Compositional, Monotone, and Non-linear Program Analysis

by

John Cyphert

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2023

Date of final oral examination: 12/15/2023

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

Thomas Reps, Professor, Computer Sciences
Loris D'Antoni, Associate Professor, Computer Sciences
Mihaela Ifrim, Associate Professor, Mathematics
Zachary Kincaid, Associate Professor, Computer Science, Princeton University
To my friends and family.
Acknowledgments

With a little help from my friends.

— The Beatles (1967)

During my graduate career I had the privilege of working with such a talented group of collaborators whose efforts directly contributed to the content of this dissertation. I would like to thank Jason Breck, Yotam Feldman, Zachary Kincaid, and Thomas Reps for if it were not for them, this would be the first and last page of my dissertation. I also had the pleasure of working with Qinheping Hu and Loris D’Antoni on a few additional projects that I was thankful to be a part of.

I wouldn’t have made it very far if it were not for the both formal and informal advisement I received. I would like to thank Alexandra Silva whose discussions during COVID helped give me perspective and keep me on the path. I would like to thank my unofficial co-advisor Zak. Zak’s mathematical precision did a lot of heavy lifting of forming the contents of this dissertation. Without him, this dissertation would not exist or would just be incorrect. I would like to thank my official advisor Tom. Tom’s experience in the field kept us grounded on how to present our work in a way that others in the community would want to read. Without him, this dissertation would not exist or would just be incoherent. I would further like to thank Tom for his genuine desire for his students to do well. Midway through my graduate career, we were working on a paper that unexpectedly required a significant change with only a few hours left to make. Tom gave his time and stayed late with us in the department to give his support for those difficult few hours. I don’t think everyone would have made that sacrifice, and I really appreciated it at the time. I would like to thank Loris and Aws Al barghouthi who made the PL group a welcoming and happy place while I was there. I would also like to thank my committee members Tom, Loris, Zak, and Mihaela Ifrim for their insightful comments and suggestions.

I believe that completing a Ph.D. is essentially impossible without the support and occasional commiseration of fellow graduate students. Before I agreed to join UW, I attended the department’s prospective graduate student welcome weekend. By pure happenstance I chatted a bit with another student who I seemed
to get along with well enough during that weekend. When I fully joined UW, I was relieved when that same person, Jordan Henkel, became one of my office mates. I quickly came to regret that feeling (only joking of course). If I had a different office mate who I didn’t play golf with or have so many nonsensical conversations with who knows how many additional papers I could have published. My closest collaborator was my fellow graduate student Jason Breck. Tom and Zak probably think higher of me than they should because 99% of my bad ideas were filtered out through Jason, and the remaining 1% were also corrected and improved through brainstorming. I would like to thank Calvin Smith and Samuel Drews. I had a lot of fun learning to play tennis with Sam. Calvin, Jason, and I had an unforgettable trip to Italy and Portugal that I will always be grateful for. Also, I have fond memories of the New York City trip I had with Calvin, Sam, Jason, and Qinheping. I would like to thank Yotam Feldman. Collaboration with Yotam came later in my graduate career, but I cherished the time working together as well as the additional conversations. Tushar Sharma was only around for my first year as a grad student, but lunches with him gave me a lot of help on what was to come. I would also like to thank Mike Vaughn and Zhicheng Cai for the conversations throughout the years.

Finally, I would like to acknowledge my non-local friends and family. Their continued support during graduate school allowed me to maintain ties to my roots. Making the rounds during my holiday visits back home is a highlight of my trips. I would like to thank my Shelby buds Josh Ensman and Jacob Robinson. Hanging out throughout high-school, a New Mexico backpacking trip, and check-ins throughout undergrad and grad school always feel like home. I would like to thank the Dorton-McMahon-Mominee-Parker-Price-Semenya family. Ask either of my parents and they will say my early mathematical curiosity did not come from them. Instead it’s almost a guarantee that it came from watching Dianne at an early age. For that reason, she’s one of the people most responsible for this document. I would also like to single out Connor for being my oldest friend and the closest thing I have to a sibling. Moving away and not being able to just hang out was one of the most difficult parts of grad school. Lastly, I would like to thank my parents. I have so far lived a very fortunate life because of their effort and sacrifices. Their influences are too many to list, so I will focus on just a couple that are relevant to my current academic success. Through my mom, I appreciated that there was a wider world out there, and through her encouragement I felt that I could be a part of that wider world. This caused me to feel that not only could I achieve my goals, but they were also worth working towards. From my dad, I inherited a natural curiosity about the world. His desire to learn how something works as well as his motivation to put forth a strong effort at whatever the job at hand is were great influences on me. I’ll never tire of his stories. I wouldn’t have gotten far in graduate school without their love and support. I’m very thankful for the life they gave me.

This work was supported in part by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE
award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from WARF; by ONR under grants N00014-17-1-2889 and N00014-19-1-2318; by a Google Ph.D. fellowship; and by the NSF under grant number 194253. Opinions, findings, conclusions, or recommendations expressed herein are those of the author and do not necessarily reflect the views of the sponsoring agencies.
Contents

Contents v

List of Tables viii

List of Figures x

Abstract xii

1 Introduction 1
  1.1 Overview of the Results 7
  1.2 Thesis Organization 15

2 Background 16
  2.1 Algebraic Analysis 16
  2.2 Gröbner Bases 22

3 Non-Linear Invariant Generation 45
  3.1 Introduction 45
  3.2 Overview and Problem Statement 47
  3.3 Additional Chapter Background 52
  3.4 The Wedge Abstract Domain 54
  3.5 Extracting Recurrences 61
  3.6 Operational Calculus Recurrence Solving 67
  3.7 Experiments 78
  3.8 Related Work 79

4 Control-Flow Refinement 86
  4.1 Introduction 87
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Overview</td>
<td>89</td>
</tr>
<tr>
<td>4.3</td>
<td>Additional Chapter Background</td>
<td>93</td>
</tr>
<tr>
<td>4.4</td>
<td>Refining Inner Loops</td>
<td>100</td>
</tr>
<tr>
<td>4.5</td>
<td>Putting Refinement to Work in a Program Analyzer</td>
<td>111</td>
</tr>
<tr>
<td>4.6</td>
<td>Experiments</td>
<td>114</td>
</tr>
<tr>
<td>4.7</td>
<td>Related Work</td>
<td>118</td>
</tr>
<tr>
<td>5</td>
<td>Simplifying Non-linear Formulas</td>
<td>120</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>120</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview</td>
<td>123</td>
</tr>
<tr>
<td>5.3</td>
<td>Additional Chapter Background</td>
<td>127</td>
</tr>
<tr>
<td>5.4</td>
<td>Reduction</td>
<td>129</td>
</tr>
<tr>
<td>5.5</td>
<td>Saturation</td>
<td>144</td>
</tr>
<tr>
<td>5.6</td>
<td>Effective Degree Order</td>
<td>151</td>
</tr>
<tr>
<td>5.7</td>
<td>Experiments</td>
<td>153</td>
</tr>
<tr>
<td>5.8</td>
<td>Detailed Statistics from Saturation Depth Experiment</td>
<td>157</td>
</tr>
<tr>
<td>5.9</td>
<td>Related Work</td>
<td>159</td>
</tr>
<tr>
<td>6</td>
<td>Solvable Algebraic Transition Systems</td>
<td>161</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>161</td>
</tr>
<tr>
<td>6.2</td>
<td>Overview</td>
<td>164</td>
</tr>
<tr>
<td>6.3</td>
<td>Additional Chapter Background</td>
<td>167</td>
</tr>
<tr>
<td>6.4</td>
<td>Solvable Reflections</td>
<td>172</td>
</tr>
<tr>
<td>6.5</td>
<td>Kleene Closure of Solvable Transition Ideals</td>
<td>179</td>
</tr>
<tr>
<td>6.6</td>
<td>Loop Summarization modulo LIRR</td>
<td>189</td>
</tr>
<tr>
<td>6.7</td>
<td>Experimental Evaluation</td>
<td>192</td>
</tr>
<tr>
<td>6.8</td>
<td>Related work</td>
<td>196</td>
</tr>
<tr>
<td>7</td>
<td>Other Contributions</td>
<td>200</td>
</tr>
<tr>
<td>7.1</td>
<td>Periodic-Rational Spectral Decomposition</td>
<td>200</td>
</tr>
<tr>
<td>7.2</td>
<td>Compositional Higher-Order Recurrence Analysis</td>
<td>204</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>208</td>
</tr>
<tr>
<td>8.1</td>
<td>Contributions</td>
<td>208</td>
</tr>
<tr>
<td>8.2</td>
<td>Limitations and Potential Future Directions</td>
<td>210</td>
</tr>
</tbody>
</table>
8.3 Concluding Remarks 213

Bibliography 214
List of Tables

3.1 Shows the results of the experiments to check non-linear assertions. Column 2 shows the total number of assertions in each benchmark suite. The two columns under each tool show the running time (in seconds) and the number of proved assertions. In each row the greatest number of assertions proved and the smallest running time are shown in boldface. 78

4.1 The results of the assertion-checking experiments. Column 2 shows the total number of programs in each benchmark suite. Columns 3-20 show analysis results under six different conditions: with abstract domain KCBR or ACI, each with or without using the refinement algorithm, and using the state-of-the-art model checkers Ultimate Automizer and SeaHorn. For each configuration, the left column indicates the total running time (in seconds), the middle column indicates the number of programs in which all assertions were proven by the analyzer, and the right column is a pair T/M where T is the number of timeouts and M is the number of out-of-memory exceptions. In each row, the smallest running time and the greatest number of assertions proved are shown in boldface. 116

5.1 Performance of AutoBound on the examples. #eq and #ineq’s are resp. equality and inequality assumptions initially given (not including instantiated axioms); #floors is the number of integer divisions (floor of division) terms in the assumptions. #c-eq and #c-ineq are resp. the number of equalities/inequalities in the generated cone’s ideal/polyhedron; #m is the number of distinct monomials in the inequalities. time is the overall execution time of AutoBound (all times in seconds). csat is the time to saturate the cone. reduce is the time to reduce w.r.t. the cone using local projection. reduce-lp is the time to reduce using linear programming instead of local projection. All experiments in this table were taken using a product saturation depth of 3. 155
5.2 The size of the cone and run-time breakdown of each examples as the saturation depth is increased until a timeout of 10 minutes. \#c-eq and \#c-ineq are resp. the number of equalities/inequalities in the generated cone’s ideal/polyhedron; \#m is the number of distinct monomials in the inequalities. time is the overall execution time of AUTOBOUND (all times in seconds). csat is the time to saturate the cone. reduce is the time to reduce w.r.t. the cone using local projection. – indicates a timeout.

5.3 Comparison of the bounds that AUTOBOUND generates for input term \( t \) (saturation depth=3) and the bounds curated by a human expert. \( \star \) indicates a result that is too large to include in the table. ed shows the (second component of the) effective-degree of the bound (the first component is 0).

6.1 Comparison of tools on the loops and NLA benchmarks. T represents the amount of time, in seconds, take by each tool not including timeouts nor out of memory exceptions. The number of timeouts is reported in parentheses. We also experienced out of memory exceptions with VeriAbs which are noted in parentheses. \#P represents the number of benchmarks proved correct. The best results in each category is bolded.

6.2 USP-Lin and USP-Quad on the multi-path ALIGATOR benchmarks.
**List of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Intersection of topics</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Some illustrative examples</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Four examples: (a) cubic-time loop nest; (b) binary search; (c) rotation in the $x, y$ plane; (d) absence of information flow.</td>
<td>48</td>
</tr>
<tr>
<td>3.2</td>
<td>Recurrences created for the nested loops in Fig. 3.1(a).</td>
<td>49</td>
</tr>
<tr>
<td>3.3</td>
<td>Inference rules for implied inequalities</td>
<td>59</td>
</tr>
<tr>
<td>3.4</td>
<td>A loop, along with its transition formula and transition wedge</td>
<td>63</td>
</tr>
<tr>
<td>3.5</td>
<td>Translation function $T_n : F \rightarrow R$. In Eqn. (3.10) $K \subseteq \mathbb{N}$ is some finite index set.</td>
<td>71</td>
</tr>
<tr>
<td>4.1</td>
<td>(a) A multi-path loop that shows the benefits of refinement on static-analysis results. (b) A multi-path loop for which refinement can produce less-precise static-analysis results.</td>
<td>90</td>
</tr>
<tr>
<td>4.2</td>
<td>(a) A refinement graph for the loop from Fig. 4.1(a), based on pair-wise feasibility. (b) The extended condensation DAG created by Alg. 8 for the graph from (a).</td>
<td>108</td>
</tr>
<tr>
<td>4.3</td>
<td>(a) An example refinement graph. (b) The extended condensation DAG without entry and exit vertices.</td>
<td>110</td>
</tr>
<tr>
<td>4.4</td>
<td>(a) A refinement graph for the loop from Fig. 4.1(b), based on pair-wise feasibility. (b) The extended condensation DAG created by Alg. 8 for the graph from (a).</td>
<td>110</td>
</tr>
<tr>
<td>4.5</td>
<td>Two examples of programs having assertions that the analyzer can prove only when using the refinement algorithm.</td>
<td>117</td>
</tr>
<tr>
<td>5.2</td>
<td>Local projections of a polyhedron.</td>
<td>133</td>
</tr>
<tr>
<td>5.3</td>
<td>Reduction by local project</td>
<td>135</td>
</tr>
<tr>
<td>5.4</td>
<td>Cone saturation overview</td>
<td>145</td>
</tr>
</tbody>
</table>
5.5 Total running time as the saturation depth is increased. Missing points indicate a time greater
than the timeout of 90 seconds. For each benchmark, the depth that suffices for both the upper
and lower bounds to be the ones given in Tab. 5.3 is marked by a larger, black marker of the same
type as the line plot for the benchmark. .......................................................... 156

6.1 A challenging example. .......................................................... 164
6.2 An abstraction of the outer-loop transition ideal .......................... 166
6.3 A solvable reflection of a transition ideal ..................................... 173

7.1 subsetSum takes an array A of n integers, and performs a brute-force search to determine whether
any non-empty subset of A’s elements sums to zero. subsetSumAux is an example of a non-linearly
recursive procedure. .......................................................... 204
Abstract

The presence of bugs in deployed software can lead to great economic and or human cost. One strategy for mitigating these losses is to prove the functional correctness of programs—or sometimes aspects of a program’s functional correctness—during development. With an appropriate analysis technique, one can guarantee that deployed software will satisfy important properties for all possible inputs.

This dissertation presents several lines of research that advance the state-of-the-art in the topic area of automatically characterizing program behavior to prove functional correctness. More specifically, this dissertation focuses on building program-analysis techniques and tools that exhibit some combination of (1) producing non-linear invariants, (2) reasoning compositionally by building up more complex program summaries from simpler ones, and (3) being predictable by satisfying a monotonicity property.

The first line of research presented in this dissertation gives a program-analysis technique that is compositional and produces non-linear invariants. The key feature of the method is how it analyzes loops. To analyze loops, loop bodies are abstracted with the newly introduced wedge abstract domain. Furthermore, wedges admit a recurrence-extraction procedure that results in a set of c-finite recurrence relations that are implied by the original loop body. In this dissertation, we solve these c-finite recurrences using a technique based on operational calculus. In combination, these advancements yield a program-analysis tool that is able to produce program summaries that include equalities and inequalities between expressions that include polynomial, exponential, and logarithmic terms. Experimental results show that our method is able to generate precise non-linear summaries that are able to prove more programs correct in comparison to other state-of-the-art tools.

The second line of research presented in this dissertation focuses on automatically improving the precision of a base compositional analysis by modifying the way the base analysis summarizes loops. Specifically, this line of research presents a method that automatically rewrites a program-analysis problem to a semantically sound alternative problem, with the goal of achieving a more-precise analysis result. In pursuit of this goal, we introduce the notion of a pre-Kleene algebra (PKA) as model for reasoning about analysis precision. A key property of PKAs is that they have a monotonicity property. Our method then refines a program-analysis
problem with respect to the laws of PKAs. Then, as long as the base analysis satisfies the axioms of PKAs, our method guarantees that the refined program-analysis problem will yield a result that is at least as good as (and often better than) the result obtained from the original analysis formulation. Although this result is technically a “no-degradation” result, an experimental evaluation showed that our method allows for an analysis to prove roughly 25% more programs correct at the expense of an approximately 50% increase in analysis time.

The third line of research presented in this dissertation introduces the optimal symbolic-bound synthesis (OSB) problem. In short, an instance of the OSB problem has as input a term $t$ and a formula $\phi$, and asks to find a symbolic term $t^*$ such that (i) $\phi$ implies that $t^*$ upper-bounds $t$, and (ii) $t^*$ exhibits some “term-desirability” properties. We present a heuristic method for finding a symbolic term $t^*$ when $t$ and $\phi$ may contain non-linear terms. Our method works by extracting an implied cone of polynomials from $\phi$ and then reducing $t$ by the cone of polynomials to obtain a sound upper bound $t^*$. At a high level, our method makes use of Gröbner-basis techniques for reducing with respect to equations, and a novel local-projection method for reducing with respect to inequalities. To show the utility of our method, we apply our techniques to the setting of bounding relevant terms, such as those representing the value of some financial asset, in Solidity smart contracts.

The fourth line of research presented in this dissertation gives another compositional program analysis that is able to produce non-linear invariants. The method follows a similar structure to the method from the first line of research; however, the subsequent method has the additional benefit of being monotone. We implemented our monotone technique in a tool named Abstractionator. Instead of wedges, Abstractionator uses solvable transition ideals as an intermediate abstraction of a loop. Abstractionator then uses a summarization technique inspired by prior complete methods based on solvable polynomial maps to summarize abstracted solvable transition ideals. Thus, by utilizing an abstraction procedure for solvable transition ideals, Abstractionator brings to bear prior complete polynomial-invariant-generation methods to the setting of programs with a more general syntactic structure. Experiments show that Abstractionator compares favorably with other program-analysis tools, especially in the case where non-linear invariants are required.
Chapter 1

Introduction

Over the past half a century, the research community has created and developed a variety of static program analysis techniques used to verify increasingly complex programs. At a high level, static analysis concerns techniques for obtaining information about the run-time behavior of a program without actually running the program on specific inputs. Instead, static-analysis techniques explore a program’s behavior for all possible inputs and all possible states that the program can reach. The trick to make this feasible is to explore the program symbolically and conservatively, with the goal of computing an abstract descriptor that includes all possible behaviors of the program (even if it additionally describes some impossible ones). The problem of generating these abstract descriptors is referred to as the invariant-generation problem. The invariants produced by an analyzer can be used to prove that a program does not exhibit erroneous behavior.

This dissertation focuses on techniques that generate numerical invariants. Such invariants can describe the possible values a numerical program variable can take on during execution. Thus, techniques for producing such invariants can be used to prove that programs are free from security vulnerabilities or meet certain safety properties. For example, numerical-invariant-generation techniques can be used to show that an array-index variable does not exceed the length of the array or that a variable in the control program of a safety-critical system, e.g. in the field of aerospace systems, does not overflow.

Numerical invariants are also useful in describing a program’s resource usage. Programs can be instrumented with additional time and/or space variables simulating the time and space usage of program statements. For example, an accurate description of such variables can be used to prove that a program runs in polynomial time. Determining a program’s resource usage can allow an analyst to establish that a program is free from (a) algorithmic-complexity vulnerabilities: such a vulnerability allows an attacker to cause a program to use an excessive amount of resources; and (b) side-channel leaks: a side-channel leak allows an attacker to obtain sensitive information based on the amount of resources a program uses during execution.
This dissertation presents work that focuses on program-analysis techniques that exhibit some combination of (i) non-linear, (ii) compositional, and (iii) monotone reasoning.

Non-linear Analysis

Many program-analysis methods are restricted to the problem of generating linear invariants of the form $a_1x_1 + \cdots + a_kx_k \leq b$, where $a_1, \ldots, a_k$ and $b$ are constants. For many applications, linear invariants provide a good balance between expressiveness and having nice computational properties. In particular, the problem of determining satisfiability of linear real (or rational) arithmetic (LRA) and linear integer arithmetic (LIA) formulas is decidable. The decidability of linear arithmetic has prompted the research community to develop practical linear arithmetic satisfiability modulo theory (SMT) solvers, e.g. Microsoft’s Z3 [42]. These powerful SMT solvers have been bootstrapped to yield countless tools for linear-invariant synthesis.

However, for other applications, linear invariants are either too imprecise (or equivalently, not expressive enough) to prove that a program is free from errors. For example, the resource usages of many popular algorithms are most accurately described by expressions involving polynomial, exponential, and logarithmic terms. Designing a program-analysis tool that is able to produce these more expressive terms presents the additional challenge of reasoning about non-linear arithmetic. Unlike the linear case, non-linear arithmetic reasoning is computationally unwieldy. In the case of non-linear rational arithmetic and non-linear integer arithmetic, determining satisfiability is undecidable. Determining satisfiability is decidable in the case of non-linear real arithmetic. The most notable decision procedure is the method of cylindrical algebraic decomposition (CAD) [32]. However, complete reasoning in the case of non-linear real arithmetic is often prohibitively expensive. CAD is doubly-exponential in the worst case, and many examples reach this upper-bound.

Due to the challenges of non-linear arithmetic reasoning, program-analysis tools that are able to produce non-linear invariants often employ heuristic and/or incomplete subroutines. This approach has a negative effect on the predictability of the overarching program analysis, which can be frustrating for users. For instance, Hawblitzel et al. [67] states “we found Z3’s theory of nonlinear arithmetic to be slow and unstable; small code changes often caused unpredictable verification failures.” The trade-off between (a) the expressiveness of invariants produced, and (b) the predictability of the method presents a critical design decision for the developer of a program-analysis tool. This dissertation presents methods that allow for more expressive non-linear invariants; however, the goal is to build techniques on strong mathematical foundations to be able to prove theorems about the overall capabilities of a program-analysis tool. For example, many of the techniques presented in this dissertation build off of the theory of Gröbner bases. Gröbner bases can be utilized to yield complete reasoning in a fragment of non-linear arithmetic [88].
Compositional Analysis

Rice’s theorem [114] establishes that determining whether a program satisfies any non-trivial semantic property is undecidable. Consequently, the capabilities of any program analysis is fundamentally limited in some way. For example, some static-analysis methods, such as many methods based on abstraction refinement [30], are able to provide both positive and negative answers to program-analysis problems at the cost of being non-terminating in general. Another popular framework for program analysis is abstract interpretation [34]. In abstract interpretation, one does not directly analyze the original program. Instead, one first computes an abstraction of the program. An abstraction is a simplification of the program that retains some details but abstracts away others. A possible abstraction of numerical program variables could be the sign domain, which abstracts away the concrete values a variable could have, and only keeps track of whether the variable is negative, zero, or positive. For example, using the sign domain a concrete state \( \langle x \mapsto 3, y \mapsto 0, z \mapsto -100 \rangle \) is abstracted by the abstract state \( \langle x \mapsto \text{pos}, y \mapsto \text{zero}, z \mapsto \text{neg} \rangle \). This abstraction is sound because the original concrete state is within the concretization of the abstraction state. With a slight abuse of notation, the concretization of \( \langle x \mapsto \text{pos}, y \mapsto \text{zero}, z \mapsto \text{neg} \rangle \) is \( \langle x \mapsto \{1, 2, \ldots\}, y \mapsto 0, z \mapsto \{-1, -2, \ldots\} \rangle \). The advantage of abstract interpretation is that by analyzing appropriate abstractions of the original program we can compute a descriptor that over-approximates all possible behaviors of the original program, whereas without abstraction, determining all possible behaviors is undecidable. The disadvantage is that abstraction introduces imprecision, and consequently we can no longer guarantee definitive answers to program-analysis questions. For example, using the sign domain, and given the abstraction \( \langle x \mapsto \text{pos}, y \mapsto \text{zero}, z \mapsto \text{neg} \rangle \) of the concrete state \( \langle x \mapsto 3, y \mapsto \{0\}, z \mapsto -100 \rangle \), we could ask the following questions:

- Is \( x \geq -2 \)? From the abstract descriptor we know that \( x \) is strictly positive. Therefore, we can definitively say that \( x \) is greater than or equal to \(-2\).

- Is \( x \leq -2 \)? Similarly, from the abstract descriptor we know that \( x > 0 \) holds, and thus we can say that \( x \leq -2 \) definitely does not hold.

- Is \( x \geq 2 \)? In this example we cannot determine the truth value of the statement \( x \geq 2 \). From the description that \( x \mapsto \text{pos} \), it may be the case that \( x \mapsto 5 \) in a concrete execution and thus \( x \geq 2 \) holds. However, it may also be the case that \( x \mapsto 1 \) in a concrete execution and thus \( x \geq 2 \) does not hold. Therefore, the best answer we could provide in this situation is that we don’t know the validity of the statement \( x \geq 2 \).

This basic example illustrates that, in general, using abstractions mean that we can no longer give only yes or no answers to program-analysis questions. Instead, we must be content with the three possible answers of
yes, no, and maybe. However, at least for the current example, we would have been able to provide more a definitive answer to the previous query of $x \geq 2$ if we used a more precise abstraction. For example, the interval domain is strictly more precise than the sign domain. The interval domain can exactly represent the concrete state $\langle x \mapsto 3, y \mapsto 0, z \mapsto -100 \rangle$ with the abstract state $\langle x \mapsto [3, 3], y \mapsto [0, 0], z \mapsto [-100, -100] \rangle$. For this abstract state, we could definitively determine that $x \geq 2$ holds. While the interval domain is more precise than the sign domain, the increased precision comes at the cost of increased complexity. That is, it is more computationally expensive to use the interval domain; moreover, the interval domain does not satisfy the ascending-chain condition (ACC). Although this dissertation does not focus on the sign domain, interval domain, or the ascending-chain condition, these simple examples serve to illustrate an inherent trade-off when abstract interpretation is applied: we want to use as precise a domain as possible, but precision needs to be balanced with computational complexity.

Traditional data-flow analysis assigns an abstract state $a$ to each program location $l$ such that $a$ over-approximates all the possible concrete states that can occur at $l$. A benefit of the abstraction-interpretation framework is that, given an appropriate abstract domain, there exists a common algorithm to compute an abstract state for each program location $l$. At a high-level, the abstract-interpretation framework computes over-approximations via an iterative fixed-point finding process. To update the abstract value of a particular program location $l$, the abstract values of predecessor (potentially also successor) program locations are propagated forwards (and/or backwards) to $l$. These abstract values are then joined with the previous abstract state assigned to $l$. With each update, the set of concrete states approximated by $l$ potentially increases, until updates no longer change the abstract states assigned to each of the program locations. The theory of abstract interpretation ensures that when this happens the abstract value assigned to each program location, say $l$, over-approximates all possible concrete states that can occur at $l$. The technique of abstract interpretation is similar to other static-analysis methods in that the computation of abstract descriptors makes use of the context of a program statement, i.e. the abstract values of predecessor and successor states, to accurately update the current abstract state.

Abstract interpretation is a powerful framework that provides a uniform method for computing abstract descriptors at arbitrary program locations. By making use of the context of a program statement, an analyzer may be able to reduce imprecision by recognizing how behavior in one program location may affect the behavior at a different far away location. However, relying on context has the disadvantage that incremental changes in one part of a program may affect the analysis results in a completely different part of the program. In other words, given a small change to a program, an analyzer that makes use of context must re-analyze the whole code base to obtain an accurate answer.

The techniques described in this dissertation are rooted in the foundation of abstract interpreta-
tion. Namely, the program-analysis techniques described in this dissertation employ abstraction to over-approximate the concrete behaviors of a given program. However, the methods described in this dissertation often differ from traditional abstract interpretation on how the over-approximation is calculated. Predominantly, the techniques in this dissertation do not make use of context, and over-approximations are not computed via an iterative fixed-point finding process. Whereas traditional data-flow analysis techniques propagate information forward or backward through a program, the techniques in this dissertation propagate information in a bottom-up manner. In other words, the techniques of this dissertation are compositional. More specifically, the techniques of this dissertation predominantly compute abstract descriptors by computing relational summaries of individual program structures. Summaries for larger program structures, such as loops and branches, are computed from the summaries of program sub-structures. This approach contrasts with traditional data-flow analysis because the summary that a bottom-up technique creates for a loop, for example, does not depend on what statements surround the loop. Instead the summary for the loop only depends on the loop condition and body.

As mentioned earlier, another distinguishing characteristic between the techniques of this dissertation and traditional abstract interpretation is that the techniques of this dissertation predominantly do not compute abstract descriptors via an iterative fixed-point finding process. Instead, because the techniques of this dissertation compute summaries bottom-up and in relation to particular program structures, the techniques of this dissertation have separate procedures for summarizing loops. This contrasts with traditional data-flow analysis, which is mostly agnostic to the program structure. Because the techniques of this dissertation treat loops differently, we are free to design methods that directly compute a closure of a loop, given some abstract descriptor of the loop body.

The compositional techniques of this dissertation have the advantage that they do not make use of context during summarization. This means that a change in one part of a program only affects the summary for program structures containing that change. In other words, in contrast to traditional abstract interpretation, when using compositional techniques a small change in one part of a program does not affect the summary of far off program parts. With the size and distribution of large code-bases, today’s world requires tools that are able to accurately summarize program parts independently of their calling context. Analysis processes can then be modularized, and summaries can be checked in with corresponding new code to verify properties of the whole system. Furthermore, these analyses are amenable to parallelization.

**Monotone Analysis**

A desirable property of many static-analysis methods that attempt to verify whether a program satisfies a given semantic property is soundness. Sound verification tools are generally constructed so that they never
state that a program does satisfy a specification when in reality the program does not. As described previously
verification tools cannot in general provide definitive answers to all questions. A sound tool is conservative,
and only gives definitive answers when it can be proved that a program does in fact satisfy the specification.
On the other hand, when a sound tool is “unsure”, it might be the case that the program does in fact satisfy
the specification, but this fact cannot be proven (e.g., given the abstract domain used in the tool). In such a
case, a sound tool would be conservative and state only that the program might not satisfy the specification.

Soundness is often a requirement for a verification tool. However, creating a sound tool is trivial. A tool
that responds to any verification question with “maybe” is trivially sound. That is, such a hypothetical tool
never gives wrong answers. Yet, such a tool would be useless in practice, because it would never tell us
something we didn’t already know. Instead, we judge the quality of a sound tool by how often it is able to
provide non-trivial answers. On one side of the spectrum of providing definitive answers is a technique that
always provides definitive answers. That is, it never responds with “maybe”. Such a tool would be complete.
Unfortunately, Rice’s theorem kicks in and as a consequence no general verification tool can be both sound
and complete. However, much prior work does contain completeness results, but they necessarily must only
apply to limited situations, i.e. programs can only be of a certain form or have a limited semantics.

Because a general tool cannot be both sound and complete, the general method for determining the quality
of a tool is through the use of empirical studies. Empirical studies are important because they demonstrate the
quality of a tool on practical examples. As a consequence, however, many techniques make use of complicated
heuristics that work in practice, but are hard to characterize and understand. In contrast, a focus of this
dissertation is to work towards creating methods that can be characterized by some higher-level property.
More specifically, this dissertation discusses methods—and the use of methods—that are monotone.

The technical definition of a monotone program analysis depends on the style of analysis used. However,
at a high-level, monotonicity means that more accurate summaries for sub-parts of a program will yield
a more accurate overall summary, i.e., “more information in yields more information out”. For example,
consider the situation where an analyst wants to verify two program properties, $p_1$ and $p_2$, using an automatic
program-analysis tool. Suppose that the tool is able to verify $p_1$ but not $p_2$. The analyst notices that there is a
valid assumption about the value of a variable that the analyzer is not taking advantage of. For example,
perhaps the value of an incoming integer is always positive. The analyst decides that they will “help out”
the analyzer by adding in this assumption in the hopes that the analyzer will then be able to prove $p_2$. If
the analyzer is monotone, then this additional assumption means that the resulting overall summary will
be no worse than the original one, and therefore, it is guaranteed that $p_1$ will still be proven true. On the
other hand, if the analyzer is not monotone, it may be the case that the additional assumption would actually
cause the analyzer to provide a summary that is not able to prove $p_1$. With a monotone program analysis, the
analyst could continue to add assumptions with the goal of proving $p_2$. Because the analysis is monotone, it is guaranteed that adding assumptions “narrows down” the behaviors considered; and by this means, one hopes to be able to prove $p_2$.

Monotonicity not only concerns situations in which assumptions are added, but more generally refers to the predictability of program analyzers. Once again, the issue of an ever-changing code-base makes monotonicity a desirable property. We do not want small, far-off code changes to have wild effects on overall summaries. Unfortunately, many analyses are not monotone. The main difficulty with many traditional abstract domains is the presence of infinite ascending chains. Such situations usually require a widening operator [37] to ensure convergence. Widening operators cause many program analyzers to be non-monotone. In contrast, many of the techniques in this dissertation do not over-approximate iterative behavior using an iterative fixed-point finding procedure. Because the techniques of this dissertation predominantly over-approximate iterative behavior by computing some closure, careful examination (or crafting) of the method can yield a monotonicity result.

Another benefit of monotone program analyses is that this high-level property allows one to reason formally about the analysis behavior. With monotonicity in hand, other methods that require monotonicity can be built on top of an underlying analyzer to yield a better overall method. For example, the main subject of Chapter 4 is the bootstrapping of monotone program analyses to yield an automatic method that reformulates program-analysis problems to a modified form that is guaranteed to produce a more precise answer.

1.1 Overview of the Results

During my time as a graduate student, I had the pleasure of working as part of a team of collaborators that included at various times, Jason Breck, Yotam Feldman, Zachary Kincaid, and Thomas Reps. This collaboration resulted in multiple successful research projects that primarily make up the results of this dissertation. The primary goal of this collaboration was to improve the capabilities of (interprocedural) compositional recurrence analysis ((I)CRA) [49]. See § 2.1 for a background on CRA. My particular focus during this collaboration was to improve the precision as well as the robustness of CRA.

Towards the goal of improving precision we took steps to develop methods to improve the expressiveness of the invariants that CRA can produce. That is, a common theme of this dissertation is the production of formulas involving non-linear terms, such as polynomial, exponential, and logarithmic terms, that summarize programs. Such a setting gives rise to the significant challenge of non-linear arithmetic reasoning. As stated earlier, full reasoning about non-linear arithmetic is either undecidable, in the case of integer or rational arithmetic, or extremely expensive, in the case of real arithmetic. Because of these theoretical challenges, we
built strictly weaker arithmetic-reasoning techniques into our program-analysis tools.

As my graduate school career progressed, we began to examine the topic of analysis robustness. That is, even though our methods were incomplete, we wanted to show that our analyses satisfied some higher-level property. The main property we explored was monotonicity. The incompleteness of non-linear arithmetic does not match well with the goal of developing monotone program analyses. That is, incomplete methods are often best-effort and/or heuristic, which stands in contrast to robust predictable techniques. Therefore, to develop non-linear monotone analyses, we focused on increasing our understanding of what exactly our non-linear reasoning techniques, or fragments of our non-linear reasoning techniques, were doing. At least for me personally, this resulted in a lot of reading into Gröbner bases and algebraic number fields. Most of these mathematical ideas are hidden in the final dissertation; however, these results were necessary in developing the compositional, monotone, and non-linear analysis presented in Chapter 6.

In this dissertation, our analyses are applied to two application domains. The first concerns checking the functional correctness of programs. Namely, our techniques are designed to check whether an assertion in a program holds for any possible execution of that program. Generally, our techniques prove an assertion holds by generating a summary for the program as a formula. We generate such a formula irrespective of the
assertion to prove. If the formula we generate implies the assertion to be proven, then we report that the assertion is verified. Otherwise, our summary was not strong enough to show the desired property. Checking assertions has the benefit of being able to show that a program does not exhibit specified bad behavior, such as exceeding the size of an allocated array. The second application domain to which we have applied our techniques is the automatic generation of bounds on the usage of some program resource, such as memory or time. The same process of generating a summary formula for a program can be used to bound resource usage. That is, the program can be augmented with additional variables that track memory or time usage. In such a case, the summary that our techniques generate will potentially bound the resource variables. These bounds then characterize the running time or memory usage of the program. The techniques presented in this dissertation are not just theoretical: they have all been implemented into fully automatic open-source tools.

The remaining sub-sections of this chapter give an outline of how the motivations of compositional, monotone, and non-linear arithmetic evolved into various research projects through my graduate career. These sub-sections serve to give an introduction to the subsequent chapters, and Fig. 1.1 shows how the subsequent chapters relate to the main ideas of this dissertation.

1.1.1 Non-Linear Invariant Generation

Chapter 3, which is based on Kincaid et al. [87], begins the journey and presents a compositional analysis that is able to generate non-linear invariants automatically. Specifically, the method is able to generate program summaries as transition formulas. The transition formulas that we generate contain equality and inequality relations over non-linear arithmetic, including polynomial, exponential, and logarithmic terms.

Our method builds on the prior work of CRA [49]. CRA summarizes loops by employing a recurrence-extraction method that approximates a loop body with a set of implied recurrence relations. Closed-form solutions for these implied recurrence relations are computed, and consequently the closed-form recurrence solutions yield an approximation for the entire loop. However, because of the limitations of the recurrences used by the original CRA work, CRA was unable to produce invariants involving exponential and logarithmic terms. Furthermore, CRA’s ability to produce polynomial invariants was limited.

Akin to CRA, the method we present in Chapter 3 follows the same basic structure of extracting and solving recurrences to summarize programs. However, the method presented in Chapter 3 extends the power of CRA by being able to extract and solve a larger class of recurrences compared with what was presented in the original CRA paper [49]. Specifically, in Chapter 3 our method targets extraction and solving of c-finite
recurrences. In short, c-finite recurrences have the form

\[ a(n) = c_1 a(n-1) + \cdots + c_d a(n-d) + c_{d+1}. \]

The Fibonacci recurrence is a classical example of a c-finite recurrence. By upgrading the class of recurrences we are willing to handle, we increase the expressiveness of the invariants that we are able to produce. However, while the method of Chapter 3 has the same structure as CRA, the wider class of recurrences introduces additional challenges, and requires upgrading the individual pieces of the CRA procedure.

For example, to make use of the more powerful class of recurrence relations, the algorithm for extracting implied recurrence relations from a loop body must be upgraded. In Chapter 3, we addressed this challenge with the introduction of the wedge domain. The wedge domain admits a symbolic-abstraction procedure, which extracts an implied wedge from a loop body. Furthermore, the wedge domain utilizes Gröbner-basis techniques to rewrite wedge equalities, and a set of inference rules to rewrite inequalities to extract implied c-finite recurrences. Similarly, to make use of the more powerful class of recurrence relations, we also introduced a more powerful recurrence-solving procedure. Our recurrence-solving procedure, introduced in Kincaid et al. [87] as the operational-calculus recurrence solver (OCRS), is based on the operational calculus of Lothar Berg [15].

The introduction of more powerful recurrences also introduced a challenge related to the interplay of recurrence extraction and recurrence solving. Namely, because our method is compositional, the recurrence-extraction procedure needs to be able to reason about the closed-form solutions produced by the recurrence solver. In general, a closed-form solution of a c-finite recurrence can be formulated as an expression that contains polynomial and exponential terms. Such expressions are mostly not an issue for the wedge domain to handle. Wedges use uninterpreted functions and inference rules to reason about exponentials and Gröbner bases, and inference rules to reason about polynomials. However, a subtlety is that in general, closed-form solutions to c-finite recurrences have constants ranging over the algebraic numbers, whereas the underlying reasoning engine of the wedge domain uses strictly rational numbers. We addressed this challenge by introducing the concept of implicitly interpreted functions (IIFs). IIFs can be used to express recurrence solutions that would require non-rational arithmetic. Thus, via this notion, our recurrence solver solves c-finite recurrences using exponentials and polynomials over purely rational arithmetic when it is able to, and when a solution would require non-rational numbers, the solver produces a closed-form solution containing an IIF. IIFs are then reasoned about as an uninterpreted function outside the recurrence solver, i.e. by the wedge domain.

By addressing these challenges, we created (and implemented) a method for summarizing general
programs using non-linear arithmetic. We applied our method to functional-correctness and resource-bound-analysis benchmarks, and achieved promising results.

1.1.2 Control-Flow Refinement

One of the most challenging program features for any program analysis is a multi-path loop—that is, a loop with a branch. In Chapter 4, which is based on Cyphert et al. [39], we present a method that aims to improve the capabilities of compositional analysis on this kind of program structure.

To illustrate the issue, consider some loop with a branch. Label one of the paths of the loop body $A$ and the other path $B$. All the possible paths of the loop can be modeled by the regular expression $(A + B)^*$. That is, for any possible concrete execution of the loop there is a corresponding word in the language represented by $(A + B)^*$. To analyze such a loop using, for example, the techniques described in Chapter 3, we would have to extract a set of implied recurrence relations from the loop body $A + B$. In this case, for a recurrence relation to be implied by the loop body, it must be implied by both the paths $A$ and $B$. Often there are few such recurrence relations that are common to both paths, and this results in a low-precision loop summary.

We observe that one reason that multi-path loops result in an unnecessary loss of precision is that the initial analysis problem considers behaviors that are not actually feasible in the concrete program. For example, recall the case of analyzing a loop $(A + B)^*$. Suppose, that in any concrete execution of the loop it is impossible for the path $A$ to follow the path $B$. Stated in language terms, suppose $BA = 0$. Syntactically, the original program-analysis problem was to analyze a loop of the form $(A + B)^*$, which is often difficult to analyze. However, with the additional knowledge that $BA = 0$, we can say that $A^*B^*$ also represents all the concrete executions of the loop. Moreover, the analysis problem of the form $A^*B^*$ is often easier than an analysis problem of the form $(A + B)^*$. We call this process of removing infeasible paths from a representative regular expression refinement.

In Chapter 3, we make the observation that refining a path-expression often leads to increased analysis precision. However, we also found that refinement does not necessarily lead to increased precision. Moreover, we found cases for which arbitrary refinement can actually lead to a worse result (see § 4.2 for an example). A key challenge addressed in Chapter 4 is to find a characterization for when a particular refinement will not degrade precision. We solved this challenge by introducing the concept of a pre-Kleene algebra (PKA). In essence, the theory of PKAs can be used to compare two path-expressions, and so long as the analysis being used respects the laws of PKA's, we can guarantee a no-degradation result. In Chapter 4, we further develop an automatic-refinement procedure that rewrites a loop with respect to the laws of PKA's. Thus, our method can be used to automatically improve the precision of any analysis that respects the laws of PKA's.
A key feature of PKA’s is monotonicity. That is, for an analysis to satisfy the laws of PKA’s, the analysis must be monotone. We would not be able to guarantee a no-degradation result without monotonicity, because, for example, even if refinement was guaranteed to improve the precision of an inner loop, we would be unable to give a no-degradation guarantee for any outer loop, if the analysis is not monotone.

Empirically, our implementation and experiments showed that using our refinement method allowed for analyses to prove roughly 25% more assertions in our benchmark programs at the expense of an approximately 50% increase in analysis time.

1.1.3 Simplifying Non-linear Formulas

Chapter 5 presents previously unpublished work that was done in collaboration with Yotam Feldman, Zachary Kincaid, and Thomas Reps. In Chapter 5, we investigate the problem of deriving consequences of non-linear formulas with the goal of the consequences being simpler for a human analyst to understand. Many of the techniques of this dissertation seek to summarize programs with non-linear formulas. These summaries contain lots of information about the dynamics of a program; however, much of this information may not be relevant for a human analyst when they are trying to get a sense of the behavior of the program. For example, one goal of an analyst would be to see if some resource, such as time, space, or the value of some financial asset does not exceed some value. A program summary may constrain such a value in a non-obvious way, and so a question would be whether we could derive simpler bounds on the resource that are more human interpretable.

Towards this end, in Chapter 5 we introduce the optimal symbolic-bound synthesis (OSB) problem:

**Given** a (potentially non-linear) formula $\phi$ representing assumptions and axioms, and an objective term $t$, **find** a term $t^*$ such that

1. (Bound) $\phi \models t \leq t^*$

2. (Optimality) For every term $s$ that satisfies the first condition, $t^* \preceq s$ holds, where $\preceq$ represents some notion of “term desirability.”

The OSB problem differs from many prior optimization techniques, because rather than finding some constant to bound a given term $t$ we want to find some term $t^*$ that may contain multiplication, division, or some other arithmetic function.

In Chapter 5, we examine the OSB problem in the context of non-linear arithmetic terms and formulas. To address the challenge of non-linear arithmetic reasoning we introduce a mathematical object which we call a cone of polynomials. A cone of polynomials has a finite representation but is able to hold on to an infinite
set of non-linear inequalities. Internally, a cone of polynomials consists of a Gröbner bases, which captures equations, and a polyhedral cone, which captures inequalities. Cones of polynomials strike a balance between expressiveness and computational feasibility—using non-linear reasoning on equalities through the ideal, and linear reasoning on inequalities through the linear cone, gives efficient yet powerful non-linear reasoning on inequalities.

We use cones of polynomials to address the OSB problem by first creating an implied cone of polynomials $C$ from the given formula $\phi$, and then reducing the term of interest $t$ by $C$ to obtain the bounding term $t^\ast$. Our reduction algorithm similarly consists of a two-stage process: (1) we reduce $t$ by the Gröbner basis; and (2) we perform a novel local-projection method that further reduces $t$ by the polyhedral cone.

We implemented our method and applied it to the setting of smart-contract verification. An example of a problem on which we tried the method was to determine the growth of certain assets via multiple calls to a real Solidity smart contract. We found that our tool is able to produce—in seconds or minutes—bounds that match or nearly-match human-produced bounds, as well as bounds where ones were previously unknown to human experts.

### 1.1.4 Monotone Non-linear Invariant Generation

In Chapter 6, we unify the previous concepts and present a compositional program analysis that produces non-linear invariants and is monotone. At the time of this writing the material of Chapter 6 is unpublished; however, the work is to appear as Cyphert and Kincaid [40].

The method in Chapter 6 has a similar structure to the method presented in Chapter 3 in that loops are analyzed by abstracting a set of recurrence relations from a loop body, which are then solved to yield a summary for the entire loop. A deficiency of the technique in Chapter 3 is that it is not monotone. We found that both the recurrence-extraction and recurrence-solving methods of Chapter 3 were not robust enough to yield a monotone analysis. Part of the issue concerns the recurrence model that we used in Chapter 3. In Chapter 3, the recurrence model that we used was that of c-finite recurrences. However, in Chapter 6 we introduce a more powerful recurrence model, which we call solvable transition ideals.

The inspiration for the use of solvable transition ideals comes from another line of prior work [94, 74, 116, 6], which is able to generate all polynomial invariants for a restricted class of loops. In short, these methods work by viewing a restricted loop body as a set of recurrence relations, i.e. a solvable polynomial map. The recurrence relations are then symbolically solved. However, rather than returning the recurrence solutions as a summary for the loop, which would run into the same problem of reasoning about algebraic numbers that was present in Chapter 3, the algorithm of Kauers and Zimmermann [82] is used to generate all polynomial relations.

\[1\text{Solvable polynomial maps are semantically equivalent to c-finite recurrences; however, they are syntactically different.}\]
between the recurrence solutions. These polynomial relations represent all polynomial invariants of the loop. To exploit this prior work, the challenge is how to be able to apply the complete polynomial-invariant generation method, which applies to a restricted class of loops, to general programs. That is, we want to create a method that can be used to summarize programs, not just single loops whose bodies satisfy a restricted syntax.

Due to our use of the more powerful intermediate model of solvable transition ideals, both a new recurrence-extraction technique and a new recurrence-solving technique are required. Our new extraction procedure is similar to the one used in Chapter 3, in that it uses Gröbner bases to rewrite polynomial equations to find implied c-finite recurrences. However, the extraction procedure in Chapter 6 extends the one from Chapter 3—not only in that it extracts the more general notion of a solvable transition ideal, but also because the extraction procedure produces the best solvable transition ideal. This property of producing a best abstraction is one of the key pieces that makes the overall process monotone, and represents a significant advance over the method of Chapter 3.

The new recurrence-solving procedure of Chapter 6 also advances the state-of-the-art. It is based on the algorithm of Kauers and Zimmermann [82]. However, that algorithm computes polynomial relations between c-finite recurrences, and, as stated, our model of solvable transition ideals is more general than c-finite recurrences. Consequently, we introduced a method that generalized the algorithm of Kauers and Zimmermann [82], while still maintaining the key property that our algorithm is a complete method for finding the set of polynomials implied by the arbitrary iteration of a solvable transition ideal.

In Chapter 6, we show that the combination of (i) an extraction procedure that computes best abstractions, with (ii) a complete solving procedure, yields an overall monotone program analysis. Moreover, we also showed that our analysis satisfies the additional axioms of a pre-Kleene algebra. Therefore, the techniques of Chapter 4 can be directly applied.

We implemented the resulting procedure and compared it with state-of-the-art analysis tools on a set of challenging benchmarks, with an additional focus on benchmarks requiring non-linear invariants. We also compared the techniques with the method of Chapter 3. Even though many of the algorithms of Chapter 6 generalize the procedures of Chapter 3, ultimately, the techniques are incomparable. This situation is mainly due to the fact that Chapter 3 is able to produce logarithmic and exponential invariants, whereas Chapter 6 only produces polynomial invariants. Furthermore, the techniques of Chapters 3 and 6 handle inequality invariants differently. Our experimental results show that not only is the method of Chapter 6 theoretically interesting, but the method also has practical relevance.
1.1.5 Further Contributions

Chapter 7 highlights two other related papers on which I was a co-author. However, I was less involved compared with the projects described in the other chapters. In § 7.1, Kincaid et al. [86] is summarized. That paper presents a method for non-linear, compositional analysis. This method differs from the techniques of Chapter 3 in how recurrences are solved. In Chapter 3 we did not precisely solve recurrences when the closed-form solution would require a non-rational number. In Kincaid et al. [86], we observed how some recurrences that at first appear to require non-rational numbers to solve can actually be solved using purely rational closed-forms. Kincaid et al. [86] describes how this alternative method can be utilized in a program analyzer.

In § 7.2, the results of Breck et al. [24] are summarized. Breck et al. [24] presents a method to summarize recursive programs using the recurrence-solving techniques presented in Chapter 3. The main idea is to examine a recursive procedure and extract a “recurrence template.” This template has a numerical parameter that relates to recursion height. If a set of these templates imply a recurrence equation, the recurrence equation can be solved to yield a summary for the procedure. An experimental highlight of this technique is being able to show that a recursive implementation of Strassen’s algorithm has running time $O(n^{\log_2(7)})$.

1.2 Thesis Organization

Chapter 2 gives background on algebraic program analysis, which is the compositional-analysis framework used throughout this dissertation. Chapter 2 also discusses the theory and techniques of Gröbner bases. Gröbner bases are an important tool used in this dissertation to reason about non-linear arithmetic. Chapter 3 discusses a (non-monotone) compositional-program-analysis technique that is able to generate non-linear numerical invariants. Chapter 4 provides a technique for improving the precision of monotone algebraic analyses. Chapter 5 discusses a technique that addresses a new kind on non-linear-reasoning problem, which was applied to the task of verifying smart contracts. Chapter 6 enhances the technique of Chapter 3 by developing a monotone compositional-program-analysis technique that is able to produce non-linear invariants. Chapter 7 summarizes an additional non-linear program-analysis technique, as well as a method for generating non-linear invariants for recursive procedures. Chapter 8 gives concluding remarks.
Chapter 2

Background

This chapter presents overall background on topics and ideas that will be used in multiple subsequent chapters. Subsequent chapters will also have their own background section where additional material will be introduced that is relevant for that particular chapter. These backgrounds are kept separate in order to maintain the flow and expository nature of this overall background chapter on algebraic program analysis and Gröbner bases.

2.1 Algebraic Analysis

In this subsection I give a gentle introduction to the algebraic approach to program analysis. Chapters 3, 4 and 6 build upon the foundation of algebraic program analysis. For a more detailed presentation as well as multiple examples of algebraic analyses see Kincaid et al. [89].

Many of the results of this thesis are framed in the context of algebraic program analysis. Algebraic program analysis, or sometimes called “analysis in the style of Tarjan” [131], consists of a two-step process. First, a regular expression is created whose language contains the set of feasible paths of a program. This task is done by solving the path-expression problem: a program’s control-flow graph (CFG) is considered to be a finite-state machine in which CFG nodes are states, and each edge is labeled by an alphabet symbol unique to that edge. Then a path-expression algorithm, such as the one presented in Tarjan [130], creates for each node \( n \) a regular expression \( R_n \) whose language, \( L(R_n) \), is the set of all paths from the CFG’s start node to \( n \). For instance, for programs with structured control-flow, program statements correspond to alphabet symbols from some alphabet \( \Sigma \); sequencing of statements corresponds to concatenation, \( (\cdot) \); conditionals correspond to the choice operator, \( (+) \); and loops correspond to a Kleene star, \( (*) \). In the second phase of analysis, each expression \( R_n \) is evaluated using an interpretation in which the regular-expression operators +, ·, and *—now treated as syntactic operators—are interpreted as some suitable (sound) operations, \( \oplus, \otimes, \),
and ⊗, respectively, in the analysis domain.

More formally, let Σ be an alphabet of program statements. We assume that an algebraic analysis consists of a structure \( \mathcal{A} = (\mathbb{A}, \oplus, \otimes, \odot, 0, 1) \), and a semantic function \( \mathcal{A}[\cdot] : \Sigma \rightarrow \mathbb{A} \). \( \mathcal{A}[\cdot] \) can be extended to regular expressions via:

\[
\begin{align*}
\mathcal{A}[R_1 + R_2] &= \mathcal{A}[R_1] \oplus \mathcal{A}[R_2] \\
\mathcal{A}[R_1 \cdot R_2] &= \mathcal{A}[R_1] \otimes \mathcal{A}[R_2] \\
\mathcal{A}[R^*] &= (\mathcal{A}[R])^\odot \\
\mathcal{A}[0] &= 0 \\
\mathcal{A}[1] &= 1
\end{align*}
\] (2.1)

With sound operations for \( \oplus, \otimes, \) and \( \odot \), \( \mathcal{A}[R_n] \) gives an overapproximating summary of the state transformation computed along the paths from procedure entry to CFG node \( n \).

The existence of the explicit \( \odot \) operator is a main differentiator between algebraic and traditional data-flow analysis. The need to supply an extra operation imposes an additional burden on the domain designer; on the other hand, the domain designer is then free to implement their own method for approximating iterative behavior, which could be some direct computation or some standard iterative fixed-point-finding process. By having an explicit iteration operator, the operators of an algebraic analysis have a direct correspondence with the three standard regular-expression operators. Therefore, a static-analysis task can be performed by reinterpreting regular expressions as domain transformers, and evaluating a set of path expressions for a program in a bottom-up manner.

Many standard program-analysis problems can be formulated as an algebraic analysis. In particular, any summary-based interprocedural analysis [124] can be cast as an algebraic analysis, including predicate abstraction [12]; affine-relation analysis [103, 104, 90, 45]; problems in the IFDS [109], IDE [118], and Weighted Pushdown System [111] frameworks; and polyhedral analysis [37, 78]. In the remainder of this section, I present a particular algebraic method for summarizing programs that will serve as a prototypical example for the approach of this dissertation.

I will illustrate algebraic program analysis using the program in Fig. 2.1a as an example. However, for a contrasting viewpoint, I will first consider analyzing the program using traditional abstract interpretation. A traditional analysis based on intervals would analyze this loop in a forward direction. First, the analysis would discover that \( x \) is within the range \([0, 0]\) at the head of the loop. Then the analyzer would propagate this information forward into the loop, and establish that \( x \) can be in the range \([1, 1]\). This new interval would be combined with the old interval \([0, 0]\) allowing the analyzer to see that \( x \) can be in the range \([0, 1]\) at the head of the loop. Then, this updated value would be propagated into the loop to establish that \( x \) can be in the range \([1, 2]\) inside the loop. This process would continue until a fixed-point is reached. Ultimately, the analyzer would determine that at the head of the loop \( x \) can take values in the range \([0, 100]\), and due to the
loop conditional the analyzer would show that the final value of $x$ would be 100. However, consider if, for whatever reason, the initial value of $x$ was changed to 5 rather than 0. The final value of $x$ would still be 100. Nevertheless, the whole analysis would have to be re-run (requiring 95 more iterations) to incorporate this new information.

Now consider analyzing this program using the alternative algebraic framework. First, we construct a regular expression whose language contains the set of feasible program paths from entry to exit. For this example, $a \cdot (b \cdot c)^* \cdot \bar{b}$ is a suitable path-expression, where $a$ denotes the statement $x=0$; $b$ and $\bar{b}$, respectively, denote the loop guard $x<100$ and its negation; and $c$ denotes the statement $x=x+1$. To analyze this program, we reinterpret the instructions of the original program, as well as the regular-expression operators, in a suitable over-approximating domain. For this particular example, consider using the domain of compositional recurrence analysis, $\text{CRA} = \langle \Phi(x, x'), \oplus, \otimes, \odot, \text{False}, \land, \lor \rangle$ [49].

