int. Conf. Supercomputing*.

[67] Meyer, C.D. 2000. *Matrix analysis and applied linear algebra*. Philadelphia, PA: SIAM.

[68] Miné, A. 2001. The octagon abstract domain. In *Working Conference on Reverse Engineering*, 310–322.

[69] ———. 2002. A few graph-based relational numerical abstract domains. In *Static Analysis Symposium*.

[70] ———. 2004. Relational abstract domains for the detection of floating-point run-time errors. In *European Symp. on Programming*.

[71] ———. 2006. The octagon abstract domain. *J. Higher-Order and Symbolic Computation* 19(1):31–100.

[72] ———. 2012. Abstract domains for bit-level machine integer and floating-point operations. In *Int. Joint Conf. on Automated Reasoning*.

[73] Monniaux, D. 2010. Automatic modular abstractions for template numerical constraints. *Logical Methods in Comp. Sci.*

[74] de Moura, L., and N. Bjørner. 2008. Z3: An efficient SMT solver. In *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*.

[75] Müller-Olm, M., and H. Seidl. 2004. Precise interprocedural analysis through linear algebra. In *Symposium on Principles of Programming Languages*.

[76] ———. 2005. Analysis of modular arithmetic. In *European Symp. on Programming*.

[77] ———. 2005. Personal communication.

[78] ———. 2007. Analysis of modular arithmetic. *Trans. on Prog. Lang. and Syst.* 29(5).

[79] Mycroft, A., and N.D. Jones. 1985. A relational framework for abstract interpretation. In *Programs as data objects*.

[80] Nelson, G., and D. Oppen. 1979. Simplification by cooperating decision procedures. *Trans. on Prog. Lang. and Syst.* 1(2).

[81] Nielson, F. 1989. Two-level semantics and abstract interpretation. *Theor. Comp. Sci.* 69:117–242.

[82] PPL. PPL: The Parma polyhedra library. Www.cs.unipr.it/ppl/.

[83] Reps, T., G. Balakrishnan, and J. Lim. 2006. Intermediate-representation recovery from low-level code. In *Part. eval. and semantics-based prog. manip.*

[84] Reps, T., M. Sagiv, and G. Yorsh. 2004. Symbolic implementation of the best transformer. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*.

[85] Reps, T., S. Schwoon, S. Jha, and D. Melski. 2005. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming* 58(1–2).

[86] Reps, T., and A. Thakur. 2016. Automating abstract interpretation. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*.

[87] Sagiv, M., T. Reps, and S. Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.* 167:131–170.

[88] Sankaranarayanan, S., F. Ivančić, I. Shlyakhter, and A. Gupta. 2006. Static analysis in disjunctive numerical domains. In *Static Analysis Symposium*.

[89] Sankaranarayanan, S., H.B. Sipma, and Z. Manna. 2005. Scalable analysis of linear systems using mathematical programming. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*.

[90] Schmidt, D.A. 1986. *Denotational semantics*. Boston, MA: Allyn and Bacon, Inc.

[91] Sen, R., and Y.N. Srikant. 2007. Executable analysis using abstract interpretation with circular linear progressions. In *Memocode*.

[92] Sharir, M., and A. Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Program flow analysis: Theory and applications*. Prentice-Hall.

[93] Sharma, T., and T. Reps. 2017. A new abstraction framework for affine transformers. In *Static Analysis Symposium*.

[94] ———. 2017. Sound bit-precise numerical domains. In *Int. Conf. on Verification, Model Checking, and Abstract Interpretation*.

[95] Sharma, T., A. Thakur, and T. Reps. 2013. An abstract domain for bit-vector inequalities. Tech. Rep. TR-1789, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI.

[96] Simon, A., and A. King. 2007. Taming the wrapping of integer arithmetic. In *Static Analysis Symposium*.

[97] Simon, A., A. King, and J.M. Howe. 2002. Two variables per linear inequality as an abstract domain. In *Int. Workshop on Logic Based Prog. Dev. and Transformation*, 71–89.

[98] Spolsky, Joel. 2007. Explaining the excel bug. `https://www.joelonsoftware.com/2007/09/26/explaining-the-excel-bug`.

[99] Storjohann, A. 2000. Algorithms for matrix canonical forms. Ph.D. thesis, ETH Zurich, Zurich, Switzerland. Diss. ETH No. 13922.

[100] Thakur, A., M. Elder, and T. Reps. 2012. Bilateral algorithms for symbolic abstraction. In *Static Analysis Symposium*.

[101] Thakur, A., and T. Reps. 2012. A method for symbolic computation of abstract operations. In *Int. Conf. on Computer Aided Verification*.

[102] Warren, H.S., Jr. 2003. *Hacker's delight*. Addison-Wesley.