This domain consists of two-vocabulary transition formulas, $\phi(x, x')$ over the program variables $x$, plus a set of primed copies $x'$. Such a formula represents a relation between pre-states (over $x$) and post-states (over $x'$). The $\otimes$ operation is relational composition, and $\oplus$ is disjunction,

$$
\phi \otimes \psi = \exists x''. \phi(x, x'') \land \psi(x'', x') \quad \phi \oplus \psi = \phi \lor \psi.
$$

(2.2)

The $\otimes$ operation consists of a two-step process. To over-approximate $\phi(x, x')^{\otimes}$, $\phi$ is abstracted to a set of over-approximating recurrence relations, representing an over-approximation of one iteration of the formula $\phi$. Then, these recurrence relations are solved to obtain a closed-form approximation for $\phi(x, x')^{\otimes}$. Returning to the example from Fig. 4.1 (a), we summarize the statements $a \overset{\text{def}}{=} x=0$ and $c \overset{\text{def}}{=} x=x+1$ with the formulas $\phi_a \overset{\text{def}}{=} x' = 0$ and $\phi_c \overset{\text{def}}{=} x' = x + 1$, respectively. Similarly, the loop guard $b$ is summarized with the formula $\phi_b \overset{\text{def}}{=} x < 100$. To analyze the program, we evaluate $\phi_a \otimes (\phi_b \otimes \phi_c)^{\otimes} \otimes \phi_b$ in a bottom-up manner.

$$
\phi_b \otimes \phi_c \overset{\text{def}}{=} x < 100 \land x' = x + 1.
$$

(2.3)
To summarize \((\phi_b \otimes \phi_c)\circ\), the recurrence relation \(x^{[k]} = x^{[k-1]} + 1\) is extracted from the formula on the right-hand side of Eqn. (2.3), where \(x^{[k]}\) denotes the value of \(x\) on the \(k\)th iteration of the loop. This recurrence relation is solved to obtain a closed-form expression indicating the value of \(x\) for an arbitrary iteration \(k\).

\[
(\phi_b \otimes \phi_c)\circ \triangleq \exists k \geq 0. x < 100 \land x' = x + k \land x' \leq 100. \tag{2.4}
\]

This loop-summary is then combined with the initial value for \(x\), which is captured by \(\phi_a\), as well as the loop-terminating condition, \(\phi_b\), to obtain a final program summary of \(x' = 100\). However, note that because algebraic program analyses summarize programs in a bottom-up manner, rather than forward or backwards, if the initial value of \(x\) is changed to 5, the loop does not have to be re-analyzed. As long as the structure of the loop did not change, the old summary is still valid.

The previous example illustrates the basic mechanics of an algebraic analysis, and gave an instance where CRA is able to characterize exactly the dynamics of a code snippet. To illustrate how imprecision can be introduced, as well as to demonstrate the type of recurrence relations extracted by CRA, consider the program in Fig. 2.1b. We have the following summary for the loop body:

\[
\phi_{\text{body}} = i < N \land i' = i + 1 \land (y' = y + i + 1 \land x' = x + 1 \lor z' = z + i + 1).
\]

Unlike the previous example, \(\phi_{\text{body}}\) contains a disjunction, and thus cannot be directly translated into a recurrence relation. In this case, CRA will determine a set of over-approximating recurrence relations \(\{r_i\}\) with \(\phi_{\text{body}} \models r_i\) for each \(i\).

First, CRA determines if there are any simple recurrence relations satisfied by the loop body. A simple recurrence is one of the form \(w^{[k]} = w^{[k-1]} + d\), where \(d\) is a constant. In this particular example, the variable \(i\) gets increased by 1 in each path through the loop. Thus program variable \(i\) is satisfied by the simple recurrence

\[
i^{[k]} = i^{[k-1]} + 1.
\]

Solving the recurrence yields

\[
i^{[k]} = i^{[0]} + k = 0 + k = k. \tag{2.5}
\]

Now consider the variables \(y\) and \(z\). In one path through the loop \(y\) gets increased and through the other path \(z\) gets increased. Because we have no information about the condition on which path is taken, there is no way to determine the exact dynamics of variable \(y\) and \(z\) with recurrence relations. However, observe that
no matter which path is taken the linear term $y + z$ gets increased by $i + 1$. In other words,

$$\phi_{\text{body}} \models y' + z' = y + z + i + 1.$$  

This equation entails

$$(y + z)^{[k]} = (y + z)^{[k-1]} + i^{[k-1]} + 1 \quad (2.6)$$

which is nearly a simple recurrence equation, but the additive term in this recurrence, $i^{[k-1]} + 1$, is not a constant. However, because $i$ satisfies a simple recurrence equation, we know the value of $i^{[k-1]}$ in terms of $k$ and $i^{[0]}$. Thus we can rewrite Eqn. (2.6) using Eqn. (2.5) on the right-hand side and obtain

$$(y + z)^{[k]} = (y + z)^{[k-1]} + k + i^{[0]}.$$  

Any recurrence of the form $t^{[k]} = t^{[k-1]} + p(k)$, where $p(k)$ is a degree $n$ polynomial over $k$ has a direct solution as a polynomial of degree $n + 1$. For the case of $y + z$, $(y + z)^{[k]} = (y + z)^{[0]} + \frac{k(k+1)}{2} + k i^{[0]}$. A recurrence such as Eqn. (2.6), in which the additive term consists of constants and/or recurrence variables for which we know the solution are referred to as stratified recurrences.

In the context of CRA, a stratified recurrence of stratum 0 is a simple recurrence. A stratified recurrence of stratum $D + 1$ is of the form

$$cx^{[k]} = cx^{[k-1]} + c'y^{[k-1]} + d,$$

where $c$ and $c'$ are rational vectors, $y$ is a vector of variables satisfying a stratified recurrence of strata less than or equal to $D$, and $d$ is a constant. Solutions of stratified recurrences are polynomials in the induction variable $k$. Thus, stratified recurrence equations can be rewritten as

$$cx^{[k]} = cx^{[k-1]} + p(k),$$

which admits a polynomial solution.

Finally, consider the variable $x$. In one path, $x$ is increased by one, and in the other $x$ stays the same. Thus, $x$ does not satisfy a simple recurrence. Furthermore, it can be checked that $x$ does not satisfy a stratified recurrence equation nor is there any linear term involving $x$ that satisfies a stratified recurrence equation. However, $\phi_{\text{body}}$ does entail the following recurrence inequations:

$$x^{[k]} \geq x^{[k-1]} \quad x^{[k]} \leq x^{[k-1]} + 1.$$
21

The right-hand side of a recurrence inequation has the same form as the right-hand side of a recurrence equation, and as far as solving is concerned, recurrence inequations are treated the same as recurrence equations. Therefore, solving these recurrence inequations and combining the results with the closed forms for $i$, $x + y$, and the loop condition yields the following summary for the program in Fig. 4.1 (b):

\[
i' = N \\
\land y' + z' = y + z + \frac{N(N + 1)}{2} \\
\land x \leq x' \leq x + N
\]

This example demonstrates the types of recurrences that CRA is able to extract and solve. In summary CRA, extracts stratified recurrence equations and inequations over linear combinations of program variables, and the solutions to these recurrences are polynomials over a loop induction variable.

2.1.1 Sound Algebraic Program Analysis

We briefly cover what we mean when we say an algebraic program analysis is sound. Let $TR = \langle TR, \cup, \circ, *, \emptyset, \text{id} \rangle$ be the Kleene algebra of transition relations. In this case, $\cup$ is relational union, $\circ$ is relational composition, and $*$ is reflexive transitive closure. The additive identity for $TR$ is the empty relation $\emptyset$, and the multiplicative identity is the identity relation $\text{id}$.

We use a transition relation to represent the input/output relationship of each member of some alphabet of actions $\Sigma$. We assume $TR$ has a corresponding semantic function, $TR[\cdot] : \Sigma \rightarrow TR$, which is extended to regular expressions over $\Sigma$ in the same manner described in Eqn. (2.1). If we take $\Sigma$ to be the set of program actions, then $TR[R]$ denotes the input/output relationship of the program paths described by $R$. In general, it is uncomputable to determine the exact value of $TR[R]$. Thus, we use an abstract domain $A$ to approximate $TR$.

We consider a sound algebraic analysis to consist of an abstract domain $A = \langle A, \oplus, \otimes, \circ, 0, 1 \rangle$, a semantic function $A[\cdot] : \Sigma \rightarrow A$, and a concretization function $\gamma : A \rightarrow TR$. The concretization function $\gamma$ defines which abstract elements of $A$ over-approximate which elements of $TR$. That is, $c \subseteq \gamma(a)$ indicates that $a \in A$ over-approximates $c \in TR$.

Definition 2.1. We consider a sound algebraic program analysis over an alphabet $\Sigma$ to consist of a triple $(A, A[\cdot], \gamma)$, where

1. $\gamma(0) = \emptyset$
2. $\gamma(1) \supseteq \text{id}$

3. $TR[r] \subseteq \gamma(A[r])$, for all $r \in \Sigma$

4. $\gamma(a_1) \cup \gamma(a_2) \subseteq \gamma(a_1 \oplus a_2)$; $\gamma(a_2) \circ \gamma(a_1) \subseteq \gamma(a_1 \odot a_2)$; $\gamma(a)^* \subseteq \gamma(a^*)$ for all $a_1, a_2, a \in A$.

Defn. 2.1 gives a correspondence between the operators of TR and the operators of A, and allows a sound result to be computed in a bottom-up manner:

**Lemma 2.2.** Let $(A, A[\cdot], \gamma)$ be a sound interpretation over an alphabet $\Sigma$. Then for any $R \in \text{regexp}_{\Sigma}$, we have $TR[R] \subseteq \gamma(A[R])$.

**Proof.** Can be directly proved via structural induction on $R$. \qed

### 2.2 Gröbner Bases

In this subsection, I give a self-contained and introductory presentation on relevant concepts from algebraic geometry. The concepts of varieties and ideals are presented. From a mathematical point of view a variety is a set of points that are solutions to a set of polynomials, and a (polynomial) ideal is a set of polynomials that are related to each other through multiplication and addition. These concepts are related in that a set of polynomials define a set of solutions, and a set of points define a set of polynomials that are zero on those set of points. This relationship is the topic of algebraic geometry. Varieties are the geometric side of the topic and ideals are the algebraic side.

From a program-analysis point of view, algebraic geometry is relevant because the multivariate-division algorithm, along with algorithms for computing Gröbner bases, are effective algorithms for manipulating polynomial ideals. In other words, these algorithms result in principled methods that can automatically reason about *non-linear* arithmetic; Chapters 3, 5 and 6 make direct use of these concepts. Those later chapters do not make reference to varieties; however, they are presented here to give a concrete connection between sets of polynomials and their solutions.

This subsection is essentially a summary of concepts from the wonderful text *Ideals, Varieties, and Algorithms* [38]. For more detail on the topic, *Ideals, Varieties, and Algorithms* provides an excellent introduction.

#### 2.2.1 Varieties and Ideals

We begin with some standard definitions of rings and fields. All the rings considered in this dissertation are commutative.
Definition 2.3. A ring, \((R, +, \cdot, 0, 1)\) is an algebraic structure that consists of a set \(R\), binary operations \(+\) (addition) and \(\cdot\) (multiplication), along with elements \(0 \in R\) and \(1 \in R\) such that the following hold:

1. \((R, +, 0)\) is an abelian group:
   - (Addition is associative) \((a + b) + c = a + (b + c)\) for all \(a, b, c \in R\).
   - (Addition is commutative) \(a + b = b + a\) for all \(a, b \in R\).
   - (Additive identity) \(a + 0 = 0 + a = a\) for all \(a \in R\).
   - (Additive inverse) For every \(a \in R\), there exists an \(-a \in R\) such that \(a + (-a) = -a + a = 0\).

2. \((R, \cdot, 1)\) is an commutative monoid:
   - (Multiplication is associative) \((a \cdot b) \cdot c = a \cdot (b \cdot c)\) for all \(a, b, c \in R\).
   - (Multiplication is commutative) \(a \cdot b = b \cdot a\) for all \(a, b \in R\).
   - (Multiplicative identity) \(a \cdot 1 = 1 \cdot a = a\) for all \(a \in R\).

3. Multiplication distributes: \(a \cdot (b + c) = a \cdot b + a \cdot c\) for all \(a, b, c \in R\).

When discussing a ring \((R, +, \cdot, 0, 1)\), we often use the set \(R\) to refer to the whole algebraic structure.

Example 2.4. The integers \(\mathbb{Z}\), the rational numbers \(\mathbb{Q}\), the real numbers \(\mathbb{R}\), and the complex numbers \(\mathbb{C}\) are rings with standard addition and multiplication.

Definition 2.5. A field, \((F, +, \cdot, 0, 1)\) is a ring such that \(0 \neq 1\) and every \(a \in F\) has a multiplicative inverse. That is, for every \(a \in F\), there exists an \(a^{-1} \in F\) such that \(a^{-1}a = aa^{-1} = 1\).

Example 2.6. \(\mathbb{Q}\), \(\mathbb{R}\), and \(\mathbb{C}\) are fields. \(\mathbb{Z}\) is not a field.

Our definitions of monomials and polynomials are the standard definitions.

Definition 2.7. Fix a set of variables \(x_1, \ldots, x_n\). A monomial is a product of the form

\[x_1^{a_1} \cdots x_n^{a_n}\]

where \(a_1, \ldots, a_n\) are non-negative integers. The total degree of a monomial is \(a_1 + \cdots + a_n\). The multi-degree of a monomial is \((a_1, \ldots, a_n) \in \mathbb{Z}_{\geq 0}^n\).

Definition 2.8. Let \(K\) be a field. A polynomial, \(f\), over \(x_1, \ldots, x_n\) is a finite linear combination of monomials with coefficients from \(K\).
Example 2.9. \(x^2yz^3\) is a monomial over the variables \(x, y,\) and \(z\). The total degree of \(x^2yz^3\) is 6 = 2 + 1 + 3. The multi-degree of \(x^2yz^3\) is \((2, 1, 3)\). \(x^2yz^3 + 2x + 3\) is an example of a polynomial over say the field of rational numbers \(\mathbb{Q}\).

Definition 2.10. The set of all polynomials with coefficients from \(\mathbb{K}\) over \(x_1, \ldots, x_n\) is a commutative ring and is denoted by \(\mathbb{K}[x_1, \ldots, x_n]\).

In this chapter, definitions and theorems, unless otherwise required, are stated in terms of a polynomial ring over generic field \(\mathbb{K}\). For the subsequent technical chapters this generality is not needed. Our theorems and implementations are designed for working over the polynomial ring over the rational numbers, \(\mathbb{Q}[x_1, \ldots, x_n]\).

Definition 2.11. A polynomial homomorphism is a ring homomorphism \(f : \mathbb{K}[X] \rightarrow \mathbb{K}[Y]\) between two polynomial rings. Provided that \(X\) is finite, a polynomial homomorphism can be represented by its action on the variables \(X\). \(f\) is a polynomial endomorphism if \(X = Y\).

The technical chapters of this dissertation do not make direct use of the following definition. However, it is given here to give intuition and a gentle introduction into the ideas of algebraic geometry.

Definition 2.12. Let \(P\) be a (possibly infinite) set of polynomials in \(\mathbb{K}[x_1, \ldots, x_n]\). The set of points \((c_1, \ldots, c_n) \in \mathbb{K}^n\) such that every polynomial in \(P\) evaluates to zero is an affine (also possibly algebraic) variety, or just variety. More succinctly, a variety \(V(P)\) is the set of points

\[V(P) = \{(c_1, \ldots, c_n) \in \mathbb{K}^n \mid p(c_1, \ldots, c_n) = 0 \forall p \in P\}\]

Varieties are the geometric side of algebraic geometry.

Example 2.13. The set of points of the unit circle in \(\mathbb{R}^2\) is the affine variety \(V(x^2 + y^2 - 1)\).

Because (in general) varieties describe an infinite set of points, we represent varieties by the set of polynomials \(P\) that generate them. However, it is not clear what makes a good set of polynomials \(P\).

Example 2.14. Consider the variety \(V(2x^2 + 3y^2 - 11, x^2 - y^2 - 3)\). It turns out that the only points that satisfy these equations are \((x = \pm 2, y = \pm 1)\). Therefore, \(V(2x^2 + 3y^2 - 11, x^2 - y^2 - 3) = V(x^2 - 4, y^2 - 1)\).

In general there are an infinite number of sets of polynomials that describe the same variety. In fact, consider a variety \(V(P)\), and a consequence \(q\) with

\[\forall p \in P. \ p(x) = 0 \models q(x) = 0.\]
Then $V(P) = V(P \cup \{q\})$.

A central question of algebraic geometry is can we find “good” polynomial consequences to represent varieties. For example, perhaps we are given some polynomials that represent a variety and we want to ask if there are purely linear polynomials that represent that same variety.

To begin to answer this question we must turn to the algebraic side of algebraic geometry. This side of the dichotomy will be embodied by ideals.

**Definition 2.15.** Let $R$ be a (commutative) ring. A subset $I \subseteq R$, is an ideal if

1. $0 \in I$
2. If $f, g \in I$, then $f + g \in I$
3. If $f \in I$, then $h \cdot f \in I$ for all $h \in R$.

In this dissertation we are interested in ideals over polynomial rings. Examples of polynomial ideals will appear soon, for instance in Ex. 2.24. The examples are delayed in order for the next few definitions and theorems to further explain notation.

**Definition 2.16.** Let $Z$ be a set of elements of $R$. Let $\langle Z \rangle$ denote the following set:

$$\langle Z \rangle = \{ r_1 z_1 + \cdots + r_s z_s \mid z_i \in Z \land r_j \in R \land s \in \mathbb{N} \}.$$  

The next lemma shows that $\langle Z \rangle$ is an ideal, and consequently we call $\langle Z \rangle$ the ideal generated by $Z$.

**General Properties of Ideals**

In this subsection, we’ll indulge in a digression about some general properties of ideals. This will go beyond the basic exploration of algebraic geometry. The reason for this generality will be to incorporate more general results. Namely, so far we have considered polynomials over a field $\mathbb{K}$. However, this excludes structures such as $\mathbb{Z}[x_1, \ldots, x_n]$.

**Lemma 2.17.** If $Z \subseteq R$, then $\langle Z \rangle$ is the smallest ideal of $R$ containing $Z$.

**Proof.** First, we show that $\langle Z \rangle$ is an ideal. Let $a, b \in \langle Z \rangle$. Then $a = a_1 z_1 + \cdots + a_s z_s$ for some $\{z_1, \ldots, z_s\} \subseteq Z$ and $\{a_1, \ldots, a_s\} \subseteq R$. Similarly, $b = b_1 z'_1 + \cdots + b_t z'_t$ for some $\{z'_1, \ldots, z'_t\} \subseteq Z$ and $\{b_1, \ldots, b_t\} \subseteq R$. Let $r \in R$.
Then, due to the properties of a ring

\[ ra = r(a_1z_1 + \ldots + a_sz_s) \]
\[ = (ra_1)z_1 + \ldots (ra_s)z_s \]

Because R is a ring \( ra_1 \in R \). Thus \( ra \in \langle Z \rangle \).

\( \{z_1, \ldots, z_s, z'_1, \ldots, z'_t\} \subseteq Z \) and \( \{a_1, \ldots, a_s, b_1, \ldots, b_t\} \subseteq R \). Thus, \( a + b \in \langle Z \rangle \). This shows that \( \langle Z \rangle \) is an ideal.

Now, to show \( \langle Z \rangle \) is the smallest ideal containing \( Z \). Let \( Y \) be an ideal with \( Z \subseteq Y \subseteq \langle Z \rangle \). Let \( x \in \langle Z \rangle \). By definition

\[ x = r_1z_1 + \cdots + r_sz_s. \]

Note that because \( Y \) is an ideal and each \( z_i \in Y \), \( r_i z_i \in Y \). Furthermore, because ideals are closed via sums, \( x \in Y \) holds as well. Therefore, \( Y = \langle Z \rangle \). \( \square \)

Lem. 2.17 shows that we can consider any ideal to be generated by some (possibly infinite) set. That is if \( X \) is an ideal already then \( \langle X \rangle = X \). We will be concerned with ideals that are finitely generated.

**Definition 2.18.** Let \( R \) be a ring. \( R \) is Noetherian, if \( R \) satisfies the ascending chain condition. That is, for every increasing sequence of ideals of \( R \), \( I_1 \subseteq \cdots \subseteq I_k \subseteq I_{k+1} \subseteq \cdots \), there exists a natural number \( n \) with \( I_n = I_{n+1} = \cdots \).

**Definition 2.19.** We say that an ideal \( I \subseteq R \) is finitely generated if there exists \( \{f_1, \ldots, f_s\} \) such that \( \langle f_1, \ldots, f_s \rangle = I \).

**Theorem 2.20.** The following statements are equivalent

1. \( R \) is a Noetherian ring.

2. Every ideal \( I \subseteq R \) is finitely generated.

**Proof.** (2) \( \implies \) (1). Consider an increasing sequence of ideals \( I_1 \subseteq \cdots \subseteq I_k \subseteq I_{k+1} \subseteq \cdots \). Let \( I = \cup_{i=1}^{\infty} I_i \).

First, we show that \( I \) is an ideal of \( R \). Let \( a, b \in I \). By definition \( a \in I_i \) and \( b \in I_j \) for some \( i \) and \( j \). Without loss of generality assume \( i \leq j \). Then \( a \in I_j \), so \( a + b \in I_j \), because \( I_j \) is an ideal. Therefore, \( a + b \in I \). Now, let \( r \in R \). \( ra \in I_i \) because \( I_i \) is an ideal. Thus, \( ra \in I \).

Because \( I \) is an ideal of \( R \), by assumption \( I \) is finitely generated. Let \( \langle f_1, \ldots, f_s \rangle = I \). Each \( f_i \) is contained in some \( I_j \), say \( f_i \in I_{j_i} \) for some \( j_i, i = 1, \ldots, s \). Let \( N = \max_{i=1,\ldots,s} j_i \). This means all the \( f_i \)’s are contained in \( I_N \). Thus,

\[ I = \langle f_1, \ldots, f_s \rangle \subseteq I_N \subseteq I_{N+1} \subseteq \cdots \subseteq I. \]
This shows $I = I_N = I_{N+1} = \ldots$, and therefore the ascending chain stabilizes at $N$.

(1) $\implies$ (2). Suppose, there was some ideal, $I$, of a Noetherian ring $R$ that was not finitely generated. Let $I$ be generated by an infinite set $X$. By the axiom of choice we can create an infinite sequence of finite sets, $X_1 \subseteq X_2 \subseteq \ldots \subseteq X_k \subseteq X_{k+1} \subseteq \ldots$, with each $X_i \subset X$, and $|X_i| = i$. In particular, consider $X = \{x_1, x_2, \ldots, x_k, \ldots\}$. Then we have $X_1 = \{x_1\}$, $X_2 = \{x_1, x_2\}$, $X_3 = \{x_1, x_2, x_3\}$, and so on. If $\langle X_k \rangle = \langle X \rangle = I$, then $I$ is generated by the finite set $X_k$. However, this contradicts the assumption that $I$ is not finitely generated. Therefore, $\langle X_k \rangle \subseteq \langle X \rangle$ for each $k$. Thus, we can construct the infinite chain of ideals

$$
\langle X_1 \rangle \subseteq \langle X_2 \rangle \subseteq \ldots \subseteq \langle X_k \rangle \subseteq \ldots
$$

However, this cannot hold because $R$ is Noetherian. Thus, $I$ must be finitely generated.

**Theorem 2.21.** (Hilbert’s Basis Theorem) Let $R$ be a Noetherian ring. Then $R[x_1, \ldots, x_n]$ is a Noetherian ring.

**Corollary 2.22.** Because fields as well as $\mathbb{Z}$ are Noetherian rings, ideals of $\mathbb{Q}[x_1, \ldots, x_n]$, $\mathbb{R}[x_1, \ldots, x_n]$, $\mathbb{C}[x_1, \ldots, x_n]$, $\mathbb{Z}[x_1, \ldots, x_n]$, and $\mathbb{K}[x_1, \ldots, x_n]$, for a field $\mathbb{K}$, are finitely generated.

**Definition 2.23.** We call $\{f_1, \ldots, f_s\}$ a basis for $I$ if $\langle f_1, \ldots, f_s \rangle = I$.

**Example 2.24.** Here we give some examples of polynomial ideals.

1. $x^2 - 2x + 2 - y \in \langle x - 1 - t, y - 1 - t^2 \rangle$, because

   $$x^2 - 2x + 2 - y = (x - 1 + t)(x - 1 - t) + (-1)(y - 1 - t^2)$$

2. The set $\langle x, y \rangle$ consists of all non-constant polynomials $\mathbb{K}[x, y]$, along with the polynomial 0.

3. Recall Ex. 2.14. It turns out that $\langle 2x^2 + 3y^2 - 11, x^2 - y^2 - 3 \rangle = \langle x^2 - 4, y^2 - 1 \rangle$. To show this, one can show that $\{2x^2 + 3y^2 - 11, x^2 - y^2 - 3\} \subseteq \langle x^2 - 4, y^2 - 1 \rangle$ and $\{x^2 - 4, y^2 - 1\} \subseteq \langle 2x^2 + 3y^2 - 11, x^2 - y^2 - 3 \rangle$.

4. (1) $= \mathbb{K}[x_1, \ldots, x_n], \langle 0 \rangle = \{0\}$

   Intuitively, one can consider a polynomial ideal $I$ as a collection of polynomial equations $\{p = 0 \mid p \in I\}$. The conditions for the collection to be an ideal can be read as inference rules: $0 = 0$, if $p = 0$ and $q = 0$ then $p + q = 0$, and if $p = 0$ then $pq = 0$. Thus, ideals are important, because, in light of Hilbert’s basis theorem, we can represent an infinite set of polynomial consequences with a finite set of generators. However, Ex. 2.24 shows that the basis for an ideal is not necessarily unique.

Now we connect varieties with ideals.
Theorem 2.25. Consider a set \( P \subseteq \mathbb{K}[x_1, \ldots, x_n] \). \( V(P) = V(\langle P \rangle) \).

Proof. Consider \( c \in V(\langle P \rangle) \). Thus \( p'(c) = 0 \) for all \( p' \in \langle P \rangle \). Since \( P \subseteq \langle P \rangle \), \( p(c) = 0 \) for all \( p \in P \). Therefore, \( V(P) \supset V(\langle P \rangle) \).

Let \( c = (c_1, \ldots, c_n) \in V(P) \). Then \( p(c) = 0 \) for all \( p \in P \). Consider some arbitrary polynomial \( p' \in \langle P \rangle \). By the definition of \( \langle P \rangle \), \( p' = h_1f_1 + \cdots + h_sf_s \) for some polynomials \( h_j \in \mathbb{K}[x_1, \ldots, x_n] \), and \( f_i \in P \).

\[
p'(c) = h_1(c)f_1(c) + \cdots + h_s(c)f_s(c) = h_1(c)0 + \cdots + h_s(c)0 = 0
\]

Because \( p' \) was arbitrary, \( p'(c) = 0 \) for all \( p' \in \langle f_1, \ldots, f_s \rangle \). Thus \( c \in V(\langle P \rangle) \), and \( V(P) \subseteq V(\langle P \rangle) \). \( \square \)

Corollary 2.26. Combining Thm. 2.25 with Hilbert’s basis theorem, we get that every variety \( V(P) \) can be written as \( V(S) \) with \( S \) finite.

One way to think about the results is that varieties are really defined by ideals. However, as we noted earlier, representations of ideals is not unique. Therefore, our representations of varieties are in general also not unique. Yet some of these representations are better than others, and in the next section we’ll describe algorithms that can be performed on ideals and manipulate ideals.

So far we’ve considered varieties of ideals. That is, given a set of polynomials as an ideal, what is the set of points that vanish on all those polynomials. The previous corollary shows that this is the same set as considering a variety of a finite set of polynomials. Thus, we get the following logical connection

Corollary 2.27. Consider a set of polynomials \( f_1, \ldots, f_s \subset \mathbb{K}[x_1, \ldots, x_n] \). For every \( g \in \langle f_1, \ldots, f_s \rangle \)

\[
\bigwedge_{i=1}^{n} f_i(x) = 0 \models g(x) = 0
\]

Proof. The set of \( x \) for which \( \bigwedge_{i=1}^{n} f_i(x) = 0 \) is exactly \( V(f_1, \ldots, f_s) \), which by Thm. 2.25 is \( V(\langle f_1, \ldots, f_s \rangle) \). Therefore, because \( g \in \langle f_1, \ldots, f_s \rangle \), \( g(x) = 0 \). \( \square \)

This gives us a way to determine polynomial consequences for an arbitrary field \( \mathbb{K} \) (assuming we can effectively compute in \( \mathbb{K} \)). However, we can ask if this is a complete method. That is, fix some field \( \mathbb{K} \) and suppose we have a polynomial \( g \) such that \( \bigwedge_{i=1}^{n} f_i(x) = 0 \models g(x) = 0 \). Is it the case that \( g \in \langle f_1, \ldots, f_s \rangle \)? In other words does ideal membership completely determine polynomial consequences for any field \( \mathbb{K} \)? Unfortunately, the answer is no.
Definition 2.28. Let \( V \subseteq \mathbb{K}^n \) be a variety. Let

\[
I(V) = \{ p \in \mathbb{K}[x_1, \ldots, x_n] \mid p(c_1, \ldots, c_n) = 0 \text{ for all } (c_1, \ldots, c_n) \in V \}
\]

It can easily be shown that \( I(V) \) is an ideal. Thus we call \( I(V) \) the ideal of \( V \), sometimes also the vanishing ideal.

In other words, \( I(V) \) consists of all polynomials that vanish on the variety \( V \). More specifically, \( I(V(f_1, \ldots, f_s)) \) consists of the set of \( g \) such that \( \wedge_{i=1}^{n} f_i(x) = 0 \Rightarrow g(x) \) holds.

Thus, our previous question can now be rephrased as: in general, what is the relationship between \( \langle f_1, \ldots, f_s \rangle \) and \( I(V(f_1, \ldots, f_s)) \). The reasoning of Cor. 2.27 says that \( \langle f_1, \ldots, f_s \rangle \subseteq I(V(f_1, \ldots, f_s)) \), but the reverse inclusion doesn’t hold in general.

Example 2.29. Consider the polynomials \( x^2, y^2 \). \( V(x^2, y^2) = (0, 0) \). Thus, \( \{x, y\} \subset I(V(x^2, y^2)) \), but neither \( x \) nor \( y \) is in \( \langle x^2, y^2 \rangle \).

The previous example illustrates the gap between determining ideal membership, e.g. \( p \in \langle f_1, \ldots, f_s \rangle \), and determining whether \( p = 0 \) is a consequence of \( f_1 = 0, \ldots, f_s = 0 \). However, Cor. 2.27 shows that we can answer positively that \( p = 0 \) is a consequence of \( f_1 = 0, \ldots, f_s = 0 \) if \( p \in \langle f_1, \ldots, f_s \rangle \). Furthermore, in the next section it will be shown that determining ideal membership is decidable.

2.2.2 Algorithms for Ideals

Definition 2.30. Let \( f_1, \ldots, f_s, g \subset \mathbb{K}[x_1, \ldots, x_n] \). The ideal-membership problem is to determine whether \( g \in \langle f_1, \ldots, f_s \rangle \).

We will show that the ideal-membership problem to be decidable, but it is not immediately obvious that it should be. One issue is that, as we observed before, the representation of \( \langle f_1, \ldots, f_s \rangle \) is not unique.

To motivate some of the subsequent definitions, we will first consider the case of univariate polynomials, and why the ideal-membership problem is easier in such a context. We’ll then look at why some of these properties don’t hold in the multivariate case, but how we can remedy the situation.

Univariate polynomials

The ring of univariate polynomials over a field \( \mathbb{K} \) admits a Euclidean function that maps polynomials to their degree. This means there is an effective division algorithm and greatest common divisor algorithm for univariate polynomials (assuming we can compute division in \( \mathbb{K} \)). This also makes the ring of univariate polynomials a principal ideal domain (PID). This means that for any ideal \( I \subseteq \mathbb{K}[x] \), there exists a polynomial
h ∈ K[x] such that I = ⟨h⟩. Furthermore, given polynomials \(f_1, \ldots, f_s \subset K[x]\), \(⟨f_1, \ldots, f_s⟩ = \langle \gcd(f_1, \ldots, f_s)\rangle\), where \(\gcd(f_1, \ldots, f_s)\) is the unique greatest common divisor of \(f_1, \ldots, f_s\).

The division algorithm works over two polynomials \(f, g\) and produces polynomials \(q, r\) such that \(f = qg + r\) and either (i) \(r = 0\), or (ii) \(r \neq 0\) and the leading term of \(g\) does not divide any term of \(r\). The basic idea of the algorithm is to arrange polynomials in decreasing degree order, start with \(r = f\), and subtract off multiples of \(g\) that remove the leading term of \(r\). This process repeats until \(r = 0\) or the leading term of \(g\) does not divide any term of \(r\). One thing to note is that during this process \(r\) will be updated and can contain new monomials that were never present in \(f\) or \(g\). However, these newly appearing monomials will have smaller degree than the previous leading term of \(r\). Therefore, the algorithm always terminates, because comparing the degree of polynomials is well-ordered. Observe the key to the algorithm’s success is the well-ordering on monomials.

With these two algorithms in hand, the ideal-membership problem can be solved for univariate polynomials. Given \(f_1, \ldots, f_s, g \subset K[x]\), compute \(h = \gcd(f_1, \ldots, f_s)\). Then divide \(g\) by \(h\). If the remainder is zero, then \(g = hq\) for some \(q\); thus, \(g \in ⟨h⟩ = ⟨f_1, \ldots, f_s⟩\).

**Multivariate Polynomials Issues**

The solution to the ideal-membership problem for multivariate polynomials works roughly the same way as for univariate polynomials. Given \(f_1, \ldots, f_s \subset K[x_1, \ldots, x_n]\), we will compute a special basis for \(⟨f_1, \ldots, f_s⟩\). Then we will divide \(g\) by the elements of this basis. If the remainder on division is 0, then \(g \in ⟨f_1, \ldots, f_s⟩\); otherwise, \(g \notin ⟨f_1, \ldots, f_s⟩\).

However, there a few issues that make this generalization not immediate. First, unlike the univariate case, the ring of polynomials \(K[x_1, \ldots, x_n]\) is not a principal ideal domain. This means that our “good” basis will in general consist of multiple polynomials, \(q_1, \ldots, q_l\). As a consequence, our division algorithm will have to consider multiple divisors at the same time. Another issue is that the success of the standard division algorithm depends on some sort of well-ordered sequence that decreases at every step of the algorithm. In the case of univariate polynomials this function is the degree of terms. However, in the case of multivariate polynomials terms have multi-degrees. Thus, we need to consider a more general ordering on monomials.

**Monomial Orders**

A remark on some new notation to be used in this section. Consider monomials over a set of variables \(x_1, \ldots, x_n\). Let \(\alpha = (a_1, \ldots, a_n) \in \mathbb{Z}_{\geq 0}^n\). We will use \(x^\alpha\) as shorthand for the monomial \(x_1^{a_1} \cdots x_n^{a_n}\). Sums and differences of values from \(\mathbb{Z}_{\geq 0}^n\) are defined point-wise.

**Definition 2.31.** Let \(<\) be a relation on the monomials \(x^\alpha, \alpha \in \mathbb{Z}_{\geq 0}^n\). \(<\) is a monomial order if:
1. \( \ll \) is a total order.

2. For any \( \gamma \in \mathbb{Z}_{\geq 0}^n \), if \( x^\alpha \ll x^\beta \), then \( x^{\alpha + \gamma} \ll x^{\beta + \gamma} \)

3. \( 1 \ll x^\alpha \) for all \( \alpha \in \mathbb{Z}_{\geq 0}^n \)

Lemma 2.32. Any monomial order is a well-ordering. This property is a corollary of Dickson’s lemma.

Definition 2.33. The lexicographic monomial order (lex) is defined as follows. Let \( \alpha, \beta \in \mathbb{Z}_{\geq 0}^n \). \( x^\alpha \gg_{\text{lex}} x^\beta \) if the leftmost nonzero entry of \( \alpha - \beta \) is positive.

For \( \alpha = (a_1, \ldots, a_n) \in \mathbb{Z}_{\geq 0}^n \), let \( |\alpha| = a_1 + \cdots + a_n \).

Definition 2.34. The graded lexicographic (grlex) or degree lexicographic (deglex) monomial order is defined as follows. Let \( \alpha, \beta \in \mathbb{Z}_{\geq 0}^n \). \( x^\alpha \gg_{\text{grlex}} x^\beta \) if \( |\alpha| > |\beta| \) or \( |\alpha| = |\beta| \) and \( x^\alpha \gg_{\text{lex}} x^\beta \).

Definition 2.35. The graded reverse lexicographic (grevlex) monomial order is defined as follows. Let \( \alpha, \beta \in \mathbb{Z}_{\geq 0}^n \). \( x^\alpha \gg_{\text{grevlex}} x^\beta \) if \( |\alpha| > |\beta| \) or \( |\alpha| = |\beta| \) and the rightmost nonzero entry of \( \alpha - \beta \) is negative.

Example 2.36. Consider monomials in \( \mathbb{K}[x, y, z] \), with \( x \) lexicographically bigger than \( y \), and \( y \) lexicographically bigger than \( z \).

- \( x^3 \gg_{\text{lex}} x^2z^2 \gg_{\text{lex}} xy^2z \gg_{\text{lex}} z^2 \)
- \( x^2z^2 \gg_{\text{grlex}} xy^2z \gg_{\text{grlex}} x^3 \gg_{\text{grlex}} z^2 \)
- \( xy^2z \gg_{\text{grevlex}} x^2z^2 \gg_{\text{grevlex}} x^3 \gg_{\text{grevlex}} z^2 \)

Definition 2.37. We say a monomial order \( \gg_{\sigma} \) is a graded order if for every \( x^\alpha \) and \( x^\beta \) with \( |\alpha| > |\beta| \) then \( x^\alpha \gg_{\sigma} x^\beta \).

For us the relevancy of graded orders is that linear monomials (and polynomials) will always be smaller than nonlinear monomials. What this means is that by using a graded order we can force algorithms to favor producing linear polynomials rather than nonlinear ones.

Example 2.38. Grlex and grevlex are graded orders, but thelex order is not.

Definition 2.39. Let \( Y \cup Z \) be a partition of a set of variables \( X \). Let \( m = m_y m_z \) and \( n = n_y n_z \) be monomials with \( m_i \) and \( n_i \) containing only \( Y \) variables, and \( m_z \) and \( n_z \) only containing \( Z \) variables. Let \( \gg_{\sigma} \) be some monomial order. The elimination order \( \gg_Y \) defines \( m \gg_Y n \) as either (1) \( m_y \gg_{\sigma} n_y \) or (2) \( m_y = n_y \) and \( m_z \gg_{\sigma} n_z \).

The basic idea of an elimination order is that any monomial involving a variable in the first block, \( Y \), will be greater than any monomial which does not contain a variable in the same block.
Algorithm 1: Multivariate Division

Input: \( f_1, \ldots, f_s, g \in \mathbb{K}[x_1, \ldots, x_n] \)
Result: \( a_1, \ldots, a_s, r \in \mathbb{K}[x_1, \ldots, x_n] \) with \( g = a_1 f_1 + \cdots + a_s f_s + r \)

1. \( a_1 \leftarrow 0, \ldots, a_s \leftarrow 0 \);
2. \( r \leftarrow 0 \);
3. while \( g \neq 0 \) do
   4. \( i \leftarrow 1 \);
   5. \( \text{nodivision} \leftarrow \text{true} \);
   6. while \( i \leq s \) and \( \text{nodivision} \) do
      7. if \( \text{LT}(f_i) \) divides \( \text{LT}(g) \) then
         8. \( a_i \leftarrow a_i + \frac{\text{LT}(g)}{\text{LT}(f_i)} \);
         9. \( g \leftarrow g - \frac{\text{LT}(g)}{\text{LT}(f_i)} f_i \);
      10. \( \text{nodivision} \leftarrow \text{false} \);
   11. if \( \text{nodivision} \) then
      12. \( r \leftarrow r + \frac{\text{LT}(g)}{\text{LT}(f_i)} \);
      13. \( g \leftarrow g - \frac{\text{LT}(g)}{\text{LT}(f_i)} \);

Example 2.40. Consider a variable set \( w, x, y, z \), and the elimination order \( \gg_{(w)} \), with the underlying order, \( \gg_\sigma \), being the grevlex order.

\[
\begin{align*}
w^2 z^2 & \gg_{(w)} w x^3 \gg_{(w)} w \gg_{(w)} x y^2 z \gg_{(w)} x^2 z^2
\end{align*}
\]

Multivariate Division

Given a monomial order, we can talk about the leading terms, \( \text{LT} \), and leading monomials, \( \text{LM} \), of multivariate polynomials. This allows the definition of a multivariate division algorithm.

Theorem 2.41. Fix a monomial order \( \gg \). Given a set of polynomials \( f_1, \ldots, f_s, g \in \mathbb{K}[x_1, \ldots, x_n] \) and a dividend \( g \), the multivariate division algorithm produces \( a_1, \ldots, a_s, r \in \mathbb{K}[x_1, \ldots, x_n] \) such that

\[ g = a_1 f_1 + \cdots + a_s f_s + r \]

with \( r = 0 \) or \( r \) contains no terms which are divisible by a \( \text{LT}(f_i) \). Furthermore

\[
\max\{\text{LT}(a_1)\text{LT}(f_1), \ldots, \text{LT}(a_s)\text{LT}(f_s), \text{LT}(r)\} = \text{LT}(f)
\]

The multi-variate division algorithm seems to give a method for determining the ideal-membership problem. That is to determine if \( g \in \langle f_1, \ldots, f_s \rangle \), one divides \( g \) by \( f_1, \ldots, f_s \). If \( r = 0 \), then \( g \in \langle f_1, \ldots, f_s \rangle \). However, \( r = 0 \) is not a necessary condition.

Example 2.42. Let \( g = x y^2 - x \) and \( f_1 = x y + 1, f_2 = y^2 - 1 \), and use a lexicographic order with \( x \) lexicographically
bigger than $y$. Then the division algorithm yields

\[
\begin{array}{c|cc}
 a_1 : & y & r \\
 a_2 : \\
 x + 1 & xy^2 & -x \\
 y - 1 & xy^2 & y \\
 & -x & -y \\
 & -y & -x \\
\end{array}
\]

This shows $xy^2 - x = y(xy + 1) + 0(y^2 - 1) + (-x - y)$. However, if one were to run the division algorithm with $f_1$ and $f_2$ flipped one would obtain

\[
xy^2 - x = x(y^2 - 1) + 0(xy + 1).
\]

Thus, $xy^2 - x \in (xy + 1, y^2 - 1)$. This example demonstrates two things. First, the division algorithm depends on the order in which the divisors are given. This means in general the remainder $r$ is not unique. Second, unlike the univariate case, division is, by itself, insufficient to solve the ideal-membership problem.

**Example 2.43.** Again consider $f_1 = xy + 1, f_2 = y^2 - 1$, but let $g = x + y$. If we were to run the division algorithm using any order of $f_1$ and $f_2$ we would obtain the equation $x + y = 0(xy + 1) + 0(y^2 - 1) + x + y$. However,

\[
x + y = y(xy + 1) - x(y^2 - 1),
\]

so $x + y \in (xy + 1, y^2 - 1)$.

**Gröbner Bases**

Consider again the ideal-membership problem. Suppose that we have polynomials $f_1, \ldots, f_s, g \in \mathbb{K}[x_1, \ldots, x_n]$. Our straw-man method for determining if $g \in (f_1, \ldots, f_s)$ was to divide $g$ by $f_1, \ldots, f_s$. The previous two examples show that this method is incomplete. Furthermore, the previous example shows that the issue is not with the division algorithm. Perhaps the situation can be remedied if we used a different basis to represent the ideal $(f_1, \ldots, f_s)$. 
**Definition 2.44.** Fix some monomial ordering. Let \( I \subseteq \mathbb{K}[x_1, \ldots, x_n] \) be an ideal different from \((0)\).

\[
\text{LT}(I) = \{ cx^\alpha \mid \exists p \in I. \text{LT}(p) = cx^\alpha \}
\]

\( \langle \text{LT}(I) \rangle \) is the ideal generated by \( \text{LT}(I) \)

**Definition 2.45.** Fix a monomial ordering. Let \( I \subseteq \mathbb{K}[x_1, \ldots, x_n] \) be an ideal. A set of polynomials \( \{q_1, \ldots, q_l\} \subset I \) is a \Gröbner basis for \( I \) if

\[
\langle \text{LT}(I) \rangle = \langle \text{LT}(q_1), \ldots, \text{LT}(q_l) \rangle
\]

An equivalent, but informal definition is a set \( q_1, \ldots, q_l \) is a \Gröbner basis for \( I \) if and only if for every \( p \in I \), \( \text{LT}(p) \) is divisible by one of the \( \text{LT}(q_i) \)'s.

**Theorem 2.46.** Let \( q_1, \ldots, q_l \) be a \Gröbner basis for an ideal \( I \). Then \( \langle q_1, \ldots, q_l \rangle = I \).

**Proof.** By definition \( \{q_1, \ldots, q_l\} \subset I \). Thus \( \langle q_1, \ldots, q_l \rangle \subseteq I \). Consider some \( f \in I \). Using the multivariate division algorithm, divide \( f \) by \( \{q_1, \ldots, q_l\} \). We obtain \( a_1, \ldots, a_l, r \) such that

\[
f = a_1q_1 + \cdots + a_lq_l + r
\]

and either \( r = 0 \) or \( r \neq 0 \) and no term of \( r \) is divisible by any \( \text{LT}(q_i) \). If \( r = 0 \) then \( f \in \langle q_1, \ldots, q_l \rangle \), and we are done. Thus, suppose that \( r \neq 0 \). Note that because \( f, a_1q_1, \ldots, a_lq_l \in I \), \( r = f - a_1q_1 - \cdots - a_lq_l \in I \). Because \( \{q_1, \ldots, q_l\} \) is a \Gröbner basis, \( \text{LT}(r) \) must be divisible by some \( \text{LT}(q_i) \). This is a contradiction, because no term of \( r \) is divisible by any \( \text{LT}(q_i) \). Thus, \( r = 0 \) and \( f \in \langle q_1, \ldots, q_l \rangle \). Because \( f \) was an arbitrary element of \( I \), \( \langle q_1, \ldots, q_l \rangle = I \).

**Example 2.47.** Consider a lex order with \( x \) lexicographically bigger than \( y \). \( \langle xy + 1, y^2 - 1 \rangle \) is not a \Gröbner basis for \( \langle xy + 1, y^2 - 1 \rangle \). To see why recall that \( x + y \in \langle xy + 1, y^2 - 1 \rangle \). Thus \( x \in \langle \text{LT}(\langle xy + 1, y^2 - 1 \rangle) \rangle \), but \( x \notin \langle xy, y^2 \rangle = \langle \text{LT}(xy + 1), \text{LT}(y^2 - 1) \rangle \).

**Example 2.48.** Though at this point it is not obvious why, \( \langle xy + 1, y^2 - 1, x + y \rangle \) is a \Gröbner basis for \( \langle xy + 1, y^2 - 1 \rangle \).

Recall the ideal-membership problem in the univariate case. There we noted that every ideal is a principal ideal. Thus, given a set of univariate polynomials, we could compute their GCD, \( h \in \mathbb{K}[x] \). Then, due to being a PID, every other member of the ideal is a polynomial multiple of \( h \). That is, every element, \( f \in \mathbb{K}[x] \), of the ideal can be written as \( hq \) for some \( q \in \mathbb{K}[x] \). Therefore, \( \deg(f) = \deg(hq) \geq \deg(h) \) for all \( f \). In other words, if we were to consider “generating” the elements of the ideal from \( h \) by writing \( hq \), we would never introduce a polynomial whose \LM is smaller than the \LM of \( h \).
However, this is not the case in the multivariate case. Consider an ideal, \( \langle f_1, \ldots, f_s \rangle \subseteq \mathbb{K}[x_1, \ldots, x_n] \). Any element \( g \) of this ideal, by definition, is written as

\[ g = h_1 f_1 + \cdots + h_s f_s \]

We ask the question: Is it possible that \( \text{LM}(g) \) is smaller (in the monomial order) than \( \text{LM}(f_i) \) for all the \( f_i \)'s? The answer is yes, as shown by Ex. 2.43. To see why this can happen consider a basis of size 2, \( f_1, f_2 \). It is certainly possible for \( \text{LT}(h_1 f_1) = -\text{LT}(h_2 f_2) \). Thus, when we add \( h_1 f_1 + h_2 f_2 \), the leading terms of the two summands may cancel. In such a case, the resulting sum will have a leading monomial that is smaller than either of the polynomials \( f_1 \) and \( f_2 \). Exactly this situation is what makes the multivariate division algorithm insufficient for the ideal-membership problem. The algorithm just doesn’t have the machinery to detect this situation. Also, fortunately, this case of when \( \text{LT}(h_1 f_1) = -\text{LT}(h_2 f_2) \) is the only way for \( \text{LM}(h_1 f_1 + h_2 f_2) \) to be smaller than \( \text{LM}(f_i) \) for \( f_1 \) and \( f_2 \). However, consider the situation if we had a Gröbner basis \( q_1, \ldots, q_t \), and some \( g \in \langle q_1, \ldots, q_t \rangle \). It may be the case that for some \( i, j \), the \( \text{LM}(h_1 q_1 + h_j q_j) \) is smaller than \( \text{LM}(q_k) \) or \( \text{LM}(q_j) \), but by the definition of a Gröbner basis there must be some other \( q_k \), such that the \( \text{LT}(q_k) \) divides \( \text{LT}(h_1 q_1 + h_j q_j) \). Thus, the division algorithm would apply.

The ideas in the preceding paragraph explain why we care about Gröbner bases. Here we’ll state the ideas formally.

**Theorem 2.49.** Let \( q_1, \ldots, q_t \subseteq \mathbb{K}[x_1, \ldots, x_n] \) be a Gröbner basis for an ideal \( I \). Let \( g \in \mathbb{K}[x_1, \ldots, x_n] \). Then there is a unique \( r \in \mathbb{K}[x_1, \ldots, x_n] \) with

1. No term of \( r \) is divisible any of the \( \text{LT}(q_i) \)
2. There is a \( h \in I \), with \( g = h + r \).

**Proof.** By Thm. 2.41, dividing \( g \) by \( q_1, \ldots, q_t \) produces \( a_1, \ldots, a_t, r \). Let \( h = a_1 q_1 + \cdots + a_t q_t \). Then we have a \( h \in I \) and \( r \) that satisfy the above conditions.

To show uniqueness, consider \( g = h + r = h' + r' \). Note \( r - r' = h - h' \in I \). Thus, \( \text{LT}(r - r') \in \langle \text{LT}(I) \rangle = \langle \text{LT}(q_1), \ldots, \text{LT}(q_t) \rangle \). By the properties of Gröbner bases, \( \text{LT}(r - r') \) must be divisible by some \( \text{LT}(q_1) \). However, by assumption, no term of \( r \) and \( r' \) is divisible by any \( \text{LT}(q_1) \). Thus, \( r - r' = 0 \). This shows \( r \) is unique.

**Definition 2.50.** Given a Gröbner basis \( G \subseteq \mathbb{K}[x_1, \ldots, x_n] \) for an ideal \( I \) and a polynomial \( p \in \mathbb{K}[x_1, \ldots, x_n] \), we use \( \text{red}_G(p) \in \mathbb{K}[x_1, \ldots, x_n] \) to denote the unique polynomial, \( r \), with (1) no term of \( r \) being divisible by any \( \text{LT}(q_1) \) for \( q_1 \in G \), and (2) \( p = h + r \) for some \( h \in I \). Phrases for \( \text{red}_G(p) \) are
• “red_G(p) is the remainder of p upon division by G”.

• “red_G(p) is the result of reducing p by G”.

• “p can be rewritten with respect to G as red_G(p)”.

**Corollary 2.51.** Let G ⊂ K[x_1, . . . , x_n] be a Gröbner basis for an ideal I. Let g ∈ K[x_1, . . . , x_n]. g ∈ I if and only if red_G(g) = 0.

**Constructing Gröbner Bases**

Cor. 2.51 shows how, given a Gröbner basis, we can solve the ideal-membership problem. The only thing left to show is to demonstrate that, given some f_1, . . . , f_s ⊂ K[x_1, . . . , x_n], we can compute a Gröbner basis for ⟨f_1, . . . , f_s⟩. Note, we’ll only briefly examine this topic, and give Buchberger’s algorithm. There are other more efficient algorithms to compute Gröbner bases that will not be explored here.

As noted earlier, the issue that prevents a set f_1, . . . , f_s from being a Gröbner basis is the possibility of there being two elements, f_i and f_j, for which there exist coefficient polynomials h_i and h_j such that LT(h_i f_i) = −LT(h_j f_j). If this happens then new leading terms can be introduced in the sum h_i f_i + h_j f_j. That is, it might be the case that LT(h_i f_i + h_j f_j) /∈ ⟨LT(f_1), . . . , LT(f_s)⟩. This actually turns out to be the only way a new smaller leading term can be introduced. Polynomials h_i f_i + h_j f_j that introduce a smaller leading term are s-polynomials.

**Definition 2.52.** Let f, g be polynomials in K[x_1, . . . , x_n]. Let cx^γ = LCM(LT(f), LT(g)). The s-polynomial of f and g denoted S(f, g) is defined as

\[ S(f, g) = \frac{cx^\gamma}{LT(f)} f - \frac{cx^\gamma}{LT(g)} g \]

**Example 2.53.** Let f = xy + 1 and g = y^2 − 1. The s-polynomial of f and g is

\[ S(f, g) = \frac{xy^2}{xy} (xy + 1) - \frac{xy^2}{y^2} (y^2 - 1) = y(xy + 1) - x(y^2 - 1) = xy^2 + y - xy^2 + x = x + y \]

The idea of s-polynomials is that they may introduce leading monomials that are smaller, in the monomial order, than the leading monomials of f and g. This observation is the key insight behind Buchberger’s algorithm. The idea is that we look at each unordered pair (f_i, f_j) from f_1, . . . , f_s and compute the s-polynomial.
Algorithm 2: Naive Buchberger’s algorithm

**Input:** $f_1, \ldots, f_s \in K[x_1, \ldots, x_n]$

**Result:** A Gröbner basis $G = q_1, \ldots, q_l \in K[x_1, \ldots, x_n]$ for $\langle f_1, \ldots, f_s \rangle$

1. $G \leftarrow \{f_1, \ldots, f_s\}$;
2. worklist $\leftarrow$ all unordered pairs of $\{f_1, \ldots, f_s\}$;
3. while worklist $\neq \emptyset$ do
   4. Remove $(g_i, g_j)$ from worklist;
   5. $s \leftarrow \text{remainder}(S(g_i, g_j), G)$;
   6. if $s \neq 0$ then
      7. worklist $\leftarrow$ worklist $\cup \{(g, s) \mid g \in G\}$;
   8. $G \leftarrow G \cup \{s\}$;

of $f_i$ and $f_j$, which is then reduced by a multivariate division. If we have a non-zero result, we have a new polynomial for our basis. We then add this new polynomial paired with previous ones to a worklist and repeat.

The key to the success of Buchberger’s algorithm is the use of a monomial order. This guarantees termination, because at every step of the algorithm $s$ is guaranteed to have an LM which is smaller than all the leading monomials in $G$. Because a monomial order is well-ordered, the leading monomials of the sequence of $s$’s must have a least element. Thus, the basis will be saturated and the algorithm will terminate.

Another way to think of Buchberger’s algorithm is that it is a generalization of Gaussian elimination. In Gaussian elimination, one repeatedly cancels a row’s leading term, which causes the leading term to be over a different variable—which, in the polynomial case, corresponds to a monomial of smaller order.

**Example 2.54.** Recall $\langle xy + 1, y^2 - 1 \rangle$ and use the lex monomial order. Buchberger’s algorithm would produce $\{xy + 1, y^2 - 1, x + y\}$.

Note that a Gröbner basis returned by Buchberger’s algorithm may not be a minimal basis. For example, consider Ex. 2.54. $\{xy + 1, y^2 - 1, x + y\}$ is a Gröbner basis for $\langle xy + 1, y^2 - 1 \rangle$. However, $\{y^2 - 1, x + y\}$ is also a Gröbner basis for $\langle xy + 1, y^2 - 1 \rangle$. To see why, note that $\text{LT}(xy + 1)$ is divisible by $\text{LT}(x + y)$, and thus can be safely removed.

**Definition 2.55.** A minimal Gröbner basis for an ideal $I$, is a Gröbner basis $G$ such that

1. The leading coefficient of each $p \in G$ is 1
2. For all $p \in G$, $\text{LT}(p) \notin \langle \text{LT}(G - \{p\}) \rangle$

**Example 2.56.** $\{y^2 - 1, x + y\}$ is a minimal Gröbner basis for $\langle xy + 1, y^2 - 1 \rangle$. 

Minimal Gröbner bases are not unique. The issue is that while a polynomial’s leading term is important, a \( p \in G \) could have multiple trailing terms that still make \( G \) a Gröbner basis. For most purposes any minimal basis will suffice.

### 2.2.3 Applications of Gröbner Bases

**Theorem 2.57.** Buchberger’s algorithm and the multivariate division algorithm give a decision procedure for the ideal-membership problem.

*Proof.* Given \( f_1, \ldots, f_s, g \subset K[x_1, \ldots, x_n] \). To determine if \( g \in \langle f_1, \ldots, f_s \rangle \), compute a Gröbner basis, \( G \), for \( \langle f_1, \ldots, f_s \rangle \) and divide \( g \) by \( G \). If the remainder upon division is 0, then \( g \in \langle f_1, \ldots, f_s \rangle \). Otherwise, \( g \not\in \langle f_1, \ldots, f_s \rangle \).

### Elimination

While Gröbner bases have a nice application to the ideal-membership problem, they can also be used for elimination.

**Example 2.58.** Consider the polynomial equations

\[
\begin{align*}
x^2 + y^2 + z^2 &= 1 \\
x^2 + z^2 &= y \\
x &= z
\end{align*}
\]

Without too much difficulty, elementary algebra could give us the solution to this system. We could substitute \( z \) in for \( x \) in the top two equations. Then the second equation would give us an expression for \( y \) in terms of \( z \). We could then substitute that expression for \( y \) in the top equation and get an equation only in terms of \( z \). If we could solve that equation, we could then use those solutions to determine solutions for \( x \) and \( y \).

An alternative way to express the solutions to the above system is by using the variety \( V(x^2 + y^2 + z^2 - 1, x^2 + z^2 - y, x - z) \). Using Thm. 2.25, we see that this variety is equal to the variety \( V(I) \) where \( I = \langle x^2 + y^2 + z^2 - 1, x^2 + z^2 - y, x - z \rangle \). Suppose that we compute a Gröbner basis for this ideal using a lex ordering with \( x \) lexicographically larger than \( y \) and \( y \).
lexicographically larger than \( z \). This would result in \( I = \langle g_1, g_2, g_3 \rangle \), where

\[
\begin{align*}
g_1 &= x - z \\
g_2 &= y - 2z^2 \\
g_3 &= z^4 + \frac{1}{2}z^2 - \frac{1}{4}
\end{align*}
\]

Note that the polynomial \( g_3 \) only involves the variable \( z \). As a consequence of Thm. 2.25, we have \( V(x^2 + y^2 + z^2 - 1, x^2 + z^2 - y, x - z) = V(x - z, y - 2z^2, z^4 + \frac{1}{2}z^2 - \frac{1}{4}) \). Thus, the solutions for \( z \) for the equation \( z^4 + \frac{1}{2}z^2 - \frac{1}{4} = 0 \) are the only possible values of \( z \) in the variety \( V(x^2 + y^2 + z^2 - 1, x^2 + z^2 - y, x - z) \). The values of \( x \) and \( y \) follow from those for \( z \).

**Definition 2.59.** Let \( Y \cup Z \) be a partition of a set of a variables \( X \), and let \( I \subseteq K[X] \) be an ideal. The elimination ideal, \( I_Y \subseteq K[Z] \) is defined as

\[
I_Y = I \cap K[Z].
\]

In words, \( I_Y \) consists of all the polynomials in \( I \) that use only the variables in the set \( Z \). The \( Y \) variables have been “projected out”.

For various applications, such as solving equations, we may want to focus attention on a particular elimination ideal. However, there is the question how does one find a basis for an elimination ideal. Fortunately, Gröbner bases directly answer this question.

**Theorem 2.60.** Let \( \gg_Y \) be an elimination order (Defn. 2.39) for a set of variables \( \{y_1, \ldots, y_l, z_{l+1}, \ldots, z_n\} = Y \cup Z \). For example \( \gg_Y \) could be a lex order with every \( Y \) variable lexicographically larger than any \( z \) variable. Let \( G \) be a Gröbner basis for an ideal \( I \subseteq K[Y, Z] \). Let

\[
G_Y = G \cap K[Z].
\]

Then \( G_Y \) is a Gröbner basis for the elimination ideal \( I_Y \).

**Proof.** Note that \( G_Y \subseteq I_Y \). Thus it suffices to show \( \langle \text{LT}(I_Y) \rangle = \langle \text{LT}(G_Y) \rangle \). To show this property, it is sufficient to show that for any \( f \in I_Y \), \( \text{LT}(f) \) is divisible by some \( \text{LT}(g) \) for \( g \in G_Y \). Note that because \( f \in I \) and \( G \) is a Gröbner basis for \( I \), then \( \text{LT}(f) \) must be divisible by some \( \text{LT}(g') \) for \( g' \in G \). What remains to show is \( g' \in G_Y \). Note that because \( f \in I_Y \), \( \text{LT}(f) \) must only contain \( Z \) variables. Therefore, because \( \text{LT}(g') \) divides \( \text{LT}(f) \), \( \text{LT}(g') \) must also only contain \( Z \) variables. Here is the crucial use of the monomial order. Because we are using an elimination order, \( g' \) cannot contain any monomial (leading or trailing) that has a variable in
the set \( Y \). Otherwise \( \text{LT}(g') \) would need to have a \( Y \) variable. This observation shows that \( g' \) only contains variables in \( Z \). Therefore, \( g' \in G_Y \).

\[ 40 \]

\[ \text{Example 2.61.} \] Recall the ideal \( I = \langle x^2 + y^2 + z^2 - 1, x^2 + z^2 - y, x - z \rangle \). We computed a Gröbner basis using the lex ordering, and obtained a basis with three polynomials \( G = \langle x - z, y - 2z^2, z^4 + \frac{1}{2}z^2 - \frac{1}{4} \rangle \). Thus, \( G_{(x,y)} = \langle z^4 + \frac{1}{2}z^2 - \frac{1}{4} \rangle \) is a Gröbner basis for the elimination ideal \( I_{(x,y)} = I \cap \mathbb{Q}[z] \). What this means is that not only is \( g_3 = z^4 + \frac{1}{2}z^2 - \frac{1}{4} \) a consequence of our original polynomials, but also that all other polynomials in \( I \) that involve only \( z \) are a polynomial multiple of \( g_3 \).

Similarly \( G_{(x)} = \langle y - 2z^2, z^4 + \frac{1}{2}z^2 - \frac{1}{4} \rangle \) is a Gröbner basis for \( I_{(x)} = I \cap \mathbb{Q}[y,z] \). Thus, all polynomials in \( I \) that involve only \( y \) and \( z \) are a polynomial combination of the two polynomials in \( G_{(x)} \).

If we wanted all consequences that only involve \( y \), we could recompute a Gröbner basis using an elimination order, \( \gg_{(x,z)} \), that puts \( y \) smallest—for example, a lex order with \( y \) lexicographically smaller than \( x \) and \( z \). The result would tell us that all polynomials in \( I \) that use just \( y \) are multiples of \( y^2 + y - 1 \).

An aside about elimination: one might hope that elimination would lead to a general method for solving polynomial equations. Intuitively, the idea would be to use Gröbner-basis techniques to eliminate variables from a set of equations, until we are left with an equation in one unknown. We would find a partial solution to this equation, and then substitute these solutions into our other equations to yield solutions to the entire system. In general, this idea does not always work. There are essentially two issues. First, partial solutions that allow 0 are problematic. Second, if our field is not algebraically closed, our partial solutions might not generalize. What follows are two examples demonstrating these issues.

\[ \text{Example 2.62.} \] Consider the system over some field \( \mathbb{K} \):

\[
\begin{align*}
xy &= 1 \\
xyz &= 1
\end{align*}
\]

Computing a Gröbner basis to eliminate \( x \) yields \( \langle y - z \rangle \). Unsurprisingly, the Gröbner basis indicates that a solution to the original equations must have \( y = z \). The set of points that satisfy this one equation is \( \langle a, a \rangle \) for any \( a \in \mathbb{K} \). Generalizing these points to include \( x \) we obtain the set of points that satisfy the above equations is \( \langle \frac{1}{a}, a, a \rangle \). However, note that if \( a = 0 \), we have a problem. That is, all assignments that have \( y = z \) generalize except the point \( (y = 0, z = 0) \). There are theorems that won’t be covered here that tell us that if \( \mathbb{K} \) is algebraically closed then the only time partial solutions don’t generalize is if there are zeros involved.
Example 2.63. Consider the system over \( \mathbb{R} \) or \( \mathbb{Q} \)
\[
\begin{align*}
x^2 &= y \\
x^2 &= z
\end{align*}
\]
If we eliminate \( x \), we would obtain \((y - z)\). However solutions where \( y \) and \( z \) are negative do not generalize in \( \mathbb{R} \) or \( \mathbb{Q} \). That is, there is no real solution for \( x^2 = a \) when \( a \) is negative. However, solutions where \( y \) and \( z \) are negative would generalize over \( \mathbb{C} \) because \( \mathbb{C} \) is algebraically closed.

An application of elimination techniques is that we can compute the inverse image of an ideal with respect to a polynomial homomorphism.

Definition 2.64. Let \( X \) and \( Y \) be finite set of variables, let \( P \subseteq \mathbb{K}[X] \) be a set of polynomials, and let \( f : \mathbb{K}[Y] \to \mathbb{K}[X] \) be a polynomial homomorphism. Then the inverse image \( f^{-1}(\langle P \rangle) \) of \( \langle P \rangle \) under \( f \) is an ideal of \( \mathbb{K}[Y] \).

Lemma 2.65. Let \( X \) and \( Y \) be finite sets of variables, let \( P \subseteq \mathbb{K}[X] \) be a set of polynomials, and let \( f : \mathbb{K}[Y] \to \mathbb{K}[X] \) be a polynomial homomorphism. Then we have
\[
\langle \text{inv.image}(f, P) \rangle = f^{-1}(\langle P \rangle).
\]

Proof. Suppose that \( G \) is a Gröbner basis for the ideal generated by \( P \cup \{y - f(y) : y \in Y\} \) with respect to an elimination order \( \ll_X \). Define \( \text{inv.image}(f, P) \) as \( G \cap \mathbb{K}[Y] \).

(O1) For each \( q \in \mathbb{K}[Y] \), we have \( q - f(q) \in \langle G \rangle \) (by induction on \( q \))

(O2) \( \langle G \rangle \cap \mathbb{K}[X] = \langle P \rangle \): Without loss of generality, we may suppose that \( P \) is a Gröbner basis for \( \langle P \rangle \) w.r.t. \( \ll_Y \). Then \( P \cup \{y - f(y) : y \in Y\} \) is also Gröbner basis w.r.t. \( \ll_Y \), and so \( \langle G \rangle \cap \mathbb{K}[X] = \langle G \cap \mathbb{K}[X] \rangle = \langle P \rangle \) by the key property of elimination orderings.

We show \( \langle \text{inv.image}(f, P) \rangle = f^{-1}(\langle P \rangle) \) by proving inclusion in both directions.

\( \subseteq \) It is sufficient to show that for all \( q \in \langle \text{inv.image}(f, P) \rangle \), we have \( f(q) \in \langle P \rangle \). Suppose \( q \in \langle \text{inv.image}(f, P) \rangle \).

Then \( q \in \langle G \rangle \) and \( q \in \mathbb{K}[Y] \). Since \( q \in \mathbb{K}[Y] \), we have \( q - f(q) \in \langle G \rangle \) by observation (O1). Since
2.2.4 Reduction

Let \( t, p_1, \ldots, p_m \in \mathbb{K}[x_1, \ldots, x_n] \) be polynomials over some field \( \mathbb{K} \). Let \( \gg \) be some monomial order. Consider the problem of finding some \( r \) with \( \text{LT}(t) \gg \text{LT}(r) \) s.t.

\[
\bigwedge_{i=1}^{m} p_i = 0 \quad \Rightarrow \quad t = r.
\]

Let \( V(P) = \{(c_1, \ldots, c_n) \in \mathbb{K}^n \mid p(c_1, \ldots, c_n) = 0 \ \forall p \in P\} \). Thus, the above problem could be stated as finding some \( r \) with \( \text{LT}(t) \gg \text{LT}(r) \) s.t.

\[
\forall x \in V([p_1, \ldots, p_m]). \ t(x) = r(x)
\]

From Thm. 2.25 we have \( V(P) = V(G) \). Consider rewriting \( t \) as the sum of some polynomial \( f \in \langle p_1, \ldots, p_m \rangle \) and some remainder, i.e. \( t = f + r \), with \( f \in \langle p_1, \ldots, p_m \rangle \) and \( r \in \mathbb{K}[x_1, \ldots, x_n] \). Because \( f \in \langle p_1, \ldots, p_m \rangle \), \( f(x) = 0 \) for each \( x \in V(p_1, \ldots, p_m) = V(G) \). Therefore,

\[
\forall x \in V([p_1, \ldots, p_m]). \ t(x) = r(x)
\]

We can (and do) use a Gröbner basis and the multivariate division algorithm to find such an \( f \in \langle p_1, \ldots, p_m \rangle \) and \( r \in \mathbb{K}[x_1, \ldots, x_n] \) where \( t = f + r \). Moreover, by finding a Gröbner basis using the desired monomial order \( \gg \), we have an optimality property of \( r \) in the following sense:
Theorem 2.67. Let $G = \{g_1, \ldots, g_s\}$ be a Gröbner basis for an ideal $I$ with respect to a monomial order $\gg$. Let $t$ be a polynomial, and $r = \text{red}_G(t)$. $r$ is the optimal remainder in the monomial order. That is,

1. $\text{LT}(t) \gg \text{LT}(r)$.

2. For any other $r'$ with $t = g' + r'$ for some $g' \in I$, $\text{LT}(r') \gg \text{LT}(r)$.

Proof. First, by definition $t = g + \text{red}_G(t)$ for some $g \in I$. Thus $\text{LT}(t) \gg \text{LT}(r)$.

Let $r'$ be some polynomial with $t = g' + r'$. Reducing $r'$ by $G$ yields $r' = g'' + r''$ for some $g'' \in I$, as well as $\text{LT}(r') \gg \text{LT}(r'')$. Moreover, $r''$ satisfies the properties of Thm. 2.49 with respect to $r'$. However, $t = g' + g'' + r''$, and because $g', g'' \in I$ so is $g' + g'' \in I$. Thus, $r''$ also satisfies the conditions of Thm. 2.49 with respect to $t$. Thus, $r = r''$ and $\text{LT}(r') \gg \text{LT}(r'') = \text{LT}(r)$.

That is, the Gröbner basis gives us a minimality property on $r$. Note, however, that this minimality does not necessarily translate to the original problem. In a sense, $r$ is optimal with respect to the ideal (and $\gg$), but not with respect to the variety. That is it is possible to have

$$t = g + r \text{ with } g(x) = 0 \forall(x) \in V([p_1, \ldots, p_m]), \text{ but } g \not\in \langle p_1, \ldots, p_m \rangle.$$ 

Example 2.68. Recall Ex. 2.29. There we had the polynomials $x^2, y^2$. Clearly, $V(x^2, y^2)$ is just the point $(0, 0)$. If we reduce $t = x + y$ by $(x^2, y^2)$ using any monomial order we get $r = x + y$. However, $g = x + y$ is 0 for every point in $V(x^2, y^2)$. Thus, $t = g + r'$ with $r' = 0$ is optimal with respect to the variety, but $r = x + y$ is optimal with respect to the ideal $(x^2, y^2)$.

Nevertheless, the method is sound. Furthermore, it allows us to think about reducing our original polynomial not by a finite set of given polynomials $p_1, \ldots, p_m$, but by an infinite set of consequences, i.e., the ideal $\langle p_1, \ldots, p_m \rangle$.

2.2.5 Ideal Operations

Lemma 2.69. Let $A$ and $B$ be ideals of some ring $R$. Then $A + B \overset{df}{=} \{a + b \mid a \in A, b \in B\}$ and $A \cap B$ are also ideals of $R$.

Proof. Clearly, $0 \in A + B$. Suppose $f, g \in A + B$. Then $f = a_1 + b_1$ and $g = a_2 + b_2$ for $a_1, a_2 \in A$ and $b_1, b_2 \in B$. Because $A$ and $B$ are ideals $a_1 + a_2 \in A$ and $b_1 + b_2 \in B$. Therefore, $f + g = a_1 + a_2 + b_1 + b_2 \in A + B$. Finally, suppose $f \in A + B$. Then $f = a + b$ for some $a \in A$ and $b \in B$. Let $h \in R$. Then because $A$ and $B$ are ideals $ha \in A$ and $hb \in B$. Thus, $hf = h(a + b) = ha + hb \in A + B$. This shows $A + B$ is an ideal of $R$. 
Clearly, \(0 \in A \cap B\). Suppose \(f, g \in A \cap B\). Then \(f, g \in A\) and \(f, g \in B\). Because \(A\) and \(B\) are ideals \(f + g \in A\) and \(f + g \in B\). Therefore, \(f + g \in A \cap B\). Finally, suppose \(f \in A \cap B\). Then for any \(h \in R\), \(hf \in A\) and \(hf \in B\). Thus, \(hf \in A \cap B\). This shows \(A \cap B\) is an ideal of \(R\). \(\square\)

Given a representation for ideals \(A\) and \(B\) it is easy to come up with a representation for \(A + B\).

**Lemma 2.70.** Let \(A = (a_1, \ldots, a_m)\) and \(B = (b_1, \ldots, b_n)\) be ideals of some ring \(R\). Then \(A + B = (a_1, \ldots, a_m, b_1, \ldots, b_n)\).

*Proof.* First we show \(A + B \subseteq (a_1, \ldots, a_m, b_1, \ldots, b_n)\). Let \(f \in A + B\). Then \(f = a + b\) for some \(a \in A\) and \(b \in B\). Therefore, \(a = g_1 a_1 + \cdots + g_m a_m\) for some \(g_1, \ldots, g_m \in R\). Similarly \(b = h_1 b_1 + \cdots + h_n b_n\) for some \(h_1, \ldots, h_n \in R\). Therefore, \(a + b = g_1 a_1 + \cdots + g_m a_m + h_1 b_1 + \cdots + h_n b_n \in (a_1, \ldots, a_m, b_1, \ldots, b_n)\).

Now we show \((a_1, \ldots, a_m, b_1, \ldots, b_n) \subseteq A + B\). Let \(f \in (a_1, \ldots, a_m, b_1, \ldots, b_n)\). Then \(f = g_1 a_1 + \cdots + g_m a_m + h_1 b_1 + \cdots + h_n b_n\) for \(g_1, \ldots, g_m, h_1, \ldots, h_n \in R\). \(g_1 a_1 + \cdots + g_m a_m \in A\) and \(h_1 b_1 + \cdots + h_n b_n \in B\). Therefore, \(f \in A + B\). \(\square\)

However, coming up with a representation for the intersection of two ideals is more challenging. In the case of polynomial ideals we can use Gröbner basis techniques to compute a representation for \(A \cap B\) assuming representations for \(A\) and \(B\). We will not formally prove the following lemma, but instead provide a sketch of the proof.

**Lemma 2.71.** Let \(A = (a_1, \ldots, a_m) \subseteq \mathbb{K}[X]\) and \(B = (b_1, \ldots, b_n) \subseteq \mathbb{K}[X]\). Let \(t\) be a fresh variable not in the set \(X\).

\[
A \cap B = (\langle ta_1, \ldots, ta_m \rangle + \langle (1-t)b_1, \ldots, (1-t)b_n \rangle) \cap \mathbb{K}[X].
\]

*Sketch.* First, let \(f \in A \cap B\). Then \(f \in \mathbb{K}[X]\), \(f \in A\), and \(f \in B\). Because \(f \in A\), \(tf \in (ta_1, \ldots, ta_m)\). Similarly, \((1-t)f \in \langle (1-t)b_1, \ldots, (1-t)b_n \rangle\). Therefore, \(f = tf + (1-t)f \in (\langle ta_1, \ldots, ta_m \rangle + \langle (1-t)b_1, \ldots, (1-t)b_n \rangle) \cap \mathbb{K}[X]\).

Now, let \(f \in (\langle ta_1, \ldots, ta_m \rangle + \langle (1-t)b_1, \ldots, (1-t)b_n \rangle) \cap \mathbb{K}[X]\). Then \(f = g(x, t) + h(x, t)\), where \(g(x, t) \in \langle ta_1, \ldots, ta_m \rangle \subseteq \mathbb{K}[X, t]\), and \(h(x, t) \in \langle (1-t)b_1, \ldots, (1-t)b_n \rangle \subseteq \mathbb{K}[X, t]\). Now the idea is that we can substitute some value in \(\mathbb{K}\) for \(t\) and get a particular ideal of \(\mathbb{K}[X]\). For example, if \(t = 1\), then \(h(x, 1) = 0\) and so \(f = g(x, 1)\). Moreover, it turns out that \(g(x, 1) \in \langle (1)a_1, \ldots, (1)a_m \rangle = A\). Thus, \(f \in A\). Similarly, if \(t = 0\), then \(g(x, 0) = 0\) and \(f = h(x, 0) \in \langle (1-0)b_1, \ldots, (1-0)b_n \rangle = B\). Thus, \(f \in A \cap B\). \(\square\)

The above lemma gives a recipe in order to compute the intersection of two polynomial ideals. We form the ideal \(tA + (1-t)B \subseteq \mathbb{K}[X, t]\). Then we use Gröbner basis elimination to eliminate the variable \(t\). The result will be a Gröbner basis for the ideal \(A \cap B\).
Chapter 3

Non-Linear Invariant Generation

Automatic generation of non-linear loop invariants is a long-standing challenge in program analysis, with many applications. For instance, reasoning about exponentials provides a way to find invariants of digital-filter programs, and reasoning about polynomials and/or logarithms is needed for establishing invariants that describe the time or memory usage of many well-known algorithms. An appealing approach to this challenge is to exploit the powerful recurrence-solving techniques that have been developed in the field of computer algebra, which can compute exact characterizations of non-linear repetitive behavior. However, there is a gap between the capabilities of recurrence solvers and the needs of program analysis: (1) loop bodies are not merely systems of recurrence relations—they may contain conditional branches, nested loops, non-deterministic assignments, etc., and (2) a client program analyzer must be able to reason about the closed-form solutions produced by a recurrence solver (e.g., to prove assertions).

This chapter presents a method for generating non-linear invariants of general loops based on analyzing recurrence relations. The key components are an abstract domain for reasoning about non-linear arithmetic, a semantics-based method for extracting recurrence relations from loop bodies, and a recurrence solver that avoids closed forms that involve complex or irrational numbers. Our technique has been implemented in a program analyzer that can analyze general loops and mutually recursive procedures. Our experiments show that our technique shows promise for non-linear assertion-checking and resource-bound generation.

3.1 Introduction

Recurrence equations have a long history, and variety of techniques for solving them are known. A natural question is to ask how these techniques can be put to work for invariant generation.

One line of work in this direction focuses on computing very accurate information about a syntactically restricted class of loops. For example, Rodríguez-Carbonell and Kapur [116] and Kovács [94] use recurrence
solving to compute all invariant polynomial equations of two (different) classes of loops. Neither technique can compute any invariants for loops that have (for example) nondeterministic assignments or nested loops.

Compositional recurrence analysis (CRA) [49] is another line of work, which focuses on over-approximate analysis of general loops rather than precise analysis of syntactically restricted loops. Recent work [85] extends the generality of CRA even further, showing how the approach can be applied to recursive procedures as well as loops. The key idea that makes CRA so broadly applicable is that it represents loop behavior using logical formulas, and uses semantics-based techniques to find implied recurrence relations. However, CRA's ability to reason about non-linear behavior is limited by the fact that it uses SMT solving, linear algebra, and polyhedral techniques to extract recurrences from loop bodies, and polynomial summation to solve them. In particular, CRA is only capable of extracting recurrence relations that can be expressed in linear integer arithmetic and that have polynomial closed forms—effectively exploiting only a fraction of what recurrence solvers (e.g., the ones used in [116, 94]) are capable of.

Extensions to the numerical-reasoning techniques underlying CRA are presented in this chapter. We demonstrate that these extensions enable CRA to establish many non-linear numerical invariants. The contributions of this chapter are three-fold:

- We present the wedge abstract domain, a numerical abstract domain capable of reasoning about non-linear arithmetic. Just as convex polyhedra represent properties in the conjunctive fragment of linear arithmetic, wedges represent properties in the conjunctive fragment of non-linear arithmetic (including polynomials, exponentials, and logarithms). The deductive power of wedges is due to polyhedral and Gröbner-basis techniques, congruence closure, and simple inference rules for non-linear functions. The key operation supported by the domain is symbolic abstraction [110, 134], which, given an arbitrary non-linear formula $\phi$, computes a wedge that over-approximates $\phi$. (See § 3.4.)

- We present a semantics-based algorithm for extracting recurrence relations that are entailed by a loop-body formula. The algorithm is based on first over-approximating the loop body by a wedge, and then using techniques from linear algebra to extract recurrences from the wedge. The algorithm can extract recurrences involving non-linear arithmetic and inter-dependent program variables; the class of recurrences that can be extracted by this algorithm corresponds to C-finite sequences [81, §4.2]. (See § 3.5.)

- We present an algorithm, OCRS, that is able to solve these recurrences, and find closed-form solutions that include polynomials, exponentials, and logarithms. OCRS is based on an automated and enhanced form of the discrete operational calculus of Berg [15]. Classically, the closed forms of C-finite sequences
involves algebraic irrational or algebraic complex numbers, but OCRS avoids non-rational numbers by using what we call *implicitly interpreted functions*. Each implicitly interpreted function is associated with a term in the logic of OCRS that exactly characterizes the function, but outside of the recurrence solver (and in particular, within the wedge domain) an implicitly interpreted function is treated as an uninterpreted function symbol. (See §3.6.)

Our approach builds upon the recent work of Kincaid et al. [85], which extended CRA so that it can analyze recursive programs using essentially the same approach that it uses to handle loops.

**Organization.** §3.2 illustrates the main features of our method via a series of examples. §3.3 presents relevant background material. §3.4 presents the wedge abstract domain. §3.5 describes the method used in our system for extracting a recurrence relation from a wedge. §3.6 presents OCRS. §3.7 presents experimental results. §3.8 discusses related work.

### 3.2 Overview and Problem Statement

Our system follows in the tradition of CRA [49] and ICRA [85] in how it combines symbolic analysis and abstract interpretation:

- It uses an abstract domain of transition formulas, and thus models non-looping program behavior precisely.

- It conservatively explores all behaviors of a loop by over-approximating the transitive closure of the loop body: a formula for the loop body is converted into a system of recurrences, which are then solved to create a transition formula that summarizes the overall action of the loop.

For brevity, we refer to the actions performed to analyze a loop as the *star operator*.

This section illustrates at a high level the main features of the star operator, focusing on how the system reasons about non-linear relationships, such as polynomials, exponentials, and logarithms. We present two examples of running-time analysis, along with two other examples, one analyzing a rotation, and one demonstrating information-flow analysis.

**Example 3.1.** Consider the loop nest shown in Fig. 3.1(a), which runs in cubic time. CRA analyzes the loop nest “bottom-up”, starting from the most deeply nested loop and moving outwards, applying the star operator at each nesting level. Nested loops lead to a design constraint in the program analyzer: the star operator must be able to reason about its own output.

---

1 An algebraic number is a complex number that is a root of a non-zero univariate polynomial with rational coefficients. Henceforth, we shorten “algebraic irrational” and “algebraic complex” to “irrational” and “complex,” respectively.
```c
int ticks = 0;
for (int a = 0; a < N; a++)
    for (int b = 0; b < N; b++)
        for (int c = 0; c < N; c++)
            ticks++;

int pos = 1, steps = 0, found = 0;
while (pos < array_size) {
    steps += 1;
    if (array[pos] == val) {
        found = 1;
        break;
    }
    if (array[pos] < val) {
        pos = 2 * pos;
    } else {
        pos = 2 * pos + 1;
    }
}
if (y == 1) goto errorlabel;

int temp, x = 1, y = 0;
for (int n = 0; n < 1; n++) {
    temp = x; x = y; y = -temp;
}
if (y == 1) goto errorlabel;
```

```c
int fib(int n, int high) {
    int f1 = 1, f2 = 0, temp = 0;
    if (high) {
        while (n > 0) {
            f1 = f1 + f2;
            f2 = f1 - f2;
            n--;
        }
    } else {
        while (n > 0) {
            temp = f2;
            f2 = f1;
            f1 = f2 + temp;
            n--;
        }
    }
    return f1;
}

void main() {
    int n1, n2, obs1, obs2, high1, high2;
    assume(n1 == n2);
    // Note: high1 might not equal high2
    obs1 = fib(n1, high1);
    obs2 = fib(n2, high2);
    assert(obs1 == obs2);
}
```

Figure 3.1: Four examples: (a) cubic-time loop nest; (b) binary search; (c) rotation in the x, y plane; (d) absence of information flow.

Fig. 3.2 shows the recurrences created for the program in Fig. 3.1(a). We use, e.g., \(c[k]\) to denote the value of variable \(c\) at the beginning of iteration \(k\) of the innermost loop (at nesting-level 3). As shown in Fig. 3.2, the body of the innermost loop can be described by a linear recurrence, and the solution is also linear. Using the solution to that recurrence, along with the fact that the inner loop iterates \(N\) times, we obtain a linear recurrence for the loop at nesting-level 2. However, the solution to this recurrence is non-linear: it involves the term \(N \cdot k\). Consequently, the body of the outer loop (at nesting-level 1) has the following non-linear transition formula:

\[
\exists k. \text{ticks}' = \text{ticks} + N \cdot k \land k \leq N \land k \geq N \land a' = a + 1,
\]

which cannot be analyzed directly using many of the classical tools of program analysis, such as Satisfiability Modulo Theories (SMT) solvers and the abstract domain of polyhedra. Thus, although each iteration of the outermost loop increases the value of \(\text{ticks}\) by \(N \cdot N\), the recurrence-extraction technique of Farzan and Kincaid [49] is unable to find the relevant recurrence. This chapter presents an extension of CRA that is able to handle polynomial recurrences such as this one, and is therefore able to prove that the above triply nested loop increases the value of \(\text{ticks}\) by exactly \(N \cdot N \cdot N\).
<table>
<thead>
<tr>
<th>Nesting Level</th>
<th>Recurrence</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>( c_{k+1} = c_k + 1 \land \text{ticks}_{k+1} = \text{ticks}_k + 1 )</td>
<td>( c_k = c_0 + k \land \text{ticks}_k = \text{ticks}_0 + k )</td>
</tr>
<tr>
<td>2</td>
<td>( b_{k+1} = b_k + 1 \land \text{ticks}_{k+1} = \text{ticks}_k + N )</td>
<td>( b_k = b_0 + k \land \text{ticks}_k = \text{ticks}_0 + N* k )</td>
</tr>
<tr>
<td>1</td>
<td>( a_{k+1} = a_k + 1 \land \text{ticks}_{k+1} = \text{ticks}_k + N + N )</td>
<td>( a_k = a_0 + k \land \text{ticks}_k = \text{ticks}_0 + N* k )</td>
</tr>
</tbody>
</table>

Figure 3.2: Recurrences created for the nested loops in Fig. 3.1(a).

To obtain tight bounds on the running time of other programs, logarithms are required. Moreover, loop bodies can contain branching code—and thus different paths through the loop body may be taken on different iterations.

**Example 3.2.** Consider the loop shown in Fig. 3.1(b), which performs a binary search over a complete binary tree that is stored in an array. Our analysis establishes that the running time of this loop is logarithmic in \( \text{array}\_\text{size} \) by first constructing a recurrence saying that, on each iteration, \( \text{steps} \) is incremented by 1 and \( \text{pos} \) (at least) doubles:

\[
\text{steps}_{k+1} = \text{steps}_k + 1 \land \text{pos}_{k+1} \geq 2 \times \text{pos}_k.
\]

Extraction is a nontrivial operation because the inner loop contains branching code. Moreover, the exact behavior of the variable \( \text{pos} \) cannot be described by a recurrence equation. Instead, we must approximate the behavior of \( \text{pos} \) by a recurrence inequality.

Next, the recurrence solver finds the solution to this recurrence:

\[
\text{steps}_k = \text{steps}_0 + k \land \text{pos}_k \geq 2^k \times \text{pos}_0,
\]

and this result, in combination with the initial conditions of the loop, establishes that whenever control reaches the head of the loop, we have \( \text{pos} \geq 2^{\text{steps}} \). The solver also establishes that, upon exit from the loop, \( \text{pos} < 2^{\text{array}\_\text{size}} \), because otherwise the loop would have exited earlier. We conclude that \( \text{steps} < 1 + \log_2(\text{array}\_\text{size}) \), if \( \text{array}\_\text{size} > 0 \), and \( \text{steps} = 0 \) otherwise.

This example shows that the analysis of a logarithmic-time algorithm sometimes requires finding exponential solutions to recurrences.

Some programs have non-linear relationships that cannot be expressed using polynomials, exponentials, and logarithms of integers.

**Example 3.3.** Consider the loop shown in Fig. 3.1(c). One iteration of the loop performs a 90-degree clockwise rotation of the point \((x, y)\) about the origin. Because of the mutual dependence between \( x \) and \( y \), we extract a matrix recurrence for this loop. Below, we show the matrix recurrence (on the left) and the solution to that recurrence that would be
obtained by a typical off-the-shelf recurrence solver (on the right, with \(i\) being the imaginary unit).

\[
\begin{bmatrix}
    x^{[k+1]} \\
    y^{[k+1]}
\end{bmatrix} =
\begin{bmatrix}
    0 & 1 \\
    -1 & 0
\end{bmatrix}
\begin{bmatrix}
    x^{[k]} \\
    y^{[k]}
\end{bmatrix}
= \begin{bmatrix}
    \frac{1}{2}((-i)^k + i^k) & \frac{1}{2}((-i)^k - i^k) \\
    -\frac{1}{2}((-i)^k - i^k) & \frac{1}{2}((-i)^k + i^k)
\end{bmatrix}
\begin{bmatrix}
    x^{[0]} \\
    y^{[0]}
\end{bmatrix}
\]

Complex exponentiation concisely represents the rotation performed by this loop, but the use of complex numbers creates a problem for the later steps of our analysis, as demonstrated below.

Suppose that we want to know whether errorlabel is reachable. We answer that question by using an SMT solver to check the satisfiability of the transition formula from program entry to errorlabel. Upon entry to the loop, \(n\) is initialized to 0 and the loop condition is “\(n < 1\)”, so the loop will always execute exactly once. Thus, the value of \(y\) at the end of the loop is \(y^{[1]} = -\frac{1}{2}((-i)^1 - i^1)x^{[0]} + \frac{1}{2}((-i)^1 - i^1)y^{[0]} = -i * i\), and the relevant part of the formula for \(y\) is:

\[y' = 1 \land y' = -i * i.\] (3.1)

If \(i\) is interpreted as the imaginary unit, then Eqn. (3.1) is satisfiable, which gives us the answer we expect: errorlabel is reachable. However, we would like to be able to use an off-the-shelf SMT solver that does not support complex numbers. Unfortunately, it is not sound to interpret Eqn. (3.1) in real arithmetic with \(i\) as a symbolic constant. In real arithmetic, we know that for all \(a\), \(a * a \geq 0\), and thus \(y' = -i * i \leq 0\), which is inconsistent with \(y' = 1\); thus, Eqn. (3.1) is unsatisfiable, which falsely suggests that errorlabel is unreachable.

To avoid problems of this kind, we developed a recurrence solver (described in § 3.6) that is able to solve recurrences like the one in this example, and communicate the solutions to the rest of our analyzer without the use of complex numbers. As a principled way to handle the issue of communication, this chapter introduces implicitly interpreted functions (IIFs) (§ 3.6.4). An IIF is a representation of the solution to a recurrence that would otherwise need to be expressed using complex or irrational numbers. Inside the recurrence solver, each IIF is associated with a term in the logic of the recurrence solver, which represents an exact representation of the function (and the recurrence solver is able to manipulate this exact representation). Outside the recurrence solver—e.g., in the wedge domain—an IIF is treated as an uninterpreted function.

The following example illustrates that IIFs retain enough information to prove some sophisticated program properties.

Example 3.4. Consider the program shown in Fig. 3.1(d), which illustrates how our analyzer is able to establish the absence of information flow (i.e., “non-interference”), using the self-composition technique of Barthe et al. [14]. For procedure fib, we will assume that variable \(n\) is a low-security input, high is a high-security input, and the return
value of \texttt{fib} is a low-security output. The information-flow property that we wish to establish is that \texttt{fib}'s return value is unaffected by the value passed in for \texttt{high}.

Secure information flow is not a safety property:
\footnote{Technically, we are referring to the so-called “termination-insensitive” secure-information-flow problem.} a safety property can be refuted by observing a finite trace of a program, whereas to refute secure information flow, one has to observe two finite traces. Barthe et al. show that secure information flow for a program \texttt{P} can be encoded as a safety property of a more complicated program that consists of \texttt{P} followed by \texttt{P'}—a second copy of \texttt{P} with its variables suitably renamed. By this means, secure information flow can be reduced to an assertion-checking problem in the program \texttt{P;P'}. They call this technique self composition.

In the program above, the self-composition technique is captured by procedure \texttt{main}, which uses two calls to \texttt{fib} rather than two copies of \texttt{fib}. In \texttt{main}, the statement \texttt{assume(n1 == n2)} ensures that the low-security inputs to the two calls on \texttt{fib} are arbitrary, but equal. The absence of any constraint on \texttt{high1} and \texttt{high2} means that they may have different values in the two calls to \texttt{fib}. The goal is to prove the assertion on the last line of \texttt{main}, which claims that the low-security outputs \texttt{obs1} and \texttt{obs2} are equal, and hence unaffected by any difference in the values of high-security inputs \texttt{high1} and \texttt{high2}.

To convince yourself that the assertion does indeed hold, observe that the two while loops in \texttt{fib} both compute the \texttt{n}\textsuperscript{th} Fibonacci number, albeit in slightly different ways. Thus, because the values of \texttt{n1} and \texttt{n2} are equal, \texttt{obs1} and \texttt{obs2} will be equal—even though \texttt{high1} and \texttt{high2} might differ, which would cause different while loops to be executed during the two calls on \texttt{fib}.

The challenge in analyzing this example is that the return value of \texttt{fib} is a complicated function of its inputs. Classically, this function would be represented using irrational numbers: the return value of \texttt{fib(n, high)} is \(\frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n)\). As in the previous example (Ex. 3.3), we avoid manipulation of non-rational numbers using IIFs.

In the program shown above, the analyzer summarizes the behaviors of the two while loops in \texttt{fib} using two IIFs. The two IIFs are associated with equal terms in the logic of the recurrence solver. Consequently, the wedge domain is able to prove the assertion because it sees it as a comparison of two identical uninterpreted functions.

**Problem Statement.** The foregoing examples illustrate several challenges that our work addresses:

1. The star operator takes as input a logical representation of the body of loop. The loop body formula typically contains disjunctions, existentially quantified variables, and inequations, and so cannot be directly interpreted as a recurrence relation. Therefore, we need an automatic method for extracting recurrences that lie within the input language of our recurrence solver. In general, these recurrences may include non-linear relationships.
2. Once we have a recurrence relation, we need to solve it. In general, an extracted recurrence may contain polynomial and exponential terms, and program variables may mutually depend on one another. We need to be able to compute a closed form that can be represented in the base logic of the program analyzer (see § 3.6.3).

The analysis method presented in this chapter addresses these challenges by the following means:

- Challenge (1) is addressed via a new abstract domain, called the wedge domain (§ 3.4), which encodes non-linear relationships by extending the abstract domain of polyhedra with new dimensions that represent non-linear terms. § 3.5 presents an algorithm that uses the wedge abstract domain to extract recurrence relations from a loop body formula.

- Challenge (2) is addressed via a new recurrence solver that we developed, based on the discrete operational calculus of Berg [15]. (See § 3.6.)

The balancing act in our work is that there is a mutual dependence between the wedge domain and the recurrence solver: each places constraints on the other. In addressing challenges (1) and (2), we faced a situation that often comes up in system building, namely, it is not always possible to use an existing component as just a black box. Although methods for solving recurrences are known (see § 3.8 for references and a discussion), the closed forms computed by these methods may involve irrational and complex numbers, which are not permitted in the base logic of our program analyzer. The desire to have IIFs for reasoning outside the recurrence solver required us to have some control over what goes on inside the recurrence solver, to identify—and return as part of the answer—terms that define IIFs.

### 3.3 Additional Chapter Background

#### 3.3.1 Linear Real Arithmetic

Fix a set of variables $z_1, ..., z_n$. A **linear term** is an expression of the form $a_1z_1 + \cdots + a_nz_n + b$, where each $a_i \in \mathbb{Q}$ and $b \in \mathbb{Q}$. We will typically write such a term as $az + b$, where $a = [a_1, ..., a_n]$ and $z = [z_1, ..., z_n]^t$ are vectors containing the term’s rational coefficients and variables, respectively. A **linear inequality** is of the form $az \geq b$. A system of linear inequalities $a_1z \geq b_1 \land \cdots \land a_mz \geq b_m$ can be written succinctly as $Az \geq b$, where $A$ is a matrix whose rows are $a_1, ..., a_m$, and $b = [b_1, ..., b_m]^t$. The set of points $z$ such that $Az \geq b$ is a (convex) polyhedron.
3.3.2 (Non)linear Real Arithmetic with Uninterpreted Functions

Let $\Sigma = \langle F, ar \rangle$ be a functional first-order vocabulary, consisting of a set of function symbols $F$ and a function $ar : F \to \mathbb{N}$ associating each function symbol with an arity. We define the syntax of a logical language that extends linear real arithmetic with the vocabulary $\Sigma$ as follows:

$$s, t \in \text{Term}(\Sigma) ::= a \in \mathbb{Q} \mid v \in \text{Var} \mid a \times t \mid s + t \mid f(t_1, \ldots, t_{ar(f)})$$

$$\phi, \psi \in \text{Formula}(\Sigma) ::= t \geq 0 \mid t > 0 \mid \phi \land \psi \mid \phi \lor \psi \mid \exists v. \phi$$

A term of the form $a, s + t$, or $a \times t$ is called linear (with the rest called non-linear). We assume that $\Sigma$ includes a vocabulary $\Sigma_N$ of designated function symbols that represent multiplication, exponentiation, logarithm, modulus, and floor, which we will write using standard notation:

$$\text{Term}(\Sigma_N) ::= \ldots \mid s \times t \mid s^t \mid \log_s(t) \mid s \mod t \mid \lfloor t \rfloor$$

We will make use of two different semantics for this logic. A linear structure $\mathcal{L}$ consists of an interpretation of each function symbol $f \in \Sigma$ as a function $f^\mathcal{L} : \mathbb{R}^{ar(f)} \to \mathbb{R}$; the semantics of numerals, addition, and numeral multiplication are the usual ones. We write $\mathcal{L} \models_L \phi$ to denote that $\mathcal{L}$ is a linear structure that satisfies the sentence $\phi$; write $\mathcal{L} \models_L \psi$ if every linear structure that satisfies the sentence $\phi$ also satisfies the sentence $\psi$; and write $\mathcal{L} \equiv_L \psi$ if $\mathcal{L} \models_L \psi$ and $\psi \models_L \phi$. A non-linear structure $\mathcal{N}$ is a linear structure that interprets the designated function symbols in $\Sigma_N$ with their usual interpretation on their domain of definition (i.e., $\times^\mathcal{N}$ is multiplication, $\log_2^\mathcal{N}$ is base-2 logarithm for positive arguments and arbitrary for non-positive arguments, etc). We use $\models_N$ and $\equiv_N$ to denote the non-linear analogues of $\models_L$ and $\equiv_L$. Obviously, $\phi \models_L \psi$ implies that $\phi \models_N \psi$, but the reverse does not hold. Note that membership of a pair of formulas in the relation $\models_L$ is decidable, but $\models_N$ is not.

We are particularly interested in formulas that represent transition relations of some fragment of a program. Formally, we suppose that $\Sigma$ is a vocabulary that contains a designated set $\text{PVar} = \{x_1, \ldots, x_n\}$ of constant symbols and also a disjoint set $\text{PVar}' = \{x'_1, \ldots, x'_n\}$ of primed copies, representing the values of program variables before and after a transition. A formula $\phi$'s concretization $\gamma_N(\phi)$ is the transition relation defined by its non-linear models:

$$\gamma_N(\phi) \overset{\text{df}}{=} \{(a, a') : \exists \mathcal{N} : \mathcal{N} \models_N \phi, a_1 = x_1^\mathcal{N}, \ldots, a_n = x_n^\mathcal{N}\}.$$
3.3.3 Reasoning About Non-Linear Arithmetic

In this chapter, we use a variety of techniques to reason about non-linear arithmetic. The following is a road map that outlines the techniques used in each section. Within the wedge domain (§ 3.4), we use Gröbner bases and congruence closure to reason about non-linear equations, and inference rules to reason about non-linear inequations. In § 3.5, we use Gröbner bases to extract non-linear recurrence relations from the body of a loop (represented by a wedge). In § 3.6, we present a recurrence solver that can solve recurrences involving polynomials and exponentials, and which uses implicitly interpreted functions (IIFs) to represent closed forms for recurrences that would otherwise require exponentiation of irrational or complex numbers.

As demonstrated in Ex. 3.2, logarithmic relationships are obtained as derived information from exponential relationships (cf. Fig. 3.3).

3.4 The Wedge Abstract Domain

The wedge abstract domain is a numerical abstract domain that can express properties involving linear and non-linear arithmetic as well as uninterpreted function symbols. Fixing a vocabulary $\Sigma$, a $\Sigma$-wedge is a simply a conjunction of ground atomic $\Sigma$-formulas:

$$w \in \text{Wedge}(\Sigma) := t \geq 0 \lor t > 0 \lor w_1 \land w_2 \quad \text{where } t \in \text{Term}(\Sigma) \text{ and has no free variables (§ 3.3.2)}$$

Note that $t = 0$ can be expressed as $t \geq 0 \land -t \geq 0$. Semantically, we are typically interested in viewing a wedge as a transition relation—that is, the natural concretization function is $\gamma_N$. The concretization function $\gamma_N$ induces an approximation pre-order on wedges: $w \sqsubseteq_N w'$ iff $\gamma_N(w) \subseteq \gamma_N(w')$. However, this pre-order is not effective, necessitating the use of a linear variation: $w \sqsubseteq_L w'$ iff $w \models_L w'$. The wedge domain’s ability to reason about non-linear operations stems from strengthening operations (§ 3.4.1) that saturate a wedge with properties that hold in all of its non-linear models.

Operations for manipulating wedges are based on viewing a wedge as a set of points in a real space with one coordinate corresponding to each term. This view is formalized in the following.

**Definition 3.5.** A $\Sigma$-coordinate system is a list of $\Sigma$-terms $\tau = [t_1, ..., t_n]$ such that:

1. Each $t_i$ is a term of the form $f(t_1, ..., t_n)$ for some $f \in \Sigma$ with no free variables. (Note that this definition includes terms of the form $x$ for a constant symbol $x \in \Sigma$). Such terms are called application terms.

2. There are no duplicates.

3. For any $i$, for any application sub-term $s$ of $t_i$, we must have $s = t_j$ for some $j < i$. 


We say that \( \tau \) \textit{admits} a term \( t \) \textit{if} all of \( t \)'s application sub-terms belong to \( \tau \) \textit{(including \( t \) itself, if \( t \) is an application term)}. We say that \( \tau \) admits a wedge \textit{if} it admits all the terms of that wedge. We say that \( \tau \) is a \textit{minimal} coordinate system for a wedge \( w \) \textit{if} \( \tau \) admits \( w \) and no proper sub-list of \( \tau \) admits \( w \).

Given a coordinate system \( \tau = [t_1, \ldots, t_n] \), we define a \textit{linearization} function \( \text{lin}_\tau \) \textit{that} expresses admissible terms and wedges in linear arithmetic over the set of variables \( \{z_1, \ldots, z_n\} \), where variable \( z_i \) denotes the \( i \)th coordinate of \( \mathbb{R}^n \) \textit{(representing the term} \( t_i \)).

\[
\text{lin}_\tau(a) \overset{\text{def}}{=} a, \text{ if } a \in \mathbb{Q} \\
\text{lin}_\tau(t_i) \overset{\text{def}}{=} z_i \\
\text{lin}_\tau(u_1 + u_2) \overset{\text{def}}{=} \text{lin}_\tau(u_1) + \text{lin}_\tau(u_2) \\
\text{lin}_\tau(k \times u) \overset{\text{def}}{=} k \times \text{lin}_\tau(u), \text{ if } k \in \mathbb{Q}
\]

Note that for a wedge \( w \), \( \text{lin}_\tau(w) \) \text{ defines a convex polyhedron, which we call the} \textbf{underlying polyhedron} of \( w \).

We also define a reverse operation, \( \text{interp}_\tau \), which maps a linear term over variables \( \{z_1, \ldots, z_n\} \) \text{ into a term:}

\[
\text{interp}_\tau(a_1 z_1 + \cdots + a_n z_n + b) \overset{\text{def}}{=} a_1 t_1 + \cdots + a_n t_n + b.
\]

For any term \( t \) and any coordinate system \( \tau \) \text{ that} admits \( t \), \( \text{true} \models_L \text{interp}_\tau(\text{lin}_\tau(t)) = t \) \text{ holds.}

\textbf{Example 3.6.} A minimal coordinate system that admits the wedge \( w \overset{\text{def}}{=} 2x^2 - y = 0 \land y - 3x + y \geq 0 \) \text{ is} \( \tau = [x, y, x^2] \). The \textit{linearization} of \( w \) \text{ is the polyhedron} \( \text{lin}_\tau(w) = -z_2 + 2z_3 = 0 \land -3z_1 + 2z_2 \geq 0 \).

### 3.4.1 Strengthening

Let \( w \) \text{ be a wedge}. A \textbf{strengthening} \text{ of} \( w \) \text{ is any wedge} \( w' \) \text{ such that} \( w \equiv_N w' \) \text{ and} \( w' \models_L w \). For example, a strengthening of the wedge \( x = y + 1 \land y^2 \geq x^2 \) \text{ is} \( x = y + 1 \land x^2 \geq 0 \land x^2 = y^2 + 2y + 1 \land y \geq 0 \). In the terminology of abstract interpretation, strengthening a wedge amounts to applying a sequence of reductive operators, each of which over-approximates the lower closure operator induced by the Galois insertion of the powerset lattice of non-linear models into the powerset lattice of linear models [62]. This section presents techniques for strengthening wedges, presented in two parts: first, inferring implied equalities; second, inferring implied inequalities.
Equalities

Let \( \tau = [t_1, \ldots, t_n] \) be a coordinate system and let \( w \) be an admissible wedge of \( \tau \). Let \( P \) denote the underlying polyhedron of \( w \), and let \( \text{aff}(P) \) denote a basis for the vector space

\[
\left\{ \begin{bmatrix} b_0 & b_1 & \ldots & b_n \end{bmatrix} : \forall \begin{bmatrix} a_1 & \ldots & a_n \end{bmatrix} \in P, b_1 a_1 + \cdots + b_n a_n + b_0 = 0 \right\}.
\]

\( \text{aff}(P) \) is a representation of the set of affine equations that are satisfied by all points in \( P \); it can be computed easily from a constraint representation of \( P \). Define \( \mathbb{I}(w, \tau) \) to be the ideal generated by \( \text{aff}(P) \) and the definitional equalities encoding multiplication and reciprocal coordinates:

\[
\mathbb{I}(w, \tau) \equiv \left\langle \{ b_1 z_1 + \cdots + b_n z_n + b_0 : \begin{bmatrix} b_0 & b_1 & \ldots & b_n \end{bmatrix} \in \text{aff}(P) \} \right\rangle \\
\cup \{ z_i - \text{lin}_\tau(s) \text{lin}_\tau(s') : t_i = s \times s' \} \\
\cup \{ \text{lin}_\tau(s)z_i - 1 : t_i = s^{-1} \land w \models_L \neg(s = 0) \}
\]

For any ideal \( I \subseteq \mathbb{Q}[z_1, \ldots, z_n] \) and admissible terms \( s \) and \( t \), we write \( I \models s = t \) iff \( \text{lin}_\tau(s) - \text{lin}_\tau(t) \in I \). We say that \( I \) is congruence closed if for all terms \( t_i = f(s_1, \ldots, s_m) \) and \( t_j = f(s'_1, \ldots, s'_m) \) of \( \tau \) such that for all \( i \) we have \( I \models s_i = s'_i \), then we have \( I \models f(s_1, \ldots, s_m) = f(s'_1, \ldots, s'_m) \). The congruence closure of an ideal \( I \) is the smallest congruence-closed ideal that contains \( I \). We say that a wedge \( w \) is equationally saturated in \( \tau \) if for every degree-1 polynomial \( a_1 z_1 + \cdots + a_n z_n + b \) in the congruence closure of the ideal \( \mathbb{I}(w, \tau) \) vanishes on the underlying polyhedron of \( w \). The equational saturation of \( w \) is the unique (up to \( \equiv_L \)) equationally saturated wedge \( w' \) such that \( w \equiv_N w' \) and for all equationally saturated \( w'' \) such that \( w \equiv_N w'' \) we have \( w'' \models_L w' \). A procedure for computing the equational saturation of a wedge is given as Algorithm 3.

**Example 3.7.** Consider the wedge \( w = (x \times 2^z = v \land v \leq (2^y + 1) \times y \land x = y \land x \leq 0) \) under the coordinate system \( \tau = [v, x, y, 2^z, x \times 2^z, 2^y, (2^y + 1) \times y] \). The association between terms and coordinates (defined by \( \tau \)) is as follows:

\[
\begin{align*}
&z_1 : v & z_2 : x & z_3 : y & z_4 : 2^z & z_5 : x \times 2^z & z_6 : 2^y & z_7 : (2^y + 1) \times y
\end{align*}
\]

The underlying polyhedron \( P \) of \( w \) is \( z_5 = z_1 \land z_1 \leq z_7 \land z_2 = z_3 \land z_2 \leq 0 \), and the ideal \( \mathbb{I}(w, \tau) \) is:

\[
\mathbb{I}(w, \tau) \equiv \left\langle \begin{array}{c}
z_5 - z_1, z_2 - z_3, z_5 - z_2 z_4, z_7 - (z_6 + 1)z_3,
\end{array} \right\rangle.
\]

We will now illustrate Algorithm 3 on the wedge \( w \). First we compute a (deglex) Gröbner basis for \( \mathbb{I}(w, \tau) \):

\[
\mathbf{G} = (z_5 - z_1, z_3 - z_2, z_2 z_4 - z_1, z_2 z_6 - z_7 + z_2 z_1 z_6 - z_4 z_7 + z_2 z_4) \text{ and set the active coordinates to be } [4, 6], \text{ the ones corresponding to the terms } 2^z \text{ and } 2^y.
\]
First iteration of the outer loop: When processing coordinate 4 in the inner loop, we add the mapping $CC[2z_1] := z_4$ to $CC$, indicating that the representative for the term $2z_1$ is $z_4$. We then process coordinate 6, and find that the term $(\text{red}_G(2))^{\text{red}_C(z_3)} = 2z_2$ already has a representative ($z_4$), and add $z_6 - z_4$ to B (i.e., we conclude from the fact that $x = y$ that $2z = 2y$). We add the equation $2y - 2z = 0$ to $w$ and compute a Gröbner basis for the ideal generated by $G$ and $B$: $G = \{z_5 - z_1, z_3 - z_2, z_6 - z_4, z_2z_4 - z_1, z_7 - z_2 - z_1\}$. Since B contains a non-trivial polynomial, namely $z_6 - z_4$, we continue to a second iteration.

Second iteration of the outer loop: Since the two application terms $2z$ and $2y$ were identified in the previous iteration, the inner loop does not produce any new equations. However, the Gröbner basis $G$ changed in the previous iteration (in particular, it now contains $z_7 - z_2 - z_1$). Since $\text{red}_G(z_7) = z_2 + z_1$, we add the equation $(2y + 1) \times y = v + x$ to $w$ (line 15). Since $w$ contains the inequality $u \leq (2y + 1) \times y$ and (now) the equation $(2y + 1) \times y = v + x$, we have $v \leq u + x$ and thus $0 \leq z$; since $w$ also contains $x \leq 0$, we have $z = 0$. Thus, the underlying polyhedron of $w$ is $z_5 = z_1 \land z_1 \leq z_7 \land z_2 = z_5 \land z_2 = 0$, and $z_2$ belongs to $\mathbb{I}(w, \tau)$. We add $z_2$ to $B$ and compute a Gröbner basis for the ideal generated by $G$ and $B$: $G = \{z_1, z_2, z_3, z_7, z_6 - z_4\}$. Since $B$ contains a non-trivial polynomial ($z_2$), we continue to a third iteration.

Third iteration of the outer loop: Again the inner loop does not produce any new equations. On line 15 we derive the equations $v = 0$, $x = 0$, $y = 0$, $x \times 2z = 0$, $(2y + 1) \times y = 0$, due to the fact that $G$ reduces $z_1$, $z_2$, $z_3$, $z_5$, and $z_7$ to 0. Since $G$ reduces every basis polynomial in $\mathbb{I}(w, \tau)$ to 0, the loop exits and the algorithm returns the following (equationally saturated) wedge:

$$v = x = y = x \times 2z = (2y + 1) \times y = 0 \land 2z = 2y.$$

Inequalities

Let $\tau = [t_1, ..., t_n]$ be a coordinate system, let $w$ be an equationally saturated admissible wedge of $\tau$, and let $G$ be the Gröbner basis of $\mathbb{I}(w, \tau)$ (w.r.t. degrevlex order). Strengthening operations for inferring inequalities that hold in all non-linear models of $w$ are presented as a set of inference rules in Fig. 3.3. The hypotheses and consequences are expressed as polynomial inequalities in the coordinates $z_1, ..., z_n$. A rule may only be applied if hypotheses and consequences are linear and admissible. Each appearance of $\text{lin}_v(t)$ is implicitly guarded by the assumption that $\tau$ admits $t$ (so for example, the Floor rule is only applied to floor terms that appear in $\tau$).

The Interval rule assumes that each function $f$ is associated with a monotone function $f^{\text{int}}$ that approximates $f$ on intervals (e.g., the interval approximation of multiplication is $[a, b] \times^{\text{int}} [c, d] \overset{\text{def}}{=} [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$). The interval constraints in the hypothesis of the rule are com-
Algorithm 3: Equational saturation

Input: Coordinate system $\tau = [t_1, ..., t_n]$ and a wedge $w$
Output: Equational saturation of $w$ in $\tau$

1. $G \leftarrow$ Gröbner basis (degrevlex) for $I(w, \tau)$;
2. $active \leftarrow \{i : t_i$ is an application$\}$;
3. repeat
4. $B \leftarrow \{0\}$;
5. /* Compute congruence closure. CC maps terms to representative coordinates */
6. CC $\leftarrow$ empty map;
7. foreach $i$ in active do
8. Let $t_i = f(s_1, ..., s_m)$;
9. Let $r_j \leftarrow \text{red}_G(\text{lin}_\tau(s_i))$ for all $j$;
10. if $\text{CC}[f(r_1, ..., r_n)]$ is defined then
11. $z_r \leftarrow \text{CC}[f(r_1, ..., r_n)]$;
12. $B \leftarrow B \cup \{\text{red}_G(z_i - z_r)\}$; /* $i$ is represented by $z_r$ - it need not be processed again */
13. Remove $i$ from active;
14. else
15. $\text{CC}[f(r_1, ..., r_n)] := z_i$; /* Add implied linear equations to $w$. Note that since $G$ is computed w.r.t degrevlex, the degree of $\text{red}_G(z_i)$ must be $\leq 1$ (i.e., it’s linear) */
16. $w \leftarrow w \land (\bigwedge_{i=1}^{n} \text{interp}_\tau(z_i = \text{red}_G(z_i))) \land (\bigwedge_{b \in B} \text{interp}_\tau(b = 0))$;
17. $B \leftarrow B \cup \{\text{red}_G(b) : b \in \text{basis for } I(w, \tau)\}$;
18. $G \leftarrow$ Gröbner basis (degrevlex) for $(G \cup B)$;
19. until $B = \{0\}$;
20. return $w$

Compared using linear programming.

Unlike the equational inference rules, which we can apply until the wedge is saturated, the inequality inference rules can be applied indefinitely without ever converging. To enforce termination, we use a simple heuristic: we apply each inference rule in sequence (first we apply the Interval rule for every term, then the Product rule for every pair of inequalities, ...), and do not use the consequence of one application of a rule as a hypothesis for the next (e.g., if an inequality was derived via the Product rule, it may not later be used as the hypothesis of another application of the Product rule). As a consequence, our algorithm for strengthening a wedge is not complete with respect to the inference rules (i.e., there are inequations that can be deduced by the inference rules that will not be be deduced by a single run of the strengthening algorithm).

### 3.4.2 Basic Operations on Wedges

We now define some basic operations for manipulating wedges: pseudo-join, meet, widening, and projection.

Let $w_1$ and $w_2$ be wedges. The **pseudo-join** of $w_1$ and $w_2$ is a wedge that is an upper bound of $w_1$ and $w_2$ with respect to the ordering $|=N$ (it is not, however, a least upper bound with respect to $|=N$, which is not computable). We compute the pseudo-join $w_1 \sqcup w_2$ by first finding a minimal coordinate system $\tau$ that
In contrast to conventional methods, the program analysis proposed in chapter makes no use of widening in the analysis of loops: a loop is analyzed by extracting recurrences and computing closed forms.

Example 3.8. Consider the wedges $y \geq 1 \land y^2 \geq x$ and $0 \geq x$. A coordinate system admitting both wedges is $[x, y, y^2]$. Strengthening $y \geq 1 \land y^2 \geq x$ yields $y \geq 1 \land y^2 \geq x \land y^2 \geq y$, and strengthening $0 \geq x$ yields $0 \geq x \land y^2 \geq 0$. The pseudo-join is $y^2 \geq x \land y^2 \geq 0$. Notice that $0 \geq x \not\models_L y^2 \geq x \land y^2 \geq 0$.

Unlike the least upper bound, greatest lower bounds of wedges are exact and easily computed. The meet $w_1 \sqcap w_2$ is simply the conjunction: $w_1 \sqcap w_2 \triangleq w_1 \land w_2$.

Widening. In contrast to conventional methods, the program analysis proposed in chapter makes no use of widening in the analysis of loops: a loop is analyzed by extracting recurrences and computing closed forms.
However, the interprocedural framework for compositional recurrence analysis does make use of widening when analyzing functions with non-linear recursion (more than one recursive call on some path through the function) [85]. Thus for completeness, we will define the widening operator for wedges.

Like the pseudo-join operation, the widening operation of wedges relies on the operations of their underlying polyhedra. Unlike pseudo-join, the dimension of the coordinate system must decrease rather than increase. Let $\tau_1$ and $\tau_2$ be minimal coordinate systems admitting $w_1$ and $w_2$, respectively. Compute a coordinate system $\tau$ that contains only the terms that are common to both $\tau_1$ and $\tau_2$—w.l.o.g. assume that it is a prefix of both $\tau_1$ and $\tau_2$. We define $w_1 \nabla w_2$ as

$$w_1 \nabla w_2 \overset{\text{def}}{=} \text{interp}_\tau(\text{project}_L(\text{lin}_{\tau_1}(w_1),|\tau|)\nabla_L\text{project}_L(\text{lin}_{\tau_2}(w_2),|\tau|))$$

where $\text{project}_L(P,n)$ denotes the projection of a polyhedron onto its first $n$ dimensions and $\nabla_L$ denotes polyhedral widening.

**Projection.** Let $\Sigma'$ be a sub-vocabulary of $\Sigma$. The result of projecting a $\Sigma$-wedge $w$ onto $\Sigma'$ is a $\Sigma'$-wedge $w'$ such that $w \models_N w'$. While sometimes we wish to eliminate function symbols of arity $\geq 1$, most commonly we are interested in the case where $\Sigma$ and $\Sigma'$ differ by a set $X$ of constant symbols, in which case projecting onto $\Sigma'$ results in an over-approximation of the formula $(\exists X.w)$.

A projection algorithm is given as Alg. 4. It operates as follows. Let $w$ be a $\Sigma$-wedge and let $\tau = [t_1, \ldots, t_n]$ be a minimal coordinate assignment admitting $w$. If $t_i$ is a $\Sigma'$-term, we mark $i$ as safe and associate $i$ with $t_i$; otherwise, it is unsafe. We then compute the Gröbner basis for the ideal $\Pi[w,\tau]$ w.r.t. an elimination ordering for $[z_i : t_i \text{ is unsafe}]$. We then traverse the unsafe terms $t_i = f(s_1, \ldots, s_m)$: if every coordinate in $\text{red}_G(\text{lin}_+(s_j))$ is safe for all $s_j$, we mark $i$ safe and associate it with an equivalent $\Sigma'$-term. We iterate this process until no more coordinates are marked safe. We then use polyhedral projection to eliminate the unsafe dimensions, and then replace the safe dimensions with their associated $\Sigma'$-terms to obtain a $\Sigma'$ wedge $w'$.

### 3.4.3 Symbolic Abstraction

Symbolic abstraction is the key operation supported by the wedge domain, and the foundation of our method for extracting recurrence relations from transition formulas (see § 3.5). Given a formula $\phi$, symbolic abstraction computes a wedge $w$ that over-approximates it (that is, $\phi \models_N w$) and is as precise as possible (noting that the latter part of the specification is informal, since the most precise wedge is not necessarily computable or even well defined). Phrased differently, symbolic abstraction computes a system of equations and inequations that are implied by a given formula.
Algorithm 5: Symbolic abstraction.

Input: Formula $\phi$ over vocabulary $\Sigma$

Output: Wedge $w$ over vocabulary $\Sigma$ such that $\phi \models_N w$

1. $w \leftarrow$ false;
2. $F \leftarrow$ Skolemize $\phi$;
3. $R \leftarrow F$;
4. while there exists a model $\mathcal{L} \models_L R$ do
   5. Let $w_1$ be an implicant of $F$ s.t. $\mathcal{L} \models_L w_1$;
      // Wedge projection: remove Skolem constants
   6. $\langle w'_1, A_3 \rangle \leftarrow$ project($w_1, \Sigma$);
      // Wedge pseudo-join
   7. $\langle w, A_{\exists} \rangle \leftarrow w \sqcap w'_1$;
   8. $R \leftarrow R \land A_{\exists} \land A_{\sqcup} \land \neg w$;
5. return $w$

Let $\phi$ be a formula that we would like to over-approximate by a wedge. If $\phi$ happens to be a purely conjunctive formula then it is already a wedge, and we may simply strengthen (§ 3.4.1) and return it. In general, $\phi$ may contain disjunctions and existential quantifiers; the necessary ingredients for treating both were given in the previous section. In principle, one may compute a wedge $w$ that over-approximates $\phi$ as follows (1) Skolemize $\phi$ and rewrite in disjunctive normal form (DNF) $\phi \equiv w_1 \lor \ldots \lor w_n$, (2) for each disjunct $w_i$, project out Skolem constants (noting that each disjunct is a wedge), resulting in a wedge $w'_i$, and (3) take the pseudo-join of all resulting wedges $w \overset{def}{=} w'_1 \sqcup \ldots \sqcup w'_n$. While this is a relatively straightforward procedure, it is worth noting that the rest of the operations of the wedge domain were designed specifically to support it. A faster algorithm that computes the same wedge—but can often avoid the worst-case behavior of DNF conversion—is given as Alg. 5.

The symbolic-abstraction procedure assumes that the wedge projection and pseudo-join operations emit justifying axioms. More precisely, $\text{project}(w, \Sigma')$ returns a pair $\langle w', A \rangle$ consisting of a $\Sigma'$-wedge and a formula $A$ such that $\text{true} \models_N A$ and $A \land w \models_L w'$. Similarly, $w_1 \sqcup w_2$ returns a pair $\langle w, A \rangle$ consisting of a wedge $w$ and a formula $A$ such that $(w_1 \lor w_2) \land A \models_L w$. Computing this extra information is straightforward: For the inferred inequalities, we track the inference rules that are applied during the course of the operation. For the inferred equalities, we must keep track of every non-trivial equation that is added to the Gröbner basis (“asserted equalities”), and whenever a non-trivial equation is added to the polyhedron $P$ (“derived equality”), we emit the implication that the asserted equalities imply the derived equality.

### 3.5 Extracting Recurrences

The fundamental question of interest in this chapter is how to compute an over-approximation of the transitive closure of a transition formula. As in CRA [49], our approach (1) computes a system of recurrence relations
that are semantically entailed by the input transition formula, and (2) finds a closed-form representation of those recurrences. In this section, we give a method for extracting recurrences from a transition formula that is based on the wedge domain (§ 3.4). A description of how this method goes beyond that of Farzan and Kincaid [49] is given in § 3.8.

In § 3.6, we describe a method for solving recurrence equations of the form

\[
\begin{bmatrix}
y_1[n] \\
\vdots \\
y_k[n]
\end{bmatrix} = A \begin{bmatrix}
y_1[n-1] \\
\vdots \\
y_k[n-1]
\end{bmatrix} + \begin{bmatrix}
r_1 \\
\vdots \\
r_k
\end{bmatrix},
\]

(or more succinctly, \(y[n] = Ay[n-1] + r(n)\)), where \(A \in \mathbb{Q}^{k \times k}\) and each \(r_1\) is a term of the form:

\[
\begin{align*}
r &\in \text{RecTerm}(K, F, n) ::= a \times r | x \times r | r_1 + r_2 | f(n) | u \\
u &\in \text{MultRecTerm}(K, n) ::= a | x | n | a^n | u_1 + u_2 | u_1 \times u_2
\end{align*}
\]

where \(K\) is a set of symbolic constants (initial values of program variables) and \(F\) is a set of implicitly interpreted functions (IIFs). The closed forms computed by the recurrence solver have the same form as \(\text{RecTerm}\) above.

In general, the behavior of a loop cannot be expressed as such a system of recurrence equations: loops may have non-deterministic assignments, conditional branches, nested loops, etc. The question is how we can exploit a solver for recurrences of this form, even though the loop itself does not have such a description. Our solution involves extracting multiple systems of recurrences for the same loop, where each system approximates some aspect of the dynamics of the loop. In the following, we present three different mechanisms for extracting recurrences from a transition formula, each building on the last.

The recurrence-extraction mechanisms all assume a wedge representation of the loop body, which eliminates the need to deal with disjunction and existentially quantified variables (both of which are present in transition formulas). Thus, the first step we take in computing recurrences entailed by a transition formula \(\phi_{\text{body}}\) is to find an over-approximating wedge \(w_{\text{body}}\) using the symbolic-abstraction procedure from § 3.4.3 (Alg. 5).

Before proceeding to the recurrence-extraction algorithms, we will formalize the problem set-up that is common to all. We fix \(\phi_{\text{body}}\) to be a loop-body transition formula. Suppose that \(PVar\) is a set of constant symbols that represent program variables, and \(PVar'\) is a disjoint set of “primed copies” of program variables. Suppose that \(\Sigma\) is a vocabulary containing \(PVar, PVar',\) and \(\Sigma_N\). Fix a \(\Sigma\)-wedge \(w_{\text{body}}\) that over-approximates the transition relation of a loop body. Let \(\tau = [t_1, ..., t_n]\) be a coordinate system admitting \(w_{\text{body}}\). Without loss of generality, suppose that the terms \(t_1, ..., t_m\) correspond to the program variables in \(PVar\) and the terms
while(*) {
    if (*) { x = 2x + y + i; }
    else { y = x + 2y + i; }
    i = i + 1;
    j = j + x;
}

(b) Loop-transition formula

\[
\begin{bmatrix}
    x' \\
    y' \\
end{bmatrix} = \begin{bmatrix}
    2 & 1 & 1 \\
    1 & 2 & 1 \\
end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    i \\
end{bmatrix} + \begin{bmatrix}
    1 \\
    0 \\
end{bmatrix}
\]

(c) Loop-transition wedge

3.5.1 Affine Recurrences

An affine recurrence is a recurrence for a linear combination of program variables. Our goal is to compute matrices \( A \in \mathbb{Q}^{k \times m} \) and \( B \in \mathbb{Q}^{k \times k} \) and a vector \( c \in \mathbb{Q}^k \) such that \( w_{\text{body}} \vdash_L A x' = B A x + c \). The relation \( A x' = B A x + c \) defines a set of affine recurrences for the linear combinations in \( A x \). Moreover, we want the affine space consisting of all valuations of \( x \) and \( x' \) that satisfy \( A x' = B A x + c \) to be least (in the sense that if \( A', B', c' \) satisfy \( w_{\text{body}} \vdash_L A' x' = B' A' x + c' \), then we have \( A x' = B A x + c \vdash_L A' x' = B' A' x + c' \)). With \( A \) in hand, we can let \( y = A x \) and \( y' = A x' \), and turn our attention to the system of recurrence equations \( y^{[n]} = B y^{[n-1]} + c \) (which matches the form in Eqn. (3.2)). Suppose that \( \phi \) is a closed-form solution for \( y^{[n]} \), in the sense that \( y^{[n]} = B y^{[n-1]} + c \iff y^{[n]} = \phi(y^{[0]}, n) \). Then the transitive closure of \( \phi_{\text{body}} \) must entail \( \exists n \geq 0 A x' = \phi(A x, n) \).

As a motivating example, consider the loop from Fig. 3.4(a) and the associated transition formula and wedge. Observe that (1) the loop is non-deterministic, and so there are no exact representations of the closed forms for the variables \( x \) and \( y \), and (2) despite the fact that there is an apparently recursive assignment to the variable \( j \), there is no exact closed form for \( j \) because it depends on variable \( x \), for which there is no closed form. For this example, the best affine recurrence is

\[
\begin{bmatrix}
    i' \\
    j' \\
end{bmatrix} = \begin{bmatrix}
    1 & 0 & 0 \\
    0 & 1 & 1 \\
end{bmatrix}
\begin{bmatrix}
    i \\
    j \\
    x' \\
end{bmatrix} + \begin{bmatrix}
    1 \\
    0 \\
end{bmatrix}
\]

(or more simply, \( i' = i + 1 \) and \( x' + y' = 2(x + y) + i \)). Note how the second row of matrix \( A \) in Eqn. (3.3)
creates the linear combinations $x' + y'$ and $x + y$ in the latter recurrence equation.

The best affine recurrence entailed by $w$ can be computed via successive approximation. We begin by explaining the procedure in the abstract, and return to the concrete example below. The first approximation $A_1 x' = M_1 x + c_1$ is merely the constraint representation of the affine hull of the underlying polyhedron of $w$, projected onto the first $2m$ dimensions (corresponding to the $x$ and $x'$ symbols). If $M_1$ factors as $BA_1$, then we are done: the best affine recurrence entailed by $w$ is $A_1 x' = BA_1 x + c_1$. If $M_1$ does not factor as $BA_1$, then the equation $A_1 x' = M_1 x + c_1$ cannot be interpreted as a recurrence. (The rowspace of $A_1$ must contain the rowspace of $M_1$—intuitively, the terms that appear on the right-hand side of a recurrence equation must be linear combinations of terms that appear on the left.) Thus, we compute a second approximation $A_2 x' = M_2 x + c_2$. For any approximation $i$, the $(i + 1)$th approximation is computed from the $i$th by finding a matrix $T_i$ whose rows form a basis for the vector space $\{v : \exists u. uA_i = vB_i\}$ and taking $A_{i+1} = T_i A_i$, $M_{i+1} = T_i M_i$, and $c_{i+1} = T_i c_i$. Eventually this process must reach a fixpoint, because each step decreases the dimension of the rowspace of $A_1$ by at least 1.

Returning to our running example Fig. 3.4, the first approximation is essentially just the wedge Fig. 3.4(c) itself:

$$
\begin{bmatrix}
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
i' \\
j' \\
x' \\
y' \\
\end{bmatrix}
= 
\begin{bmatrix}
1 & 0 & 2 & 2 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
i \\
j \\
x \\
y \\
\end{bmatrix}
+ 
\begin{bmatrix}
0 \\
1 \\
0 \\
\end{bmatrix}
$$

Observe that the rowspace of $A_1$ does not contain $\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$, the third row of $M_1$ (i.e., $j'$ appears on the right-hand side of an equation but we cannot isolate $j'$ on the left). We find a matrix $T_1$ whose rows generate the space $\{v : \exists u. uA_1 = vB_1\}$ pictured below to the left (intuitively, $T_1$ is a linear transformation that projects out $j'$), and multiply both sides of the equation to get the equation pictured below to the right:

$$
\begin{bmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
i' \\
j' \\
x' \\
y' \\
\end{bmatrix}
= 
\begin{bmatrix}
1 & 0 & 0 & 0 \\
1 & 0 & 2 & 2 \\
\end{bmatrix}
\begin{bmatrix}
i \\
j \\
x \\
y \\
\end{bmatrix}
+ 
\begin{bmatrix}
1 \\
0 \\
\end{bmatrix}
$$

The rowspace of $A_2$ does contain the rowspace of $M_2$, so are done. We factor $M_2$ as $BA_2$ and obtain the affine recurrence shown in Eqn. (3.3).
3.5.2 Stratified Recurrences

Consider the simple loop `while(*){ y += x*x; x++; }`. The affine recurrence extracted by the algorithm in the preceding section is \( x' = x + 1 \): no recurrence is extracted for \( y \) because its recurrence has a non-linear dependence on \( x \). However, because there is no circular dependence of \( x \) on \( y \), we can arrange the recurrences into strata and leverage the closed form of \( x \) to compute a closed form for \( y \):

<table>
<thead>
<tr>
<th>Stratum</th>
<th>Recurrence</th>
<th>Closed form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( x' = x + 1 )</td>
<td>( x^{[n]} = x^{[0]} + n )</td>
</tr>
<tr>
<td>1</td>
<td>( y' = y + x \times x )</td>
<td>( y^{[n]} = y^{[0]} + \sum_{i=0}^{n-1} x^{[i]} = y^{[0]} + \sum_{i=0}^{n-1} (x^{[0]} + i)^2 )</td>
</tr>
</tbody>
</table>

More generally, we can compute a sequence of stratified recurrences such that each recurrence in the sequence may have non-linear dependencies on previous terms in the sequence:

\[
\begin{align*}
\text{w}_{\text{body}} \models L \ & A_0 x' = B_0 A_0 x + c \\
\text{w}_{\text{body}} \models L \ & A_1 x' = B_1 A_1 x + p_1(A_0 x) \\
& \vdots \\
\text{w}_{\text{body}} \models L \ & A_i x' = B_i A_i x + p_i(A_0 x, \ldots, A_{i-1} x)
\end{align*}
\]

where \( A_0 x' = B_0 A_0 x + c \) is an affine recurrence and each \( p_i(A_0 x, \ldots, A_{i-1} x) \) is a polynomial over the terms in \( A_0 x, \ldots, A_{i-1} x \). For each stratum \( j \), the closed form \( \text{cl}_i \) is computed as the solution to a single matrix recurrence (of the form in Eqn. (3.2)) by substituting closed forms of previous strata into the next:

\[
\begin{align*}
y_0^{[n]} &= B_0 y_0^{[n-1]} + c \iff y_0^{[n]} = \text{cl}_0(y_0^{[0]}, n) \\
y_1^{[n]} &= B_1 y_1^{[n-1]} + p_1(\text{cl}_0(y_0^{[n]}, n)) \iff y_1^{[n]} = \text{cl}_1(y_0^{[0]}, y_1^{[0]}, n) \\
& \vdots \\
y_i^{[n]} &= B_i y_i^{[n-1]} + p_i(\text{cl}_0(y_0^{[n]}, n), \ldots, \text{cl}_{i-1}(y_0^{[n]}, \ldots, y_{i-1}^{[n]}, n)) \iff y_i^{[n]} = \text{cl}_i(y_0^{[0]}, \ldots, y_{i-1}^{[0]}, n)
\end{align*}
\]

Observing that substitution of \( \text{MultRecTerms} \) into a polynomial yields a \( \text{MultRecTerm} \), this procedure works as long as no \( \text{cl}_i(y_0^{[0]}, \ldots, y_{i-1}^{[0]}, n) \) contains an IIF.

The procedure for extracting stratified recurrences similarly proceeds iteratively on strata, beginning at stratum 0 (the best affine recurrence satisfied by the input wedge, as computed in § 3.5.1), and continuing until reaching a stratum where no new recurrences are found. For each stratum \( i \), we compute the recurrence equation

\[ A_i x' = B_i A_i x + p_i(A_0 x, \ldots, A_{i-1} x) \]
as follows. Let \( T = \{ t_1, ..., t_k \} \) be a set consisting of all linear terms of the form \( ax \), where \( a \) is a row of one of the matrices \( A_0, ..., A_{i-1} \); \( T \) corresponds to the set of linear terms for which there is a recurrence equation in some stratum \( < i \) (and thus, we know a closed form for each \( t_j \)). For each term \( t_j \in T \) create a fresh variable \( r_j \) and define \( I_i \) to be the ideal generated by \( I(w_{\text{body}}, \tau) \) along with the (linear) polynomials \( \{ r_j - t_j : 1 \leq j \leq k \} \).

Compute a Gröbner basis \( G_i \) for \( I_i \) w.r.t. an elimination order for \( \{ z_{2m+1}, ..., z_n \} \), with the \( r \) variables ordered before the \( z \) variables. Recall that the variables \( z_{2m+1}, ..., z_n \) correspond to non-linear terms over program variables, so reduction by \( G_i \) projects out non-linear terms over the program variables, but in doing so may introduce polynomials of the new \( r \) variables (which is desirable, since those are the polynomials for which we can compute closed forms). Compute an initial system of equations

\[
A_{i,1}x' = M_{i,1}x + p_{i,1}
\]

where \( A_{i,1} \) and \( M_{i,1} \) are rational matrices, and \( p \) is a vector of polynomials in \( \mathbb{Q}[t_1, ..., t_k] \) by taking the constraint representation of the affine hull of the underlying polyhedron of \( w \), reducing each equation by \( G_i \), collecting the subset of equations that have the appropriate form, and finally replacing each \((z, \tau)\) variable with its corresponding \((x, x', t_i)\) term. Then perform the same fixpoint algorithm from § 3.5.1 to reduce this system of equations to a system of recurrence equations of the form \( A_i x' = B_i A_i x + p_{i,1} \), and proceed to the next stratum \( i + 1 \).

### 3.5.3 Recurrence Inequations

In § 3.5.1 and § 3.5.2, we described methods for describing the exact behavior of linear terms over program variables using recurrence equations. In this sub-section, our goal shifts to finding inequations that bound linear terms. Our goal is to compute a system of recurrence inequations

\[
 w_{\text{body}} \models_1 A x' \leq B Ax + p(A_0 x, ..., A_k x)
\]

where \( A \in \mathbb{Q}^{k \times m} \) is rational matrix, \( B \in \mathbb{Q}^{k \times k} \) is a non-negative matrix (that is, a matrix with no negative entries), and \( p(A_0 x, ..., A_k x) \) is a polynomial over \( A_0 x, ..., A_k x \), where \( A_0, ..., A_k \) are as in § 3.5.2.

The procedure for extracting recurrence inequations operates similarly to the one for affine recurrences, except that rather than finding some \( i \) such that \( A_i \) and \( M_i \) generate the same row space, we require that \( A_i \) and \( M_i \) generate the same cone. Given a matrix \( A \) with rows \( a_1, ..., a_k \), the (convex) cone generated by the rows of \( A \) is as follows:

\[
\text{cone}(A) \overset{\text{def}}{=} \left\{ \lambda_1 a_1 + \cdots + \lambda_k a_k : \lambda_1 \geq 0, ..., \lambda_k \geq 0 \right\}
\]
The requirement that \( A_i \) and \( M_i \) generate the same cone guarantees that there exists a factorization \( M_i = BA_i \) where \( B \) is non-negative. A second complication is that the iterative algorithm may fail to terminate because the lattice of polyhedra (in contrast to the lattice of vector spaces) has infinite ascending chains. We use polyhedral widening to guarantee termination of the successive-approximation algorithm.

### 3.6 Operational Calculus Recurrence Solving

In this section, we present the Operational Calculus Recurrence Solver (OCRS). Ultimately, we present a method (Alg. 7) that solves recurrences of the form of Eqn. (3.2). However, in order to build intuition we show how the mathematics of the operational calculus can be used to solved univariate recurrences. We then extend that method to solve first-order matrix recurrences.

#### 3.6.1 Introduction

Operational calculus \([100]\) is a technique for transforming a problem in analysis into an algebraic problem. Differencing and summation correspond to algebraic operators in a certain underlying algebra. Most commonly, operational-calculus techniques are used to solve differential equations. However, using the operational-calculus algebra of Berg \([15, \text{Ch. II}]\) it is possible to solve recurrence relations using operational calculus. The analysis problem of solving a recurrence equation is transformed into an equation or equation system, which is then solved by algebraic manipulations. A solution to the original problem can be read out of a solution to the algebraic problem.

### The Ring of Functions

In this section, we review the univariate operational calculus of Berg \([15, \text{Ch. II}]\). The ring (Defn. 2.3) of functions \( \mathcal{R} \) is defined as \( \mathcal{R} \overset{\text{def}}{=} (\mathbb{R}, +, \cdot, 0, 1) \), where \( \mathbb{R} = \mathbb{N} \to \mathbb{Q} \).

It is often helpful to consider a ring element \( r \in \mathcal{R} \) as an infinite sequence \( (r[0], r[1], r[2], \ldots) \). For \( r \in \mathcal{R} \), we use both \( r(i) \) and \( r[i] \) to denote the \( i \)th element of \( r \). (We use \( r(i) \) when we wish to emphasize that \( r \) is the function \( \lambda i. r(i) \), and \( r[i] \) when we wish to emphasize that \( r \) is the sequence \( \langle r[i] \rangle \).)

Ring addition is pointwise addition: for all \( a, b \in \mathcal{R} \), \( a + b \overset{\text{def}}{=} \lambda i. a(i) + b(i) \), so the additive identity element, \( 0 \), is \( \lambda i. 0 = \langle 0, 0, 0, \ldots \rangle \). Ring multiplication is the following convolution-product difference:

\[ a \odot b = \sum_{i=0}^{\infty} a(i) b(i-1) \]

Berg uses \( \mathbb{N} \to \mathbb{C} \), where \( \mathbb{C} \) is the set of complex numbers. However, our goal is to compute closed forms for recurrences that can be expressed in the term language of the logic defined in § 3.3.2, which requires using \( \mathbb{Q} \).
Definition 3.9. For all \( a, b \in \mathbb{R} \),

\[
a \cdot b \overset{\text{def}}{=} \lambda n \sum_{\nu=0}^{n} a^{[\nu]} b^{[n-\nu]} - \sum_{\nu=0}^{n-1} a^{[\nu]} b^{[n-1-\nu]}.
\] (3.4)

We will often omit multiplication operators, and write \( a \cdot b \) as \( ab \). The multiplicative identity element, \( 1 \), is \( \lambda i = \langle 1, 1, 1, \ldots \rangle \). It is easy to show that multiplication is commutative and that multiplication distributes over addition [15, §7].

The constant elements of \( \mathbb{R} \) are those of the form \( c \overset{\text{def}}{=} \lambda n. c = \langle c, c, c, \ldots \rangle \). When either argument of a multiplication is a constant, ring multiplication acts like scalar multiplication: for all \( b \in \mathbb{R} \), \( c \cdot b = \lambda n. cb^{[n]} \).

When both arguments are scalars, ring multiplication is isomorphic to ordinary multiplication: \( c \cdot d = cd \).

It is sometimes useful to write Eqn. (3.4) as

\[
a \cdot b \overset{\text{def}}{=} \lambda n. \sum_{\nu=0}^{n-1} a^{[\nu]} (b^{[n-\nu]} - b^{[n-1-\nu]}) + a^{[n]} b^{[0]}.
\] (3.5)

\( \text{Id} \) denotes the identity function \( \lambda n. n = \langle 0, 1, 2, \ldots \rangle \). Using Eqn. (3.5), note that for all \( a \in \mathbb{R} \),

\[
\text{Id} \cdot a = a \cdot \text{Id} = \lambda n. \sum_{\nu=0}^{n-1} a^{[\nu]} \left( (n - \nu) - (n - 1 - \nu) \right) + a^{[n]} 0 = \lambda n. \sum_{\nu=0}^{n-1} a^{[\nu]}.
\]

In other words, \( \text{Id} \) acts as a summation operator, producing the sequence of (exclusive) partial sums of \( a \).

For instance, \( \text{Id} \cdot \text{Id} = \langle 0, 0, 1, 3, 6, 10, 15, \ldots \rangle = \lambda n. \binom{n}{2} \).

Let \( v \overset{\text{def}}{=} \langle 0, 1, 1, 1, \ldots \rangle \). Let \( c^{[n]} \) denote an arbitrary infinite sequence \( \langle c^{[0]}, c^{[1]}, \ldots \rangle \). Then

\[
v \cdot c^{[n]} = \lambda n. \begin{cases} 0 & \text{if } n = 0 \\ c^{[n-1]} & \text{if } n \geq 1 \end{cases}
\]

and hence ring multiplication by \( v \) performs a right shift (and ring multiplication by \( v^m \) performs a right shift of \( c^{[n]} \) by \( m \) places.).

The Field of Operators

Ring \( \mathbb{R} \) has no divisors of 0, and hence can be extended to a field (Defn. 2.5) \( \mathbb{F} \overset{\text{def}}{=} (F, +, \cdot, 0, 1) \) with \( \mathbb{R} \subseteq \mathbb{F} \). An element of \( \mathbb{F} \) is called an operator. Some operators are elements of \( \mathbb{R} \); however, there are interesting operators

\footnote{As is standard, for \( n \geq k \), the binomial coefficient \( \binom{n}{k} \) denotes \( \frac{n!}{k!(n-k)!} \). We define \( \binom{n}{0} = 1 \) and, for all \( k > n \), \( \binom{n}{k} = 0 \).}
that are not members of $\mathbb{R}$. For instance, the equation

$$qv = 1 \quad (3.6)$$

has no solution in $\mathbb{R}$. However, because $v \neq 0$, there is an operator $q \in \mathbb{F}$ that solves Eqn. (3.6). Intuitively, $q$ should be a left-shift operator.

As Berg points out [15, §8], an operator can be interpreted as a sequence with a finite number of non-zero elements at negative-index positions (whereas a ring element is a sequence with no non-zero elements at negative-index positions). Therefore, it is easy to show that the appropriate representation for $q$, the operator that solves Eqn. (3.6), is

$$q = \begin{array}{ccccccc}
\ldots & -2 & -1 & 0 & 1 & 2 & 3 & \ldots \\
\end{array}$$

Let $y^{[n]}$ denote an arbitrary infinite sequence $\langle y[0], y[1], \ldots \rangle$, and let $y^{[n+1]}$ denote $\langle y[1], y[2], \ldots \rangle$. The following general formula expresses $y^{[n+1]}$ in terms of $y^{[n]}$, $y[0]$, and $q$. It works by multiplying $y^{[n]}$ by $q$ to shift the sequence to the left by one position, and subtracting away the value that would otherwise appear at index position $-1$, giving us $y^{[n+1]}$:

$$y^{[n+1]} = q \cdot y^{[n]} - (q - 1)y[0]. \quad (3.7)$$

We can depict some of the infinite sequences that appear in Eqn. (3.7) as follows:

$$y^{[n]} = \begin{array}{ccccccc}
\ldots & -2 & -1 & 0 & 1 & 2 & 3 & \ldots \\
\end{array}$$

$$y^{[n+1]} = \begin{array}{ccccccc}
\end{array}$$

$$q \cdot y[0] = \begin{array}{ccccccc}
\end{array}$$

$$(q - 1)y[0] = \begin{array}{ccccccc}
\ldots & 0 & y[0] & 0 & 0 & 0 & 0 & \ldots \\
\end{array}$$

Henceforth, sequences like the ones above will be written as follows:

$$y^{[n]} = \langle y[0], y[1], y[2], y[3], \ldots \rangle \quad q \cdot y^{[n]} = \langle y[0] || y[1], y[2], y[3], \ldots \rangle$$

$$y^{[n+1]} = \langle y[1], y[2], y[3], y[4], \ldots \rangle \quad (q - 1)y[0] = \langle y[0] || 0, 0, 0, 0, \ldots \rangle$$

where $||$ is used to separate the negative-index positions (if any) from the non-negative positions. The values at all negative-index positions that are not shown are assumed to be 0.
Using this notation, the operators \( q \) and \((q - 1)\) have the sequences shown below:

\[
q = \langle 1 \parallel 1, 1, 1, \ldots \rangle \quad (q - 1) = \langle 1 \parallel 0, 0, 0, \ldots \rangle
\]

Eqn. (3.7) can be generalized to create \( y^{[n+m]} = \langle y^{[m]}, y^{[m+1]}, y^{[m+2]}, y^{[m+3]}, \ldots \rangle \) from \( y^{[n]} \) as follows:

\[
y^{[n+m]} = q^m \cdot y^{[n]} - \sum_{\mu=0}^{m-1} q^\mu \cdot (q - 1) y^{[m-1-\mu]}.
\] (3.8)

In this case, it is necessary to knock out the \( m \) values \( y^{[0]}, y^{[1]}, \ldots, y^{[m-1]} \) from positions \(-m, -m + 1, \ldots, -1\), respectively. Eqn. (3.8) can be understood by recognizing that

\[
(q - 1)y^{[m-1-\mu]} = \langle y^{[m-1-\mu]} \parallel 0, 0, 0, \ldots \rangle,
\]

which is then shifted \( \mu \) positions to the left by \( q^\mu \).

**Translation Rules**

In some cases, functions that are expressed in terms of operators can also be expressed purely in terms of ordinary algebra. For example, consider the functional sequence \( \lambda n.\alpha^n = \langle \alpha^0, \alpha^1, \alpha^2, \ldots \rangle \) for \( n \in \mathbb{N} \) and \( \alpha \in \mathbb{Q} \). By Eqn. (3.7) and \( \alpha^0 = 1 \) we have:

\[
\alpha^{n+1} = q\alpha^n - (q - 1)\alpha^0,
\]

\[
q\alpha^n - \alpha^{n+1} = (q - 1)
\]

\[
\alpha^n(q - \alpha) = (q - 1)
\]

\[
\alpha^n = \frac{q - 1}{q - \alpha}
\]

Thus we have a way of translating a recognizable functional sequence in standard algebra to the operational-calculus algebra. We will write this translation as:

\[
T_n \left( \frac{q - 1}{q - \alpha} \right) = \alpha^n
\]

Fig. 3.5 shows the above rule along with other useful translation rules.
\[ T_n(c) = c \quad (3.9) \]

\[ T_n \left( \sum_{k \in K} g_k \right) = \sum_{k \in K} T_n(g_k) \quad (3.10) \]

\[ T_n(gf) = cT_n(f) \quad (3.11) \]

\[ T_n \left( \frac{1}{(q - 1)^i} \right) = \binom{n}{i} \quad (3.12) \]

\[ T_n \left( \frac{q - 1}{q - k} \right) = k^n \quad (3.13) \]

\[ T_n \left( \frac{q - 1}{(q - k)^c} \right) = \binom{n}{c} k^{n-c} \quad (3.14) \]

\[ T_n \left( \frac{1}{q - k} \right) = \frac{k^n - 1}{k - 1} \quad (3.15) \]

\[ T_n \left( \frac{1}{(q - k)^c} \right) = \binom{n}{c-1} - T_n \left( \frac{1}{(q - k)^{c+1}} \right) \quad (3.16) \]

Figure 3.5: Translation function \( T_n : F \rightarrow R \). In Eqn. (3.10) \( K \subseteq \mathbb{N} \) is some finite index set.

Algorithm 6: Univariate Operational Calculus Recurrence Solving

**Input**: A recurrence inequation \( r \), a variable to solve for \( y \), and an index variable \( n \)

**Output**: The closed form solution for \( y \)

1. \( \text{OpR} \leftarrow \text{Translate } r \text{ into } F \text{ using Eqn. (3.8) and } T_n^{-1} \); 
2. \( \text{OpR} \leftarrow \text{Solve for } y \text{ in } \text{OpR} \); 
3. \( \text{OpR} \leftarrow \text{Perform partial-fraction decomposition on } \text{OpR} \); 
4. \( S \leftarrow T_n(\text{OpR}) \); 
5. \( \text{return } S \)

3.6.2 Solving First-Order Univariate Recurrences

The properties of the algebra described in § 3.6.1 give us a nice algebraic way of solving recurrences of the following form:

\[ y^{[n+1]} = \alpha \cdot y^{[n]} + \sum_i f_i(n) \quad \text{where } f_i(n) = \text{poly}_i(n) \cdot \beta_i^n \quad \text{and } \text{poly}_i(n) \in \mathbb{Q}[n] \quad (3.17) \]

Alg. 6 presents a method for solving such univariate recurrences.

Explanation of Operational Calculus Recurrence Solving

Between each step of the algorithm we simplify expressions using a simplification algorithm, given in Cohen [31], which performs standard algebraic simplifications, such as collecting like terms and combining exponents. Simplification helps to maintain a more canonical form for expression matching in later steps.

Consider how Step 1 of Alg. 6, translates a recurrence of the form of Eqn. (3.17). We can use Eqn. (3.7) to write \( y^{[n+1]} \) as \( qy^{[n]} - (q - 1)y^{[0]} \). For the right-hand side of Eqn. (3.17), \( \alpha \cdot y^{[n]} \) remains the same, except that the constant \( \alpha \) becomes the constant sequence \( \alpha \). What remains to describe is how we obtain \( T_n^{-1}(\sum_i f_i(n)) \).

As a quick detour, note that for all the rules in Fig. 3.5, polynomials are represented using a binomial operator. Thus we must have an appropriate way to represent polynomials using binomials. This transformation
can be performed via the following relation, where \( \{ \frac{d}{j} \} \) denotes a Stirling number of the second kind:

\[
\begin{align*}
\frac{d^d}{j!} = \frac{d^d}{j!} \binom{n}{j} = d^d = \sum_{j=0}^{d} \binom{n}{j} \end{align*}
\] (3.18)

To perform the translation \( T_n^{-1} (\sum_i f_i(n)) \), first note that by Eqn. (3.10), we can pull the summation out. We now seek a characterization of \( T_n^{-1} (\text{poly}_i(n) \ast \beta_i^n) \). First, consider rewriting \( \text{poly}_i(n) \) using Eqn. (3.18), where \( d = \deg(\text{poly}_i(n)) \).

\[
\begin{align*}
f_i(n) = \beta_i^n \ast \text{poly}_i(n) = \beta_i^n \ast \sum_{j=0}^{d} \nu_{i,j} \binom{n}{j} = \sum_{j=0}^{d} \beta_i^j \ast \nu_{i,j} \binom{n}{j} \beta_i^{n-j}
\end{align*}
\]

We can use the general transformation

\[
T_n^{-1} \left( \binom{n}{c} k^{n-c} \right) = \frac{q-1}{(q-k)^{c+1}}
\]

to see that every \( f_i(n) \) can be translated into the following form in the operational-calculus algebra:

\[
T_n^{-1}(f_i(n)) = \sum_{j=0}^{d} \frac{\gamma_{i,j}(q-1)}{(q-\beta_i)^{c+1}}
\]

where \( \gamma_{i,j} \) is an appropriate constant. Thus, after Step 1 Alg. 6 produces an equation of the form

\[
q \cdot y^{\lfloor n \rfloor} - (q-1) y^{\lfloor 0 \rfloor} = \alpha \cdot y^{\lfloor n \rfloor} + \sum_{i} \sum_{j=0}^{d} \frac{\gamma_{i,j}(q-1)}{(q-\beta_i)^{c+1}}
\]

The algorithm continues, solving the above equation for \( y_n \) in Step 2.

\[
y^{\lfloor n \rfloor} = \frac{q-1}{q-\alpha} y^{\lfloor 0 \rfloor} + \sum_{i} \frac{\gamma_{i}(q-1)}{(q-\alpha)(q-\beta_i)^{c+1}}
\]

Note the double summation has been replaced by a single summation with a new index, \( l \).

Thus, after Step 2, we need to transform all the summands to a form that has a translation rule in Fig. 3.5. We can accomplish this goal by performing partial-fraction decomposition on all the summands (Step 3).

\[
\frac{q-1}{q-\alpha} y^{\lfloor 0 \rfloor} + \sum_{k} \frac{\alpha_k}{(q-b_k)^{c_k}}
\]

where \( \alpha_k \) and \( b_k \) are constants resulting from partial-fraction decomposition, and every \( c_k \) is a non-negative integer. After Step 3, the right-hand side of the equation will be a sum of terms, each of which has a translation
rule in Fig. 3.5 that can be applied to translate the term back into standard algebra (Step 4). Finally, the algorithm removes binomials and simplifies the result.

The preceding four paragraphs have shown that Alg. 6 obtains a closed-form solution to any recurrence of the form given in Eqn. (3.17). The steps used in Alg. 6 never get stuck; i.e., operations such as partial-fraction decomposition in Step 3 can always be performed. In other words, we have proven the following theorem:

**Theorem 3.10.** Alg. 6 is complete for the class of recurrences given in Eqn. (3.17).

**Example 3.11.** Consider the following example, which generates an exponential closed form.

\[ y^{[n+1]} = 2y^{[n]} + 3^n \]

*Step 1 generates the following equation in \( \mathcal{F} \):

\[
q \cdot y^{[n]} - (q - 1)y^{[0]} = \mathcal{T}_n^{-1} \left( 2y^{[n]} + 3^n \right)
\]

\[
q \cdot y^{[n]} - (q - 1)y^{[0]} = 2y^{[n]} + \frac{q - 1}{q - 3}
\]

By Eqn. (3.11), Eqn. (3.9), Eqn. (3.10), Eqn. (3.13), and Eqn. (3.7).

After solving for \( y^{[n]} \), Step 2 produces

\[
y^{[n]} = \frac{q - 1}{q - 2}y^{[0]} + \frac{q - 1}{(q - 3)(q - 2)}
\]

*Step 3 performs partial-fraction decomposition on the right-hand side of the equation where appropriate, obtaining

\[
y^{[n]} = \frac{q - 1}{q - 2}y^{[0]} + \frac{2}{q - 3} - \frac{1}{q - 2}
\]

Finally, Step 4 translates the resultant expression back to standard algebra and the algorithm performs a final simplification.

\[
y^{[n]} = \mathcal{T}_n \left( \frac{q - 1}{q - 2}y^{[0]} + \frac{2}{q - 3} - \frac{1}{q - 2} \right)
\]

\[
= 2^n y^{[0]} + 2 \frac{3^n - 1}{3 - 1} - \frac{2^n - 1}{2 - 1}
\]

\[
= 2^n (y^{[0]} - 1) + 3^n
\]
3.6.3 Solving First-Order Matrix Recurrences

We now give details about solving what we call matrix recurrences. Consider a vector of recurrence variables indexed by \( n + 1 \), \( x^{[n+1]} = [x_1^{[n+1]}, x_2^{[n+1]}, \ldots, x_{k}^{[n+1]}]^t \); their copies, indexed by \( n \), \( x^{[n]} = [x_1^{[n]}, x_2^{[n]}, \ldots, x_{k}^{[n]}]^t \); a matrix \( A \) with entries in \( \mathbb{Q} \); and a vector \( b = [f_1(n), f_2(n), \ldots, f_k(n)]^t \) in which each \( f_i(n) \) is a \( \text{RecTerm} \) as given in §3.5. We seek a closed-form solution for \( x^{[n]} \) in the recurrence

\[
x^{[n+1]} = Ax^{[n]} + b
\]

(3.19)

Let us consider how to extend the method from §3.6.1 and §3.6.2 to solve such an equation.

Left-Hand Side of Eqn. (3.19). We seek a rule for the vector \( x^{[n+1]} \) that is analogous to Eqn. (3.7). If we consider applying Eqn. (3.7) to each of the entries of \( x^{[n+1]} \), we have the following:

\[
x^{[n+1]} = \left[ \begin{array}{c}
q \cdot x_1^{[n]} - (q - 1)x_1^{[0]} \\
\vdots \\
q \cdot x_k^{[n]} - (q - 1)x_k^{[0]}
\end{array} \right] = \left[ \begin{array}{c}
q \cdot x_1^{[n]} \\
\vdots \\
q \cdot x_k^{[n]}
\end{array} \right] - \left[ \begin{array}{c}
(q - 1)x_1^{[0]} \\
\vdots \\
(q - 1)x_k^{[0]}
\end{array} \right] = qI x^{[n]} - x^{[0]}(q - 1)
\]

(3.20)

\( I \) denotes the identity matrix of dimension \( k \times k \), and \( x^{[0]} \) denotes the vector of symbols \( [x_1^{[0]}, \ldots, x_k^{[0]}]^t \).

Right-Hand Side of Eqn. (3.19). We now consider rewriting the right-hand side of Eqn. (3.19). In other words, we seek to know \( T_n^{-1} (Ax^{[n]} + b) \), where we interpret \( T_n^{-1} \) as being applied to each row of the vector \( Ax^{[n]} + b \). By Eqns. (3.10) and (3.11), we have

\[
T_n^{-1} \left( Ax^{[n]} + b \right) = Ax^{[n]} + T_n^{-1} (b).
\]

(3.21)

By Eqns. (3.20) and (3.21), we can rewrite the right-hand side of Eqn. (3.19) as follows:

\[
qI x^{[n]} - (q - 1)x^{[0]} = Ax^{[n]} + T_n^{-1} (b)
\]

(3.22)
Algorithm 7: Matrix Operational Calculus Recurrence Solving

**Input**: A vector of recurrence variables indexed by \( n + 1 \), \( \mathbf{x}^{(n+1)} \), an \( m \times m \) matrix \( \mathbf{A} \), a \( m \)-length vector of recurrence variables indexed by \( n \), \( \mathbf{x}^{(n)} \), and a \( m \)-length additive vector \( \mathbf{b} \)

**Output**: The closed form solution for \( \mathbf{x}^{(n)} \), in \( \mathbf{x}^{(n+1)} = \mathbf{A}\mathbf{x}^{(n)} + \mathbf{b} \)

1. \( \mathbf{b} \leftarrow \text{Simplify } \mathbf{b} ; \)
2. \( \mathbf{Opb} \leftarrow \mathbf{x}^{[0]}(q - 1) + \mathcal{T}^{-1}_n(\mathbf{b}) ; \quad */ \mathbf{Opb} \text{ is } m \times 1 */ \)
3. \( \mathbf{D} \leftarrow (q\mathbf{I} - \mathbf{A})^{-1} ; \quad */ \mathbf{D} \text{ is } m \times m. \text{ Each entry is } \frac{f}{g} \text{ with } f, g \in \mathbb{Q}[q] */ \)
4. \( \mathbf{V} \leftarrow \mathbf{D} \ast \mathbf{Opb} ; \quad */ \mathbf{V} \text{ is } m \times 1 */ \)
5. \( \mathbf{S} \leftarrow \text{initial } m \times 1 \text{ vector;} \)
6. for \( i = 1 \) to \( m \) do
7. \( \frac{f_j}{g_j} \leftarrow \text{row } i \text{ of } \mathbf{V} ; \)
8. \( \mathcal{T} \leftarrow \text{Factor } g \text{ into } g_1, \ldots, g_s \text{ irreducible factors in } \mathbb{Q}[q] ; \)
9. \( \frac{f_1}{g_1} + \cdots + \frac{f_s}{g_s} \leftarrow \text{partial-fraction decomposition of } \frac{f}{g_1 \cdots g_s} ; \)
10. \( \mathbf{R} \leftarrow 0 ; \)
11. for each \( \frac{f_j}{g_j} \) do
12. if a translation rule \( \mathcal{T}_n \left( \frac{f}{g} \right) \) applies then
13. \( \mathbf{R} \leftarrow \mathbf{R} + \mathcal{T}_n \left( \frac{f_j}{g_j} \right) \)
14. else
15. \( \mathbf{R} \leftarrow \mathbf{R} + \mathcal{IIF} \left( \frac{f_j}{g_j} \right) \)
16. end if
17. Row \( i \) of \( \mathbf{S} \leftarrow \mathbf{R} ; \)
18. end for
19. return \( \mathbf{S} \)

Eqn. (3.22) gives us a closed form for \( \mathbf{x}^{(n)} \) as a vector of elements in \( \mathcal{F} \). To obtain the corresponding solution in standard algebra, we must apply \( \mathcal{T}_n \) to each entry in the resulting vector.

Consequently, a general solution to a matrix recurrence of the form given in Eqn. (3.19) can be expressed as follows:

\[
\mathbf{x}^{(n)} = \mathcal{T}_n \left( (q\mathbf{I} - \mathbf{A})^{-1}(\mathbf{x}^{[0]}(q - 1) + \mathcal{T}^{-1}_n(\mathbf{b})) \right)
\]

(3.23)

Alg. 7 gives a procedure that implements Eqn. (3.23), where some additional details have been given in Steps 8 and 9. Unlike the univariate case, Alg. 7 may not always return a polynomial or exponential closed form for the class of matrix recurrences defined by Eqn. (3.19). In particular, note that all translation rules in Fig. 3.5 involve rational functions in which the denominator is a function with only linear terms. In other words, for there always to be an appropriate translation rule in Fig. 3.5, we must be able to find all roots of the polynomials in the denominator in Step 8. This requirement presents several challenges. First, it has high implementation complexity: it demands an implementation of algebraic numbers, including, in general, algebraic numbers that arise as roots of polynomials of degree \( \geq 5 \), which may not be expressible in terms of radicals. Second, one or more root may be complex, which means that even if there is a translation into standard algebra, the translation uses complex exponentiation, which (as shown in Ex. 3.3) is inadmissible in non-linear real arithmetic. Thus, to mitigate these difficulties, Step 8 finds all rational roots for the polynomial.
However, the restricted scope of Step 8 means that at Step 12, Alg. 7 can have rational terms that have no corresponding translations in Fig. 3.5. When that happens, Alg. 7 creates an implicitly interpreted function (IIF). An IIF is essentially an uninterpreted function, but has an association with its defining term—the rational term from Step 9 for which Fig. 3.5 had no rule.

IIFs occur exactly when the input matrix $A$ contains at least one irrational or complex eigenvalue. IIFs are beneficial because they give the wedge domain an exact and canonical characterization of the recurrence solution, albeit one with one or more “uninterpreted functions.” However, because the system retains the exact characterization of each IIF in the recurrence solver’s logic, an IIF has an explicit meaning inside the recurrence solver. Consequently, when presented with an IIF, the recurrence solver can manipulate and operate over the defining term as appropriate. For an example of the creation of an IIF, see Ex. 3.13; for a demonstration of the utility of IIFs, see Ex. 3.4.

### 3.6.4 Matrix Examples

**Example 3.12.** Consider the following matrix recurrence:

$$
\begin{bmatrix}
  x_{[k+1]} \\
  y_{[k+1]}
\end{bmatrix} =
\begin{bmatrix}
  1 & 1 \\
  -2 & 4
\end{bmatrix}
\begin{bmatrix}
  x_{[k]} \\
  y_{[k]}
\end{bmatrix} +
\begin{bmatrix}
  0 \\
  1
\end{bmatrix}
$$

For this example, we have

$$
\mathbf{b} =
\begin{bmatrix}
  0 & 1
\end{bmatrix}^t,
\mathbf{x}^{[0]} =
\begin{bmatrix}
  x_{[0]} \\
  y_{[0]}
\end{bmatrix}^t,
T_n^{-1}(\mathbf{b}) =
\begin{bmatrix}
  0 & 1
\end{bmatrix}^t.
$$

The additive vector, $\mathbf{b}$ in this recurrence is already simplified, and the translation, $T_n^{-1}(\mathbf{b})$, is given above. Thus, performing Steps 1 and 2 leaves us with the vector

$$
\begin{bmatrix}
  x^{[0]}(q - 1) \\
  y^{[0]}(q - 1) + 1
\end{bmatrix}^t.
$$

Step 3 symbolically computes

$$
\begin{bmatrix}
  q - 1 & -1 \\
  2 & (q - 4)
\end{bmatrix}^{-1} =
\begin{bmatrix}
  \frac{q - 4}{q^2 - 2q + 5} & \frac{1}{q^2 - 2q + 5} \\
  -\frac{2}{q^2 - 2q + 6} & \frac{q - 1}{q^2 - 2q + 6}
\end{bmatrix}
$$

After Step 4, $\mathbf{V}$ equals the following vector:

$$
\begin{bmatrix}
  \frac{(q^2 - 5q + 4)x^{[0]} + (q - 1)y^{[0]} + 1}{q^2 - 5q + 6} \\
  \frac{(q^2 - 2q + 1)y^{[0]} - (q - 1)2x^{[0]} + q - 1}{q^2 - 2q + 6}
\end{bmatrix}^t.
$$
The term \( q^2 - 5q + 6 \) can be factored into \((q - 3)(q - 2)\), so Steps 8 and 9 produce

\[
\begin{bmatrix}
\frac{1 - 2x^{(0)} + 2y^{(0)}}{q - 2} & -\frac{1 + 2x^{(0)} - y^{(0)}}{q - 2} + x^{(0)} \cdot 2 - 4x^{(0)} + 4y^{(0)} + \frac{1 + 2x^{(0)} - y^{(0)}}{q - 2} + y^{(0)}
\end{bmatrix}^t
\]

Finally, Step 13 applies \( \mathcal{T}_k \) to each term of each entry of the above vector to obtain closed-form expressions for both \( x^{[k]} \) and \( y^{[k]} \).

\[
x^{[k]} = \frac{1}{2} + (2x^{[0]} - y^{[0]} - 1)2^k + (-x^{[0]} + y^{[0]} + \frac{1}{2})3^k \quad y^{[k]} = (2x^{[0]} - y^{[0]} - 1)2^k + (-2x^{[0]} + 2y^{[0]} + 1)3^k
\]

**Example 3.13.** Consider the following matrix recurrence:

\[
\begin{bmatrix}
y^{[k+1]} \\
z^{[k+1]}
\end{bmatrix} =
\begin{bmatrix}
1 & 1 \\
\cdot9 & 1
\end{bmatrix}
\begin{bmatrix}
y^{[k]} \\
z^{[k]}
\end{bmatrix}
\]

To simplify the example, suppose we know that the values of \( y^{[0]} \) and \( z^{[0]} \) are 1 and 0, respectively. If this information is used in Steps 1–4 of Alg. 7, we are left with the following equation:

\[
\begin{bmatrix}
y^{[k]} \\
z^{[k]}
\end{bmatrix}^t = \begin{bmatrix}
\frac{(q-1)^2}{q-1} & -\frac{9(q-1)}{q-1}
\end{bmatrix}
\]

Unfortunately, \( q^2 - 2q + 10 \) is irreducible in \( \mathbb{Q}[q] \), and so we are not able to provide a solution for the recurrence only using rational coefficients. Step 9 (partial-fraction decomposition) yields

\[
\begin{bmatrix}
y^{[k]} \\
z^{[k]}
\end{bmatrix}^t = \begin{bmatrix}
1 - \frac{9}{q^2 - 2q + 10} & \frac{9}{q - 2q + 10}
\end{bmatrix}
\]

We can pull out the constant coefficients in the above closed form to yield the following closed-form expressions for \( y^{[k]} \) and \( z^{[k]} \).

\[
\begin{bmatrix}
y^{[k]} \\
z^{[k]}
\end{bmatrix}^t = \begin{bmatrix}
1 - 9\mathcal{T}_k \left( \frac{1}{q^2 - 2q + 10} \right) & -9\mathcal{T}_k \left( \frac{q}{q^2 - 2q + 10} \right) + 9\mathcal{T}_k \left( \frac{1}{q^2 - 2q + 10} \right)
\end{bmatrix}.
\]

(3.24)

There is no direct translation for the remaining terms. Thus, at this point Alg. 7 creates IIFs for each term involving \( \mathcal{T}_k \) to use in the solutions returned to the wedge domain. In the example above, the returned values are \( 1 - 9\text{IIF}\left(\frac{1}{q^2 - 2q + 10}\right)(k) \) for \( y^{[k]} \) and \(-9\text{IIF}\left(\frac{q}{q^2 - 2q + 10}\right)(k) + 9\text{IIF}\left(\frac{1}{q^2 - 2q + 10}\right)(k) \) for \( z^{[k]} \).
3.7 Experiments

Our techniques are implemented in a tool called ICRA, which uses Z3’s UFLRA solver [42], and Apron’s NewPolka polyhedron domain [76]. Our experiments were designed to answer the following questions:

1. How often is ICRA able to prove assertions for programs that have non-linear invariants?

2. How often is ICRA able to generate upper bounds on the resource usage (e.g., running time) of programs?

Programs with Non-Linear Invariants. To address the question of ICRA’s capabilities for proving assertions in programs that have non-linear invariants, we used a suite of 77 small programs with true assertions. These programs can be divided into three categories:

- 46 programs are from a suite used to test the invariant-generating tool HOLA [44].

- 21 programs have assertions that concern functional-correctness properties. These programs all have some non-linear structure. We created these benchmarks to test ICRA’s non-linear capabilities.

- 10 programs are self-composed programs in the style described in § 3.2. The Fibonacci program in § 3.2 is an example of a program in this category. The other nine programs were obtained from [8, 13, 132].

We compared ICRA on these 77 programs against three state-of-the-art software model checkers: Ultimate Automizer [68] from SV-COMP16 and CPAchecker [17] from SV-COMP17, both based on predicate abstraction; and SeaHorn [66] version 0.1.0, a Horn-clause solver based on IC3. Timings (with a timeout limit of 60 seconds) were taken on a virtual machine (using Oracle VirtualBox), with a guest OS of Ubuntu 14.04, host of Microsoft Windows 7 Enterprise, and a 3.2 GHz quad-core Intel Core i5-4570 host CPU.

Tab. 3.1 shows that ICRA performs significantly better than the other tools on these 77 programs. In every category, ICRA was able to prove more assertions than any other tool, except for SeaHorn on the HOLA programs. Also, in terms of time ICRA was significantly faster than the other tools in all categories. This
result is mostly due to ICRA having fewer timeouts compared with the other tools. ICRA had only 2 timeouts, where as SeaHorn had the next fewest with 17.

Resource-Bound Generation. To address the question of ICRA's capabilities for generating upper bounds on resource usage (in this case, upper bounds on running time), we ran experiments on 8 small benchmark programs. Five programs were taken from Brockschmidt et al. [25], two examples are from § 3.2 (Ex. 3.1 and Ex. 3.2), and one additional program is described below.

Example 3.14. The following program illustrates the relevance of non-linear invariant generation to resource-bound analysis. To upper-bound this program’s running time, an analyzer must determine an upper bound on c that holds after the first loop:

```c
void main(int a) {
    int b = 0, c = 1; assume(a >= 1 && c>= 1);
    while(a > b) { b ++; c += a; if (nondet()) { return; } }
    while(c >= 0) { c --; if (nondet()) { return; } }
}
```

We compared ICRA on these 8 programs against two other resource-bound analysis tools, CoFloCo [57] and PUBS [3]. Timings (with a timeout of 60 seconds) were taken on a virtual machine (using Oracle VirtualBox), with a guest OS of Ubuntu 14.04, host OS of Fedora Core 6, and a 3.2 GHz quad-core Intel Core i5-4570 host CPU.

The number of upper bounds on running time generated by ICRA was 7; by CoFloCo, 7; and by PUBS, 4. The total time taken by ICRA was 11.0 seconds; by CoFloCo, 2.5 seconds, and by PUBS, 0.5 seconds. ICRA generated bounds on three examples where PUBS did not generate a bound, and on one example where CoFloCo did not generate a bound. From these experiments, we conclude that ICRA shows promise as a resource-bound-analysis tool.

3.8 Related Work

Non-Linear Abstract Domains. There is a significant amount of prior work for reasoning about uninterpreted function symbols [65, 28, 59] and non-linear arithmetic [103, 33, 10, 63]. A common theme in this work, including our own, is to introduce new dimensions in some relational abstract domain (such as polyhedra [37] or octagons [101]) to represent the values of terms that cannot be expressed in the domain per se.

Various techniques can be used to propagate information about the alien terms (borrowing the terminology of Chang and Leino [28]) to the base domain. For example, Chang and Leino's congruence-closure abstract domain extends a base domain with terms built from arbitrary function symbols, and uses congruence-closure techniques to infer equations between terms [28]. Colón [33] extends the domain of affine relations
with dimensions corresponding to all monomials less than some fixed degree, and infers new affine relations by closing under degree-bounded products; Bagnara et al. [10] generalize this idea, extending polyhedra to reason about polynomial inequalities. Gulavani and Gulwani [63] present a framework for extending an abstract domain with alien functions using user-provided axioms for inference. As a specific instance, they extend polyhedra with terms involving multiplication, log, square root, and exponentiation, and give a set of axioms for reasoning about them. The wedge domain uses many of these ideas: the equational-saturation procedure for the wedge domain (Alg. 3) infers equalities using congruence closure (like Chang and Leino [28]) and Gröbner-basis techniques (which generalize [33]); the wedge domain deduces inequalities by applying a set of inference rules (like Gulavani and Gulwani [63]); the Product rule generalizes the degree-bounded products of Bagnara et al. [10].

In abstract-interpretation terms, such techniques are using semantic reductions [35] to improve the representation of an abstract-domain element. It is often useful to apply semantic reductions repeatedly, as a “best-effort” method to improve the representation, à la Granger’s technique of local decreasing iteration [62].

The design goals of wedges differ considerably from those of previous work, however. In particular, previous work has focused on abstract domains for iterative abstract interpreters, which compute loop invariants by evaluating the program under an abstract semantics until convergence on a property that abstracts all reachable states. Most such domains require fixing a priori the dimensions of the relational abstract domain (e.g., to all monomials of degree \( \leq 2 \)) to enforce convergence. In contrast, wedges are designed for recurrence analysis, wherein the fundamental problem of interest is symbolic abstraction (§ 3.4.3) (a problem not addressed by previous work). The association between non-linear terms and dimensions is more fluid in wedges, because there is no iterative process that we need to force to converge. Moreover, since abstract-domain operations are relatively infrequent in our analysis (in comparison to iterative program analysis), wedges are able to make use of computationally intensive operations, such as Gröbner-basis computation.

Other Methods for Non-Linear Invariant Generation. A variety of other methods have been developed for generating non-linear invariants [119, 106, 127]. Srikanth et al. [127] recently developed a method for determining if a program satisfies a given resource bound, which can be expressed using polynomials, exponentials, and logarithms. The approach uses an interpolating theorem prover for non-linear arithmetic. The method is property-directed, and thus fundamentally non-compositional.

Other Program Analyzers that Perform Recurrence Analysis. There are a number of program analyzers that find loop invariants by solving recurrence relations [4, 20, 53, 116, 9, 94, 2, 19, 7, 23, 77, 49, 106, 85, 73].

The techniques presented in this chapter build upon previous work on compositional recurrence analysis
(CRA) [49] and its interprocedural extension ICRA [85]. CRA uses recurrence analysis to approximate the transitive closure of arbitrary loops (rather than compute it exactly for a limited class of loops). Inspired by the methods used in so-called “Newtonian program analysis” [46, 113], the ICRA analysis framework of Kincaid et al. [85] handles loops and recursion in a uniform way. In essence, their technique reduces arbitrary recursive programs (including ones with non-linear mutual recursion) to a sequence of ever-more-precise tail-recursive problems over a more complicated abstract domain. The latter are then analyzed using only slight generalizations of the CRA technique for extracting and solving recurrence relations for loops [49].

In this chapter, we present techniques that improve (I)CRA’s ability to generate non-linear invariants. In particular, [49]’s formulation of CRA is capable of extracting linear recurrence relations that have polynomial closed forms. It has limited support for reasoning about non-linear arithmetic: it uses a linearization technique that over-approximates a non-linear formula by a linear one. For example, consider the formula

\[ x' = x + 1 \land y' = y + x \times x \land 0 \leq x \leq 5. \]

Farzan and Kincaid’s algorithm would compute the following linearization:

\[ x' = x + 1 \land y \leq y' \leq y + 25, \]

and thus lose the precise recurrence for \( y \): \( y' = y + x \times x \). In contrast, the method we present in this chapter handles recurrence relations that involve polynomials and exponentials—and whose closed-form solutions include polynomials and exponentials (and therefore logarithmic relationships are implicit, as well). Consequently, this chapter’s techniques are different from those used by Farzan and Kincaid, for both extracting and solving recurrences. The wedge domain retains non-linear information rather than abstracting it away, thus allowing us to extract the recurrence \( y' = y + x \times x \) for the example above.

Another distinction that can be drawn between the techniques of this chapter and Farzan and Kincaid’s algorithm is that we extract and solve matrix recurrences. Matrix recurrences generalize the stratified recurrences of Farzan and Kincaid, which (like our stratified recurrences) orders recurrences into layers to aid solving. Unlike our notion of stratified recurrence (which is essentially a mechanism to cope with non-linear arithmetic), Farzan and Kincaid consider only linear arithmetic—all Farzan/Kincaid-style stratified recurrences appear in our matrix recurrences at stratum 0.

In both this chapter and Farzan and Kincaid [49], the key operation used for extracting recurrences is symbolic abstraction [110, 134]. For more about symbolic abstraction in program analysis, see Thakur [133], Reps and Thakur [112].
Compared to other program analyzers that are based on the principle of solving recurrence equations, the advantage of the technique described in this chapter is that it can analyze general loops, including loops that contain (i) branching, (ii) nested loops, and (iii) non-deterministic assignments.

An important distinction between the recurrence solver presented in § 3.6 and others that appear in the literature is that our solver will produce closed forms in terms of IIFs even if the closed form cannot be represented (e.g.) as an elementary function. Thus, for example, while the solver from Rodríguez-Carbonell and Kapur [116] accepts the same class of stratified recurrences as our algorithm (§ 3.5.2), it only succeeds in computing closed forms in the case that all eigenvalues of every transformation matrix are real.

There exist algorithms for solving even broader classes of recurrences, such as the work of Humenberger et al. [73]. For example, their technique can find closed forms that include factorials, whereas ours cannot. The methods of Rodríguez-Carbonell and Kapur [116] and Humenberger et al. [73] both search for polynomial equalities, whereas our technique can find invariants that are not polynomial equalities; also, neither Rodríguez-Carbonell and Kapur [116] nor Humenberger et al. [73] handle nested loops, whereas our technique does. Below, we show two examples of single-path loops that can be handled by our implementation (ICRA), but not by Aligator, which is the implementation of the techniques from Humenberger et al. [73].

```c
int x = 1, y = 1; for(n = 0; n < N; n++) { x *= 2; y *= 3; }
```

```c
int a = 5, b = 3, c = 1; for(n = 0; n < N; n++) { a *= a; b += a; c -= a; }
```

In the first loop, no polynomial equality holds between \(x\) and \(y\), so Aligator finds no invariant; ICRA is able to prove that \(x \leq y\). In the second loop, ICRA cannot find closed-form solutions for \(a\), \(b\), or \(c\), but it is able to prove the invariant \(b + c = 4\) by finding a closed form for the sum \((b' + c' = b + c)\); Aligator finds no invariant (it reports that the loop is “not P-solvable”). This example illustrates an advantage of ICRA: when it encounters a recurrence that it cannot solve, it may still produce a non-trivial overapproximation, unlike Aligator, which finds no invariants when it cannot solve the recurrences.

Recurrence Relations that Define C-Finite Sequences. The sequences defined by stratified recurrence relations fall into the class of C-finite sequences—sequences that satisfy linear recurrence relations with constant coefficients [81, Chapter 4].

**Definition 3.15.** ([94, Def. 3.2][81, §4.2]) A **C-finite recurrence** is a homogeneous linear recurrence equation with constant coefficients

\[ f[n+r] = a_{r-1}f[n+r-1] + \ldots + a_1f[n+1] + a_0f[n], \]

where \(n \in \mathbb{N}\), and \(a_0, \ldots, a_{r-1}\) are constants with \(a_0 \neq 0\). The recurrence is of order \(r \in \mathbb{N}\).

A sequence \((f[n])_{n=0}^{\infty}\) is a **C-finite sequence** if it satisfies a C-finite recurrence.

It is natural to ask whether stratified recurrence relations and C-finite recurrences are “equivalent.”
The sequences defined by stratified recurrence relations are C-finite sequences. The essential argument appears in Kovács [94]: a system of recurrences \( x[n] = Ax[n-1] + f(n) \) (with \( f(n) \) a sum of polynomials multiplied by exponentials) can be transformed into a set of inhomogeneous C-finite recurrences, \( x_i[n] = a_{i,1}x_i[n] + \cdots a_{i,k}x_i[n-k] + g(n) \) (where \( g(n) \) has the same form as \( f(n) \)), each of which can in turn be transformed into a homogeneous C-finite recurrence.

The question is whether every C-finite sequence \( S \) is definable by a stratified recurrence relation. Suppose that \( S \) satisfies a C-finite recurrence \( R \) of order \( r \). Then one can construct a program \( P \) (with a loop over loop-counter \( n \)) that computes \( S \): the program will have one variable that holds the \( n^{th} \) element of \( S \), and \( r-1 \) variables to hold onto the “lagging” values from prior iterations. All assignments would have linear expressions on the right-hand side. From \( P \), one obtains a matrix recurrence \( M \).

In terms of the components of ICRA, the recurrence-extraction method from § 3.5 would extract from \( P \) the matrix recurrence \( M \) (at stratum 0). OCRS would solve \( M \), and would return an answer, which would contain one or more IIFs if the (standard) closed-form solution involves non-rationals (irrational numbers or complex numbers).

In this sense, § 3.6 does not represent a method that can solve recurrences beyond what existing solvers can handle: there are other algorithms that might be applied to “solve” the recurrence relations that ICRA is capable of extracting.

The above equivalence argument must be tempered by the fact that not every program \( Q \) with a loop containing a variable that generates a C-finite sequence will be in the “normal form” enjoyed by \( P \) in case \( \Leftarrow \). For such a \( Q \), ICRA would extract a system of stratified matrix recurrences (involving multiple strata). If one knew the C-finite recurrence \( R_Q \) for \( Q \), one could construct another program \( Q’ \) in the normal form from case \( \Leftarrow \); for \( Q’ \), ICRA would extract a single matrix recurrence at stratum-0. (ICRA does not attempt to perform such a source-to-source transformation.)

Moreover, the notion of “solving” is predicated on the definition of “closed form,” where closed forms need to be consumable by the client of the solver. In our work, the ability to process the output of the recurrence solver (§ 3.6) as the input to the recurrence extractor (§ 3.5) is essential for handling nested loops, as illustrated by Ex. 3.1. Although it is possible for the ICRA recurrence extractor to extract recurrences that the solver cannot solve, in practice, the solver presented in § 3.6 is well-matched to the class of recurrences that are extracted by the algorithm given in § 3.5.

Closed forms produced by the method given in § 3.6 differ from those used in previous work (even if the class of sequences does not). Classical algorithms admit algebraic numbers, including complex numbers, in closed-form solutions. However, in analyzing an outer loop, the wedge domain needs to make use of answers
obtained from the analysis of an inner loop. Because the wedge domain uses theorem proving modulo the theory of linear real arithmetic as a subroutine in the recurrence-extraction algorithm (§3.4 and §3.5), it would obviously not be able to make use of answers that involve complex numbers. Similarly, it would not be able to make use of answers that involve irrational numbers because the theory of linear real arithmetic admits only rational coefficients. Consequently, the solver described in § 3.6 disallows irrational numbers and complex numbers from appearing in solutions, but allows there to be IIFs.

Finally, there is a syntactic difference between stratified recurrence relations and C-finite recurrences. The syntactic form of a system of stratified recurrence relations indicates explicitly the order in which individual recurrence equations should be solved. For instance, consider the program shown below on the left, from which ICRA would extract the loop-body transition formula shown on the right.

```c
while(*) { z = z + x*y;
    tmp = y;
    y = 2*x;
    x = x + tmp + 1; }
```

\[\phi \overset{\text{def}}{=} x' = x + y + 1 \land y' = 2 \times x \land z' = z + x \times y\]

The recurrence-extraction technique from § 3.5 would identify the following system of stratified recurrence relations:

- **Stratum 0:**
  \[x[n] = x[n-1] + y[n-1] + 1\]
  \[y[n] = 2x[n-1]\]

- **Stratum 1:**
  \[z[n] = z[n-1] + x[n-1]y[n-1]\] (3.25)

The recurrence equations at stratum 0 are solved first; their solutions are used to solve the recurrence equation at stratum 1.

The x, y, and z sequences that satisfy Eqn. (3.25) are all C-finite sequences. For instance, by writing

\[x[n] - x[n-1] - 2x[n-2] - 1 = 0\]
\[x[n+1] - x[n] - 2x[n-1] - 1 = 0\]

and subtracting the first line from the second, we see that the x sequence must satisfy

\[x[n+1] - 2x[n] - x[n-1] + 2x[n-2] = 0,\] (3.26)

which is equivalent to the 3\(^{rd}\)-order C-finite recurrence

\[x[n+3] = 2x[n+2] + x[n+1] - 2x[n].\]
The recurrence equation for \( z \), at stratum 1, is syntactically not a C-finite recurrence because it contains the non-linear term \( x^{[n-1]}y^{[n-1]} \).  

The syntactic distinction between stratified recurrence relations and C-finite recurrences is important from the viewpoint of recurrence extraction. In particular, the loop-body formula \( \phi \) does not entail Eqn. (3.26): \( x^{[n+1]} \) and \( x^{[n-2]} \) do not belong to the vocabulary of \( \phi \) (which refers to only two copies of \( x \)—one in the pre-state and one in the post-state).

**Generating Functions.** Operational calculus is a transform method: a recurrence relation is converted into an algebraic problem; the algebraic problem is solved by algebraic manipulations; a solution in ordinary algebra is read out of the solution to the algebraic problem. Generating functions [138] are another transform method that can be used to solve recurrence relations [55]. With generating functions, a sequence \( a = \langle a_0, a_1, a_2, \ldots \rangle \) is represented by the sum (or power series) \( \sum_{n=0}^{\infty} a_n x^n \). Multiplication of two sequences is performed as the following convolution: \( \sum_{n=0}^{\infty} a_n x^n \ast \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} \left( \sum_{i=0}^{n} a_i b_{n-i} \right) x^n \). Thus, the multiplication operator for generating functions is a convolution, whereas the multiplication in operational calculus is a convolution difference (cf. Eqn. (3.4)).

Operational calculus and generating functions create different terms for subsequent algebraic manipulation. Simpler terms might lead to a faster and simpler solver. However, as shown by the following examples, neither technique can be said to have an advantage in this respect:

**Operational calculus creates simpler terms:** In operational calculus, the constant sequence \( \lambda_n \cdot c \) is represented by \( c = \langle c, c, \ldots \rangle \), whereas the generating function is \( \frac{c}{1-x} \). The respective values for the sequence \( \lambda_n \cdot n^2 \) are \( \frac{2}{(q-1)^2} + \frac{1}{q-1} \) and \( \frac{x^{[x+1]}}{(1-x)^2} \).

**Generating functions create simpler terms:** The generating function for the geometric sequence \( \lambda_n \cdot \alpha^n \) is \( \frac{1}{1-\alpha x} \), whereas the operational-calculus value is \( \frac{1}{q-\alpha} \).

**Operational calculus and generating functions create terms of similar complexity:** The left-shift operators are \( q \) and \( \frac{1}{x} \), respectively; the right-shift operators are \( v \) and \( x \), respectively.

**Operational Calculus.** We are not aware of other work that solves recurrences automatically using operational calculus; Berg [15] discusses recurrence solving in the style of a mathematics textbook, but provides no explicit algorithm. § 3.6.2 describes our mechanization of Berg’s approach. § 3.6.3 concerns an extension to operational calculus not found in Berg (or elsewhere, as far as we know).

---

6A C-finite recurrence can contain only linear terms; however, a C-finite sequence may satisfy a recurrence relation that contains a non-linear term.
Chapter 4

Control-Flow Refinement

Algebraic program analyses compute information about a program’s behavior by first (a) computing a valid path expression—i.e., a regular expression that recognizes all feasible execution paths (and usually more)—and then (b) interpreting the path expression in a semantic algebra that defines the analysis. There are an infinite number of different regular expressions that qualify as valid path expressions, which raises the question “Which one should we choose?” While any choice yields a sound result, for many analyses the choice can have a drastic effect on the precision of the results obtained. This chapter investigates the following two questions:

1. What does it mean for one valid path expression to be “better” than another?

2. Can we compute a valid path expression that is “better,” and if so, how?

We show that it is not satisfactory to compare two path expressions $E_1$ and $E_2$ solely by means of the languages that they generate. Counter to one’s intuition, it is possible for $L(E_2) \subseteq L(E_1)$, yet for $E_2$ to produce a less-precise analysis result than $E_1$—and thus we would not want to perform the transformation $E_1 \rightarrow E_2$. However, the exclusion of paths so as to analyze a smaller language of paths is exactly the refinement criterion used by some prior methods.

In this chapter, we develop an algorithm that takes as input a valid path expression $E$, and returns a valid path expression $E'$ that is guaranteed to yield analysis results that are at least as good as those obtained using $E$. While the algorithm sometimes returns $E$ itself, it typically does not: (i) we prove a no-degradation result for the algorithm’s base case—for transforming a leaf loop (i.e., a most-deeply-nested loop); (ii) at a non-leaf loop $L$, the algorithm treats each loop $L'$ in the body of $L$ as an indivisible atom, and applies the leaf-loop algorithm to $L'$; the no-degradation result carries over to (ii), as well. Our experiments show that the technique has a substantial impact: the loop-refinement algorithm allows the implementation of
Compositional Recurrence Analysis to prove over 25% more assertions for a collection of challenging loop micro-benchmarks.

4.1 Introduction

Tarjan [131] introduced the idea of using regular expressions as a kind of “most-general program-analysis method.” This style of analysis is also sometimes referred to as algebraic program analysis [48, 85, 89] (see § 2.1). Specific program-analysis problems are solved by first solving the path-expression problem: a program’s control-flow graph (CFG) is considered to be a finite-state machine in which CFG nodes are states, and each edge is labeled by an alphabet symbol unique to that edge. Tarjan’s path-expression method [130] creates for each node \( n \) a regular expression \( R_n \) whose language, \( L(R_n) \), is the set of all paths from the CFG’s start node to \( n \). The “client” program-analysis problem is then solved by evaluating each regular expression \( R_n \) bottom-up, using an interpretation in which the regular-expression operators \( + \), \( \cdot \), and \( \ast \)—now treated as syntactic operators—are interpreted as some suitable (sound) operations, \( \oplus \), \( \otimes \), and \( \oslash \), respectively, in the analysis domain.

The idea of applying transformations to the intermediate representation (IR) of a program as a way to improve the results of static analysis has a long history (see § 4.7). For instance, many people have taken advantage of loop unrolling as a way to potentially improve analysis precision. In the nomenclature of algebraic program analysis, the question can be phrased as follows:¹

\[
\text{There are many different regular expressions that represent the same set of concrete-action sequences; are some of these regular expressions better than others? In other words, can the evaluation of one regular expression produce more precise analysis results, compared with a structurally different—but equivalent—regular expression? (4.1)}
\]

Not surprisingly, the answer is “Yes. At the level of analysis, the structure of the regular expression matters.”

In this chapter, we focus on a particular sub-problem related to loops: given a regular expression of the form \( R = (r_1 + \ldots + r_n)\ast \) and a set of forbidden (infeasible) subpaths, compute a new regular expression \( R' \) such that (a) the forbidden subpaths are eliminated from \( R' \), and (b) the precision of the analysis results obtained with \( R' \) are at least as good as those obtained with \( R \).

Past work that has studied refinement has worked only with requirement (a). For that situation—studied in the context of iterative program analysis under the name control-flow refinement [11, 64, 125, 57]—the solution exploits the observation that the syntax of a loop (or the structure of the IR that represents the loop)

¹“Equivalent” is used in the sense that both expressions represent the same set of concrete-action sequences.
may cause more paths through the loop to be considered than are actually possible. A set of infeasible paths can be characterized by a set of forbidden subwords; consequently, the language of paths not containing forbidden subwords is regular, and therefore recognized by a regular expression, say F. One can then obtain the analysis results for the loop by evaluating F instead of R.

Counter to one’s intuition, however, the analysis of F can yield a less-precise summary of the loop than merely evaluating \((r_1 + \ldots + r_m)^*\). (See § 4.2.2.) This conundrum suggests the following problem: find a regular expression R' that is between F and R—i.e., \(L(F) \subseteq L(R') \subseteq L(R)\)—and which gives analysis results that are no worse than those obtained with R. In this chapter, we give an algorithm for computing such an R'.

As just mentioned, it is unsatisfactory to compare regular expressions solely by means of the languages that they generate: the comparison must account for the fact that they are both interpreted in the abstract domain A in use. In particular, we wish to establish that path expression R' yields A-analysis results that are at least as good as those obtained with R. Moreover, we wish to do so by some means other than (i) creating R', (ii) evaluating R' in A, and (iii) comparing the result to the value of R evaluated in A. Instead, we wish to reason about the relative precision of the A-interpretations of R and R'. We address this issue by axiomatizing the properties of an abstract domain that influence the relative precision obtained using different regular expressions. Our name for the formalism that we introduce is “pre-Kleene algebra” (PKA) (§ 4.3). By assuming that abstract domain A satisfies the PKA axioms, we can use algebraic reasoning to prove properties of the transformation algorithm.

The second problem addressed in this chapter is how to use our understanding of the simple case above in a program analyzer. There are two relevant problems: (1) how to generalize the results to arbitrary regular expressions, and (2) how to recognize infeasible sub-paths. We exploit the compositional nature of algebraic program analysis to provide answers to both questions. The result is a generic algorithm that provably does not degrade the precision of any algebraic program analysis meeting our conditions, and, as our experiments show, often improves the precision.

Our experiments show that the technique has a substantial impact: the loop-refinement algorithm allows the implementation of Compositional Recurrence Analysis (CRA) [49] to prove over 25% more assertions for a collection of challenging loop micro-benchmarks.

**Contributions.** Our work makes the following contributions:

- We prove a no-degradation result (Thm. 4.14) for the algorithm’s base case, which transforms a leaf loop.

- At non-leaf loops, the algorithm applies the leaf-loop algorithm greedily, bottom-up (§ 4.5). We show that the non-degradation result carries over to the non-leaf loops, as well. Consequently, our bottom-up
algorithm for refinement is guaranteed to yield analysis results that are at least as good as those obtained without refinement (Thm. 4.16).

For an algebraic-analysis framework, the benefits of this approach are three-fold:

1. The algorithm can be used to (often) improve the precision of any algebraic program analysis that uses an abstract domain that satisfies the PKA axioms.

2. The refinement method can be incorporated in a uniform way. Refinement is invoked at the level of the pre-existing star operator, which only has to consider regular expressions of the form R*, where R has already been analyzed. Moreover, checks for infeasible paths in R can be made in the abstract domain (e.g., using an SMT solver). The infeasible paths can be used to refine R* to create an alternative expression R'. Then R' is evaluated in place of R*, and the result is used in evaluating the parent expression.

3. Thanks to the analysis method developed by Kincaid et al. [85], the algorithm also applies to programs with recursive procedure calls.

- We have incorporated an implementation of our refinement algorithm into the algebraic-analysis tool described by Farzan and Kincaid [49]. § 4.6 presents empirical results that demonstrate the practical benefits of our method.

§ 4.2 motivates the problem of path-expression refinement by showing how the precision of an example analysis can be increased by soundly modifying the path-expression that is used. It also gives an example showing that naive refinement can lead to worse analysis results. § 4.3 gives background on Kleene algebras, algebraic program analysis, and control-flow refinement, and introduces the formalism of pre-Kleene algebras. § 4.7 discusses related work.

### 4.2 Overview

In this section, we present two motivating examples based on a Tarjan-style analysis. We consider an example abstract domain equipped with extend, combine, and iteration operators, denoted by \( \otimes \), \( \oplus \), and \( \odot \), respectively. For this example, the elements of the abstract domain are two-vocabulary transition formulas in Presburger arithmetic, \( \phi(x, x') \), and the \( \otimes \) and \( \oplus \) operations are the same operations defined in Eqn. (2.2). For the \( \odot \) operator, we consider a two-step process. To compute \( \phi^{\odot} \), first compute the best approximation of \( \phi \) as an octagonal relation \( w(x, x') \) using optimization modulo theories [123, 97]; then compute an existential Presburger formula representing the transitive closure of \( w(x, x') \) using the algorithm of Bozga et al. [22].
Consider the program in Fig. 4.1(a). There are two paths through the loop’s body. Let A denote the path in which the then branch is avoided, and B denote the path that follows the then branch. Let $\phi_A$ and $\phi_B$ be the transition formulas for paths A and B, respectively.

$$\phi_A \overset{\text{def}}{=} (x < 100 \land x' = x + 1 \land y' = y)$$

$$\phi_B \overset{\text{def}}{=} (x < 100 \land x' = x + 1 \land x' > 50 \land y' = y + 1)$$

The simplest path expression that represents the set of paths from just before the loop to just after the loop is $R \overset{\text{def}}{=} (A + B)^*$. To analyze the loop, we evaluate $R$ with A and B replaced by $\phi_A$ and $\phi_B$, respectively; + replaced by $\oplus$; and * replaced by $\odot$: $(\phi_A \oplus \phi_B) \odot = (\phi_A \lor \phi_B) \odot$. To evaluate $(\phi_A \lor \phi_B) \odot$, we first abstract $\phi_A \lor \phi_B$ to create an over-approximating formula for the loop-body’s transition relation. In particular, we create the most-precise over-approximation of $\phi_A \lor \phi_B$ that is in conjunctive form:

$$w_{\text{body}} = (x < 100 \land x' = x + 1 \land y' = y \land y' - y \geq 0).$$

In $w_{\text{body}}$, we have only an inequality that relates $y$ and $y'$.

Even with the most-precise computation of the closure $w_{\text{body}}^\odot$, the best property that can be deduced for the program’s final state from $(x = 0 \land y = 50) \odot w_{\text{body}}^\odot \odot (x \geq 100)$ is $x = 100 \land 100 \geq y \geq 50$. However, it is not difficult to see that $x = 100 \land y = 100$ always holds after the program executes.

The main issue is that $(A + B)^*$ describes a larger set of paths than the set of feasible execution paths. In particular, as observed by Sharma et al. [125], once the predicate $x > 50$ is true on some iteration, it will
continue to be true on all later iterations: path A will never execute after path B. One way that this property can be discovered is by showing that the formula

\[ \phi_B \otimes \phi_A = (x < 100 \land x + 1 > 50 \land x + 1 < 100 \land x + 2 < 50 \land x' = x + 2 \land y' = y + 1) \]

is unsatisfiable, which can be established easily by an SMT solver. Consequently, we can refine the regular expression R to \( R' \equiv A^*B^* \), which is both (i) sound with respect to the program’s semantics, and (ii) more closely represents the feasible execution paths of the program. In other words, the set of feasible paths of the program is contained in \( L(A^*B^*) \), and \( L(A^*B^*) \subseteq L((A + B)^*) \).

Now consider the evaluation of \( (x = 0 \land y = 50) \otimes \phi_A^w \otimes \phi_B^w \). It happens that \( \phi_A \) and \( \phi_B \) are octagons already, so \( \phi_A^w \) and \( \phi_B^w \) can be computed via the methods of Bozga et al. [22]. \( (x = 0 \land y = 50) \otimes \phi_A^w \) yields \( (x = 50 \land y = 50) \) and \( (x = 50 \land y = 50) \otimes \phi_B^w \) becomes \( (x = 100 \land y = 100) \). Thus the analysis of \( A^*B^* \) allows us to conclude \( x = 100 \land y = 100 \) after the loop. This example shows that a simple rewrite of the regular expression being analyzed can produce a more precise static-analysis result.

### 4.2.2 A Transformation that Leads to a Worse Result

Consider the program in Fig. 4.1(b). The loop body has three paths, which we call A, B, and C. The loop can be described by the regular expression \( R_1 \equiv (A^*B^*). \) Let \( \phi_A, \phi_B, \) and \( \phi_C \) be two-vocabulary formulas representing the transition relations for paths A, B, and C, respectively. We have

\[
\begin{align*}
\phi_A & \equiv (\text{pos} > 0 \land i' = i + 1 \land j' = j + 1 \land \text{pos}' = 1) \\
\phi_B & \equiv (\text{pos} > 0 \land i' = i + 1 \land j' = j + 1 \land \text{pos}' = 0) \\
\phi_C & \equiv (\text{pos} \leq 0 \land i' = i + 1 \land j' = j + 1 \land \text{pos}' = 1)
\end{align*}
\]

Let us now evaluate \( R_1 \) in the abstract domain defined at the beginning of this section: \( (\phi_A \oplus \phi_B \oplus \phi_C)^w \). When we abstract the disjunction \( \phi_A \lor \phi_B \lor \phi_C \) we obtain the formula \( w_{\text{Body}} \):

\[
w_{\text{Body}} \equiv (i' = i + 1 \land j' = j + 1 \land 0 \leq \text{pos}' \land \text{pos}' \leq 1).
\]

The closure of this formula, \( w_{\text{Body}}^w \), along with the fact that \( i = 0 \land j = 0 \) before the loop, implies that \( i = j \) holds after the loop.

Now consider a plausible alternative way of analyzing the program in Fig. 4.1(b). By inspecting the individual paths of the loop, we observe that \( \phi_A \otimes \phi_C, \phi_B \otimes \phi_A, \phi_B \otimes \phi_B, \) and \( \phi_C \otimes \phi_C \) are all unsatisfiable.
Using this observation, we refine the original regular expression $R_1$ to obtain a new regular expression, $R_2 \triangleq (\epsilon + C)(A + BC)^*(\epsilon + B)$ whose words never contain the subsequences $\ldots AC \ldots, \ldots BA \ldots, \ldots BB \ldots,$ and $\ldots CC \ldots$. Note that $L(R_2) \subset L(R_1)$, and that $R_2$ has fewer disjunctions that appear under a $*$ operator. For these reasons, we might expect that $R_2$ would produce a more-precise result than $R_1$. Counter-intuitively, $R_2$ produces a less-precise result.

To see why, consider the evaluation of $R_2$ in the abstract domain: $(1 \oplus \phi_C) \odot (\phi_A \oplus \phi_B \odot \phi_C)^{\oplus} \odot (1 \oplus \phi_B)$. We first compute

$$\phi_{BC} \triangleq \phi_B \odot \phi_C = (\text{pos} > 0 \land i' = i + 2 \land j' = j + 2 \land \text{pos'} = 1).$$

We then combine $\phi_A$ with $\phi_{BC}$ to obtain the formula:

$$\phi_{A+BC} \triangleq (i' = i + 1 \land j' = j + 1 \land \text{pos} \geq 0 \land \text{pos'} = 1) \lor (i' = i + 2 \land j' = j + 2 \land \text{pos} \geq 0 \land \text{pos'} = 1).$$

As a next step, we abstract $\phi_{A+BC}$ into the formula $w'$:

$$w' \triangleq (i + 1 \leq i' \land i' \leq i + 2 \land j + 1 \leq j' \land j' \leq j + 2 \land \text{pos'} = 1).$$

Notice that $w'$ does not imply that $i$ and $j$ increase by the same amount. The analysis continues by extending $(1 \oplus \phi_C)$ first with $(w')^{\oplus}$ and then with $(1 \oplus \phi_C)$. From the resulting formula, together with the fact that $i = 0 \land j = 0$ holds before the loop, after the loop one is able to establish only that $i \leq 2 \ast j$ and $j \leq 2 \ast i$. In particular, we can no longer draw the conclusion that $i = j$ holds, which the evaluation of the “cruder” regular expression $R_1$ succeeded in establishing.

This example demonstrates that an arbitrary—even plausible—refinement of a path expression can actually degrade analysis results, compared with using the original path expression. In general, degradation arises because of an interaction between the expressiveness of the abstract domain in use, and the refinement step. In the example above, the common increments to $i$ and $j$ in the three original disjuncts are lost because in $A + BC$, $A$ involves one loop iteration, whereas $BC$ involves two iterations. To retain the information that the increments to $i$ and $j$ are “synchronized,” the formula $w'$ that abstracts $\phi_{A+BC}$ would need to have either the subformula $j' - i' = j - i$ or the subformula $\exists k. i' = i + k \land j' = j + k$. The first subformula uses four variables, and each conjunct of the second subformula uses three variables; however, the octagon domain can only express conjunctions of two-variable inequalities.
4.2.3 Transformation in the Presence of Nested Loops

In the preceding examples, we applied transformations to regular expressions that do not contain nested stars (i.e., leaf loops); these expressions were derived from programs that do not contain nested loops. In general, however, we also want to analyze programs that do contain nested loops. We approach such problems by computing a regular expression that has nested stars. Beginning with an innermost occurrence of a star operator—i.e., a subexpression of the form \( R^* \), where \( R \) is star-free)—we apply our transformation, compute a summary, and substitute the summary in place of the innermost star. When this process has been performed for all innermost occurrences of star in the body of a non-innermost occurrence of star, we can analyze the body of the latter occurrence as if the body were star-free. We repeat this process until we have summarized the entire procedure.

Example 4.1. Consider again the program in Fig. 4.1(a). As explained earlier in this section, the paths in the loop body can be refined from \( R \overset{\text{def}}{=} (A + B)^* \) to \( R' \overset{\text{def}}{=} A^*B^* \). Suppose that the loop had been nested inside an enclosing loop, and that the path expression for the loop-nest had the form \( E \overset{\text{def}}{=} (\alpha((\beta + \gamma)(A + B)^*(\delta + \nu))^*\mu) \). Then the refinement of \( R \) would create \( E_1 \overset{\text{def}}{=} (\alpha ((\beta + \gamma)A^*B^*(\delta + \nu))^*\mu) \). The leaf-loop algorithm would then be applied to \( S \overset{\text{def}}{=} ((\beta + \gamma)A^*B^*(\delta + \nu))^* \), with \( A^*B^* \) treated as an indivisible atom. This approach is tantamount to transforming \( S \) into "let \( c = A^*B^* \) in \( ((\beta + \gamma)c(\delta + \nu))^* \), and applying the leaf-loop algorithm to \( T \overset{\text{def}}{=} ((\beta + \gamma)c(\delta + \nu))^* \). □

As shown in the earlier examples of this section, the leaf-loop algorithm requires us to have some way to check for and remove infeasible paths. The summarization of inner loops gives us a way to perform infeasibility checks even in the presence of nested loops, by using (the interpretations of) the summary values in place of starred subexpressions. For instance, in Ex. 4.1, we could multiply out \( (\beta + \gamma)c(\delta + \nu) \) to obtain the sum of products \( \beta c \delta + \beta c \nu + \gamma c \delta + \gamma c \nu \), and then test the abstract values of all words in \( \{\beta c \delta, \beta c \nu, \gamma c \delta, \gamma c \nu\} \cdot \{\beta c \delta, \beta c \nu, \gamma c \delta, \gamma c \nu\} \) for pair-wise infeasibility.

4.3 Additional Chapter Background

4.3.1 Regular Languages and Algebra

Definition 4.2. The set of regular expressions over an alphabet \( \Sigma \) is defined by the following grammar:

\[
\text{regexp}_\Sigma := 0 | 1 | a \in \Sigma | \text{regexp}_\Sigma + \text{regexp}_\Sigma | \text{regexp}_\Sigma \cdot \text{regexp}_\Sigma | \text{regexp}_\Sigma^*
\]
(Because $\Sigma$ will always be implicit from context, the subscript on regexp will henceforth be omitted.) For simplicity, we sometimes write “$a \cdot b$” as “$ab$.” The star height of a regular expression $R \in \text{regexp}$ is the maximum nesting depth of a star in $R$.

A regular expression $R \in \text{regexp}$ denotes a regular language, $L(R)$, defined as follows:

$$L(R) \overset{\text{def}}{=} \begin{cases} \emptyset & \text{if } R = 0 \\ \{\epsilon\} & \text{if } R = 1 \\ \{a\} & \text{if } R = a \in \Sigma \\ L(R_1) \cup L(R_2) & \text{if } R = R_1 + R_2 \\ L(R_1) \otimes L(R_2) & \text{if } R = R_1 \cdot R_2 \\ \bigcup_{i=0}^{\infty} L(R_1)^i & \text{if } R = (R_1)^* \end{cases}$$

where $\epsilon$ denotes the empty word (of length 0), $\otimes$ denotes language concatenation: $L_1 \otimes L_2 \overset{\text{def}}{=} \{xy \mid x \in L_1 \land y \in L_2\}$, and $L(R_1)^i$ denotes the $i$-fold concatenation of $L(R_1)$ with itself (e.g., $L(a^*) = \emptyset + \{a\} + \{aa\} + \{aaa\} + \ldots$).

We now give the axiomatizations of Kleene algebras and a family of strictly weaker structures, which we call pre-Kleene algebras.

Both of these structures have “combine,” “extend,” and “iteration” operators. The following definition for a Kleene algebra was given by Kozen [95].

**Definition 4.3.** A Kleene algebra (KA), $\langle A, +, \cdot, *, 0, 1 \rangle$, is a set $A$ (called the carrier) equipped with two binary operations $\cdot$ and $+$, a unary operation $*$, and distinguished elements 0 and 1, and such that the following conditions hold.

1. (Semiring) The binary operators $\cdot$ and $+$ satisfy the following axioms, which define a semiring

   a) (Associativity of $\cdot$ and $+$) $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $a(bc) = (ab)c$, for all $a, b, c \in A$

   b) (Commutativity of $+$) $a + b = b + a$, for all $a, b \in A$

   c) (Distributivity) $a(b + c) = ab + ac$ and $(b + c)a = ba + ca$, for all $a, b, c \in A$

   d) (Identity elements) There exists an element $0 \in A$ such that, for all $a \in A$, $a + 0 = 0 + a = a$. There exists an element $1 \in A$ such that, for all $a \in A$, $a1 = 1a = a$

   e) (Annihilation) $a0 = 0a = 0$, for all $a \in A$

2. (Idempotence) It is also required that $+$ be idempotent: $a + a = a$, for all $a \in A$
3. (Iteration) The above axioms imply that relation $\leq$ defined by $a \leq b \iff a + b = b$ is a partial order. Using this partial order, the $\star$ operator satisfies the following axioms

a) $1 + a(a^*) \leq a^*$, for all $a \in A$

b) $1 + (a^*)a \leq a^*$, for all $a \in A$

c) If $a, x \in A$ with $ax \leq x$, then $a^*x \leq x$

d) If $a, x \in A$ with $xa \leq x$, then $xa^* \leq x$

Two important examples of Kleene algebras are (1) $\text{Reg}_\Sigma$—the algebra of regular languages over an alphabet $\Sigma$—where the carrier is the set of regular languages, $+$ is union, $\cdot$ is concatenation, $\star$ is Kleene-star, $0$ is the empty language, and $1$ is the singleton language containing the empty word; and (2) $\text{Rel}_X$—the algebra of relations over a set $X$—where the carrier is the set of binary relations on $X$, $+$ is union, $\cdot$ is relational composition, $\star$ is reflexive transitive closure, $0$ is the empty relation, and $1$ is the identity relation.

**Definition 4.4.** A pre-Kleene algebra (PKA), $\langle A, +, \cdot, \star, 0, 1 \rangle$, is a set $A$ equipped with two binary operations $\cdot$ and $+$, a unary operation $\star$, and distinguished elements $0$ and $1$, and such that the Semiring and Idempotence axioms hold, and additionally we have the following iteration axioms:

1. (Reflexivity) $1 \leq a^*$, for all $a \in A$

2. (Extensivity) $a \leq a^*$, for all $a \in A$

3. (Transitivity) $a^* \cdot a^* = a^*$, for all $a \in A$

4. (Monotonicity) if $a, b \in A$ with $a \leq b$ then $a^* \leq b^*$. Equivalently $a^* \leq (a + b)^*$, for all $a, b \in A$

5. (Unrolling) $(a^n)^* \leq a^*$ for $n \in \mathbb{N}$ and $a \in A$, where $a^0 = 1$ and $a^n = a \cdot \ldots \cdot a$

The abstract domain from § 4.2 is an example of a pre-Kleene algebra. Two more examples follow. All three examples are pre-Kleene algebras, but none are Kleene algebras.

**Example 4.5.** Fix a set of variables $x$ and a set of primed copies $x'$. Define an algebraic structure $\text{ACI} = \langle F, \oplus, \otimes, \star, 0, 1 \rangle$, inspired by the analysis of Ancourt et al. [7], as follows:

- The carrier $F$ consists of all formulas in the existential fragment of Presburger arithmetic over the symbols $x'$ and $x''$ (i.e., transition formulas), quotiented by logical equivalence.

- $\phi \otimes \psi \overset{\text{def}}{=} \exists x''. \phi(x, x'') \land \psi(x'', x')$ is relational composition.

- $\phi \oplus \psi \overset{\text{def}}{=} \phi \lor \psi$ is disjunction.
• $0 \overset{\text{def}}{=} \text{false}.$

• $1 \overset{\text{def}}{=} \bigwedge_i x'_i = x_i.$

• $\phi^\odot$ is defined as follows.

  – Let $\text{pre}(\phi)$ be the convex hull of the formula $(\exists x'. \phi)$, which can be computed using the algorithm of Farzan and Kincaid [49]. The formula $\text{pre}(\phi)$ is a precondition of the loop $\phi$, and must hold on entry.

  – Similarly, let $\text{post}(\phi)$ be the convex hull of the formula $(\exists x. \phi)$.

  – For each variable $x_i$, introduce a new variable $\delta_i$, which we use to represent the change in $x_i$ over the action of $\phi$. Compute the convex hull of the formula $(\exists x, x'. (\phi \land \bigwedge_i \delta_i = x'_i - x_i)$, and write it as $A \delta \geq b$. Let $\Delta(\phi)$ be the formula $\Delta(\phi) \overset{\text{def}}{=} A x' \geq x + kb$

  – Finally, define $\phi^\odot \overset{\text{def}}{=} \exists k. ((k = 0 \land \bigwedge_i x'_i = x_i) \lor (k \geq 1 \land \text{pre}(\phi) \land \text{post}(\phi))) \land \Delta(\phi)).$

Showing that ACI satisfies the semiring axioms, idempotence of $+$, and reflexivity, extensivity, and transitivity of $\odot$ is straightforward. It is also not difficult to see that ACI fails to satisfy the KA axioms for iteration, because it fails to compute exact transitive closure.

The pre-Kleene algebra monotonicity and unrolling axioms are more subtle. First, observe that the natural order $\leq$ in ACI is entailment ($\phi \odot \psi = \psi$ if and only if $\phi \models \psi$). So the monotonicity law is that if $\phi \models \psi$, then $\phi^\odot \models \psi^\odot$. This holds because the iteration operator is defined in terms of convex hulls of projections, and both projection and hull are monotone. To show that the unrolling axiom holds, let $n$ be arbitrary. Clearly we have $\text{pre}(\phi^n) \models \text{pre}(\phi)$ and $\text{post}(\phi^n) \models \text{post}(\phi)$; it remains only to show that $B x' \geq x + c \models A x' \geq x + nb$, where $\Delta(\phi) = A x' \geq x + b$ and $\Delta(\phi) = B x' \geq x + c$. Since $\phi \models A x' \geq x + b$, we must have $\phi^n \models A x' \geq x + nb$; since $\Delta(\phi^n)$ entails every inequation of the form $bx' \geq bx + c$ that is entailed by $\phi^n$, we have the result.

Example 4.6. Fix a set of variables $x$ and a set of primed copies $x'$. Let $P$ be a finite set of predicate symbols in some theory $T$ with a decidable existential fragment (e.g., the theory of bitvectors, arrays, etc.) over the variables $x$. Define an algebraic structure $PA_P = \langle T, \oplus, \otimes, 0, 1 \rangle$, as follows:

• $T$ is the set of existential $T$-formulas over $x$ and $x'$

• $\oplus, \otimes, 0, and 1$ are as in Ex. 4.5

• $\phi^\odot \overset{\text{def}}{=} \bigwedge (p(x) \Rightarrow p(x') : p \in P : \phi \models p(x) \Rightarrow p(x'))$

It is straightforward to check that $PA_P$ forms a pre-Kleene algebra.
We are interested in determining which facts are true in Kleene algebras and pre-Kleene algebras. However, we are also interested in viewing expressions over ·, +, and ∗ as syntactic objects, because—as we see from the examples in § 4.2—the structure of expressions matters for analysis. To make a notational distinction, we use

\[ E \models_{KA} R = R' \]

to denote that \( R = R' \) holds in any Kleene algebra that satisfies each equation in a set \( E \). We use similar notation for pre-Kleene algebras. For example, for both Kleene algebras and pre-Kleene algebras, we have

\[ a + (b + c) = (a + b) + c. \]

Because \( a + (b + c) \) and \((a + b) + c \) are syntactically different, we no longer write

\[ a + (b + c) = (a + b) + c. \]

Instead we write \( \models_{KA} a + (b + c) = (a + b) + c \) and \( \models_{PKA} a + (b + c) = (a + b) + c. \)

Note that \( \models_{KA} R = R' \) if and only if \( L(R) = L(R') \) [95].

The axioms of pre-Kleene algebras are implied by the axioms of Kleene algebras, so every Kleene algebra is also a pre-Kleene algebra; however, the converse is not true. For example,

\[ 1 + a a^* = a^* \] (4.2)

is a property that holds in any Kleene algebra, but does not necessarily hold in a pre-Kleene algebra.

The motivation for introducing pre-Kleene algebras is to capture a broader class of program analyses, for which the axioms of Kleene algebras are too strong. In particular, the Kleene algebra iteration axioms imply that for any element \( a \), \( a^* \) must be equal to the least fixed-point of the function \( f_a(x) = 1 + ax \). For many abstract domains of interest, least fixed-points of such functions are not computable (and may not even exist). The weaker iteration axioms for pre-Kleene algebras allow us to study domains in which the ∗ operator is imprecise (i.e., \( a^* \) is a post fixed-point of \( f_a \)).

Naturally, one may ask if the iteration axioms for pre-Kleene algebras are still too strong. We note that there is general strategy (see Kincaid [84]) for designing operators that over-approximate the transitive closure of transition formulas, which always results in an iteration operator satisfying the pre-Kleene algebra iteration axioms. Namely, if we compute \( R^\circ \) by (i) computing the best abstraction of \( R \) within some class of transition relations for which transitive closure is computable, and (ii) computing the exact transitive closure of that abstraction, then \( R^\circ \) satisfies the pre-Kleene algebra axioms.

The gap between Eqn. (4.2), which holds in KA, and the weaker property “\( 1 + a a^* = a^* \),” which holds in PKA, provides evidence that PKA captures a fundamental property of program analysis vis a vis program transformations: the analysis of an unrolled loop (“\( 1 + a a^* \)” can give more precise results than the original loop (“\( a^* \)”)). In contrast, the KA axioms do not allow algebraic reasoning about such possible improvements.
because they treat an unrolled loop ("1 + aa") as being equivalent to the original loop ("a*").

The axiomatization of pre-Kleene algebras is relaxed enough to capture some interesting domains, such as the three introduced above, while being expressive enough to allow for non-trivial refinements.

4.3.2 Rewriting Regular Expressions

Consider an algebraic analysis, with abstract domain \( D \), where the goal is to approximate \( TR[R] \) for some set of program paths, \( L(R) \). As described in § 2.1, an over-approximation can be obtained merely by evaluating \( R \) over \( D[R] \). However, as indicated from the examples of § 4.2, the structure of \( R \) may lead to an imprecise analysis result: perhaps there is another suitable expression \( R' \) that leads to a better analysis result. If such an \( R' \) exists, we would like to evaluate \( R' \) instead of \( R \).

We must consider when it is “suitable” to analyze \( R' \) instead of \( R \). Namely, if we are going to evaluate \( D[R'] \) instead of \( D[R] \), we need to make sure that \( D[R'] \) is still a sound analysis.

**Definition 4.7.** A rewrite, \( R \rightarrow R' \), is sound if \( TR[R] \subseteq TR[R'] \).

This definition says that a rewrite is sound if the semantics of the new path-expression approximates the semantics of the old path-expression. This definition ensures that the analysis of the new regular expression still approximates the original task because, by Lem. 2.2, \( TR[R'] \subseteq \gamma(D[R']) \). Thus, if \( TR[R] \subseteq TR[R'] \), then \( TR[R] \subseteq \gamma(D[R']) \). Ideally, we would like to have a regular expression that leads to as precise an analysis result as possible; the challenge would be to find a most-precise regular expression among the set of all sound rewrites—that is, “most precise” in terms of \( \gamma(D[R]) \) and \( \subseteq \). Unfortunately, such an expression may not exist (and even if it does, it can only be computed under strong assumptions about the abstract domain).

The main problem is that the set of considered rewrites is too large: there is an infinite space of possible rewrites, so how do we select one?

One approach for selecting a sound rewrite is based on removing infeasible words from \( R \). Suppose that we have discovered some set of infeasible words \( Inf \). That is, for every \( w \in Inf \) we have \( TR[w] = \emptyset \).

Consider creating the following regular expression: \( R_{inf} = \sum_{w \in Inf} w \). We can now create the rewrite \( R \rightarrow R' \), by computing a regular expression \( R' \) with \( L(R') = L(R) \cap L(R_{inf})^c \). The rewrite is sound, because \( L(R) = L(R') \cup L(R_{inf}) \) implies \( TR[R] = TR[R'] \cup TR[R_{inf}] = TR[R'] \). Written another way, \( R' \) is a sound rewrite because \( E \models_{KA} R' = R \), where \( E \) is the set of equations \( E = \{ w = \emptyset \mid w \in Inf \} \). We call this type of rewrite a refinement.

**Definition 4.8.** Let \( E \) be some set of equations denoting infeasible paths. A rewrite, \( R \rightarrow R' \), is a refinement if \( E \models_{KA} R' = R \) and \( L(R') \subseteq L(R) \).
Refinements provide a natural method for producing alternative analysis tasks, which in practice often give better analysis results. However, as shown in the analysis of Fig. 4.1(b) an arbitrary refinement can actually produce worse results. One could imagine rectifying this issue by creating an analysis framework that refines a given analysis task $R$ to a series of refined regular expressions $R_1, \ldots, R_k$, using methods in the style of previous work on control-flow refinement [11, 64, 125, 57]. The analysis would then analyze each expression $R_i$, as well as the original expression $R$, and then combine the results. We do not find this approach satisfactory, because:

1. It blows up the number of analysis tasks.
2. We do not assume our domains have a "meet" operator, so there might not be a meaningful method to combine the results.
3. Arbitrary refinements do not give us any analytical understanding of what makes one regular expression better than alternative expressions—for the purpose of analysis.

Unfortunately, without any structure on $\gamma$ or domain $D$, we cannot understand the analysis precision obtained using a given regular expression, just from the regular expression itself. We remedy this situation by requiring $D$ to have some algebraic properties that turn out to give some traction on the analysis-precision question.

Within this chapter, we assume that each abstract domain $D$ satisfies the axioms of a **pre-Kleene algebra** (PKA).

In essence, we use the concept of a PKA to axiomatize the precision properties of abstract domains. Under our domain assumption, we have

$$\text{if } \models_{PKA} R \leq R' \text{ then } D[R] \leq D[R'].$$

We can now use the order $\leq$ of a pre-Kleene algebra to compare the relative analysis precision of regular expressions.

**Theorem 4.9.** Let $\Sigma$ be an alphabet, and let $R, R' \in \text{regexp}_\Sigma$ with $\models_{PKA} R' \leq R$. Then for any algebraic program analysis $(D, D[\cdot], \gamma)$ over $\Sigma$, we have $\gamma(D[R']) \subseteq \gamma(D[R])$; i.e., $\gamma$ is monotone with respect to $\leq_{PKA}$.

**Proof.** $\models_{PKA} R' \leq R$ is equivalent to saying $\models_{PKA} R' + R = R$, which means $D[R'] \oplus D[R] = D[R]$. By Defn. 2.1 $\gamma(D[R']) \cup \gamma(D[R]) \subseteq \gamma(D[R'] \oplus D[R]) = \gamma(D[R])$. Therefore, $\gamma(D[R']) \subseteq \gamma(D[R']) \cup \gamma(D[R]) \subseteq \gamma(D[R])$. \qed
Thm. 4.9 gives us the analytical understanding we have been looking for. Thm. 4.9 says that if $\models_{PKA} R' \leq R$ holds, then the analysis of $R'$ will be at least as precise as—and possibly more precise than—the analysis of $R$.

We use $\leq_{PKA}$ to drive the design of a procedure that, given an initial analysis task $R$ and a set of infeasible paths $E$, refines $R$ to $R'$ with (i) $E \models_{KA} R' = R$ and $L(R') \subseteq L(R)$ (refinement, Defn. 4.8) and (ii) $\models_{PKA} R' \leq R$ (potentially precision-improving, Thm. 4.9).

By using a refinement procedure that satisfies the specification above, we can analyze $R'$ instead of $R$: as long as our domain is a pre-Kleene algebra, we are guaranteed that $R'$ will give no worse results, compared with $R$.

Note that a procedure that satisfies the above specification is still useful for a domain that is not a pre-Kleene algebra: such a refinement method will produce a sound analysis task $R'$ from $R$ whenever $E \models_{KA} R' = R$, for some infeasible paths $E$. However, if the domain is not a pre-Kleene algebra, then $\models_{PKA} R' \leq R$ does not imply that $D[R'] \subseteq D[R]$, and the conclusion of Thm. 4.9 does not hold. Even though we lose the guarantee that precision does not degrade with domains that are not pre-Kleene algebras, in practice we find that our refinement procedure improves analysis results (see § 4.6).

### 4.4 Refining Inner Loops

In this section, we address the problem of refining an innermost-loop based on infeasible paths: we are given an expression $R^*$ where $R$ is star-free; we would like to refine $R^*$ to $R'$ without having $R'$ decrease analysis precision. We use the order of a pre-Kleene algebra to say when a refinement does not decrease precision. That is, we want a method that refines $R^*$ while still having $\models_{PKA} R' \leq R^*$, because by Thm. 4.9 we know the analysis of $R'$ will be at least as good as the analysis of $R^*$.

Given $R^*$, we first take $R$ to a different form. Using the distributivity axioms of a pre-Kleene algebra, we rewrite $R$ into the form $a_1 + a_2 + \ldots + a_k$, for some set of words $A \overset{\text{def}}{=} \{a_1, a_2, \ldots, a_k\}$. Due to the axioms of the operators $+$ and $\cdot$ in a pre-Kleene algebra, $\models_{PKA} R^* = (a_1 + \ldots + a_k)^*$; i.e., taking $R^*$ to $(a_1 + \ldots + a_k)^*$ does not change analysis precision. Once $R^*$ is in this form, it becomes easy to determine an expression of infeasible word sequences, $R_{inf}$. Because we have $\gamma(0) = 0$, if $D[a_1 \cdot \ldots \cdot a_{i,n}] = 0$ then $TR[a_i, \ldots, a_{i,n}] = 0$. For instance, we can test all sequences of a small number of elements $\{a_i\} \subseteq A$.

We now consider the problem of refining the loop $(a_1 + \ldots + a_k)^*$ into another expression $R'$ by removing infeasible sequences in $L(R_{inf})$, while maintaining that $\models_{PKA} R' \leq R^*$.

For the remainder of this section, we will consider $\{a_1, a_2, \ldots, a_n\}$ to be the alphabet under consideration; we are interested in words over $\{a_1, a_2, \ldots, a_n\}$, where each $a_i$ is considered to have no “internal” structure.
4.4.1 Form for Refinements

Before we present the algorithm for refinement, we first ask the following question. We are going to refine \((a_1 + \ldots + a_k)^*\) to \(R'\) with \(\models_{\text{PKA}} R' \leq (a_1 + \ldots + a_k)^*\). What are the properties that \(R'\) must satisfy?

**Definition 4.10.** We say that a regular expression \(R\) over some alphabet \(S\) satisfies the equal-unrolling property if \(R\) has star-height 0 or 1 and for every starred sub-expression \((R_1)^*\) of \(R\), we have that every pair of words \(w_1\) and \(w_2\) in \(L(R_1)\), \(|w_1| = |w_2|\).

In other words, every word in \(L(R_1)\) must have the same length. An alternative way to express Defn. 4.10 is as follows: \(R\) satisfies the equal-unrolling property if \(R\) can be written in the following form

\[
R = \sum_i R_i^1 (R_i^1)^* \cdots R_i^n (R_i^n)^* \tag{4.3}
\]

where each \(R_i^j\) is a \(*\)-free expression over the alphabet \(S\) and \(R_i^j = \sum_{w \in S'} w\) and \(S' \subseteq (S)^{n_{i,j}}\) for some \(n_{i,j}\), and \((S)^n\) denotes the set \({a_{i,1}, \ldots, a_{i,n}| a_{i,j} \in S}\).

**Theorem 4.11.** Let \(R'\) be a regular expression, and \(a_1, \ldots, a_k\) be letters. \(\models_{\text{PKA}} R' \leq (a_1 + \ldots + a_k)^*\) if and only if \(R'\) satisfies the equal-unrolling property.

**Proof.** \((\Leftarrow)\) Suppose that \(R'\) satisfies the equal-unrolling property. Then \(R'\) can be written in the form shown in Eqn. (4.3). Consider some \(R_i^j\). Since \(R_i^j\) is \(*\)-free, then there exists a longest word for \(R_i^j\). Let \(N\) be the length of this longest word. By the definition of \(\leq\) in a pre-Kleene algebra, we must have

\[
\models_{\text{PKA}} R_i^j \leq \sum_{i=0}^{N} (a_1 + \ldots + a_k)^i \leq \sum_{i=0}^{N} ((a_1 + \ldots + a_k)^i)^* \leq \sum_{i=0}^{N} (a_1 + \ldots + a_k)^* \leq (a_1 + \ldots + a_k)^*
\]

With the last three steps due to extensivity (2), unrolling (5), and idempotence (2) respectively. Now consider some \(R_{i,j} = \sum_{w \in S'} w\), where \(S' \subseteq (S)^{n_{i,j}}\) for some \(n_{i,j}\). By the definition of \(\leq\) in a pre-Kleene algebra, \(\models_{\text{PKA}} R_{i,j} \leq (a_1 + \ldots + a_k)^{n_{i,j}}\). Thus, due to monotonicity of \(*\) (4) and unrolling (5), \(\models_{\text{PKA}} (R_i^j)^* \leq ((a_1 + \ldots + a_k)^{n_{i,j}})^* \leq (a_1 + \ldots + a_k)^*\). Because we have \(\models_{\text{PKA}} R_{i,j}^1 \leq (a_1 + \ldots + a_k)^*\) and \(\models_{\text{PKA}} R_{i,j}^n \leq (a_1 + \ldots + a_k)^*\) for every \(i\) and \(j\), we can conclude \(\models_{\text{PKA}} R' \leq (a_1 + \ldots + a_k)^*\) due to transitivity (3) and idempotence (2).

\((\Rightarrow)\) [sketch] Suppose that the property \(\models_{\text{PKA}} R' \leq (a_1 + \ldots + a_k)^*\) holds. By the completeness of first-order logic, there must exist a proof of this fact, denoted by \(\vdash_{\text{PKA}} R' \leq (a_1 + \ldots + a_k)^*\). The axioms for the proof system are the axioms of pre-Kleene algebra (in equational form). Other relevant inference rules
are the usual ones for first-order logic with equality, including transitivity, reflexivity, and symmetry rules and functional consistency (e.g., \(\vdash_{\text{PKA}} E_1 = E'_1\) and \(\vdash_{\text{PKA}} E_2 = E'_2\) then \(\vdash_{\text{PKA}} E_1 + E_2 = E'_1 + E'_2\)).

We now prove, using induction on the height of the proof tree for the judgement \(\vdash_{\text{PKA}} E = E'\), that \(E\) satisfies the equal-unrolling property iff \(E'\) does, and \(E\) and \(E'\) have the same star-height. To prove the base case, we consider the pre-Kleene axioms in equational form. Then we simply check that if the left-hand side of each equality axiom satisfies the equal-unrolling property then the right-hand side does as well. For example, due to extensivity (2) we have \(\vdash_{\text{PKA}} a^a + a = a^a\) and both the left-hand and right-hand sides of the equality either satisfy the equal-unrolling property or both do not, and both sides have the same star-height. Checking the other axioms follows a similar pattern.

Now for the inductive step. The induction hypothesis says that for every sub-proof-tree with root \(\vdash_{\text{PKA}} F = F'\) in the proof \(\vdash_{\text{PKA}} E = E'\), if \(F\) satisfies the equal-unrolling property iff \(F'\) does, and \(F\) and \(F'\) have the same star-height. Consider the last step of the proof \(\vdash_{\text{PKA}} E = E'\). Consider the inference rule

\[
\frac{\vdash_{\text{PKA}} E_1 = E'_1 \quad \vdash_{\text{PKA}} E_2 = E'_2}{\vdash_{\text{PKA}} E_1 + E_2 = E'_1 + E'_2}
\]

Now suppose \(E_1 + E_2\) satisfies the equal-unrolling property. Then \(E_1\) and \(E_2\) must also satisfy it as well. By the induction hypothesis, it must be the case that \(E'_1 + E'_2\) also satisfy the equal-unrolling property. Therefore, \(E'_1 + E'_2\) must also satisfy the equal-unrolling property. Also, \(E_1\) has the same star-height as \(E'_1\) and \(E_2\) has the same star-height as \(E'_2\). Therefore, the star-height of \(E_1 + E_2\) must be the same as the star-height of \(E'_1 + E'_2\). The reasoning for the \(\cdot\) inference rule is similar. Finally, suppose that the last step of \(\vdash_{\text{PKA}} E = E'\) was of the form \(\vdash_{\text{PKA}} (E_1)^* = (E'_1)^*\) under the assumption \(\vdash_{\text{PKA}} E_1 = E'_1\). Suppose that \((E_1)^*\) satisfies the equal-unrolling property. Because the equal unrolling property only applies to expressions of star-height 0 or 1, \(E_1\) must be star-free and all word lengths be equal. By the induction hypothesis \(E'_1\) must also be star-free. It then becomes the case that words in \(L(E'_1)\) must have the same length. Thus \((E'_1)^*\) has star-height 1 and satisfies the equal-unrolling property. What we have shown is that if we have \(\vdash_{\text{PKA}} E = E'\) and \(E\) satisfies the equal-unrolling property then so must \(E'\), and \(E\) has the same star-height as \(E'\). Our initial assumption of the \((\Rightarrow)\) direction was that \(\vdash_{\text{PKA}} R' \leq (a_1 + . . . + a_k)^*\). Equivalently, \(\vdash_{\text{PKA}} (a_1 + . . . + a_k)^* = (a_1 + . . . + a_k)^* + R'\). \((a_1 + . . . + a_k)^*\) satisfies the equal-unrolling property and has star-height 1. Therefore, \(R'\) must have star-height 0 or 1 and also satisfy the equal-unrolling property.

Essentially, Thm. 4.11 says that for any \(R''\) with \(\vdash_{\text{PKA}} R'' \leq (a_1 + . . . + a_k)^*\), \(R''\) can be rewritten into a sum-of-products form \(R'\), and \(R'\) must satisfy two important properties. One, \(R'\) must have a star-height no
Theorem 4.13. Let $G$ focus on a strongly-connected component under a star. Because the cyclic behavior of $G$ is captured by its strongly-connected components, we can conclude that $\{L_{\Pi^j_k}(v_i)\}$, where each word has the same length. As an example, using Thm. 4.11 we can directly conclude that $\models_{PKA} A^*B^* \leq (A + B)^*$, because $A^*B^*$ has the form shown in Eqn. (4.3). Also, Thm. 4.11 shows that $\not\models_{PKA} (\epsilon + C)(A + BC)^*(\epsilon + B) \leq (A + B + C)^*$ because $A$ and $BC$ come from unrolling $(A + B + C)$, but they come from unrollings of different lengths.

4.4.2 Refinement Graphs

Consider the problem of refining the loop $(a_1 + \ldots + a_k)^*$ to another expression $R'$ by removing some infeasible sequences denoted by an expression $R_{\inf}$, while maintaining that $\models_{PKA} R' \leq (a_1 + \ldots + a_k)^*$. Note that it is possible to generate an expression $R''$ that removes all the infeasible sequences from $R_{\inf}$ by finding an $R''$ such that $L(R'') = L((a_1 + \ldots + a_k)^*) \cap L(R_{\inf})^c$. However it might be the case that neither $R''$ nor any other equivalent expression comes with any analysis-precision guarantees.

Definition 4.12. Suppose that we have an expression $(a_1 + \ldots + a_k)^*$ and an expression $R_{\inf}$ denoting infeasible words. Let $G = (V, E, \Lambda)$ be a directed graph with vertices $V$, edges $E$, and a labeling function $\Lambda : V \rightarrow \Sigma$. Let $\text{Paths}(G)$ be the set of paths of $G$. A path is a sequence of vertices $v_1v_2\cdots v_n$ where for each $i < n$, $(v_i, v_{i+1}) \in E$. We define $L_{\text{Paths}}(G) = \{\Lambda(v_1)\cdots \Lambda(v_n) \mid v_1\cdots v_n \in \text{Paths}(G)\}$. We say that $G$ is a refinement graph with respect to $R_{\inf}$ if $L_{\text{Paths}}(G) = L((a_1 + \ldots + a_k)^*) \cap L(R_{\inf})^c$.

Given the appropriate refinement graph, our problem now becomes one of finding an expression $R'$ with $L_{\text{Paths}}(G) \subseteq L(R') \subseteq L((a_1 + \ldots + a_k)^*)$ and $\models_{PKA} R' \leq (a_1 + \ldots + a_k)^*$. Because of Thm. 4.11, we only need to consider regular expressions of the form shown in Eqn. (4.3) to ensure $\models_{PKA} R' \leq (a_1 + \ldots + a_k)^*$. Note that being careful about the $*$’s in $R'$ is the important piece to make sure $R'$ has the form given in Eqn. (4.3).

Consequently, we now consider the cyclic portions of $G$, and determine how to represent these cyclic behaviors using an expression $R'$ with a star-height no greater than 1 and an equal number of unrollings under a star. Because the cyclic behavior of $G$ is captured by its strongly-connected components, we can focus on a strongly-connected components $S$.

Theorem 4.13. Let $S$ be a non-trivial strongly-connected component. If

1. The cycles in $S$ have the same length
2. There is a vertex common to all the cycles in $S$
then

There exists an expression of the form shown in Eqn. (4.3) whose language exactly matches\[ L_{\text{Paths}}(S). \]

Proof. This proof is constructive. Let \( \{C_1, \ldots, C_N\} \) be the set of simple cycles of \( S \). Because we have \((1) \land (2),\) \( |C_i| = n \) for all \( i \), and there exists a vertex, \( e \), common to all the cycles.

Now rotate all the cycles \( C_i \) to \( C'_i \), where each \( C'_i \) starts with \( e \). Let \( \langle v_{i,1}, \ldots, v_{i,n} \rangle \) be the sequence of vertices for a cycle \( C'_i \). After rotation, \( v_{i,1} = v_{j,1} = e \) for each \( i, j \). Let \( a_i = \Lambda(v_i) \) for each \( i \). Let \( w_i = a_{i,1} \cdot \ldots \cdot a_{i,n} \). Consider the expression \((\sum_{i=1}^{N} w_i)^*\). Since all the cycles \( C'_i \) start and end with the same vertex, we have

\[ L((\sum_{i=1}^{N} w_i)^*) \subseteq L_{\text{Paths}}(S). \]

However, there is a difference between \( L((\sum_{i=1}^{N} w_i)^*) \) and \( L_{\text{Paths}}(S) \). For example, suppose that we have one cycle in \( S \) with length 2. That is, \( C'_1 = \langle v_{1,1}, v_{1,2} \rangle \). \( a_{1,1}a_{1,2}a_{1,1} \in L_{\text{Paths}}(S) \), but \( a_{1,1}a_{1,2}a_{1,1} \notin L((a_{1,1}a_{1,2})^*) \). We have captured the cyclic behavior of \( S \) with \((\sum_{i=1}^{N} w_i)^*\); however, we need to consider how a path can enter a cycle and how it can leave a cycle.

We can solve this issue by creating a new graph \( S' \), where \( S' \) has a vertex \( v_{\text{star}} \) with label \((\sum_{i=1}^{N} w_i)^*\), and creating head and tail chains in \( S' \). These chains act as an extension of the cyclic behavior of the component, showing how control can enter the cycle and how control can leave the cycle. These heads and tails can be created by creating a head and tail vertex for every vertex in the component, and then chaining the head vertices together according to the paths in the cycle; and similarly for the tail vertices. In short, \( S' \) contains the vertex \( v_{\text{star}} \) that captures the cyclic behavior, as well as head chains going into \( v_{\text{star}} \), and tail chains coming from \( v_{\text{star}} \).

\( S' \) is a DAG, for which \( L_{\text{Paths}}(S') = L_{\text{Paths}}(S) \). Furthermore, since \( S' \) is a DAG, an expression \( R' \) of the form shown in Eqn. (4.3) such that \( L(R') = L_{\text{Paths}}(S') \) can be constructed by applying an existing path-expression algorithm.

4.4.3 Refinement Procedure

In § 4.4.2, we saw that it is possible to exactly capture the language of a refinement graph \( G \) with an expression \( R' \) of the form shown in Eqn. (4.3), when for each strongly-connected components \( S \), all the simple cycles of \( S \) are the same length and share a common vertex. We use this understanding to develop Alg. 8, which takes in a refinement graph and determines such an \( R' \).

The basic process of Alg. 8 is to identify cyclic behavior, to capture cyclic behavior using one *, and then to connect that cyclic behavior together in a new graph. This process is similar to graph condensation. In
Algorithm 8: SafeRefinement(G)

**Data:** G = (V, E, Λ) is a refinement graph

/* Create the Extended Condensation DAG Cond */

1. SCCs ← strongly connected components of G;
2. Head ← Empty Map; Tail ← Empty Map;
3. Cond = (V_c, E_c, Λ_c) ← Empty graph; /* Initialize the Extended Condensation DAG */
4. for each SCC of SCCs do
   5. (V_{comp}, E_{comp}, Λ_{comp}) ← ProccessSCC(SCC, Head, Tail, Λ);
6. V_c ← V_c ∪ V_{comp}; E_c ← E_c ∪ E_{comp}; Λ_c ∪ Λ_{comp};
7. for (u, w) ∈ E do /* Connect the sccs together */
8.   if u and w are not in the same scc then
9.     E_c ← E_c ∪ (Tail(u), Head(w));
10. Add entry and exit vertices to Cond with an edge from entry to every other vertex in Cond and an edge from every other vertex in Cond to exit;
11. Compute an expression for all paths from entry to exit in Cond using the labeling function Λ_c

Graph condensation, one vertex is created for each strongly connected component, which can be identified via the algorithm of Tarjan [129]; then, if there was an edge in the original graph between any vertices in two strongly connected components c₁ and c₂, then an edge is added between the vertices in the condensed graph that represent c₁ and c₂. The resulting condensed graph is a directed acyclic graph (DAG). The main conceptual difference with the standard algorithm and Alg. 8 is that strongly connected components are not necessarily condensed to a single vertex in Alg. 8. However, the basic structure of (i) process strongly connected components, and then (ii) connect the result remains. Because our algorithm follows this process, we call the graph it produces the extended condensation DAG.

To construct the extended condensation DAG, Alg. 8 calls Alg. 9 to process each strongly connected component. Alg. 9 performs a few tasks. At the very least, Alg. 9 works to create an over-approximating regular expression that captures the cyclic behavior of the component using a regular expression with *-height at most 1. The first few lines of Alg. 9 check to see if the incoming component is trivial or not. If a strongly connected component has a single vertex, then the branch at line (2) is taken. In this case, the algorithm will either return a new vertex with associated labeling a₁, or a₁*.

If the incoming strongly connected component has more than one vertex, all the simple cycles of the component are found. This task can be done via the algorithm of Johnson [80]. Then a check on the cycles is made. If the cycles have different lengths, or if they do not share a common vertex, then algorithm returns a new vertex with associated labeling (∑ a_i)*, where the a_i’s are the labels associated with the vertices of the component.

If the condition at line (11) is false, then, by Thm. 4.13, it is possible to capture the cyclic behavior of the component without adding any additional paths. Alg. 9 follows the construction from Thm. 4.13 in this case.

After the strongly connected components have been processed by Alg. 9, Alg. 8 hooks together the
Algorithm 9: ProcessSCC(SCC, Head, Tail, Λ_G)

Data: SCC a strongly connected component containing vertices v_1, ..., v_n
1 (V, E, Λ) ← Empty refinement graph;
2 if SCC has a single vertex v_1 then
3 if (v_1, v_1) ∉ SCC then
4 v' ← New vertex; Λ(v') ← Λ_G(v_1);
5 Head(v_1) ← (v'), Tail(v_1) ← (v');
6 return (i.e', ∅, Λ)
7 v' ← New vertex; Λ(v') = (Λ(v_1))^*;
8 Head(v_1) ← (v'), Tail(v_1) ← (v');
9 return (i.e', ∅, Λ)
10 Cycles ← all simple cycles of SCC including self-loops;
11 if the cycles of SCC have different lengths or there is no vertex common to all cycles in Cycles then
12 v ← New vertex; Λ(v) = (∑_{i=1,...,n} Λ_G(v_i))^*;
13 Head(v_i) ← (v), Tail(v_i) ← (v), for i = 1, ..., n;
14 return (i.e, ∅, Λ)
15 e ← a vertex that is common to all the cycles in Cycles;
16 Permute all the cycles in Cycles to start with e;
17 v_{star} ← New vertex; Λ(v_{star}) ← (∑_{(v_{i,j}, v_{i,j+1}) ∈ Cycles} Λ_G(v_{i,j}) ⋯ Λ_G(v_{i,m}))^*;
18 Head(e) ← v_{star}; V ← V ∪ {v_{star}};
19 for i = 1, ..., n do
20 t_i ← New vertex; Tail(v_i) ← t_i; Λ(t_i) ← Λ_G(v_i);
21 V ← V ∪ {t_i};
22 if v_i ≠ e then
23 h_i ← New vertex; Head(v_i) ← h_i; Λ(h_i) ← Λ_G(v_i);
24 V ← V ∪ {h_i};
25 for (v_{i,j}, ..., v_{i,m}) ∈ Cycles do
26 for k = 2, ..., m - 1 do
27 E ← E ∪ {(h_{i,k}, h_{i,k+1})};
28 for k = 1, ..., m - 1 do
29 E ← E ∪ {(t_{i,k}, t_{i,k+1})};
30 E ← E ∪ {(h_{i,m}, v_{star})} ∪ {(v_{star}, t_i)};
31 return (V, E, Λ)

resulting DAGs returned by Alg. 9 into the extended condensation DAG. Because every vertex is in some strongly connected component, perhaps a trivial one, Alg. 9 will consider every vertex in the refinement graph. Therefore, every vertex of the refinement graph will have an entry in both Head and Tail. With this in mind, Alg. 8 hooks together the DAGs returned by Alg. 9 by looking at each edge (v_i, v_j) in the refinement graph, and checking to see if v_i and v_j belong to different strongly connected components. If v_i and v_j are associated with different components, then Alg. 8 adds the edge (Tail(v_i), Head(v_j)) to the extended condensation DAG. Finally, Alg. 8 adds an entry and exit vertex to the extended condensation DAG, and adds the edges (entry, u) and (u, exit) for every vertex u. The algorithm finishes by determining an expression for all paths from entry to exit, using the labeling function of the extended condensation DAG. This task can be accomplished by moving the label of each vertex to its incoming edges, and using a traditional path-expression algorithm to capture all paths from entry to exit. The resulting path expression
We construct a refinement graph for this problem, by computing pair-wise feasibility of the summands of \( L_{\text{paths}}(G) \), where \( G \) is the original refinement graph, while satisfying the form of Thm. 4.11.

We now consider the complexity of Alg. 8. If we measure the complexity of Alg. 8 in terms of the size of graph \( G \), the running time is dominated by the time for finding simple cycles in Alg. 9. The best known algorithm for this problem is Johnson’s Algorithm, which has complexity \( O((|V| + |E|)(c + 1)) \), where \( c \) is the number of simple cycles in the graph. Johnson notes that \( c \) can be exponential in \( |V| \). However, we have found in practice that refinement graphs tend to be fairly simple (see § 4.6).

We now demonstrate Alg. 8 by analyzing in detail the loop in Fig. 4.1(a), with an algebraic analysis that uses our refinement algorithm at the evaluation of an iteration operator. We consider the same abstract domain described at the beginning of § 4.2. We denote the statements and conditions of the program, as well as their semantics in the abstract domain as follows:

\[
\begin{align*}
    s_1 := x &= 0 & D[s_1] &= \phi_{s_1} := x' = 0 \land y' = y \\
    s_2 := y &= 50 & D[s_2] &= \phi_{s_2} := y' = 50 \land x' = x \\
    c_{1,t} := [x < 100] & D[c_{1,t}] = \phi_{c_{1,t}} := x < 100 \land x' = x \land y' = y \\
    c_{1,f} := [x \geq 100] & D[c_{1,f}] = \phi_{c_{1,f}} := x \geq 100 \land x' = x \land y' = y \\
    s_3 := x &= x + 1 & D[s_3] &= \phi_{s_3} := x' = x + 1 \land y' = y \\
    c_{2,t} := [x > 50] & D[c_{2,t}] = \phi_{c_{2,t}} := x > 50 \land x' = x \land y' = y \\
    c_{2,f} := [x \leq 50] & D[c_{2,f}] = \phi_{c_{2,f}} := x \leq 50 \land x' = x \land y' = y \\
    s_4 := y &= y + 1 & D[s_4] &= \phi_{s_4} := y' = y + 1 \land x' = x
\end{align*}
\]

Suppose that the path-expression that described the set of paths of the procedure is

\[ s_1 \cdot s_2 \cdot (c_{1,t} \cdot s_3 \cdot (c_{2,t} \cdot s_4 + c_{2,f}))^* \cdot c_{1,f}. \]

The task of the analysis is then to evaluate \( D[s_1 \cdot s_2 \cdot (c_{1,t} \cdot s_3 \cdot (c_{2,t} \cdot s_4 + c_{2,f}))^* \cdot c_{1,f}] \). Consider refining the loop \( (c_{1,t} \cdot s_3 \cdot (c_{2,t} \cdot s_4 + c_{2,f}))^* \) using algorithm Alg. 8. First, we take the loop to the form \( (a_1 + \ldots + a_k)^* \).

We can do this by distributing \( \cdot \) through \( + \) to achieve the following:

\[ (c_{1,t} \cdot s_3 \cdot c_{2,t} \cdot s_4 + c_{1,t} \cdot s_3 \cdot c_{2,f})^* \]

Associate \( A \) with \( c_{1,t} \cdot s_3 \cdot c_{2,t} \cdot s_4 \) and \( B \) with \( c_{1,t} \cdot s_3 \cdot c_{2,f} \). We then have the goal of refining \( (A + B)^* \). We construct a refinement graph for this problem, by computing pair-wise feasibility of the summands of \( (A + B)^* \). We find that \( D[BA] = 0 \), but \( D[AA], D[AB], D[BB] \neq 0 \). Thus, we construct the refinement graph
depicted in Fig. 4.2(a), where the labeling of the vertices is shown.

Call the vertex with associated label A, \( z_1 \), and the vertex with associated label B, \( z_2 \).

The first phase of the algorithm is to build an extended condensation graph (DAG). First, as with traditional graph condensation, the strongly connected components are identified. For the graph in Fig. 4.2(a) there are two strongly connected components: vertex \( z_1 \), with a self-loop; and vertex \( z_2 \), also with a self-loop. Thus, Alg. 8 will call Alg. 9 on both of these components.

Consider the call on Alg. 9 with the strongly connected component that consists of vertex \( z_1 \). In this case, the strongly connected component has a single vertex. Therefore, in Alg. 9 the first then branch is taken. Alg. 9 then checks to see if \( z_1 \) has a self-loop. In the case of the graph in Fig. 4.2(a), this property is true. Therefore, a new vertex is created and associated with a label \( A^* \). Call this new vertex \( z'_1 \). Alg. 9 adds the mapping from \( z_1 \) to \( z'_1 \) in both Head and Tail, and then returns the graph containing just \( z'_1 \). The same process occurs with \( z_2 \).

Control is then returned to Alg. 8. The extended condensation DAG Cond just contains vertices \( z'_1 \) and \( z'_2 \) at this point. The final for-loop in Alg. 8 then adds the edge \((z'_1, z'_2)\) to the graph, because \((z_1, z_2)\) was in the original graph. After these vertices have been connected, the extended condensation DAG has been almost completely constructed. All that remains is to add entry and exit vertices, with an edge from entry to all other vertices, and an edge from all vertices to exit. The point of the entry and exit vertices is to indicate that, in the original expression, it is possible for the loop to start and end on any summand. The resulting DAG is shown in Fig. 4.2(b). Finally, the algorithm creates a refinement by computing an expression for all paths from entry to exit over the labels of Cond. For our example, the resulting expression is \( A^* + B^* + A^*B^* \). It can easily be shown that \( |=_{PKA} A^* + B^* + A^*B^* = A^*B^* \), which was the refined expression given in § 4.2.\(^2\)

We now give an example to demonstrate some of the more complicated aspects of Alg. 9. Suppose that we

\(^2\)Actually, the easiest way to obtain \( A^*B^* \) is to recognize that the edge from entry to \( z'_2 \) and the edge from \( z'_1 \) to exit are extraneous. Our actual implementation removes such extraneous edges, and thus the extended condensation DAG would just be a chain. However, for this presentation, we do not give the details on how to remove such edges, because precision is not affected in either case.
have as input the refinement graph shown in Fig. 4.3(a). Consider calling Alg. 9 with the non-trivial strongly connected component that consists of the vertices $z_1, z_2, \text{ and } z_3$ with respective associated labels $a, b, \text{ and } c$. This strongly connected component has more than one vertex, so the first branch is not taken. The algorithm then computes all the simple cycles of the strongly connected component, which can be accomplished by Johnson’s algorithm [80]. The simple cycles of this component are $\langle z_1, z_2 \rangle$ and $\langle z_2, z_3 \rangle$. Both of these cycles have length 2, and share the common vertex $z_2$, which means that this cyclic behavior can be captured by a single $*$, without adding any infeasible paths. That is, the branch at line (11) is not taken. The algorithm then permutes the cycles so that they start with the same vertex. Thus, after line (16), the cycles in $\text{Cycles}$ are $\langle z_2, z_1 \rangle$ and $\langle z_2, z_3 \rangle$. The algorithm captures the cyclic behavior of the component by creating a new vertex $v_{\text{star}}$ with associated labeling $(ba + bc)^*$. Thus, we have captured the cyclic behavior of this component with the labeling of $v_{\text{star}}$. The remaining problem is that the regular expression $(ba + bc)^*$ says that each path must start with $b$; however, $a \in L_{\text{Paths}}(G)$ but $a \not\in L((ba + bc)^*)$. Thus, we have to create heads and tails for this component to indicate how control can get to the cyclic part, and how control can leave the cyclic part. For $z_1$ and $z_3$, (i) the head and tail vertices $z_1^h$ and $z_3^1$ are created and given labels that are the same as $z_1$’s label; and (ii) $z_3^h$ and $z_3^1$ are created and given labels that are the same as $z_3$’s label. For $z_2$, we only create a tail vertex $z_2^t$, because starting the component with labeling $b$ is captured by the labeling of $v_{\text{star}}$. Alg. 9 also populates the Head and Tail maps as follows:

$$\begin{align*}
\text{Head}(z_1) &= z_1^h \\
\text{Tail}(z_1) &= z_1^1 \\
\text{Head}(z_2) &= v_{\text{star}} \\
\text{Tail}(z_2) &= z_2^1 \\
\text{Head}(z_3) &= z_3^h \\
\text{Tail}(z_3) &= z_3^1
\end{align*}$$

The algorithm returns control back to Alg. 8. Alg. 8 also processes the other (trivial) strongly connected components of the refinement graph. Once Alg. 8 has processed all the strongly connected components, the next task is to connect the components together. This task is accomplished by connecting a tail vertex to a head vertex whenever such an edge exists in the original graph. For this example, in the original graph there is an edge between the vertex with labeling $e$ and the vertex with labeling $c$. Consequently, Alg. 8 adds an edge in the extended condensation DAG between the tail vertex with labeling $e$ and the head vertex with labeling $c$. Alg. 8 repeats this action for all of the edges in the original refinement graph. Thus, excluding entry and exit vertices, we are left with the extended condensation DAG depicted in Fig. 4.3(b). To finish this example, Alg. 8 adds entry and exit vertices, and then computes a regular expression for all paths from entry to exit over the labels of the resulting graph.

In § 4.2, we observed in the example of the analysis of the program in Fig. 4.1(b) that a plausible refinement lead to a worse analysis result compared with just evaluating a regular expression that reflected the syntax of
the program. We originally had the path expression \((A + B + C)^*\). We now show how our algorithm avoids producing the refinement discussed in § 4.2. Instead, Alg. 8 returns the original path expression \((A + B + C)^*\).

Based on the infeasible paths noted in § 4.2, we would obtain the refinement graph depicted in Fig. 4.4(a). The only strongly connected component of this graph is the whole graph. Thus, Alg. 8 would call Alg. 9 with the graph depicted in Fig. 4.4(a). This graph is not a trivial strongly connected component, so Alg. 9 would compute all the simple cycles of the strongly connected component, which are (in terms of labelings) \(\langle A \rangle\), \(\langle B, C \rangle\), and \(\langle A, B, C \rangle\). Note that these cycles do not share a common length. Consequently, the then branch at line (11) is taken, and a new vertex is created with the labeling \((A + B + C)^*\). Control returns to Alg. 8, and the extended condensation DAG depicted in Fig. 4.4(b) is created. Thus, the final expression created for this example is exactly the same expression it was given as input, namely, \((A + B + C)^*\).

**Theorem 4.14.** Let \(E\) be some set of infeasible paths and let \(R_{inf}\) be a regular expression such that \(L(R_{inf}) = E\). Now let \(G\) be a refinement graph with respect to \(R_{inf}\) and \((a_1 + \ldots + a_n)^*\). Alg. 8 will produce a regular expression \(R'\) with

1. \(E \models_{KA} R' = (a_1 + \ldots + a_n)^*\) and \(L(R') \subseteq L(a_1 + \ldots + a_n)^*\)

2. \(\models_{PKA} R' \subseteq (a_1 + \ldots + a_n)^*\)
Alg. 10 determines infeasible sequences of precision guarantees for PKA domains. Suppose that Lemma 4.15.

\[
(\sum_{(v_{1},...,v_{m})}\in Cycles(\Lambda_{G}(v_{1}) \cdot \ldots \cdot \Lambda_{G}(v_{l}, m)))^* \quad or \quad (\sum_{i=1,...,n} \Lambda_{G}(v_{i}))^*. \quad Both \ of \ these \ satisfy \ the \ form \ of \ interest, \ and \ are \ the \ only \ star \ expressions \ in \ R'. \quad Thus, \ Alg. \ 8 \ always \ produces \ an \ expression \ of \ the \ form \ show \ in \ Eqn. \ (4.3). \quad That \ is, \ Alg. \ 8 \ returns \ an \ expression \ R', \ where \ R' \ has \ star-height \ no \ greater \ than \ 1, \ and \ every \ summand \ under \ a \ star \ in \ R' \ has \ the \ same \ length. \quad By \ Thm. \ 4.11, \ this \ property \ is \ enough \ to \ conclude \ that \ \equiv_{PKA} R' \leq (a_{1} + \ldots + a_{n})^* \ holds.
\]

In regards to (1), \(\mathcal{E} \equiv_{KA} R' = (a_{1} + \ldots + a_{n})^* \) holds because the only difference between \(R' \) and \((a_{1} + \ldots + a_{n})^* \) is the (possible) removal of some of the infeasible paths in \(\mathcal{E} \). As for \(L(R') \subseteq L(a_{1} + \ldots + a_{n})^* \), \(L((a_{1} + \ldots + a_{n})^*) \) contains all words over the alphabet \(\{a_{1} + \ldots + a_{n}\} \). Alg. 8 creates a regular expression \(R' \) over the same alphabet, so trivially \(L(R') \subseteq L(a_{1} + \ldots + a_{n})^* \).

Thm. 4.14 shows that Alg. 8 meets the goal (4.3.2) described in § 4.3.2 for an expression \(R^* \) where \(R \) is star-free. That is, Alg. 8 is a procedure that can be used to refine a most inner loop, and provides analysis precision guarantees for PKA domains.

### 4.5 Putting Refinement to Work in a Program Analyzer

**Refining Arbitrary Regular Expressions.** In § 4.4, we gave an approach for refining regular expressions of the restricted form \(R^* \), where \(R \) is star-free, to another regular expression \(R' \). We showed in Thm. 4.14 that \(L(R') \subseteq L(R^*) \) and \(\equiv_{PKA} R' \leq R^* \). In this section, we show how such a method can be incorporated to rewrite an arbitrary regular expression \(E \) to another expression \(E' \) with \(L(E') \subseteq L(E) \) and \(\equiv_{PKA} E' \leq E \).

The method works bottom-up. Suppose that we have an expression \(R^* \) where \(R \) is not star-free. Alg. 10 first takes \(R \) to a “sum-of-products” form. Then Alg. 10 calls itself recursively to refine each of the summands, \(A_{i} \), of the transformed \(R \) to obtain refined expressions \(A_{i}' \). For each \(A_{i}' \), Alg. 10 associates a new label \(a_{i} \).

Alg. 10 determines infeasible sequences of \(a_{i} \)'s by analyzing sequences of \(A_{i}' \)'s. Alg. 10 then uses the methods from § 4.4 to refine the expression \((a_{1} + \ldots + a_{n})^* \) based on the detected infeasible sequences. Finally, Alg. 10 returns the resulting refined expression with each \(a_{i} \) replaced by \(A_{i}' \). As to the complexity of Alg. 10, we note that line (8) can cause an exponential blow-up in the size of the regular expression when the body of a loop is transformed to a sum-of-products form. In other words, \((A_{1} + \ldots + A_{n}) \) can be exponentially larger than \(R \).

We now prove that Alg. 10 refines an arbitrary expression \(E \) and gives back an expression \(E' \) with \(\equiv_{PKA} E' \leq E \). First, we give a lemma about pre-Kleene algebras.

**Lemma 4.15.** Suppose that \(\equiv_{PKA} a_{1} \leq b_{1} \) and \(\equiv_{PKA} a_{2} \leq b_{2} \) hold. Then \(\equiv_{PKA} a_{1} + a_{2} \leq b_{1} + b_{2} \) and \(\equiv_{PKA} a_{1}a_{2} \leq b_{1}b_{2} \) hold.
Algorithm 10: GeneralRefinement(E)

Data: E is a regular expression over an alphabet \( \Sigma \)

1. If E is a label in \( \Sigma \), then
2. Return E
3. If \( E=R_1 + R_2 \), then
4. Return GeneralRefinement(\( R_1 \)) + GeneralRefinement(\( R_2 \))
5. If \( E=R_1 \cdot R_2 \), then
6. Return GeneralRefinement(\( R_1 \)) \cdot GeneralRefinement(\( R_2 \))
7. If \( E=R^* \), then
8. \( (A_1 + \ldots + A_n) \leftarrow \) distribute \( \cdot \) through \( + \) in \( R \);
9. Associate a new alphabet symbol \( a_i \) with each summand \( A_i \);
10. \( A_i' \leftarrow \) GeneralRefinement(\( A_i \));
11. \( D[a_i] \leftarrow D[A_i'] \);
12. Identify some set Seq of sequences of \( a_i \)'s; /* Candidate infeasible sequences */
13. For all sequences \( a_{i_1} \cdot \ldots \cdot a_{i_n} \) of Seq do
14. If \( D[a_{i_1} \cdot \ldots \cdot a_{i_n}] = 0 \) then
15. Inf \( \leftarrow \) Inf \cup \( a_{i_1} \cdot \ldots \cdot a_{i_n} \);
16. Build a refinement graph G out of Inf and \( a_i \)'s;
17. \( E' \leftarrow \) SafeRefinement(\( G \)); /* Alg. 8 */
18. Return Replace all occurrences of \( a_i \) in \( E' \) with \( A_i' \)

Proof. \( |=_{PKA} a_1 \leq b_1 \) means \( |=_{PKA} a_1 + b_1 \leq b_1 \). Similar for \( a_2 \) and \( b_2 \).

\[
|=_{PKA} a_1 + a_2 \leq a_1 + a_2 + b_1 + b_2 = b_1 + b_2
\]

\[
|=_{PKA} a_1 a_2 \leq a_1 a_2 + a_1 b_2 + a_2 b_1 + a_2 b_2 = (a_1 + b_1)(a_2 + b_2) = b_1 b_2
\]

\[\square\]

Theorem 4.16. Suppose that E is a regular expression, and \( E' = \) GeneralRefinement(\( E \)). Then

1. \( L(E') \subseteq L(E) \)
2. \( |=_{PKA} E' \leq E \)

Proof. The proof is by structural induction on E.

The base case is when E is just a label, in which case the conditions hold trivially.

For the recursive cases, let \( E'_\text{sub} = \) GeneralRefinement(\( E_\text{sub} \)) for each sub-expression \( E_\text{sub} \) of E. The induction hypothesis states that for each sub-expression \( E_\text{sub} \) of E, \( L(E'_\text{sub}) \subseteq L(E_\text{sub}) \) and \( |=_{PKA} E'_\text{sub} \leq E_\text{sub} \)

1. The cases for \( E = R_1 + R_2 \) and \( E = R_1 \cdot R_2 \) follow directly from Lem. 4.15.
2. If \( E = R^* \), Alg. 8 first converts R to have the form \( A_1 + \ldots + A_n \), and then refines each \( A_i \) into \( A'_i \), calling Alg. 10 recursively. It then uses Alg. 8 to build a refined regular expression \( E' \) using the labels \( \{a_i\} \).
Note that by Thm. 4.14, \( L(E') \subseteq L((a_1 + ... + a_n)^*) \) and \( \models_{\text{PKA}} E' \leq (a_1 + ... + a_n)^* \). By Lem. 4.15, as well as the monotonicity axiom of star for a pre-Kleene algebra (Defn. 4.4, item 4), if we replace each occurrence of \( a_i \) in \( E' \) with \( A_i \) to obtain \( E_A \), we can conclude \( L(E_A) \subseteq L((A_1 + ... + A_n)^*) \) and \( \models_{\text{PKA}} E_A \leq (A_1 + ... + A_n)^* = E \). Furthermore, because of the induction hypothesis, we have \( L(A_1') \subseteq L(A_1) \) and \( \models_{\text{PKA}} A_1' \leq A_1 \). Consequently, if we replace each \( a_i \) in \( E' \) with \( A_i' \) to obtain \( E_A' \), we can conclude \( L(E_A') \subseteq L(E_A) \subseteq L((A_1 + ... + A_n)^*) \) and \( \models_{\text{PKA}} E_A' \leq E_A \leq (A_1 + ... + A_n)^* = E \). The algorithm returns \( E_A' \).

\[\square\]

Thm. 4.16 shows that for the most part Alg. 10 fits the description of a refinement procedure that meets the goal described in § 4.3.2 for any arbitrary expression. The only difference is that Alg. 10 does not refine \( R \) based on some externally given set of infeasible paths \( \mathcal{E} \). Instead Alg. 10 refines sub-expressions based on an internal strategy for detecting infeasible path sequences (lines (12)–(15)). The benefits of this design are described below.

**Recognizing Infeasible Sub-Paths.** The second issue that must be addressed to put our refinement technique to work in a program analyzer is that of recognizing infeasible sub-paths. lines (12)–(15) of Alg. 10 constitute a mechanism for identifying a set of infeasible sub-paths, which could be used with different policies. There are a few interesting points to observe about this mechanism.

- It exploits compositionality: the values in the sequences \( a_{i,1} \cdot ... \cdot a_{i,n} \) used in the test \( D[a_{i,1} \cdot ... \cdot a_{i,n}] = 0 \) involve summary values computed for subterms of \( R \) (where \( E \) has the form \( R^* \)). The mechanism applies to non-leaf loops because a summary, in the form of an abstract-domain value, will have been computed for each more-deeply-nested loop contained within \( R \).

- Because the test \( D[a_{i,1} \cdot ... \cdot a_{i,n}] = 0 \) merely involves evaluation in the abstract domain, the test is decidable.

- Because the test is performed using the *same* abstract domain employed everywhere else in the analyzer, there should be a good “impedance match” with the rest of the analyzer. That is, the test will only cause a sub-path to be excluded if the abstract domain has enough fidelity to observe the properties that cause the sub-path to be infeasible.

These properties contrast with the method Sharma et al. [125] use to identify splitter predicates: their method also works bottom-up, but for non-leaf loops they need to rely on a separate method for identifying loop invariants of inner loops.
In our experiments, we used a simple policy of checking all pairs of summands when $R$ is put in sum-of-products form, as discussed in the paragraph just after Ex. 4.1.

### 4.6 Experiments

Our experiments were designed to answer the questions posed below.

The algorithm given in § 4.4 refines a regular expression $R$ into a refinement $R'$ for which it is guaranteed that the results obtained with $R'$ are no worse than those obtained with $R$. However, in practice, we would also like a refinement to provide better answers. One measure of success is whether employing the refinement algorithm improves some “down-stream task” that uses the analysis results.

#### Experimental Question 1: Does our refinement algorithm allow an analysis to prove more assertions in practice?

In practice, we often find that the most expensive part of an analysis is the evaluation of the $*$ operators in a regular expression. The refinement procedure works bottom-up, repeatedly taking a regular expression of the form $R^*$, where $R$ is star-free, and producing another expression $R'$. At each level, there is only one star operator to evaluate; however, there is no a priori bound on the number of star operators in $R'$. Thus, it could be the case that the refinement procedure increases precision, at the cost of substantially increased analysis time.

#### Experimental Question 2: Do our refinements greatly increase analysis time in practice?

We would also like to understand how well our approach performs compared to alternative methods for static program analysis. It is possible that our techniques increase analysis precision in practice, but not enough to make algebraic program analyses competitive with other analysis techniques.

#### Experimental Question 3: How does the performance of an algebraic program analyzer that uses our refinement techniques compare to that of state-of-the-art model checkers?

### 4.6.1 Experimental Setup

We implemented Alg. 10 as an extension of the implementation of Compositional Recurrence Analysis (CRA) [49]. At each star, our implementation checks the feasibility of pairs of labels $a_i$, $a_j$ to see whether $a_i a_j$ or
\(a_1 a_1\) are infeasible action sequences. We tested the algorithm using two different abstract domains, which we call KCBR (the technique of Chapter 3) and ACI [7] (Ex. 4.5). The KCBR domain is more expressive than the ACI domain, but does not satisfy (all of) the axioms of a pre-Kleene algebra (Defn. 4.4). Thus, there is no longer a guarantee that the results from analyzing the refined regular expression are no worse than the results from analyzing the original expression. However, because our refinements are always sound in the sense of Defn. 4.7, the resulting analysis will still be a (sound) over-approximation. The ACI domain always satisfies the axioms, so the refinement algorithm is guaranteed to give analysis results that are at least as good as those obtained without using refinement.

To answer the experimental questions, we ran our implementation(s), as well as the software model checkers Ultimate Automizer [68] version 0.1.23 and SeaHorn [66] version 0.1.0, on several suites of micro-benchmark programs containing only true assertions. In Tab. 4.1, we report for each suite (i) the number of programs for which the analyzer was able to prove all assertions, (ii) the total analysis time, and (iii) the number of timeout and out-of-memory exceptions. Timings (with a timeout limit of 300 seconds) were taken on a virtual machine (using Oracle VirtualBox) with 8GB of RAM, with a guest OS of Ubuntu 14.04, host OS of CentOS 6.9, and a 3.2 GHz quad-core Intel Core i5-4570 host CPU. The programs come from the following sources:

- 35 programs are from a suite used to test the resource-bound-analysis tool C4B [27].
- 46 programs are from a suite used to test the invariant-generating tool HOLA [44].
- 96 programs are from the \textit{Integers and Control Flow—Loops} subcategory of SV-COMP16 [128].
- 47 programs are benchmarks containing multi-path loops that we created to test the analyzer.

Answers to the three experimental questions are given in § 4.6.2, § 4.6.3, and § 4.6.4.

4.6.2 EQ1: Does Refinement Allow More Assertions to be Proven?

In short, the answer is, “Yes, the refinement algorithm allows the analyzer to prove more assertions in practice. Overall, refinement helped both KCBR and ACI prove over 25% more assertions than without refinement.”

There was one instance where the additional overhead of refinement increased the analyzer’s memory usage enough that the analysis could not complete successfully. This example was in the \textit{loops} suite, and is the reason that, for both the KCBR and ACI domains, the number of assertions proved decreased for that suite when using refinement. For all other suites, the refinement algorithm allowed both the KCBR and ACI domains to prove strictly more assertions than when the analysis was performed without refinement.
Table 4.1: The results of the assertion-checking experiments. Column 2 shows the total number of programs in each benchmark suite. Columns 3-20 show analysis results under six different conditions: with abstract domain KCBR or ACI, each with or without using the refinement algorithm, and using the state-of-the-art model checkers Ultimate Automizer and SeaHorn. For each configuration, the left column indicates the total running time (in seconds), the middle column indicates the number of programs in which all assertions were proven by the analyzer, and the right column is a pair T/M where T is the number of timeouts and M is the number of out-of-memory exceptions. In each row, the smallest running time and the greatest number of assertions proved are shown in boldface.

<table>
<thead>
<tr>
<th>Benchmark Suite</th>
<th>Total #P</th>
<th>KCBR Time #P #E</th>
<th>KCBR + refinement Time #P #E</th>
<th>ACI Time #P #E</th>
<th>ACI + refinement Time #P #E</th>
<th>UAutomizer Time #P #E</th>
<th>SeaHorn Time #P #E</th>
</tr>
</thead>
<tbody>
<tr>
<td>C4B</td>
<td>35</td>
<td>37.7 19 0/0</td>
<td>50.4 28 0/0</td>
<td>36.0 19 0/0</td>
<td>49.5 28 0/0</td>
<td>4118.7 24 11/0</td>
<td>1861.2 29 6/0</td>
</tr>
<tr>
<td>HOLA</td>
<td>46</td>
<td>62.0 33 0/0</td>
<td>90.1 42 0/0</td>
<td>59.6 30 0/0</td>
<td>86.5 37 0/0</td>
<td>2467.3 36 7/0</td>
<td>1212.8 38 4/0</td>
</tr>
<tr>
<td>SV/loop-accel</td>
<td>19</td>
<td>13.2 13 0/0</td>
<td>17.6 15 0/0</td>
<td>12.6 12 0/0</td>
<td>17.4 13 0/0</td>
<td>2554.0 11 7/0</td>
<td>907.3 15 3/0</td>
</tr>
<tr>
<td>SV/loop-lit</td>
<td>19</td>
<td>32.1 18 0/0</td>
<td>56.6 19 0/0</td>
<td>31.5 18 0/0</td>
<td>53.2 19 0/0</td>
<td>2221.2 12 7/0</td>
<td>474.1 17 1/0</td>
</tr>
<tr>
<td>SV/loop-new</td>
<td>15</td>
<td>12.0 10 0/0</td>
<td>16.1 13 0/0</td>
<td>11.4 10 0/0</td>
<td>15.8 12 0/0</td>
<td>940.1 13 2/0</td>
<td>302.9 13 1/0</td>
</tr>
<tr>
<td>SV/loops</td>
<td>8</td>
<td>6.2 6 0/0</td>
<td>8.1 7 0/0</td>
<td>6.1 4 0/0</td>
<td>6.0 5 0/0</td>
<td>1469.3 4 4/0</td>
<td>301.8 6 1/0</td>
</tr>
<tr>
<td>misc</td>
<td>33</td>
<td>45.7 22 0/0</td>
<td>68.4 21 0/1</td>
<td>44.3 22 0/0</td>
<td>68.9 21 0/1</td>
<td>2275.2 26 6/0</td>
<td>910.4 26 3/0</td>
</tr>
<tr>
<td>misc</td>
<td>47</td>
<td>39.4 20 0/0</td>
<td>70.3 36 0/0</td>
<td>38.2 19 0/0</td>
<td>68.0 33 0/0</td>
<td>1909.4 43 4/0</td>
<td>8.7 43 0/0</td>
</tr>
<tr>
<td>Total</td>
<td>222</td>
<td>248.3 141 0/0</td>
<td>377.6 181 0/1</td>
<td>239.7 134 0/0</td>
<td>365.3 168 0/1</td>
<td>17955.5 169 48/0</td>
<td>5979.2 187 19/0</td>
</tr>
</tbody>
</table>

Consider the programs shown in Fig. 4.5. These programs each contain one assertion. When using either the KCBR or the ACI domain, neither assertion can be proven by the analyzer without using refinement, but both assertions can be proven with refinement.

The program shown in Fig. 4.5(a) is *leapsum2.c* from the misc suite. The loop body in this program has two paths; we will use the label A for the path that takes the then-branch of the conditional statement, and the label B for the path that takes the else-branch. When we check the feasibility of path sequences, we discover that an iteration taking path A can only be followed an iteration taking path B, and vice versa. As a result, we can refine a regular expression of the form \((A + B)^*\) to one of the form \((\epsilon + B)(A \cdot B)^*(\epsilon + A)\). The refined star body \(A \cdot B\) can be described by the transition formula \(x' = x + 2 \land y' = y + 2 \land z' = 1 \land t' = t + 1\). This formula implies that the difference between \(x\) and \(y\) after the star \((A \cdot B)^*\) is the same as before the star, which is the key step in proving that the assertion holds. In contrast, without refinement, the analyzer produces a summary for the loop by summarizing its two paths and joining them with a disjunction; the resulting summary is imprecise and does not imply that the difference between \(x\) and \(y\) remains bounded between 1 and \(-1\).

The program shown in Fig. 4.5(b) is *maxequals_linear_2.c* from the misc suite. This program illustrates a less-obvious application of refinement. There are two paths A and B through the loop body, depending on whether \(x\) is assigned the left (A) or the right (B) subexpression of the max macro, i.e., depending on whether or not \(x > 75 - 10 \cdot t\) holds. The subexpression \(75 - 10 \cdot t\) always decreases from one iteration to the next, so any execution of the loop proceeds in two phases: a first phase of at most one iteration in which B occurs, and a second phase in which only A occurs. Therefore, when refinement is used, the analyzer can
void main(int N) {
    int x = 1; int y = 2; int z = 1;
    for (int t = 0; t < N; t++) {
        if (z > 0) {
            x = x + 2;
            z = -1;
        } else {
            y = y + 2;
            z = 1;
        }
    }
    assert(x + z == y);
}

#define max(A, B) ((A > B) ? A : B)

int x;

void loop() {
    for (int t = 2; t <= 10; t++) {
        x = max(x, 75 - 10 * t);
    }
}

void main() {
    x = 0;
    loop();
    assert(x == 55);
}

(a) misc/leapsum2.c

(b) misc/maxequals_linear_2.c

Figure 4.5: Two examples of programs having assertions that the analyzer can prove only when using the refinement algorithm.

conclude that x will ultimately have the value 55, which is the value of 75 - 10 * t on the first iteration. This property suffices to prove the assertion. Without refinement, the analyzer fails to find any upper bound on x that is implied by the loop body’s transition formula, and therefore the analyzer cannot prove the assertion.

4.6.3 EQ2: Does Refinement Greatly Increase Analysis Time?

In short, the answer is, “The refinement algorithm caused an increase in overall analysis time of about 50%.”

Because both the KCBR and ACI domains are based on transition formulas, the infeasibility check on line (14) of Alg. 10 results in a call to an SMT solver. Experimentally, we found that these feasibility checks are often the main contributor to the increased analysis time when using refinement. As a corollary, we find that while in principle Alg. 8 can be exponential time for certain complicated refinement graphs, in our experience Alg. 8 does not significantly contribute to increased analysis time.

4.6.4 EQ3: How Does CRA With Refinement Compare with State-of-the-Art Model Checkers?

In terms of assertions proved, we found experimentally that equipping the KCBR and ACI domains with refinement made CRA—with each domain—competitive with Ultimate Automizer and SeaHorn. Restricting the comparison to KCBR versus SeaHorn, we see that refinement allows KCBR to at least tie SeaHorn, in terms of assertions proved, for the HOLA, loop-accel, and loop-lit suites, whereas KCBR without refinement is well behind SeaHorn for these suites. Furthermore, refinement allows KCBR to overtake SeaHorn for the loop-new category and increase its lead in the loop-invgen category.

In terms of total analysis time, CRA with the KCBR and ACI domains (either with or without refinement) completes the test suite in a fraction of the time taken by Ultimate Automizer or SeaHorn. The slowest variant of CRA—based on the KCBR domain with refinement—is more than forty-seven times faster than Ultimate Automizer and fifteen times faster than SeaHorn. However, these speedup ratios are due, in part, to the fact
that CRA with KCBR or ACI never timed out on any example, whereas Ultimate Automizer and SeaHorn, which are both based on abstraction refinement, timed out on many examples. Thus, the total times given in Tab. 4.1 are highly dependent on the chosen timeout value, here 300 seconds. Nevertheless, the figures of >47x and >15x improvement are valid in the sense that we optimistically credit Ultimate Automizer and SeaHorn as having completed in 300 seconds for the examples on which they time out.

If we exclude programs for which Ultimate Automizer timed out, then Ultimate Automizer took 20 seconds on average to analyze a program. Similarly, SeaHorn took 1.4 seconds on average for programs on which it did not time out. These numbers should be compared to 1.08 and 1.7 seconds per program for the fastest and slowest variants of CRA. However, there were example programs for which Ultimate Automizer and SeaHorn took much longer than average. Excluding timeouts, the maximum time that each tool took to analyze a program was 10.1, 12.0, 9.9, 11.5, 261.7, and 168.9 seconds for KCBR, KCBR+refinement, ACI, ACI+refinement, Ultimate Automizer, and SeaHorn, respectively.

4.7 Related Work

Cousot and Cousot [36] developed a general theory of semantically justified program transformation based on abstract interpretation. They view a syntactic program as an abstraction of its semantics, and a syntactic transformation as a (conceptual) decompilation of an associated semantic transformation. They develop associated correctness conditions for when a syntactic transformation is an over-approximation of a semantic transformation. Their methodology aims to provide a conceptual framework for proving that, compared to the original program, each program produced by some transformation algorithm has some desired property in the concrete semantics. A similar approach could be used in our context, but we have the slightly different goal of being able to show that, for each transformation $R \rightarrow R'$, certain desirable properties hold for both the concrete semantics of $R$ and $R'$ and the abstract semantics of $R$ and $R'$.

Our work represents a framework that can be instantiated with different abstract domains. It comes equipped with a transformation algorithm taking, e.g., $R$ to $R'$, for which (i) the concrete semantics of $R'$ is sound with respect to the concrete semantics of $R$, and (ii) the abstract semantics of $R'$ yields a value that is sometimes better—and never worse—than the abstract semantics of $R$. The framework uses two related algebras, Kleene algebra and pre-Kleene algebra, to characterize the concrete and abstract properties on which the transformation algorithm relies. We are then able to use algebraic reasoning to prove properties of the transformation algorithm. In a similar vein, Kot and Kozen [93] consider an axiomatization that is weaker than Kleene algebra, which they use in an algorithm to compute the closure of a matrix with respect to a cutset of the control-flow graph. This weaker axiomatization is incomparable to the pre-Kleene axiomatization.
given in this chapter. The axiomatization used in Kot and Kozen [93] does not assume full distributivity laws, which we do. However, they assume the ascending-chain condition, which allows them to consider a more restrictive $*$ than the ones considered in this chapter. Kozen [96] also uses an axiomatization based on Kleene algebra, called Kleene algebra with tests (KAT) in a static-analysis context. This work differs from ours in that Kozen [96] uses a complete equational theory and careful manual reasoning, while we use an approximate abstract domain and automatic reasoning. At a high level, Kozen [96] annotates a program’s path expression, say $R$, with a (security) policy to obtain an $R'$. Then, to prove that $R$ satisfies the policy, Kozen [96] uses a special theorem prover to show, using our notation, $E \models_{KAT} R \leq R'$, for some manually chosen set $E$. We, on the other hand, automatically determine a set of infeasible paths $E$, and refine our original expression $E$ to an expression $E'$ with $E \models_{PKA} E' \leq E$.

The idea of applying transformations to a program’s IR as a way to improve the results of static analysis has a long history. In some work, transformations are explicit, such as the abstraction-refinement method used in SLAM [12], approaches based on isolating hot paths [54, 5, 99], and techniques for rewriting loops [11, 64, 125, 57]. In other work, transformations are implicit: extra information—typically information about the execution context—is used to label values that arise during the course of an analysis. This approach is tantamount to splitting lazily the elements of the IR, where each duplicated IR element is then associated with a single analysis value. The latter idea appears in numerous places, going back at least to the work of Holley and Rosen [70] on “Qualified data flow problems.” Other instances of the idea include the call-strings approach to interprocedural dataflow analysis [124], weighted pushdown systems [21, 111], and trace partitioning [115]. It has been used in such systems as ESP [41], Archer [139], and Saturn [43], and in analysis libraries such as Moped [122] and WALi [83].

Our work performs an explicit rewrite of the IR used by the analyzer (namely, a regular expression). It was directly inspired by experience with an implementation of the algorithm of Sharma et al. [125]. While that algorithm works well for some examples, it inspired us to investigate whether an algorithm loop-transformation could provide an “improvement guarantee.” While our algorithm does not guarantee improved analysis results, it does guarantee not to produce worse (or incomparable) results (see Thm. 4.16).

There is a huge literature on loop transformations for the purpose of optimizing a program’s execution time: transformations are typically performed on innermost loops, or loop nests. However, that work is more a client of an analysis, and focused on reducing the execution time, whereas the focus of our work is on improving analysis precision.
Chapter 5

Simplifying Non-linear Formulas

The problem of finding a constant bound on a term given a set of assumptions has wide applications in optimization as well as program analysis. However, in many contexts the objective term may be unbounded. Still, some sort of symbolic bound may be useful. In this chapter, we introduce the optimal symbolic-bound synthesis problem, and a technique that tackles this problem for non-linear arithmetic with function symbols.

This allows us to automatically produce symbolic bounds on arithmetic expressions from a set of both equality and inequality assumptions. Our solution employs a novel combination of powerful mathematical objects—Gröbner bases together with polyhedral cones—to represent an infinite set of implied inequalities. We obtain a sound symbolic bound by reducing the objective term by this infinite set.

We implemented our method in a tool, AutoBound, which we tested on problems originating from real Solidity programs. We find that AutoBound yields relevant bounds in each case, matching or nearly-matching upper bounds produced by a human analyst on the same set of programs.

5.1 Introduction

In this chapter we introduce and address the following problem, which we call the optimal symbolic-bound synthesis (OSB) problem:

Given a (potentially non-linear) arithmetic formula \( \phi \) representing assumptions and axioms and an objective term \( t \), find an arithmetic term \( t^* \) such that

1. (Bound) \( \phi \models t \leq t^* \)

2. (Optimality) For every term \( s \) that satisfies the first condition, \( t^* \leq s \) holds, where \( \leq \) represents some notion of “term desirability.”
A solution to this problem has many applications in the automatic analysis of programs. For example, a common program-analysis strategy is to extract the semantics of a program as a logical formula, and then reason about the resulting formula for some downstream task. One such downstream task is to verify that some resource, such as time, space, or the value of some financial asset does not exceed some value.

The problem of finding constant bounds on a term $t$ given assumptions $\phi$ is commonly addressed in the field of optimization. However, in the context of program analysis we are often not interested in constant bounds on such a term—either because $t$ is unbounded, or because the bound on $t$ is so loose as to be uninformative. An alternative approach—the one adopted in this chapter—is to find a symbolic bound, given assumptions $\phi$.

The OSB problem as given above is very general. Namely, we have yet to specify any restrictions on $\phi$, $\models$, or the term-desirability order $\preceq$. In future we would like others to consider methods to address OSB problems for various instantiations of $\models$ and $\preceq$. In this chapter, we consider the OSB problem in the context in which $\phi$, $\models$ and $t$ are interpreted over non-linear arithmetic, and $\preceq$ gives a intuitive, human notion of a simpler term. Moreover, we do not place a restriction on the form of $\phi$. That is, $\phi$ is an arithmetic formula with the usual boolean connectives.

This setting introduces the significant challenge of non-linear arithmetic reasoning: (i) in the case of rational arithmetic, it is undecidable to determine whether any $t'$ is a bound on $t$, let alone find an optimal bound; (ii) in the case of real arithmetic, reasoning is often prohibitively expensive. In the setting of finding bounds, the challenge is finding a finite object to represent the infinite set of upper-bounds implied by the formula $\phi$. In the case of linear arithmetic, convex polyhedra can be represented finitely, and can completely represent the set of inequality consequences of a linear formula $\phi$. Moreover, manipulating polyhedra is often reasonably efficient. However, in the non-linear rational case no such complete object exists, and in the non-linear real case manipulating the corresponding object is computationally challenging.

To address this challenge we introduce a mathematical object we call a cone of polynomials (§ 5.4.3) to hold on to an infinite set of non-linear inequalities. A cone of polynomials, consists of a polynomial ideal (§ 2.2), which captures equations, and a polyhedral cone (§ 5.3.1), which captures inequalities. Cones of polynomials strike a balance between expressiveness and computational feasibility—using non-linear reasoning on equalities through the ideal, and linear reasoning on inequalities through the linear cone, gives efficient yet powerful non-linear reasoning on inequalities.

A cone of polynomials is similar to a wedge from Chapter 3 in that both are conjunctive and use Gröbner bases and congruence-closure for reasoning about equations and linear techniques for reasoning about

---

1See the discussion on Positivstellensatz in § 5.9.
inequalities. However, a cone of polynomials contrasts with a wedge in how we construct them from a formula via saturation.

We utilize cones of polynomials to address the non-linear OSB problem in a two-step process: (1) From \( \phi \) create an implied cone of polynomials \( C \). That is, \( C \) is an infinite collection of inequalities, each implied by \( \phi \). (2) Reduce the term \( t \) by \( C \) to obtain \( t^* \).

Due to the difficulties of non-linear arithmetic the first step is necessarily incomplete. However, the second step (reduction) is complete with respect to cones of polynomials. Our reduction method for cones of polynomials makes use of a sub-algorithm that reduces a linear term by a polyhedron. That is, in § 5.4.2 we give an algorithm (Alg. 12) that solves the OSB problem, where \( \phi \) is a conjunction of linear inequalities (a polyhedron), \( \models \) is interpreted as linear arithmetic, and \( \preceq \) is an order that encodes preferability of the dimensions of the returned bound. This method makes use of a novel local projection (§ 5.4.2) method. Local projection can be seen as an incomplete method of quantifier elimination for polyhedra that avoids the blow-up of a full-projection method such as Fourier-Motzkin. Nevertheless, local projection suffices in the case of the OSB problem for polyhedra. In § 5.7, we compare our reduction method based on local projection with a, perhaps more obvious, approach based on multi-objective linear programming. We find that our algorithm solves the problem much more efficiently than the LP approach.

In § 5.4.3, we show (Thm. 5.20) how the polyhedral OSB solution can be extended to the setting of cones of polynomials. This means that in particular we are able to completely solve OSB with respect to a polynomial \( t \) and a polynomial cone \( C \), which has the property that the result, \( t^* \), is optimal with respect to any other bound \( s \) implied by the cone \( C \). This method works for desirability orders \( \preceq \) that are monomials orders.

With these methods in hand, § 5.5 shifts to the following problem: Given a formula \( \phi \), extract an implied cone \( C \) for which the methods from § 5.4 can be applied. Due to the issues of non-linear arithmetic, such an extraction process will necessarily be incomplete. However, in § 5.5 we give a heuristic method for extracting a cone from a non-linear formula that works well in practice (§ 5.7). Moreover, our method allows \( \phi \) and \( t \) to contain function symbols, such as \( \lfloor \cdot \rfloor \) and \( \frac{1}{\cdot} \) (reciprocal), both of which are outside the signature of polynomial arithmetic. Overall, our two-step method is sound with respect to non-linear arithmetic augmented with additional function symbols.

In § 5.6, we introduce the effective-degree order. Effective-degree is essentially a degree monomial orders, extended to the case of non-polynomial function symbols. Effective-degree orders capture the intuitive notion that terms with fewer products are simpler. Also, variable restriction can be encoded as an effective-degree order.

Our saturation and reduction methods combined with effective degree results in a powerful yet practical method for addressing the non-linear OSB problem. In § 5.7, we give experimental results that show our
class Rebase {
    uint base
    uint elastic
    function add(amount) {
        a2base = toBase(amount)
        elastic += amount
        base += a2base
    }
    function toBase(x) {
        x = this.base / this.elastic
    }
}

(a) Extracted Solidity code.

(b) Overview of the method

method, using effective-degree as a term desirability order, produces interesting and relevant bounds using a set of benchmarks extracted from Solidity code by industry experts in smart-contract verification. Our tool is able to produce in seconds or minutes bounds which match or nearly-match human-produced bounds, as well as bounds where ones were previously unknown to human experts.

Contributions.

1. The introduction of the optimal symbolic-bound synthesis problem

2. The local-projection method for projecting polyhedra (§ 5.4.1)

3. Algorithms for reducing a term t by a polyhedron (§ 5.4.2) and a polynomial cone (§ 5.4.3)

4. A saturation method that extracts a polynomial cone from a non-linear formula with additional function symbols (§ 5.5)

5. The introduction of the effective-degree order on terms, which is amenable to automation, and in practice results in useful bounds (§ 5.6)

6. An experimental evaluation demonstrating the power and practicality of our method (§ 5.7)

§ 5.9 discusses related work.

5.2 Overview

To motivate the optimal symbolic-bound synthesis problem, as well as understand how we address it, consider the code in Fig. 5.1a. This code presents us with an interesting non-linear inequational-reasoning problem, which arises from a common smart-contract pattern. A typical “rebase” or “elastic” smart contract holds some amount of “tokens,” which can vary over time, and each user holds a certain amount of “shares”
in the tokens. While the number of tokens may vary (e.g., to control the price), the given number of shares that the user holds should correspond to a largely-unchanging percentage of the total tokens. The utility class `Rebase`, which is based on real-world Solidity code\footnote{https://github.com/sushiswap/BoringSolidity/blob/master/contracts/libraries/BoringRebase.sol}, tracks the total number of tokens in `elastic` and the total number of shares in `base`. The function `add` increases the number of tokens, and the number of available shares accordingly. However, a given amount \( v \) should be represented by the same number of shares even after an `add` operation. Thus, for a given values \( v \) and \( a \), if we execute the sequence
\[
x = \text{toBase}(v) \; ; \; \text{add}(a) \; ; \; y = \text{toBase}(v),
\]
the term \( t = x - y \) should be 0, or close to 0. Plugging in concrete value shows that \( t \) is not identically 0, but how far from 0 is it? The answer can depend on \( v \) and \( a \), as well as the initial values of `elastic`, and `base`, so a precise characterization involves a symbolic expression. Indeed, a verification expert that analyzed this problem came up with the bound \( t \leq 1 + \left\lfloor \frac{v}{e + a} \right\rfloor \), where \( e \) is the initial value of `elastic`. Can we automate this creative process of generating a bound, which for humans often involves much trial-and-error, while even validating a guess for a bound is challenging? In this case, can we automatically find lower and upper symbolic bounds on the term \( t = x-y? \)

The same question can be translated to an OSB problem by writing the following conjunctive formula that represents the assumptions about the initial state, together with the program’s execution:
\[
\phi \triangleq x = \left\lceil \frac{vb}{e} \right\rceil \wedge y = \left\lceil \frac{vb'}{e'} \right\rceil \wedge a2 = \left\lceil \frac{ab}{e} \right\rceil \wedge e' = e + a \wedge b' = b + a2 \wedge a, b, e, v \geq 0.
\]
The goal is to produce a term \( t^* \) such that \( \phi \models x - y \leq t^* \). Furthermore, we are interested in terms that are “insightful” in some sense. For example, we would like to produce a bound that does not contain any temporary variables \( e' \), \( a2 \), or \( b' \), as well as the variables \( x \) and \( y \). This variable restriction does not alone determine a desirable bound, but for this example we at least require a bound to satisfy this constraint. For this example, as well as our experiments, we use the effective-degree order (§ 5.6) as a stand-in for term desirability. Using effective-degree we can encode variable restriction into the order, ensuring that the variables \( e' \), \( a2 \), \( b' \), \( x \), and \( y \) are absent from the bound we produce if such a bound exists. Effective-degree goes further and roughly minimizes the number of products in the result.

Fig. 5.1b gives an outline of our method. The first step to produce an implied cone is to purify the formula \( \phi \) into a formula using only polynomial symbols. We do this by introducing a new variable for each non-polynomial function, and placing the variable assignment in a foreign-function map. That is, for every non-polynomial function symbol \( f(w) \) we introduce a new variable, \( u \), add \( u \mapsto f(w) \) to our map and replace
f(w) with u. Purifying the formula ϕ we obtain,

(\phi') \equiv x = u_4 \land y = u_5 \land a2 = u_6 \land e' = e + a \land b' = b + a2 \land a, b, e, v \geq 0

TM = \{u_1 \mapsto e^{-1}, u_2 \mapsto e'^{-1}, u_3 \mapsto e^{-1}, u_4 \mapsto [vb'u_1], u_5 \mapsto [vb'u_2], u_6 \mapsto [ab'u_3]\}

The purpose of purification is to produce a \( \phi' \) which contains no function symbols. The original formula \( \phi \) is equivalent to \( \phi' \land \bigwedge_{u \mapsto f(w)} u = f(w) \). By making this separation of \( \phi \) in \( \phi' \) and TM we can create methods that can separately work on \( \phi' \), TM, or the combination of \( \phi' \) and TM when required. We then strengthen \( \phi' \) by adding properties of the functions in TM. This is the Instantiate Axioms step in Fig. 5.1b. For example, \( u_4 \) represents a floor term and so satisfies more properties than a generic polynomial variable. Thus, after purification our method uses the term map TM to instantiate axioms for floor and inverse for each occurrence of a floor and inverse term appearing in the map, and adds them to \( \phi' \). That is, we create \( \phi'' \) by adding the instantiated axioms \( (u_1 e = 1)^4, (u_2 e' = 1), ..., vb'u_1 - 1 \leq u_4 \leq vb'u_1, vb'u_3 - 1 \leq u_5 \leq vb'u_3, ..., e \geq 0 \implies u_1 \geq 0, vb'u_1 \geq 0 \implies u_4 \geq 0, \) etc, to \( \phi' \). At this point \( \phi'' \) is

\[ \phi'' \equiv \phi' \land (e \geq 0 \implies u_1 \geq 0) \land \cdots \land (eu_1 = 1) \land \cdots \land (vb'u_1 - u_4 \geq 0) \land \cdots \]

After axioms have been instantiated, \( \phi'' \) and the term map TM are used to construct a cone of polynomials (§ 5.4.3). A cone of polynomials is a composite of a polynomial ideal (§ 2.2) and a polyhedral cone (§ 5.3.1). The ideal and polyhedral cone are each represented by a finite set of basis equations and inequalities, respectively. The ideal consists of its basis equations, as well as all other equations that are polynomially implied by the basis equations. That is, the ideal consists of polynomials \( p_1, \ldots, p_k \), representing assumptions \( p_1 = 0 \), as well as any polynomial of the form \( h_1 p_1 + \cdots + h_k p_k \) for polynomials \( h_1, \ldots, h_k \). The polyhedral cone consists of its basis inequalities as well as all other inequalities that are linearly implied by the basis inequalities. That is, the polyhedron consists of polynomials \( q_1, \ldots, q_r \), representing assumptions \( q_1 \geq 0 \), as well as any other polynomial of the form \( \lambda_1 q_1 + \cdots + \lambda_r q_r \) for scalar \( \lambda_1 \geq 0 \). Overall, the cone consists of terms of the form \( p + q \) where \( p \) is a member of the ideal and \( q \) is a member of the polyhedron. Because \( p \) is an implied equation and \( q \) is an implied inequality, we have \( p + q \geq 0 \).

We call the process of creating a cone of polynomials from \( \phi'' \) and TM saturation. We describe the saturation process in § 5.5 using the running example from this section. At a high level, saturation is

---

3Formally we may consider the axioms for function symbols in the formula \( \phi \) to be given in \( \phi \). However, for convenience, our system automatically instantiates the mentioned \([\_\_]\) and \( \frac{1}{\_} \) axioms.

4On its own \( u_1 e = 1 \) is not a sound axiom, due to division by zero issues. However, in our applications (§ 5.7) it can be assumed this division by zero never happens, thanks to SafeMath libraries in Solidity. A more generally-applicable axiom would be \( e = 0 \lor u_1 e = 1 \) (and explicitly assuming \( e \neq 0 \) in the input), which our system can likewise handle.
an iterative process that extracts equalities and inequalities that are implied by $\phi''$ and TM. A cone of polynomials is created by adding extracted equalities to the ideal part and by adding extracted inequalities to the polyhedral cone part. The methods that we use include congruence closure (§ 5.5.1), linear consequence finding (§ 5.5.2), and “taking products” (§ 5.5.3). By taking products, we mean that from an inequality $w \geq 0$ and $z \geq 0$, we can derive $w^2 \geq 0$, $wz \geq 0$, $z^2 \geq 0$, etc. There are an infinite set of products we could add so our method takes products up to a given saturation depth. In our experiments a saturation depth of 3 worked well. By bounding the set of products we add as well as the use of our consequence finding method makes saturation incomplete for full non-linear arithmetic. However, our experiments show saturation works well in practice (§ 5.7).

As detailed in § 5.5, saturation produces the following cone of polynomials on the running example:

$$C \triangleq (x - u_4, y - u_5, a^2 - u_6, e' - e - a, b' - b - a^2, u_3 - u_1, eu_1 - 1, (e + a)u_2 - 1) + [e, a, v, b, e^2, ea, . . ., vbu_1 - u_4, vu_2u_6, vbv_2 - vbu_1 + vu_2, u_5 - vu_2u_6 - vbv_2 + 1, 1];$$

$(x - u_4, . . .)$ is the ideal and $[e, . . ., 1]$ is the polyhedral cone. In other words, saturation extracted the equations $x - u_4 = 0$, $y - u_5 = 0$, etc. as well as the inequalities $e \geq 0$, $a \geq 0$, and many more.\footnote{For this example, saturation extracted 814 inequalities.}

The next step in addressing the OSB problem is to reduce our term of interest by $C$. That is, we need to find the best $t^*$ such that the cone implies $t \leq t^*$. Equivalently, the problem is to find the best $t^*$ such that the cone contains $t^* - t$. Our reduction procedure for polynomials and polynomial cones works by first reducing the polynomial of interest by the ideal, and then reducing the result by the polyhedron. The process of reducing the polynomial by the ideal is a standard method in computational algebraic geometry (§ 2.2); however, we present a novel polyhedral reduction method (§ 5.4.2), which in turn uses a novel projection method (§ 5.4.1).

The main idea of our polyhedral reduction method is to order the dimensions of the polyhedron, which in our setting correspond to monomials, and successively project out the worst dimension of the polyhedron\footnote{For example, any dimension that corresponds to a monomial that involves the unwanted variables $e'$, $a^2$, or $b'$.} until the term of interest $t$ becomes unbounded. We show (Thm. 5.11) that the bound on $t$ right before $t$ becomes unbounded is optimal in the order of dimensions. In § 5.4.3, we show that the combination of the standard ideal reduction with the polyhedral reduction yields a reduction method for the combined cone.

For the example from Fig. 5.1a, we instantiate an effective-degree order that favors terms without temporary variables. From the saturated cone of polynomials $C$, we have the following equations in the basis of the
ideal, $x - u_4$ and $y - u_5$, as well as the following inequalities in the basis of the polyhedral cone:

$$vbu_1 - u_4 \geq 0 \quad va2u_2 + vbu_2 - vbu_1 + vu_2 \geq 0 \quad u_5 - va2u_2 - vbu_2 + 1 \geq 0$$

Reducing $x - y$ by the equations yields $u_4 - u_5$. The polyhedral reduction method can then be seen as rewriting $u_4 - u_5$ to $vu_2 + 1$ via the justification

$$vu_2 + 1 - (u_4 - u_5) = vbu_1 - u_4 + va2u_2 + vbu_2 - vbu_1 + vu_2 + u_5 - va2u_2 - vbu_2 + 1$$

The right-hand-side is non-negative. Thus, $x - y \leq vu_2 + 1$. Before returning the final result, our system unpurifies this bound by replacing $u_2$ with its definition in TM. Consequently, our system returns the final result as “$x - y \leq \frac{v}{a + e} + 1$.” Our system can also be used to automatically find a lower bound for a term. In our example, the lower bound that it finds is “$x - y \geq -1$.”

These bounds, $-1 \leq x - y \leq \frac{v}{a + e} + 1$, which the implementation of our method found on the order of seconds, are very nearly the bounds, $0 \leq x - y \leq \left\lfloor \frac{v}{a + e} \right\rfloor + 1$, found manually by a human analyst. Differences between the bound we compute automatically and the bound produced by a human sometimes stem from slightly different preferences in the tension between the bound’s simplicity and tightness, but in this case a deeper issue is at play. Our method has a limited capacity to perform inequality reasoning inside a floor term; for instance, we do not produce the inequality $\lfloor t_1 \rfloor \leq \lfloor t_2 \rfloor$ even when $t_1 \leq t_2$ is known, if $t_1$ or $t_2$ are not present in the input formula. We do obtain the slightly weaker $\lfloor t_1 \rfloor \leq t_2$, which, for instance, does not precisely cancel with $-\lfloor t_2 \rfloor$, leading to slightly weaker bounds.

Our initial experience with the system is that it is able to produce interesting upper bounds that are challenging to come up with manually. In one case (fixed point integer arithmetic—see § 5.7), we asked a human analyst to propose a bound for a problem they knew, but had previously attempted only a bound in the other direction (whereas our system computes both at the same time). After approximately fifteen minutes, and correcting the derivation at least once, they came up with a bound that nearly matches the bound that our system generated in less than a second.

### 5.3 Additional Chapter Background

Our method is based on the construction and manipulation of a cone, which as stated consists of a polynomial ideal to hold on to equations and a linear cone to hold on to inequalities. Part of our contribution is the use and manipulation of this composite object. However, we borrow many techniques and ideas from the study of the individual components.
Overall, our method works for any ordered field. That is, our techniques are sound with respect to the theory of ordered fields. We will write $\models_{OF}$ to denote entailment modulo the theory of ordered fields when we want to indicate soundness. Since $\mathbb{R}$ and $\mathbb{Q}$ are ordered fields, $\models_{OF}$ implies entailment with respect to non-linear real and non-linear rational arithmetic.

### 5.3.1 Polyhedral Cones

In this section, we give background on polyhedral cones. Mirroring the process of using ideals to represent equations, we use polyhedral cones to represent inequalities. The reader should keep in mind that our method uses two different kinds of cones. We have an inner cone which is used to hold on to linear inequalities and an outer cone which consists of an ideal and the inner cone. The inner cone is a polyhedral cone and is the main subject on this section. We will describe the outer cone in more detail in § 5.4.3. To make the distinction between the two concepts clear, we will use the terms “polyhedral cone” and “cone of polynomials” to refer to the inner and outer cone, respectively.

**Definition 5.1.** Let $\mathbb{K}$ be an ordered field (e.g., $\mathbb{R}$ or $\mathbb{Q}$) and $V$ be a vector space over $\mathbb{K}$. A polyhedral cone $C$ is the conic combination of finitely many vectors. That is, there is a set of vectors $\{v_1, \ldots, v_n\}$ with $C = \{\lambda_1 v_1 + \cdots + \lambda_n v_n \mid \lambda_i \geq 0\}$. We use $C = [v_1, \ldots, v_n]$ to denote that $C$ is generated by the vectors $\{v_1, \ldots, v_n\}$.

While we use polyhedral cones to represent a set of linear consequences, we frame some of our reduction algorithms (§ 5.4.2) in terms of convex polyhedra. Fortunately, there is a very strong connection between polyhedral cones and convex polyhedra. There are multiple equivalent definitions for a convex polyhedron that lead to different representations. In this chapter we only represent a polyhedron using a set of inequality constraints, sometimes called the constraint representation.

**Definition 5.2.** Let $\mathbb{K}$ be an ordered field. A linear constraint over variables $x_1, \ldots, x_n$ is of the form $a_1 x_1 + \cdots + a_n x_n + b \geq 0$, $a_1 x_1 + \cdots + a_n x_n + b > 0$, or $a_1 x_1 + \cdots + a_n x_n + b = 0$, where $a_1, \ldots, a_n, b \in \mathbb{K}$. A (convex) polyhedron is the set of points of $\mathbb{K}^n$ satisfying a set of linear constraints.

Because each equality can be represented as two inequalities, we could consider polyhedra to not have equality constraints. However, having explicit equalities can allow algorithms to be more efficient in their calculations. We do not take a strong stance on whether all of the equalities of a polyhedron are explicit or not. In § 5.4, we sometimes consider equality constraints as being explicitly part of a polyhedron, but our methods work the same if the equalities are implicit.
If we look at the constraints of the polyhedron as given inequality assumptions, taking conical combinations of the constraints give a sound set of inequality consequences. Moreover, Farkas’ Lemma shows that this set of consequences is also complete.

Lemma 5.3. (Variant of Farkas’ Lemma) Let $P$ be a non-empty polyhedron with non-strict constraints $N = \{c_1 \geq 0, \ldots, c_k \geq 0\}$ and strict constraints $S = \{s_1 > 0, \ldots, s_l > 0\}$. Let $P^*$ denote the polyhedral cone $[c_1, \ldots, c_k, s_1, \ldots, s_l, 1]$. Then $P \models_{\text{LRA}} t \geq 0$ if and only if $t \in P^*$.

We close this section by giving observing that a polyhedral cone that represents inequalities can also represent equalities in the sense that for some vector $v \in C$ it is possible for $-v \in C$ to hold as well. If $C$ is holding onto non-negative vectors, we would have $v \geq 0$ and $-v \geq 0$, so $v = 0$.

Definition 5.4. If the only vector $v$ of $C$ with $-v \in C$ is $0$ then $C$ is called salient.

In the case of polyhedral cones we can determine if a cone is salient by looking at the generators.

Lemma 5.5. If $C = [v_1, \ldots, v_n]$ is not salient then there is an $i \in \{1, \ldots, n\}$ with $v_i \in \{v_1, \ldots, v_n\}$ and $-v_i \in C$.

Proof. Let $C$ be a non-salient cone. Then there exists a vector $f \in C$ and $-f \in C$. Thus $f = \lambda_1 v_1 + \cdots + \lambda_n v_n$ and $-f = \lambda'_1 v_1 + \cdots + \lambda'_n v_n$. There must be some $\lambda'_i$ non-zero. Without loss of generality let $\lambda'_1 > 0$. Then $\lambda'_1 (-v_1) = f + \lambda'_2 v_2 + \cdots + \lambda'_n v_n$. Substituting the equation for $f$ we have $\lambda'_1 (-v_1) = \lambda_1 v_1 + \cdots + \lambda_n v_n + \lambda'_2 v_2 + \cdots + \lambda'_n v_n$. Dividing by $\lambda'_1$ gives $-v_1 = \frac{\lambda_1}{\lambda'_1} v_1 + \frac{\lambda_2 + \lambda'_2}{\lambda'_1} v_2 + \cdots + \frac{\lambda_n + \lambda'_n}{\lambda'_1} v_n$. Thus $-v_1 \in C$. Also $v_1 \in C$. \qed

Remark 5.6. Lem. 5.3 can be modified to also say something about the strict inequality consequences of $P$. However, we would have to add the condition that the “witness” of $t \in P^*$ has at least one of the multiples on the strict inequality constraints non-zero. Consequently the same machinery presented here can be used to hold on to and decide strict inequalities. We just need to make sure that we keep appropriate track of which constraints are strict and which ones are not. AutoBound does distinguish between strict and non-strict inequalities, and so it is possible for the tool to return an bound $t^*$ and know that $t < t^*$ rather than $t \leq t^*$. However, for presentation purposes in the subsequent sections we focus on the non-strict case.

5.4 Reduction

In this section we present our algorithm for efficiently reducing a term w.r.t. a cone of polynomials. We first present its key technical component, the algorithm for local projection (§ 5.4.1), then explain how to use it to perform reduction w.r.t. an (ordinary) polyhedron (§ 5.4.2), and finally extend it to operate w.r.t. the extra ideal to handle the more general case of a cone of polynomials (§ 5.4.3).
5.4.1 Local Projection

An important polyhedral operation is projection. Our reduction method uses a weaker projection operation, which we call local projection. We could use a standard polyhedral-projection operation such as Fourier-Motzkin elimination to yield the same result. However, using full Fourier-Motzkin elimination to remove a single variable from a polyhedron of \( n \) constraints can result in \( O(n^2) \) constraints in the projection. Projecting out \( d \) variables can result in \( O((\frac{n}{d})^2) \) constraints, although many are redundant. The number of necessary constraints grows as a single exponential, at the expense of some additional work detecting redundant constraints at each step. In contrast to this complexity, using local projection to remove a single variable is linear in time and space. Thus, projecting out \( d \) variables takes \( O(dn) \) time and space. The caveat is that local projection only results in a subset of the real projection result, but, as we will show, the real projection result can be finitely covered by local projections. In the worst case the number of partitions for projecting out a single variable is \( O(n^2) \), so local projection does not give a theoretical advantage compared to Fourier-Motzkin.

However, in our case, we often do not need to compute the full projection result. Instead, we only require parts of it, and so using local projection gives us a lazy method for computing objects of interest. Local projection can also be understood as a method of model-based projection [92, Section 5], specialized to the setting of polyhedra. Komuravelli et al. [92] give a model-based projection for LRA based on the quantifier-elimination technique of Loos and Weispfenning [98]. Thus, our specialization is very similar to these prior methods.

**Definition 5.7.** Let \( P \) be a polyhedron with dimension \( d_i \). \( \text{Proj}(P, d_i) = \{ m \models_{\text{LRA}} \exists d_i. P \} \).

Let \( P \) be a polyhedron represented by a conjunction of equality and inequality constraints, and let \( x \) be a dimension of \( P \). The constraints of \( P \) can be divided and rewritten as follows:

- Let \( E_x \equiv \{ x = -\frac{f_i}{e_i} \mid e_i x + f_i = 0 \in P \} \), where each \( e_i \) is a constant and each \( f_i \) is \( x \)-free.

- Let \( L_x \equiv \{ x \geq -\frac{b_i}{a_i} \mid a_i x + b_i \geq 0 \in P \} \) and \( L_x^r \equiv \{ x > -\frac{b_i}{a_i} \mid a_i x + b_i' \geq 0 \in P \} \), where each \( a_i, a_i' \) is a constant greater than 0, and each \( b_i, b_i' \) is \( x \)-free.

- Let \( U_x \equiv \{ x \leq -\frac{d_i}{c_i} \mid c_i x + d_i = 0 \in P \} \) and \( U_x^r \equiv \{ x < -\frac{d_i}{c_i} \mid c_i x + d_i' > 0 \in P \} \), where each \( c_i, c_i' \) is a constant less than 0, and each \( d_i, d_i' \) is \( x \)-free.

- Other constraints, \( C \), not involving \( x \).

**Local Projection.** Let \( x \) be some dimension of a polyhedron \( P \) that is represented by equality constraints \( E_x \), lower-bound constraints \( L_x \) and \( L_x^r \), upper-bound constraints \( U_x \) and \( U_x^r \), and other constraints \( C \). Let \( m \) be a model of \( P \). The local projection of \( x \) from \( P \) w.r.t. \( m \), denoted by \( \text{LProj}(m, P, x) \), is a polyhedron defined by a set of constraints as follows:
• If $E_x$ is not empty, then let $x = e \in E_x$. Then $\text{LProj}(m, P, x)$ is

$$E_x[x \mapsto e] \cup L_x[x \mapsto e] \cup L^x_x[x \mapsto e] \cup U_x[x \mapsto e] \cup U^x_x[x \mapsto e] \cup C.$$ 

• If $E_x$, $L_x$, and $L^x_x$ are empty, then $\text{LProj}(m, P, x) \equiv C$

• If $E_x$ is empty, but either $L_x$ or $L^x_x$ is non-empty, then let

$$L'_x = \{ lb \mid x \geq lb \in L_x \} \cup \{ lb' \mid x > lb' \in L^x_x \}.$$ Let $lb^* \in \arg \max_{L'_x} \text{eval}(lb, m)$.

- If $lb^*$ corresponds to a non-strict constraint, then $\text{LProj}(m, P, x)$ is

$$\{ lb \leq lb^* \mid lb \in L'_x \} \cup \{ lb^* \leq ub \mid x \leq ub \in U_x \} \cup \{ lb^* < ub' \mid x < ub' \in U^x_x \} \cup C$$

- If $lb^*$ corresponds to a strict constraint, then $\text{LProj}(m, P, x)$ is

$$\{ lb \leq lb^* \mid lb \in L'_x \} \cup \{ lb^* < ub \mid x \leq ub \in U_x \} \cup \{ lb^* < ub' \mid x < ub' \in U^x_x \} \cup C$$

The idea of a local projection is identical to a full projection except in local projection we only consider the lower bound $lb^*$ that is binding with respect to the given model. In general there are other models with different lower bounds, so a full projection needs to consider these alternative cases. However, because there are only finitely many possible binding lower bounds, local projection finitely covers the full project. These ideas are formally captured by Lem. 5.8. For a more detailed comparison of local projection versus full projection see the proof of the lemma.

**Lemma 5.8.** Let $P$, be a polyhedron. For a model $m \models P$ and a dimension $x$, the following are true:

1. $m \models \text{LProj}(m, P, x)$

2. $\text{LProj}(m, P, x) \models \text{Proj}(P, x)$

3. $\{ \text{LProj}(m, P, x) \mid m \models P \}$ is a finite set

**Proof.** We begin by first giving a logical account of $\text{Proj}(P, x)$. That is, we consider different cases and construct a quantifier free formula $\psi$ and show it’s equivalent to $\text{Proj}(P, x)$. We then compare local projection with the result to show the lemma.

Let $E_x$, $L_x$, $L^x_x$, $U_x$, $U^x_x$, and $C$ be the constraints of $P$. $\text{Proj}(P, x)$ is $\exists x. \phi$ where $\phi$ is a conjunction of all of the $E_x$, $L_x$, $L^x_x$, $U_x$, $U^x_x$, and $C$ constraints. To compute a quantifier-free formula equivalent to $\exists x. \phi$ we
consider a few cases. In each case, to see that the resulting quantifier-free formula, say \( \psi \), is equivalent to \( \exists x. \phi \), it suffices to construct a value for \( x \) from any model of \( \psi \) such that \( \phi \) is satisfied.

**E\( _{x} \) is not empty.** To create an equivalent quantifier-free formula, we pick and rewrite some equality from \( E_{x} \), say \( x = e \). Then we substitute \( e \) in for \( x \) in all the constraints in \( E_{x} \), \( L_{x} \), and \( U_{x} \). That is

\[
\psi \overset{\text{def}}{=} \bigwedge E_{x}[x \mapsto e] \cup L_{x}[x \mapsto e] \cup L_{x}^{*}[x \mapsto e] \cup U_{x}[x \mapsto e] \cup U_{x}^{*}[x \mapsto e] \cup C.
\]

To construct a satisfying model of \( \phi \) from a model \( m \) of \( \psi \), we set \( x = eval(e, m) \), where \( eval(e, m) \) is evaluated at \( m \).

**E\( _{x} \) is empty and \( L_{x} \cup L_{x}^{*} \) is empty.** In this case, \( x \) has no lower bound. That is, \( x \) can get arbitrarily small. In this case, \( \psi \overset{\text{def}}{=} \bigwedge C \). To construct a satisfying model of \( \phi \) from a model \( m \) of \( \psi \), we find the smallest upper-bound constraint, and ensure that is satisfied. For example, we set \( x = -1 + \min(\min_{x \leq ub \in U_{x}} eval(ub, m), \min_{x < ub' \in U_{x}^{*}} eval(ub', m)) \).

**E\( _{x} \) is empty and \( L_{x} \cup U_{x} \) is not empty.** This is the interesting case. Let \( L_{x}' = \{lb | x \geq lb \in L_{x}\} \cup \{lb' | lb' > 0 \in L_{x}^{*}\} \). Consider the set of models where \( lb^{*} \) is the binding lower-bound. That is, \( m \) is a model with \( lb^{*} \in \arg \max_{L_{x}'} eval(lb, m) \). The set of models for which \( lb^{*} \) is the binding lower-bound are models that satisfy \( eval(lb^{*}, m) \geq eval(lb, m) \) for each \( lb \in L_{x}' \). Furthermore, if \( x \geq lb^{*} \) (or \( x > lb^{*} \), if \( lb^{*} \) corresponds an element of \( L_{x}^{*} \)), then all the constraints on \( x \) are satisfied. So far, all we have done is restrict the scope to the set of models where \( lb^{*} \) is the binding lower-bound. A formula that represents this case is

\[
\phi_{lb^{*}} \overset{\text{def}}{=} \left( \bigwedge_{lb \in L_{x}^{'}} lb \leq lb^{*} \right) \land \left( \bigwedge_{x \leq ub \in U_{x}} lb^{*} \leq x \leq ub \right) \land \left( \bigwedge_{x < ub' \in U_{x}^{*}} lb^{*} \leq x < ub' \right) \land (\bigwedge C).
\]

(The lower-bounds on \( x \) in the above formula would be strict if \( lb^{*} \) corresponds to a strict bound.) Restricting to the case of \( lb^{*} \) being the binding lower bound, \( \phi_{lb^{*}} \) is equivalent to \( \phi \). Finally, we can simply drop \( x \) from the above formula, but keep transitively implied constraints. That is, let

\[
\psi_{lb^{*}} \overset{\text{def}}{=} \left( \bigwedge_{lb \in L_{x}^{'}} lb \leq lb^{*} \right) \land \left( \bigwedge_{x \leq ub \in U_{x}} lb^{*} \leq x \leq ub \right) \land \left( \bigwedge_{x < ub' \in U_{x}^{*}} lb^{*} < ub' \right) \land (\bigwedge C).
\]

(Once again the middle inequality may be strict, depending on \( \frac{lb^{*}}{a} \).) \( \psi_{lb^{*}} \) is equivalent to \( \exists x. \phi_{lb^{*}} \). To get a value for \( x \) from a model \( m' \) of \( \psi_{lb^{*}} \), we set

\[
x = \frac{1}{2}(eval(lb^{*}, m') + \min(\min_{x \leq ub \in U_{x}} eval(ub, m'), \min_{x < ub' \in U_{x}^{*}} eval(ub', m'))).
\]
That is, $x$ is the mid-point of the most binding lower-bound and the most binding upper-bound. Thus, $x$ will satisfy all lower-bounds and all upper-bounds in $\phi_{lb}$. All of this reasoning was under the assumption that a particular lower-bound $lb^*$ was binding. However, assuming no lower-bound is redundant means that any $lb \in L'_x$ could be a binding lower-bound. Therefore, to construct a quantifier-free formula $\text{Proj}(P, x)$ we simply take a disjunction among all possible lower bounds.

$$\exists x. \phi \equiv \bigvee_{lb \in L'_x} \psi_{lb}.$$  

From this presentation, it is clear how local projection compares with full projection. In the cases when $P$ has an equality constraint involving $x$, or when $x$ has no lower-bounds in $P$, $\text{LProj}(m, P, x)$ and $\text{Proj}(P, x)$ are equivalent. Otherwise, $\text{LProj}(m, P, x)$ exactly matches one disjunct of $\text{Proj}(P, x)$ as presented above. Thus, $\text{LProj}(m, P, x) \models \text{Proj}(P, x)$. The number of disjuncts of $\text{Proj}(P, x)$ is finite, so $\{\text{LProj}(m, P, x) \mid m \models P\}$. Furthermore, every disjunct of $\text{Proj}(P, x)$ is satisfied by some $m$. 

Fig. 5.2 gives a geometric picture of local projection and projection. Consider Fig. 5.2a, where the goal is to project out the $z$ dimension. Take the red region for example. Any model in the red region has the lower-front facing triangle as a binding constraint; therefore, local projecting to the $x$-$y$ plane with respect to such a model yields the red-triangle. The union of the red, gray, olive, and blue regions give the full projection. Fig. 5.2b is a similar diagram, but for projecting out $y$ then $z$. Fig. 5.2c shows the result of projecting out $z$ then $y$. The result is a line segment in the $x$ dimension. In Figs. 5.2b and 5.2c, the resulting projections are depicted as being slightly displaced from the $x$-axis for clarity.

Local projection can also be used to project out multiple dimensions by projecting out each dimension sequentially.
Definition 5.9. Given a list of dimensions, we use $\text{LProj}(m, P, [d_1, \ldots, d_k])$ to denote $\text{LProj}(m, \text{LProj}(m, \ldots, \text{LProj}(m, P, d_1), \ldots, d_{k-1}), d_k)$. Similarly, for $\text{Proj}(P, [d_1, \ldots, d_k])$.

Crucially, locally projecting out a set of variables has the same relationship to the full projection of the same variables as we have in Lem. 5.8 for locally projecting out a single variable.

Theorem 5.10. Let $P$ be a polyhedron. For a $m \models P$ and a list of dimensions $[d_1, \ldots, d_k]$, the following are true

1. $m \models \text{LProj}(m, P, [d_1, \ldots, d_k])$

2. $\text{LProj}(m, P, [d_1, \ldots, d_k]) \models \text{Proj}(P, [d_1, \ldots, d_k])$

3. $\{\text{LProj}(m, P, [d_1, \ldots, d_k]) \mid m \models P\}$ is a finite set

Proof. These properties can be shown by induction on the length of the list of dimensions. The base case is covered by Lem. 5.8. For the inductive step, suppose that the theorem holds for $\text{LProj}(m, P, [d_1, \ldots, d_{k-1}])$. By Lem. 5.8, $\text{LProj}(m, P, [d_1, \ldots, d_{k-1}, d_k])$, entails $\text{LProj}(m, P, [d_1, \ldots, d_{k-1}])$, and finitely covers $\text{LProj}(m, P, [d_1, \ldots, d_{k-1}])$. Thus, by the inductive hypothesis, the theorem holds.

It is well known that when performing a full projection the order in which the dimensions are presented does not matter; however, in the case of local projection, the order does matter. To see why, compare Figs. 5.2b and 5.2c. In Fig. 5.2b the red, green, gray, and blue line segments are the possible results from projecting out $y$ then $z$. The line segment we actually get depends on the model, $m$, local projection is instantiated with. However, in Fig. 5.2c, either of the olive line segments (the ones furthest from the axis) are possible results from projecting out $z$ then $y$, once again depending on the model local projection is instantiated with. There is no corresponding segment in Fig. 5.2b to either of the olive line segments. That is, no matter the model used, there is no way to yield either of the olive segments in Fig. 5.2c, by projecting out $y$ then $z$.

However, Thm. 5.10 ensures that the set of possible local projections is finite, and that they exactly cover the full projection.

5.4.2 Polyhedral Reduction

In this section we present our algorithm for optimally reducing a polyhedron $P$ with respect to a linear term $t$ and an order on dimensions $\ll$. Alg. 12 will produce a bound $t^*$ over dimensions $d_j, \ldots, d_k$ such that $P \models_{\text{LRA}} t \leq t^*$ and $t^*$ is optimal with respect to $\ll$. That is, for any $b$ with $\models_{\text{LRA}} t \leq b$, then $b$ is an expression over the dimensions $d_j, \ldots, d_k$ with $d_n \ll d_i$. Another way to think about the optimality is the “leading dimension” of $t^*$ is minimal.
Algorithm 11: Conjecture

Input: Polyhedron $P$, model $m$ of $P$, dimensions $d_1, \ldots, d_k$ and $T$
Result: Conjectured upper bounds on $T$
1 $P' \leftarrow P$; $P'' \leftarrow \text{LProj}(m, P', d_1)$; $i \leftarrow 2$;
2 while $T$ has an upper bound in $P''$ do
3 $P' \leftarrow P''$;
4 $P'' \leftarrow \text{LProj}(m, P'', d_i)$;
5 $i \leftarrow i + 1$;
6 $B \leftarrow \{\}$;
7 for $c \geq 0 \in P'$ do
8 \quad if Coefficient of $T$ in $c$ is negative then
9 \quad \quad $B \leftarrow B \cup \{c \geq 0\}$;
10 return $B$;

Algorithm 12: Polyhedral reduction

Input: Polyhedron $P$, term $t$ over variables $d'_1, \ldots, d'_k$, order $\ll$
Result: $r$ with $P \models t \leq r$
1 $T \leftarrow$ new dimension;
2 $P \leftarrow P \cup \{T = t\}$;
3 $d_1, \ldots, d_k \leftarrow \text{sort}((d'_1, \ldots, d'_k), \gg)$;
4 $U \leftarrow \{\}$;
5 while $m \models P \land \bigwedge_{u \in U} \neg u$ do
6 \quad $U \leftarrow U \cup \text{conjecture}(P, m, [d_1, \ldots, d_k], T)$;
7 $r \leftarrow \text{filter true bounds from } U$;
8 return $r$;

Fig. 5.3 gives a geometric representation of Alg. 12. Suppose that we wish to upper-bound some term $t$ that is an expression over $x$ and $y$, under the assumption that $x$ and $y$ are restricted to the (unbounded) dark-blue region. Let $T = t$. Then the floating orange plane is the term to optimize. Suppose that we favor an upper-bound containing $x$ over an upper-bound containing $y$. In Fig. 5.3, the optimal upper-bound corresponds to the constraint $u_{1,x}$.

The algorithm lazily explores the polyhedron by getting a model of the floating orange polyhedron. Suppose that the first model sampled by Alg. 12, say $m_1$, has an assignment for $T$ smaller than the constant $u_{2,c}$ shown in Fig. 5.3. Alg. 12 calls Alg. 11 with $m_1$. Alg. 11 explores the local projection of the floating orange plane with respect to the model $m_1$. Note that on initial call to Alg. 11, $T$ always has an upper bound, namely $t$. Thus Alg. 11 will successively project out dimensions until $T$ is unbounded. In the case of Fig. 5.3, the
dimension $y$ is locally projected out first, yielding the orange region in the $x$-$T$ plane. This region does have an upper-bound, $u_{1,x}$, which can simply be read off the constraint representation of the orange region. However, at this point it is unknown if there is another bound in fewer dimensions. So, Alg. 11 continues and locally projects out the $x$ dimension obtaining the interval $l_{1,c} \leq T \leq u_{2,c}$. There are no more dimensions to project, so Alg. 11 returns the conjectured upper-bound of $u_{2,c}$. Note that this is not a true bound. That is, there is a model of the floating orange plane that is strictly larger than $u_{2,c}$.

This situation means that the while loop in Alg. 12 will execute again with a new model $m_2$ that is still within the polyhedron, but is strictly larger than $u_{2,c}$. Thus, it will pass off this new model to Alg. 11, to get new conjectured bounds. Alg. 11 will, again, project out the dimensions $y$ then $x$ to obtain the interval $u_{2,c} \leq T$. However, in this case projecting out $y$ then $x$ gives an interval that is unbounded above. Thus, Alg. 11 will go back one step, when $T$ still had an upper-bound, namely $T \leq u_{1,x}$. Note that the upper-bound $u_{1,x}$ is an upper-bound containing the variable $x$. Because $u_{1,x}$ is the most optimal upper-bound for $T$ using the model $m_2$, Alg. 11 will return $u_{1,x}$. In this case $u_{1,x}$ is a true upper-bound. There is no value of $T$ that is strictly greater than $u_{1,x}$. For this reason, the loop in Alg. 12 will terminate with $U = \{u_{2,c}, u_{1,x}\}$. For the loop to terminate, one of these bounds must be true. So, the algorithm finishes by filtering $U$ by which ones are true bounds, that is which $ub \in U$ have $P \models_{\text{LRA}} T \leq ub$. This check can easily be accomplished by an SMT solver. In short, such a bound must exists in $U$ because any “upper-bound” face of the full projection polyhedron with minimal dimensions $d_i, \ldots, d_k$ is a true upper bound on $T$. Moreover, Alg. 11 only returns “upper-bound” faces of local projections that are in $d_i, \ldots, d_k$ dimensions or fewer. If the upper-bound Alg. 11 returns is in fewer dimensions then it’s not a true bound, and another model will be sampled in Alg. 12. Since local projection finitely covers the full projection eventually a model will be sampled to produce a face of the full projection in $d_i, \ldots, d_k$. For a more detailed explanation, see the proof of Thm. 5.11.

For this particular choice of $m_1$ and $m_2$, Alg. 12 took two rounds to find a true upper-bound. However, if instead $m_2$ was selected as the first model, then the true bound would have been found and returned in only one round. For this reason, the performance of Alg. 12 is heavily dependent on the models returned by the SMT solver.

**Theorem 5.11.** Let $t^*$ be a term produced by Alg. 12 for inputs $P$, $t$, and $\ll$. Let $d_1, \ldots, d_k$ be the dimensions of $P$ and $t$ sorted greatest-to-smallest with respect to $\ll$. Let $d_j, \ldots, d_k$ be the dimensions used in $t^*$. Then the following are true of $t^*$:

1. $P \models_{\text{LRA}} t \leq t^*$

2. $t^*$ is optimal in the sense that for any other $b$ with $P \models_{\text{LRA}} t \leq b$, then $b$ is an expression over the dimensions $d_i, \ldots, d_k$ with $d_j \ll d_i$.
**Proof.** To show the theorem, we first show that the full project can be used to solve the problem. Then we show how Algs. 11 and 12 construct the relevant constraints of the full project in a terminating procedure. Let \( d_1, \ldots, d_k \) be the dimensions of \( P \) and \( t \) sorted greatest to smallest with respect to \(<\). Let \( P' \) be \( P \) with the added constraint that \( T = t \). Now let \( \text{Proj}(P', [d_1, \ldots, d_{j-1}]) \) be the most minimal polyhedron where \( T \) has an upper bound. That is, \( \text{Proj}(P', [d_1, \ldots, d_{j-1}]) \) is a polyhedron where there is a constraint of the form \( aT + b \geq 0, a < 0, b \) is \( T \) free, and \( \text{Proj}(P', [d_1, \ldots, d_j]) \) has no such upper-bound on \( T \). In such a case, \( t^* = -\frac{b}{a} \) is an upper-bound that satisfies the conditions of the theorem. To show this we need to show that \( t^* \) is indeed an upper-bound, and that it is optimal.

Note that from \( \text{Proj}(P', [d_1, \ldots, d_{j-1}]) \) we have \( \text{Proj}(P', [d_1, \ldots, d_{j-1}]) \models LRA T \leq -\frac{b}{a} \). By properties of projection we have \( \text{Proj}(P', [d_1, \ldots, d_{j-1}]) \models \exists d_1, \ldots, d_{j-1}. P' \). Thus, we have \( P' \models LRA \exists d_1, \ldots, d_{j-1}. P' \equiv \text{Proj}(P', [d_1, \ldots, d_{j-1}]) \). Furthermore, because \( T \) was a fresh variable \( P \equiv \exists T. (P \land T = t) \). Thus, we have the following chain of entailment:

\[
P \equiv \exists T. ([P \land T = t] \land (T = t)) \models LRA \exists T. ([\exists d_1, \ldots, d_{j-1}. P'] \land (T = t)) \equiv \\
\exists T. [\text{Proj}(P', [d_1, \ldots, d_{j-1}]) \land (T = t)] \models LRA \exists T. [T \leq -\frac{b}{a} \land (T = t)] \equiv t \leq -\frac{b}{a}
\]

To show optimality, suppose there was a better bound \( b \). In other words there is a \( b \) with \( \exists T. ([\exists d_1, \ldots, d_i. P] \land (T = t)] \models LRA \exists T. [T \leq b \land T = t] \) with \( j < i \). That is, \( b \) is in a lower dimensional space than \( t^* \). From the previous entailment and the property of projection we have \( \exists T. ([\exists d_{j+1}, \ldots, d_i. \text{Proj}(P, [d_1, \ldots, d_j])]) \land (T = t)] \models LRA \exists T. [T \leq b \land T = t] \). Further we can drop the initial existential quantifiers, and say if the previous entailment holds than so does \( [\text{Proj}(P, [d_1, \ldots, d_j])]) \land (T = t)] \models LRA T \leq b \land T = t \). However this is contradicts the assumption \( \text{Proj}(P, [d_1, \ldots, d_j]) \) has \( T \) unbounded above. We can get a counterexample by making \( T \) arbitrarily big. This shows that \( t^* \) is optimal.

Now we must show that Algs. 11 and 12 recovers the appropriate bounds from the full projection in a finite time. First, observe that the conjectured bounds returned by Alg. 11 cannot be higher-dimensional than \( d_1, \ldots, d_k \). That is, in the dimension ordering \(<\), the conjectured bounds returned by Alg. 11 are at least as good as the true optimal bound \( t^* \). For this not to be the case, we would have to have \( T \) bounded in \( L\text{Proj}([m, P', [d_1, \ldots, d_{j-1}]) \), but not in \( L\text{Proj}([m, P', [d_1, \ldots, d_i]) \) for some \( d_{i+1} \gg d_j \). However, this would contradict Thm. 5.10, i.e. we would have \( L\text{Proj}([m, P', [d_1, \ldots, d_i]) \not\models LRA \text{Proj}(P', [d_1, \ldots, d_i]) \).

So, in the \(<\) order, conjectured bounds are no worse than true bounds. However, conjectured bounds can be better, in the \(<\) order, than true bounds. In such cases the loop in Alg. 12 will not terminate. That is, consider the situation where the set of conjectured bounds in line 6, \( U \), does not contain a true bound. In such case, at least one of the bounds in \( U \) can be violated, and so there is a model \( m \) such that \( m \models P \land \bigwedge_{u \in U} \neg u \). In
such a case the algorithm loops and produces new conjectured bounds. This shows that the while loop in Alg. 12 can only terminate when a true optimal bound is conjectured.

Finally we must argue Alg. 12 always terminates. This amounts to showing that Alg. 12 will always produce a model such that Alg. 11 will return a true bound. From Thm. 5.10 we have \( \text{LProj}(m, P', [d_1, \ldots, d_{j-1}]) \mid m \models P' \) finitely covers \( \text{Proj}(P', [d_1, \ldots, d_{j-1}]) \). Thus, there is some set of models such that \( \text{LProj}(m, P', [d_1, \ldots, d_{j-1}]) \) has a constraint which is a true bound and \( \text{LProj}(m, P', [d_1, \ldots, d_j]) \) has \( T \) unbounded. Furthermore, the loop condition in Alg. 12 ensures we continue to pick models that do not lead to the same local project. Thus, eventually some model which leads to a true bound must be chosen.

\[ \square \]

Thm. 5.11 says Alg. 12 solves the OSB problem for polyhedra and “dimensional orders”. Furthermore, due to Farkas’ lemma, an optimal bound can be found with respect to polyhedral cones.

**Corollary 5.12.** Let \( Q \) be a polyhedral cone over \( \mathbb{K}^k \) (\( \mathbb{Q}^k \) or \( \mathbb{R}^k \)). Let \( \ll \) be an order on the dimensions of \( \mathbb{K}^k \) and \( t \) a vector in \( \mathbb{K}^k \). Alg. 12 can be used to solve the problem of finding a \( t^* \) with the following properties:

1. \( t^* - t \in Q \)
2. \( t^* \) is optimal in the sense that for any other \( b \) with \( b - t \in Q \), then \( b \) is an expression over the dimensions \( d_1, \ldots, d_k \) with \( d_j \ll d_i \).

**Proof.** From \( Q = [q_1, \ldots, q_r, 1] \) we can construct an appropriate polyhedron \( P \) where \( P^* = Q \). That is, make a polyhedron \( P = [q_1 \geq 0, \ldots, q_r \geq 0] \). Then, by combining Thm. 5.11 with Lem. 5.3 we obtain the desired result.

\[ \square \]

**LP Reduction**

Our main approach for reducing with respect to a polyhedral cone is Alg. 12. However, an alternative method based on linear programming is also possible. The idea is based on the observation that given a polyhedral cone \( Q = [q_1, \ldots, q_r, 1] \) and terms \( t \) and \( t' \) over dimensions \( d_1, \ldots, d_n \) and constant dimension \( d_{n+1} \), it is possible to check whether \( r - t \in Q \) with an LP query. That is, let \( t^e_j, t'^e_j, \) and \( q^e_{ij} \) denote the coefficient of \( t, t', \) and \( q_1 \) on the \( j \)th dimension. Then \( t' - t \in Q \) if and only if there are non-negative \( \lambda_1, \ldots, \lambda_{r+1} \) with \( t'^e_j - t^e_j = \lambda_1 q^e_{ij} + \cdots + \lambda_r q^e_{ij} \) for each \( 1 \leq j \leq n \) and \( t'^e_{n+1} - t^e_{n+1} = \lambda_1 q^e_{n+1} + \cdots + \lambda_r q^e_{r,n+1} + \lambda_{r+1} \). This system can be used to decide if a given concrete \( t' \) has the property \( t' - t \in Q \); moreover, the system can also represent the space of \( t' \) with this property by leaving each \( t'^e_j \) undetermined, i.e., by considering the linear system over the variables \( t'^e_1, \ldots, t'^e_{n+1}, \lambda_1, \ldots, \lambda_{r+1} \). Therefore, it is possible to reduce the OSB problem over a polyhedral cone (and consequently a polyhedron) to linear programming with a lexicographic objective function [75]. Without loss of generality assume the dimensions \( d_1, \ldots, d_n \) are ordered in terms of preference.
An optimal $t^*$ can be found by asking whether $t^* c_j - t^*_j = \lambda_1 q^c_{1,j} + \cdots + \lambda_r q^c_{r,j}$ for each $1 \leq j \leq n$ and $t^* c_{n+1} - t^*_{n+1} = \lambda_1 q^c_{1,n+1} + \cdots + \lambda_r q^c_{r,n+1} + \lambda_{r+1}$ has a solution with $t^* c_1 = 0, \ldots, t^* c_n = 0$. If not, then we see if there is a solution with $t^* c_1 = 0, \ldots, t^* c_{n-1} = 0$, and then $t^* c_1 = 0, \ldots, t^* c_{n-2} = 0$, etc. until a solution is found. We found in practice Alg. 12 was much faster (see § 5.7).

### 5.4.3 Cone of Polynomials

In this section, we extend the results from the previous section to the case of a cone of polynomials.

**Definition 5.13.** Let $p_1, \ldots, p_k$, $q_1, \ldots, q_r$ be polynomials. The cone of polynomials $C$ generated by $p_1, \ldots, p_k$ and $q_1, \ldots, q_r$ is the set

$$C = \langle p_1, \ldots, p_k \rangle + [q_1, \ldots, q_r, 1] = \{ p + q \mid p \in \langle p_1, \ldots, p_k \rangle, q \in [q_1, \ldots, q_r, 1] \}. $$

At first glance there seems to be a slight mismatch in the definition. We defined polynomial ideals as consisting of polynomials, whereas polyhedral cones are defined as a collection of vectors. However, there is no issue because polynomials can be viewed as vectors in the infinite-dimensional vector space that has monomials as basis vectors. That is, we can add the polynomials from the ideal with the vectors from the polyhedral cone by adding coefficients via common monomials.

A cone of polynomials generated by $p_1, \ldots, p_k$ and $q_1, \ldots, q_r$ gives a sound set of consequences for assumptions $p_1(x) = 0, \ldots, p_k(x) = 0$ and $q_1(x) \geq 0, \ldots, q_r(x) \geq 0$.

**Lemma 5.14.** *(Soundness)* Let $g \in C = \langle p_1, \ldots, p_k \rangle + [q_1, \ldots, q_r, 1]$. Then,

$$\bigwedge_{i=1}^k p_i(x) = 0 \land \bigwedge_{j=1}^r q_j(x) \geq 0 \models \text{OF} \ g(x) \geq 0.$$  

**Proof.** Because $g \in C$, there exists polynomials $h_1, \ldots, h_k$ and non-negative scalars $\lambda_1, \ldots, \lambda_{r+1}$ such that

$$g = h_1 p_1 + \cdots + h_k p_k + \lambda_1 q_1 + \cdots + \lambda_r q_r + \lambda_{r+1}.$$  

Given the one-to-one correspondence between polynomial expressions and polynomial functions in an
ordered field we have

\[ g(x) = h_1(x)p_1(x) + \cdots + h_k(x)p_k(x) + \lambda_1 q_1(x) + \cdots + \lambda_r q_r(x) + \lambda_{r+1} \]

\[ = h_1(x)0 + \cdots + h_k(x)0 + \lambda_1 q_1(x) + \cdots + \lambda_r q_r(x) + \lambda_{r+1} \]

\[ \geq \lambda_1 0 + \cdots + \lambda_r 0 + \lambda_{r+1} \]

\[ \geq 0 \]

Lem. 5.14 is the main reason for our interest in cones of polynomials. Furthermore, we will show that we can perform reduction on a cone of polynomials. However, we take this moment to discuss the power of this object and the issue of completeness. In the linear case, by Farkas’ lemma, polyhedral cones are complete with respect to a conjunction of linear inequalities; however, there is no such analogue for the case of a cone of polynomials\(^7\). That is, cones of polynomials are incomplete for non-linear arithmetic.

**Example 5.15.** From an empty context, we have \( \models_{OF} (x^n)^2 \geq 0 \) for any \( n \in \mathbb{N} \). If cones of polynomials were complete we would need to have \( (x^n)^2 \) in the “empty cone” \( \langle 0 \rangle + [1] \).

On the other hand because of the inclusion of the ideal, a cone of polynomials does hold onto some non-linear consequences.

**Example 5.16.** \( x^2 - 2x + 2 \in \langle x - 1 - t, y - 1 - t^2 \rangle + [y, 1] \)

\[ x^2 - 2x + 2 = (x - 1 + t)(x - 1 - t) + (-1)(y - 1 - t^2) + (1)y + (0)(1). \]

Thus, a cone of polynomials can establish the consequence \( x^2 - 2x + 2 \geq 0 \) from the assumptions \( x - 1 - t = 0 \), \( y - 1 - t^2 = 0 \), \( y \geq 0 \).

The use of cones of polynomials balances expressiveness and computational feasibility.

Before we give the method for reducing a polynomial with respect to a cone of polynomials, we need to introduce the idea of a *reduced* cone of polynomials. The only difference is that in the case of reduced cone we require the \( q_1, \ldots, q_r \) polynomials to be reduced with respect to a Gröbner basis for the ideal \( \langle p_1, \ldots, p_k \rangle \).

\(^7\)In the case of real arithmetic, positivstellensatz theorems are the analogue of Farkas’ lemma for a different kind of cone object. However, they exhibit computational difficulties. See § 5.9 for more discussion.
**Definition 5.17.** Let $C = \langle p_1, \ldots, p_k \rangle + [q_1, \ldots, q_r, 1]$ be a cone of polynomials and $\ll$ a monomial order. $C$ is reduced with respect to $\ll$ if $p_1, \ldots, p_k$ is a Gröbner basis for $\langle p_1, \ldots, p_k \rangle$ and for every $q_i$ we have that no monomial of $q_i$ is divisible by any of $\text{LM}(p_1), \ldots, \text{LM}(p_k)$.

**Theorem 5.18.** Let $C = \langle p_1, \ldots, p_k \rangle + [q_1, \ldots, q_r, 1]$ be a cone of polynomials. Let $B = \{g_1, \ldots, g_s\}$ be a Gröbner basis for $\langle p_1, \ldots, p_k \rangle$ and let $t_i = \text{red}_B(q_i)$ for each $q_i$. Then $C' = \langle g_1, \ldots, g_s \rangle + [t_1, \ldots, t_r, 1]$ is a reduced cone with $C = C'$.

**Proof.** From Thm. 2.49 we have that $C'$ is reduced, so all we need to do is show $C = C'$. First, because $g_1, \ldots, g_s$ is a Gröbner basis we have $\langle g_1, \ldots, g_s \rangle = \langle p_1, \ldots, p_k \rangle$. So we need to show $C = \langle g_1, \ldots, g_s \rangle + [q_1, \ldots, q_r, 1]$ is equal to $C' = \langle g_1, \ldots, g_s \rangle + [t_1, \ldots, t_r, 1]$. Let $g \in C$. Then

$$g = h_1 g_1 + \cdots + h_s g_s + \lambda_1 q_1 + \cdots + \lambda_r q_r + \lambda_{r+1}$$

for polynomials $h_1, \ldots, h_s$ and non-negative scalars $\lambda_1, \ldots, \lambda_{r+1}$. By construction of each $t_i$ we have $q_i = f_i + t_i$ for $f_i \in \langle g_1, \ldots, g_s \rangle$.

$$g = h_1 g_1 + \cdots + h_s g_s + \lambda_1(h_1^1 g_1 + \cdots + h_s^1 g_s + t_1) + \cdots + \lambda_r(h_1^r g_1 + \cdots + h_s^r g_s + t_r) + \lambda_{r+1}$$

The above can be reorganized as

$$g = (h_1 + \lambda_1 h_1^1 + \cdots + \lambda_r h_s^r) g_1 + \cdots + (h_s + \lambda_1 h_1^1 + \cdots + \lambda_r h_s^r) g_s + \lambda_1 t_1 + \cdots + \lambda_r t_r + \lambda_{r+1}$$

Each $h_i + \lambda_1 h_i^1 + \cdots + \lambda_r h_s^r$ is a polynomial so $g \in C'$.

For the other direction of inclusion the argument is symmetric. Suppose $g \in C'$. Then

$$g = h_1 g_1 + \cdots + h_s g_s + \lambda_1 t_1 + \cdots + \lambda_r t_r + \lambda_{r+1}$$

for polynomials $h_1, \ldots, h_s$ and non-negative scalars $\lambda_1, \ldots, \lambda_{r+1}$. By construction of each $t_i$ we have $t_i = q_i - f_i$.
for $f_i \in \langle g_1, \ldots, g_s \rangle$.

$$g = h_1 g_1 + \cdots + h_s g_s + \lambda_1 (q_1 - (h_1^1 g_1 + \cdots + h_s^1 g_s)) + \cdots + \lambda_r (q_r - (h_1^r g_1 + \cdots + h_s^r g_s)) + \lambda_{r+1}$$

The above can be reorganized as

$$g = (h_1 - \lambda_1 h_1^1 - \cdots - \lambda_r h_1^r) g_1 + \cdots + (h_s - \lambda_1 h_s^1 - \cdots - \lambda_r h_s^r) g_s + \lambda_1 q_1 + \cdots + \lambda_r q_r + \lambda_{r+1}$$

Each $h_i - \lambda_1 h_i^1 - \cdots - \lambda_r h_i^r$ is a polynomial so $g \in C$.

**Reduction**

With the notion of a reduced cone in hand, we immediately arrive at a method to reduce a polynomial $t$ by a cone $C$ with respect to a monomial order $\ll$. All we need to do is reduce $t$ by the equality part of $C$, i.e., the Gröbner basis, and then reduce the result by the polyhedral-cone part, using the method from § 5.4.2. More explicitly, given an arbitrary monomial order $\ll$, cone of polynomials $C$, and polynomial $t$, we can reduce $t$ by $C$, obtaining $t^\ast$, using the following steps:

1. From $C = \langle p_1, \ldots, p_k \rangle + [q_1, \ldots, q_r, 1]$, compute a Gröbner basis $B = \langle g_1, \ldots, g_s \rangle$ for $\langle p_1, \ldots, p_k \rangle$, and for each $q_i$ compute $t_i = \text{red}_B(q_i)$. From $B$ and $\{t_1, \ldots, t_r\}$ construct the reduced cone $C' = \langle g_1, \ldots, g_s \rangle + [t_1, \ldots, t_r, 1]$.

2. “Equality reduce” $t$ by the Gröbner basis $B$. That is, let $t' = \text{red}_B(t)$.

3. “Inequality reduce” $t'$ with Alg. 12 to obtain the result $t^\ast$. That is, treat the polynomials of $\{t, t_1, \ldots, t_r\}$ as vectors in the finite-dimensional subspace of $K[X]$ spanned by the monomials present in $\{t, t_1, \ldots, t_r\}$, and run Alg. 12.

**Definition 5.19.** We denote the process of reducing a polynomial $t$ by a cone of polynomials $C$ with respect to a monomial order $\ll$, $\text{cred}_C(t)$.

**Theorem 5.20.** Let $t$ be a polynomial, $C$ a cone of polynomials, and $\ll$ a monomial order. Let $t^\ast = \text{cred}_C(t)$. $t^\ast$ has the following properties:
1. \( t^* - t \in C \)

2. \( t^* \) is optimal in the sense that for any other \( b \) with \( b - t \in C \), then \( \text{LM}(b) \gg \text{LM}(t^*) \).

Proof. Let \( C' = \langle g_1, \ldots, g_s \rangle + [t_1, \ldots, t_r, 1] \) be a reduced cone equal to \( C \). Let \( t' = \text{red}_B(t) \). Then \( t = h_1 g_1 + \ldots h_s g_s + t' \). Let \( t'' \) be the result of Alg. 12 on \( t' \). By Cor. 5.12 \( t'' - t' \in [t_1, \ldots, t_r, 1] \), so

\[
\begin{align*}
t'' - t' &= \lambda_1 t_1 + \cdots + \lambda_r t_r + \lambda_{r+1} \\
t'' - (t - (h_1 g_1 + \cdots + h_s g_s)) &= \lambda_1 t_1 + \cdots + \lambda_r t_r + \lambda_{r+1} \\
t'' - t &= (\lambda_1 t_1 + \cdots + \lambda_r t_r + \lambda_{r+1} - (h_1 g_1 + \cdots + h_s g_s) + \lambda_1 t_1 + \cdots + \lambda_r t_r + \lambda_{r+1})
\end{align*}
\]

for non-negative \( \lambda_1, \ldots, \lambda_{r+1} \). The last line is the witness for \( t'' - t \in C' = C \).

To show optimality consider some arbitrary \( b \) with \( b - t \in C' \). This means

\[
\begin{align*}
b - t &= h'_1 g_1 + \cdots + h'_s g_s + \lambda'_1 t_1 + \cdots + \lambda'_r t_r + \lambda'_{r+1} \\
b &= t + h'_1 g_1 + \cdots + h'_s g_s + \lambda'_1 t_1 + \cdots + \lambda'_r t_r + \lambda'_{r+1}
\end{align*}
\]

Consider \( \text{red}_B(b) = b' \). By Thm. 2.49 \( b' \) has the unique property that \( b = g + b' \) for \( g \in \langle g_1, \ldots, g_s \rangle \) and \( b' \) has no term divisible by \( \text{LM}(g_1), \ldots, \text{LM}(g_s) \). Consider the term \( t' + \lambda'_1 t_1 + \cdots + \lambda'_{r} t_r + \lambda'_{r+1} \). Because \( t' = \text{red}_B(t) \), \( t' \) has no term divisible by \( \text{LM}(g_1), \ldots, \text{LM}(g_s) \). Also because \( C' \) is a reduced cone no \( \lambda_i t_i \) is divisible by \( \text{LM}(g_1), \ldots, \text{LM}(g_s) \) either. Finally, because \( t = h_1 g_1 + \cdots + h_s g_s + t' \) we have

\[
b = (h'_1 g_1 + \cdots + h'_s g_s + h'_1 g_1 + \cdots + h'_s g_s) + (t' + \lambda'_1 t_1 + \cdots + \lambda'_r t_r + \lambda'_{r+1})
\]

This shows that \( \text{red}_B(b) = t' + \lambda'_1 t_1 + \cdots + \lambda'_r t_r + \lambda'_{r+1} \). So \( \text{red}_B(b) - t' \in [t_1, \ldots, t_r, 1] \). However, by Cor. 5.12 \( t'' \) is the minimal term with this property. Thus, \( t'' \) must be an expression in fewer dimensions than \( \text{red}_B(b) \).

In other words, by interpreting \( t'' \) and \( \text{red}_B(b) \) as polynomials \( \text{LM}(t'') \ll \text{LM}(\text{red}_B(b)) \ll \text{LM}(b) \). Thus, for arbitrary \( b \) in \( C' \), \( \text{LM}(t'') \ll \text{LM}(b) \).

\[\Box\]

Example 5.21. Consider the example from § 5.2. In that example, we said that we had the equations \( x - u_4 \) and \( y - u_5 \) in the basis of the ideal, and the inequalities \( \text{vbu}_1 - u_4 \geq 0 \), \( \text{va}_2 u_2 + \text{vbu}_2 - \text{vbu}_1 + u_2 \geq 0 \), and \( u_5 - \text{va}_2 u_2 - \text{vbu}_2 + 1 \geq 0 \) in the basis of the polyhedral cone. The ideal and polyhedral cone created for this example has many more equations and inequalities, but these are the ones that are relevant for reduction. Using the new
terminology, in § 5.2 we reduced $t = x - y$ by the reduced cone

$$C = \langle x - u_4, y - u_5, \ldots \rangle + \{vbu_1 - u_4, va_2u_2 + vbu_2, vbu_2 + vbu_2 - vbu_1 + vbu_2, u_5 - va_2u_2 - vbu_2 + 1, \ldots, 1\}.$$  

To reduce $x - y$ by $C$ we first equality reduce $x - y$ by the ideal, and obtain $u_4 - u_5$. Then, to reduce $u_4 - u_5$ by the polyhedral cone we treat each unique monomial as a separate dimension, and run Alg. 12. For example, we might create the map

$$\{d_1 \mapsto u_4, d_2 \mapsto u_5, d_3 \mapsto bvu_1, d_4 \mapsto va_2u_2, d_5 \mapsto bvu_2, d_6 \mapsto bvu_1, d_7 \mapsto vbu_2, \ldots\}.$$  

We then reduce $d_1 - d_2$ by the polyhedron $P \overset{\text{def}}{=} \{d_3 - d_1 \geq 0, d_4 + d_5 - d_3 + d_7 \geq 0, d_2 - d_4 - d_5 + 1 \geq 0, \ldots\}$ and get the result $d_2 + 1$, or equivalently, $vu_2 + 1$. By Thm. 5.20, $vu_2 + 1 - (x - y) \in C$, and $vu_2 + 1$ is optimal. Furthermore, by Lem. 5.14, these equalities and inequalities entail $x - y \leq vu_2 + 1$.

**Optimality.** Thm. 5.20 gives optimality with respect to a monomial order $\ll$ which is a total order on monomials. However, when extending $\ll$ to polynomials, the comparison becomes a pre-order. For example $x + y$, and $x$ have the same leading monomial if $x \gg y$. Furthermore, coefficients are not compared in the monomial order (for example, $5x$ and $2x$ are equivalent in the monomial order). For this reason, there can be multiple distinct optimal terms that satisfy the conditions of Thm. 5.20. The reduction method is not guaranteed to return all of the optimal terms. Thm. 5.20 guarantees that the reduction will return one of the optimal bounds.

### 5.5 Saturation

In this section, we give a heuristic method for the non-linear OSB problem. The idea is that from a formula $\phi$ we extract implied polynomial equalities and polynomial inequalities, and construct a cone of polynomials from the result. We illustrate the saturation process of this chapter using the example from § 5.2.

**Purification.** The first step is to *purify* all the terms in the formula. For each non-polynomial function symbol, we introduce a fresh variable, replace all occurrences of the function with the introduced symbol, and remember the assignment in a foreign-function map. We also perform this process with respect to the function arguments as well. Thus, the foreign-function map consists of assignments $u \mapsto f(p)$, where $f$ is a non-polynomial function symbol, but $p$ has also been purified and is therefore a polynomial. The result of purification on the example from § 5.2 results in the following formula and foreign-function map:

$$\phi' \overset{\text{def}}{=} x = u_4 \land y = u_5 \land a = u_6 \land e' = e + a \land b' = b + a \land b, e, v \geq 0$$

$$\text{TM} \overset{\text{def}}{=} \{u_1 \mapsto e^{-1}, u_2 \mapsto e'^{-1}, u_3 \mapsto e^{-1}, u_4 \mapsto [vbu_1], u_5 \mapsto [vbu_2], u_6 \mapsto [abu_3]\}$$
As mentioned in § 5.2, we can also consider additional axioms satisfied by the non-polynomial functions. For example, \( \varepsilon \geq 0 \iff \frac{1}{\varepsilon} \geq 0 \). Formally, we consider these instantiated axioms as being provided by the user in the original formula; however, for convenience, in our implementation and experiments, we use templates and instantiate these axioms automatically using the foreign-function map, and then conjoin the result onto the original formula.

Once the formula has been purified and a function map created, the task becomes to extract an implied cone from the combination of the function map and the purified formula. We refer to this process as saturation. Within a saturation step, two flavors of implied equalities and inequalities can be produced. There are ones that are (linearly) implied by the formula, and there are others that are implied by the cone, but not in the cone. For example, consider the cone \( C_1 = (0) + [x, 1] \). \( x \in C \) corresponds to \( x \geq 0 \), which implies \( x^2 \geq 0 \); however, \( x^2 \not\in C_1 \). If we add \( x^2 \) to \( C_1 \), we get \( C_2 = (0) + [x^2, x, 1] \) with \( C_1 \subsetneq C_2 \). Such a step, where we add implied equalities and inequalities to a cone that are not members of the cone, is referred to as a strengthening step. Fig. 5.4 gives an overview of our saturation method. The process is iterative until no more equalities or inequalities can be added to the cone of polynomials.

### 5.5.1 Equality Saturation

This subsection covers steps (1), (2), and (3) of Fig. 5.4. We first assume we have some new implied equations \( \{p_1 = 0, \ldots, p_l = 0\} \) and inequalities \( \{q_1 \geq 0, \ldots, q_r \geq 0\} \), which have been produced from a yet-to-be-explained method. We take the equations, add them to an ideal, and compute a Gröbner basis; we take the inequalities and add them to a polyhedral cone. For the example from § 5.2, suppose that we are given the equations \( x = u_4, y = u_5, a^2 = u_6, e' = e + a, b' = b + a^2, eu_1 = 1, e'u_2 = 1, eu_3 = 1 \), and no inequalities. We add the equalities to an ideal and compute a Gröbner basis, which for this example would yield: \( \langle x - u_4, y - u_5, a^2 - u_6, e' - e - a, b' - b - a^2, eu_1 - 1, eu_2 - 1, au_2 - 1, eu_3 - 1 \rangle \). We have now finished
we check whether \( p \) axioms: example, after step (3) we have the following cone, foreign-function map, and formula with instantiated the cone reduced in the sense of Defn. 5.19. This reduction is step (3) in Fig. 5.4. Returning to the running eventually terminate.

Stabilizes. Once the ideal there are only finitely many pairs of functions to check for equality, so closure will By Hilbert's basis theorem we can only add finitely many equations to the ideal before it eventually

Proof. Closure terminates.

Lemma 5.22. Closure terminates.

Proof. By Hilbert's basis theorem we can only add finitely many equations to the ideal before it eventually stabilizes. Once the ideal there are only finitely many pairs of functions to check for equality, so closure will eventually terminate.

After equality saturation, we reduce the set of inequalities by the newly returned Gröbner basis to keep the cone reduced in the sense of Defn. 5.19. This reduction is step (3) in Fig. 5.4. Returning to the running example, after step (3) we have the following cone, foreign-function map, and formula with instantiated axioms:

\[
\phi'' \overset{\text{def}}{=} a, b, e, v \geq 0 \land (e \geq 0 \implies u_1 \geq 0) \land (v b u_1 \geq 0 \implies u_4 \geq 0) \land \ldots
\]

\[
C \overset{\text{def}}{=} (x - u_4, y - u_5, a_2 - u_6, e' - e - a, b' - b - a_2, e u_1 - 1, e u_2 + a u_2 - 1, e u_3 - 1) + [1]
\]

\[
T M \overset{\text{def}}{=} \{ u_1 \mapsto e^{-1}, u_2 \mapsto e'^{-1}, u_3 \mapsto e^{-1}, u_4 \mapsto [v b u_1], u_5 \mapsto [v b' u_3], u_6 \mapsto [a b u_2] \}\]

Step (2) is a strengthening step, where we perform a type of congruence-closure process on the ideal and the foreign-function map. Consider the running example. In the foreign-function map we have \( u_1 \mapsto e^{-1} \) and \( u_3 \mapsto e^{-1} \). By the axiom \( w = w' \implies (w) = f(w') \), for any function \( f \), it is clear that \( u_1 = u_3 \). However, \( u_1 - u_3 \) is not a member of the ideal. The purpose of step (2) (Closure) is to find these equalities.

Our closure algorithm works by considering each pair of assignments \( w \mapsto f(p) \) and \( w' \mapsto f'(p') \) in the foreign-function map, where \( f \) and \( f' \) are the same function symbol. To check if the arguments are equal we check whether \( p - p' \) is a member of the ideal which by Cor. 2.51 can be done by checking if the result of reducing \( p - p' \) by the Gröbner basis is 0. If the result is 0 then we have that the ideal entails \( p = p' \), so \( f(p) = f'(p') \) and \( w = w' \). In this case we add the new equality \( w - w' \) (meaning \( w - w' = 0 \)) into the ideal and compute a new Gröbner basis. The new ideal might uncover new equalities of the map, so we have to check each pair of functions again until no new equalities are discovered.

Lemma 5.22. Closure terminates.

Proof. By Hilbert’s basis theorem we can only add finitely many equations to the ideal before it eventually stabilizes. Once the ideal there are only finitely many pairs of functions to check for equality, so closure will eventually terminate.
5.5.2 Consequence Finding

In step (4), our goal is two-fold. First, we want to make the polyhedral cone of inequalities salient in the sense of Defn. 5.4; second, we want to extract inequalities implied by the formula. For both of these goals, we generate a set of potential consequences and use a linear SMT solver to filter the potential consequences down to a set of true consequences.

We first need to explain the relevance of making the polyhedral cone salient. Suppose that we had the cone of polynomials $C = \langle p_1, \ldots, p_k \rangle + [q_1, \ldots, q_r, 1]$. The polyhedral cone $P = [q_1, \ldots, q_r, 1]$ represents inequalities; i.e., $v \in P$ corresponds to $v \geq 0$. If $P$ is not salient, then there exists a polynomial $f \in P$ and $-f \in P$, implying that we can derive $f \geq 0$ and $-f \geq 0$, so $f = 0$; however, assuming that the cone was reduced, $f \notin \langle p_1, \ldots, p_k \rangle$. Ideal reasoning is stronger than polyhedral-cone reasoning, so in this situation if we created $C' = \langle f, p_1, \ldots, p_k \rangle + Q$, where $Q$ is $P$ with $f$ removed, we would have $C \subseteq C'$. Fortunately, we can reformulate Lem. 5.5 to say that if there are no implied equations among $\{q_1, \ldots, q_r\}$ then there are no implied equations in all of $P$. Thus, we can make $P$ salient by asking if any of the equalities $q_1 = 0, \ldots, q_r = 0$ are implied. If so, these are newly discovered equations that will be added to the ideal in step (1) on the next saturation round.

Also, in this step we want to extract other inequalities that are implied by the formula. Consider the running example. We have that $C$ implies $e \geq 0$ and from $\phi''$ we have $e \geq 0 \implies u_1 \geq 0$. Thus, $u_1 \geq 0$, but $u_1$ is not a member of $C$. To extract $u_1 \geq 0$ as a true consequence, we generate a finite list of potential consequences by adding each atom of the negation normal form of $\phi''$ as a potential consequence. For example, from $\phi''$ some potential consequences are $e < 0$, $u_1 \geq 0$, $vbu_1 < 0$, and $u_4 \geq 0$. Note that even in the linear case this method is incomplete. That is, there are inequalities that are implied by the formula that are not present in the formula. For example, $(x \geq 0) \land (x \leq 1 \implies y \geq x) \land (1 \leq x \leq 2 \implies y \geq -x + 2) \land (x \geq 2 \implies y \geq \frac{1}{2}x - 1)$ entails $y \geq 0$, but $y \geq 0$ is not found in any atom of the negation normal form of the formula.\(^8\)

We collect both the potential equality consequences and the potential inequality consequences into a list, reduce them with the ideal, and then use a Houdini [56] like algorithm using a linear SMT solver to filter the potential consequences to a list of true consequences. We use a linear SMT solver as opposed to a non-linear one to avoid the aforementioned issues of non-linear reasoning. That is, we replace each monomial with a fresh variable before determining true consequences. For the running example, we do not yet have any known inequalities, so we do not have any inequalities to potentially upgrade to equalities; however, from the formula $\phi''$ we generate the potential consequences $Cons = \{e \geq 0, b \geq 0, e \geq 0, v \geq 0, e < 0, u_1 \geq 0, vbu_1 < 0\}$.

\(^8\)In this step we can also extract equalities. We can generate potential equalities along with inequalities by looking at the formula for equality atoms. However, if we only want to look for inequalities, saturation will still work because inequalities that are actually equalities will get upgraded in some later round of saturation.
0, u_4 \geq 0, \ldots, vbu_1 - u_4 \geq 0, \ldots}. We then filter Cons to Cons^* where

$$\text{Cons}^* = \{ c \in \text{Cons} \mid \phi'' \land x - u_4 = 0 \land y - u_5 = 0 \land a2 - u_6 \land \ldots \land u_3 - u_1 = 0 \models \text{LRA} \ c \}.$$ 

What Cons^* gives is a set of equalities and inequalities that we will add to the cone of polynomials. However, we have to take one more step before we add the inequalities. For this example, Cons^* = \{ e \geq 0, \ldots, vbu_1 - u_4, \ldots \}.

### 5.5.3 Taking Products

Before we add inequalities to the cone, we “take products,” which is a strengthening step indicated as step (5) in Fig. 5.4. The process of taking products is one of the main reasons our method gives interesting answers, and it is what leads to the main expense of the overall method. Suppose that from step (4) we obtain the inequalities \( x \geq 0, y \geq 0, z \geq 0 \). In non-linear arithmetic, from these inequalities we have \( x^2 \geq 0, xy \geq 0, xz \geq 0, y^2 \geq 0, yz \geq 0, z^2 \geq 0, x^3 \geq 0, \) etc. Moreover, all of these “product” inequalities are not members of \([x, y, z]\). We could strengthen the cone by adding all of these product inequalities; however, the set of all of these products is infinite.

In our implementation, we heuristically “cut-off” the depth of products at some parameterized value \( N \). That is, we assign each inequality \( w \geq 0 \) in the cone a depth \( i \), which we denote by \( w \geq_i 0 \). Newly discovered inequalities, i.e., the ones produced from step (4), have a depth of 1. Product inequalities can be generated by the rule \( w \geq_i 0 \) and \( z \geq_j 0 \) yields \( wz \geq_{i+j} 0 \). When we add inequalities to the cone, we make sure to add all products that have a depth less than or equal to \( N \). For example, suppose that the polyhedral cone \([x, y, x^2, xy, y^2]\) corresponds to the following inequalities with indicated depths \([x \geq_1 0, y \geq_1 0, x^2 \geq_2 0, xy \geq_2 0, y^2 \geq_{2, 1}, \) \( N = 2 \), and we have the newly discovered inequality \( z \geq_1 0 \). We make sure to take all products within the new inequalities \((z^2 \geq_2 0)\), as well products with the polyhedral basis \((xz \geq_2 0, yz \geq_2 0)\), to obtain the new inequalities \([z \geq_1, z^2 \geq_2 0, xz \geq_2 0, yz \geq_2 0]\). Thus, after taking products and adding the results to the cone of polynomials we would have \( C = \langle p_1, \ldots, p_k \rangle + [x, y, x^2, xy, y^2, z, z^2, xz, yz, 1] \).

For the running example we use a saturation depth of \( N = 3 \), and would generate many inequalities from Cons^* from § 5.5.2. Generated inequalities would include \( e \geq_1 0, e^2 \geq_2 0, e^3 \geq_3 0, b \geq_1 0, eb \geq_2 0, vbu_1 \geq_3 0, \) and many more, all of which would be added to the cone of polynomials.
5.5.4 Putting it All Together

For the running example, after going through one round of saturation, we have the following:

\[
\phi'' \overset{\text{def}}{=} (vbu_1 \geq 0 \implies u_4 \geq 0) \land \ldots
\]

\[
C \overset{\text{def}}{=} \langle x - u_4, y - u_5, a_2 - u_6, e' - e - a, b' - b - a_2, eu_1 - 1, eu_2 + au_2 - 1, u_3 - u_1 \rangle + [e, e^2, e^3, b, \ldots, vbu_1, \ldots, 1]
\]

\[
TM \overset{\text{def}}{=} \{ u_1 \mapsto e^{-1}, u_2 \mapsto e'^{-1}, u_3 \mapsto e^{-1}, u_4 \mapsto [vbu_1], u_5 \mapsto [v'b'u_2], u_6 \mapsto [abu_3] \}
\]

However, going through the saturation steps again will generate even more information. For example, in \(\phi''\) we have \(vbu_1 \geq 0 \implies u_4 \geq 0\). When we performed consequence finding in step (4), \(u_4\) was not a true consequence, because it was not linearly implied by \(C\) or \(\phi''\). However, by taking products of the inequalities \(v \geq 0\), \(b \geq 0\), and \(u_1 \geq 0\), we now have \(vbu_1\) as a basis polynomial in the polyhedral cone. Therefore, now \(u_4 \geq 0\) is linearly implied by \(C\) and \(\phi''\), so running through the steps again would establish \(u_4 \geq 0\). Similarly, it may be possible for new equations to be generated in steps (2) or (4), as well.

The saturation process starts with an “empty” cone \(C_0 = \langle 0 \rangle + [1]\). Then, at each step of saturation a stronger cone is produced that is still implied by the original formula. That is, starting from \(C_0\) a sequence of cones is produced \(C_0, C_1, \ldots, C_n\) where \(C_i\) is created by running one step of saturation on \(C_{i-1}\). Each cone is implied by the original formula together with the foreign-function map. Furthermore, \(C_{i-1} \subset C_i\) for \(i \leq n\).

**Theorem 5.23.** For each \(i\). We have for every \(c \in C_i\),

\[
\phi \land \bigwedge_{u \mapsto f(p) \in TM} \phi(u = f(p)) \models_{UFOF} c \geq 0.
\]

Where \(\models_{UFOF}\) denotes entailment w.r.t. the theory of ordered fields with uninterpreted functions.

**Proof.** Here we give a proof sketch of the idea. First note that by Lem. 5.14, to show every element of the cone is sound, we only need to show that each element of the basis of the ideal is an implied equation and each element of the basis of the polyhedral cone is an implied inequality. We prove the theorem by induction on \(i\). For \(C_0 = \langle 0 \rangle + [1]\), we have the only elements of \(C_0\) are positive scalars.

For the inductive step, suppose the theorem holds for \(C_i\). Saturation creates \(C_{i+1}\) from \(C_i\) in a few different ways. To show the theorem we must consider each possibility. Step (2) (closure) can add new equations to the ideal. In this step we add the equation \(w - w'\) to the ideal when we have \(w \mapsto f(p)\) and \(w' \mapsto f(p')\) and \(p = p'\) is implied by the cone \(C_i\). By the inductive hypothesis \(p = p'\) is implied by the

---

\(^9\)To simplify the presentation, we have removed some clauses from \(\phi''\) that are no longer useful.
formula and function map, so by the function axiom $p = p'$ entails $f(p) = f(p')$ we have $w = w'$ entailed by the formula and function map. Another option to create $C_{i+1}$ is through consequence finding. We query an SMT solver with a formula equivalent to $\phi \land \bigwedge_{u \mapsto f(p) \in \text{TM}} (u = f(p)) \land C_i \models_{\text{LRA cons}}$ which implies $\phi \land \bigwedge_{u \mapsto f(p) \in \text{TM}} (u = f(p)) \land C_i \models_{\text{UF cons}}$. For any consequence cons that is an inequality we also “take products”, which is also sound with respect to ordered fields. These are the ways new equalities and inequalities get added to $C_i$, and so the resulting cone $C_{i+1}$ is also sound. Thus by induction the theorem holds.

What is key, though, is that saturation will eventually terminate. That is, $C_n = C_{n+1}$ for some $n \in \mathbb{N}$. The reason is two-fold. With respect to inequalities, because we limit inequalities with a saturation bound, there are a finite set of potential inequalities (coming from the finite formula) and inequality products because products from a finite set give a finite set. With respect to equalities, we can appeal to Hilbert’s basis theorem (see § 5.3) and say that the set of equalities must eventually stabilize because every polynomial ideal is finitely generated.

For the running example, running the saturation procedure until it stabilizes—using a saturation bound of 3—produces a cone $C = \langle x - u_4, y - u_5, \ldots + [vbu_1 - u_4, vu_2 u_6 + vbu_2 - vbu_1 + vu_2, u_5, vu_2 u_6 - vbu_2 + 1, \ldots, 1]\rangle$. Reducing $x - y$ using the procedure from § 5.4.3 gives the upper bound $vu_2 + 1$.

Comparison with Wedges

The saturation process of this chapter serves the same purpose of wedge symbolic abstraction from § 3.4. The methods are similar in how they perform equality saturation using Gröbner bases and congruence closure. However, the methods differ in a few ways. One major way the methods differ is how they generate new inequalities. The wedge domain can generate new “base” inequalities via polyhedral join, which we found to be too expensive for this application. In contrast, in this chapter we generate new “base” inequalities via a heuristic consequence finding method. Both methods also differ in how they generate new inequalities from “base” inequalities. The wedge domain heuristically applies inference rules, whereas in this chapter we systematically take the product of discovered inequalities until we reach a specified bound, $N$. Ultimately, the methods are incomparable. The wedge domain’s use of the more powerful polyhedral join allows it to discover additional information that would not be discovered by the saturation method of this chapter. On the other hand, the systematic application of the product rule in this chapter allows the method to discover information outside what the wedge domain would be able to find.
5.6 Effective Degree Order

The methods of § 5.4 can reduce a polynomial \( t \) by a cone of polynomials \( C \) with respect to an arbitrary monomial order. Thus, a user can use the results of saturation, a purified term to rewrite \( t \), a cone of polynomials \( C \), and a foreign-function map \( TM \), to create an arbitrary monomial order for some downstream task. However, determining an appropriate order is a challenging task. In this section we present the \textit{effective-degree order} which is the monomial order we use in \textsc{AutoBound}.

Our definition for effective degree includes a set of variables \( W \), specified by the user, which indicate variables to keep. That is, by specifying a set \( W \) the user is indicating that any term containing a variable not in \( W \) is worse than a term containing only \( W \) variables. In this way, the set \( W \) encodes a “variable restriction” on the preference of terms. Incorporating variable restriction into a traditional monomial order is straightforward. However, in our setting we have the additional challenge of function symbols in the foreign-function map \( TM \). Suppose we have the assignment \( x \mapsto f(p) \) in \( TM \). If \( p \) contains only \( W \) variables, then in the monomial order the variable \( x \) should also be thought of as referring to only \( W \) variables. However, it may be the case that \( p \) contains some variables \textit{not} in \( W \), but there could be another polynomial \( p' \) with \( p = p' \) and \( p' \) \textit{does} contain only \( W \) variables. For example, consider the assignment \( u_2 \mapsto e' - 1 \) in the running example, and let \( W = \{a, b, e, v\} \). \( e' \) \textit{is not} in \( W \), but we have \( e' = e + a \) implied by the ideal. In other words, we have \( u_2 \mapsto (e + a)^{-1} \), and now \( u_2 \) refers to a function containing only \( W \) variables.

To solve this challenge we must first “reduce” the foreign-function map \( TM \) using the ideal in \( C \). The idea is that we have an initial definition for effective degree with variable restriction. We then reduce each polynomial \( p \) in each assignment in \( TM \). Each reduction may rewrite \( p \) towards another \( p' \) which is lower in effective degree and thus may only contain \( W \) variables. We then update the effective degree order and repeat until the process stabilizes. \(^{10}\)

Let \( W \) be a set of variables, \( TM : Y \to \text{Term} \) a foreign-function map. Let \( m \) be a monomial over variables \( X \) with \( Y \subseteq X \). The \textit{effective-degree} of a variable \( x \) denoted \( \text{eff.deg}^W_{STM}(m) \) is a pair of natural numbers and is defined recursively as follows:

\[
\text{eff.deg}^W_{STM}(x) \overset{\text{def}}{=} \begin{cases} 
(0, 1) & \text{if } x \not\in TM \text{ and } x \in W \\
(1, 0) & \text{if } x \not\in TM \text{ and } x \not\in W \\
\text{LM}(p) & x \mapsto f(p) \in TM
\end{cases}
\]

\[
\text{eff.deg}^W_{STM}(n) \overset{\text{def}}{=} (0, 0) \quad \text{eff.deg}^W_{STM}(st) \overset{\text{def}}{=} \text{eff.deg}^W_{STM}(s) + \text{eff.deg}^W_{STM}(t)
\]

\(^{10}\)In our experiments we always use an effective-degree order so this reduction step occurs during closure, discussed in § 5.5.1, and not as a separate step.
In the above $+$ is taken pointwise, and $\text{LM}(p)$ is $\max(\text{eff.deg}_{\text{TM}}(m_1), \ldots, \text{eff.deg}_{\text{TM}}(m_n))$ with max taken in lexicographic order w.r.t. the monomials $m_1, \ldots, m_n$ of $p$. To make $\text{eff.deg}_{\text{TM}}(\_)$ a total order we break ties between $\text{eff.deg}_{\text{TM}}(x)$ and $\text{LM}(p)$ for $x \mapsto f(p) \in \text{TM}$ by taking $x \gg \text{LM}(p)$.

**Example 5.24.** Consider the un-reduced map $\text{TM}$ from earlier, and let $W = \{a, b, e, v\}$

\[
\text{TM} = \{u_1 \mapsto e^{-1}, u_2 \mapsto e'^{-1}, u_3 \mapsto e^{-1}, u_4 \mapsto [vbu_1], u_5 \mapsto [vb'u_2], u_6 \mapsto [abu_3]\}
\]

$\text{eff.deg}_{\text{TM}}(u_5) = \text{eff.deg}_{\text{TM}}'(v) + \text{eff.deg}_{\text{TM}}'(b') + \text{eff.deg}_{\text{TM}}'(u_2) = (0, 1) + (1, 0) + (1, 0) = (2, 1)$. Now consider the reduced map $\text{TM}'$, using the ideal of $C$:

\[
\text{TM}' = \{u_1 \mapsto e^{-1}, u_2 \mapsto (e + a)^{-1}, u_3 \mapsto e^{-1}, u_4 \mapsto [vbu_1], u_5 \mapsto [v(b + u_6)u_2], u_6 \mapsto [abu_3]\}
\]

$\text{eff.deg}_{\text{TM}}'(u_5) = (0, 5)$.

**Lemma 5.25.** Reducing a foreign-function map $\text{TM}$ using effective degree eventually stabilizes.

**Proof.** We must check two things to show the lemma. First, we need to make sure that effective degree remains well-defined during the reduction process. Namely, if the map $\text{TM}$ contains a cycle then effective degree is not well-defined. Let $>$ be a partial order on variables of $\text{TM}$ such that $u > u'$ if $u' \in \text{vars}(p)$ with $u \mapsto f(p) \in \text{TM}$. If such a partial order exists the map is well-defined. In the effective degree order we require $u \gg \text{LM}(p)$. Thus, if $u' \in p$ we must have $u \gg u'$. Moreover at every step of reducing the map we have $u \gg \text{LM}(p) \gg \text{LM}(\text{red}_B(p)) \gg u''$ for every $u''$ in $\text{red}_B(p)$. Thus, assuming $\text{TM}$ is well-formed to begin with, at every step of reduction we can take the partial order that shows $\text{TM}$ is well-defined to be the effective degree order.

To show termination, we define the effective degree of $\text{TM}$ as $\max_{u \mapsto f(p) \in \text{TM}} \text{eff.deg}_{\text{TM}}(u)$. Consider reducing the map with respect to some basis $B$ to create some map $\text{TM}'$. By Thm. 2.67, we have that $\text{eff.deg}_{\text{TM}}(u) = \text{LM}(p) \gg \text{LM}(\text{red}_B(p)) = \text{eff.deg}_{\text{TM}'}(u)$ for each $u \mapsto f(p)$ in $\text{TM}$. Thus the effective degree of $\text{TM}'$ is smaller than the effective degree of $\text{TM}$. Moreover, $\text{eff.deg}_{\text{TM}}(x) \in \mathbb{N} \times \mathbb{N}$ is well-ordered with minimal element $(0, 0)$. Thus, termination is guaranteed. \hfill $\Box$

In §5.7 we use effective degree to mean the effective degree w.r.t. a reduced map $\text{TM}$.
5.7 Experiments

We implemented our technique in a tool called AutoBound, handling arithmetic terms with floors and divisions.\textsuperscript{11} We use Z3\textsuperscript{12} for SMT solving, and the FGb library\textsuperscript{13} for Gröbner-basis calculations. Our evaluation of AutoBound targets the following questions, grouped in two categories:

1. **Performance** These questions address the run-time behavior of AutoBound, and its scalability, and how the time breaks down by component. We study:

   a) How much overall time does it take for AutoBound to produce a bound?

   b) How does the overall time break down into the time spent in the saturation step and in the reduction step?

   c) What is the size of the (representation of the) cone that AutoBound generates—how many equalities are produced to form the ideal, and how many inequalities over how many distinct monomials are produced to form the polyhedral cone? (The number of monomials produced gives the number of dimensions of the corresponding polyhedron and the number of inequalities gives the number of constraints of the polyhedron.)

   d) How does Alg. 12 compare with the naive method of polyhedral-cone reduction that is based on linear programming, outlined in § 5.4.2.\textsuperscript{13}

   e) How does AutoBound scale as the saturation bound is increased?

2. **Bound quality** These questions examine the output bound that AutoBound synthesizes.

   a) Is the bound optimal w.r.t. the effective-degree order?

   b) Is the bound desirable—tight yet meaningful?

   c) What saturation depth suffices for AutoBound to synthesize desirable bounds?

We investigated these questions using a set of benchmarks that we collected from Solidity code provided to us by industry experts in smart-contract verification, which we modelled in our tool as a set of equational and inequational assumptions over initial and intermediate variables in the program. Overall, we find that AutoBound is able to produce optimal, meaningful bounds in a few seconds in nearly all the benchmarks. We begin with a brief description of the benchmarks.

\textsuperscript{11} We used in saturation (§ 5.5) the equality axiom for $e \cdot \frac{1}{2}$ mentioned in § 5.2, and the inequality axioms $\forall e. e - 1 \leq |e| \leq e, \forall e. e \geq 0 \implies \frac{1}{2} \geq 0$, and $\forall e. e \geq 0 \implies |e| \geq 0$.

\textsuperscript{12} https://www-polsys.lip6.fr/~jcf/FGb/index.html.

\textsuperscript{13} In our experiments we used Z3’s Optimize module to solve the multi-objective linear program.
5.7.1 Benchmarks

We briefly describe each of the problems that we considered in our evaluation.

*Elastic.* This is the example that was described in detail in § 5.2.

*Fixed-point arithmetic.* Here, non-integer numbers are represented by multiplying them with a scaling factor $sf$, i.e., a rational number $x \in \mathbb{Q}$ is approximately represented by rounding $x \cdot sf$ to an integer. Multiplication and division need to correct for the double occurrence of the scaling factor, dividing by $sf$ upon multiplication and multiplying by $sf$ upon division. All these divisions are integer divisions and so not exact. We are interested in what happens when a number represented by $a \in \mathbb{Z}$ is multiplied and then divided by the same number represented by $b \in \mathbb{N}$, that is, in the term $t = \left\lfloor \frac{ab}{b} \cdot sf \right\rfloor$, and seek a bound over any of the variables present, under the inequational assumptions $b \geq 0, sf \geq 0$. Note the nested structure of floor terms.

*Manual price.* There is an auction with a price that decreases linearly with time. The price in time $\tilde{t} \in [\text{beginTime}, \text{endTime}]$ is computed by

$$\text{drop}(\tilde{t}) = \left\lfloor \frac{\text{startPrice} - \text{minimumPrice}}{\text{endTime} - \text{startTime}} \right\rfloor, \quad \text{price}(\tilde{t}) = \text{startPrice} - \text{drop}(\tilde{t}) \cdot (\tilde{t} - \text{startTime}).$$

To show that the price always resides in the expected interval, we want bounds on the term $t = \text{price}(\tilde{t})$, under the inequational assumptions $\text{startTime} \leq \tilde{t} \leq \text{endTime}, \text{minimumPrice} \leq \text{startPrice}$, and over any of the variables present.

*Manual-price monotonicity.* In the context of the previous benchmark, to show monotonicity of the price with time, we are interested in an upper bound on the term $t_2 = \text{price}(\tilde{t}_2) - \text{price}(\tilde{t}_1)$ under the inequational assumptions $\text{startTime} \leq \tilde{t}_1 \leq \tilde{t}_2 \leq \text{endTime}, \text{minimumPrice} \leq \text{startPrice}$, and over any of the variables present.

*Token.* In one implementation of an “elastic” smart contract by TrustToken\textsuperscript{14}, one property of interest is that a client cannot increase their gain by splitting transactions; specifically, that $\text{balanceSplit}$, the balance of a client after $\text{withdraw}(x); \text{withdraw}(x)$, is not greater than $\text{balanceJoined}$, the balance after executing $\text{withdraw}(2x)$ from the same initial state. Each $\text{withdraw}$ operation modifies the client’s balance, including a fee for the operation, as well as the total supply held by the contract, in ways that involve several multiplications and divisions with another quantity called $\text{funds}$. We are interested in bounds for the term $t = \text{balanceSplit} - \text{balanceJoined}$, under inequational assumptions that the client’s balance and total supply are large enough to withdraw $2x$, and that $\text{funds}$ is non-negative, over the variables $x$, the initial values of the client’s balance and supply, and $\text{funds}$ (but not over value of intermediate variables).

*NIRN.* In a different implementation of an “elastic” smart contract by NIRN\textsuperscript{15}, the property of interest is that

\textsuperscript{14}https://github.com/trusttoken
\textsuperscript{15}https://github.com/indexed-finance/nirn
Table 5.1: Performance of AutoBound on the examples. #eq and #ineq’s are resp. equality and inequality assumptions initially given (not including instantiated axioms); #floors is the number of integer divisions (floor of division) terms in the assumptions. #c-eq and #c-ineq are resp. the number of equalities/inequalities in the generated cone’s ideal/polyhedron; #m is the number of distinct monomials in the inequalities. Time is the overall execution time of AutoBound (all times in seconds). csat is the time to saturate the cone. reduce is the time to reduce w.r.t. the cone using local projection. reduce-lp is the time to reduce using linear programming instead of local projection. All experiments in this table were taken using a product saturation depth of 3.

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>#eq</th>
<th>#in</th>
<th>#floors</th>
<th>#c-eq</th>
<th>#c-in</th>
<th>#c-m</th>
<th>time (s)</th>
<th>csat (s)</th>
<th>reduce (s)</th>
<th>reduce-lp (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>elastic</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>8</td>
<td>814</td>
<td>413</td>
<td>2.4</td>
<td>1.4</td>
<td>1.0</td>
<td>8.5</td>
</tr>
<tr>
<td>fixedPointInt</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>162</td>
<td>131</td>
<td>0.3</td>
<td>0.2</td>
<td>0.1</td>
<td>0.7</td>
</tr>
<tr>
<td>manualPrice</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>163</td>
<td>168</td>
<td>0.5</td>
<td>0.4</td>
<td>0.1</td>
<td>0.6</td>
</tr>
<tr>
<td>manualPriceMonotone</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>218</td>
<td>228</td>
<td>0.7</td>
<td>0.6</td>
<td>0.1</td>
<td>1.7</td>
</tr>
<tr>
<td>tokent</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>13</td>
<td>815</td>
<td>288</td>
<td>2.0</td>
<td>1.1</td>
<td>0.9</td>
<td>2.8</td>
</tr>
<tr>
<td>nirn</td>
<td>10</td>
<td>5</td>
<td>6</td>
<td>16</td>
<td>4057</td>
<td>1351</td>
<td>81.0</td>
<td>9.6</td>
<td>71.4</td>
<td>963.1</td>
</tr>
</tbody>
</table>

when a client deposits an amount and receives a corresponding amount of shares (see § 5.2), the number of shares does not vary much at different times for the same amount. Specifically, the question is what happens under an execution of \( \text{shares1} = \text{deposit}(x); \ \text{shares2} = \text{deposit}(x) \) to the term \( t = \text{shares1} - \text{shares2} \), under appropriate inequational assumptions that ensure that deposit operations execute successfully. We would like to find a bound for \( t \) defined over the variable \( x \) and the initial values alone (not over values of intermediate variables). Each \text{deposit} operation modifies the total supply and the client’s balance, in a calculation that is more involved than in the benchmark \text{Token}, described above.

### 5.7.2 Performance

Tab. 5.1 displays the running time of AutoBound on each of the benchmarks with saturation depth 3. Time measurements are averaged over 5 runs, running on an Ubuntu 18.04 VM over an X1 Carbon 5th Generation with an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz processor.

**Overall time**

In the majority of our benchmarks, AutoBound is able to produce the bound in less than a few seconds. The only exception is the NIRN benchmark, where the upper bound requires about a minute and a half, which we attribute to the larger number of floor and division terms and accordingly a larger resulting cone.

**Breakdown**

In most benchmarks, saturation-time is slightly larger than reduce-time. The only exception is NIRN, where reduce-time dominates. This may indicate that the time to reduce increases more steeply then the time saturate as the cone size increases. This is also apparent in the trends when saturation depth is increased (shown in § 5.8).
Cone size

The size of the cone affects the running time of both saturation and reduction. The number of equalities in the ideal grows with the number of equational assumptions in the input, although not steeply. The number of inequalities in the cone grows rather steeply with the number of inequational assumptions, and especially with the number of integer divisions—these growth patterns are likely because floor and division terms trigger the instantiation of inequational axioms, which in the saturation process further amplifies the number of inequalities because of taking products of inequational assumptions and inequational terms from axiom instantiations.

Reduction with Linear Programming

In all benchmarks, performing the reduction step using linear programming is significantly slower than using our local projection method, often in one or more orders-of-magnitude, justifying our use of local projection for reduction.

Scalability with Saturation Depth

Fig. 5.5 shows how the overall time of AutoBound changes as the saturation depth is increased, using a timeout of 90 seconds (each data point averaged over 3 runs). We see that running time increases steeply, which is expected because the number of product terms grows exponentially with the saturation depth. We are pleasantly surprised by the fact that AutoBound tolerates even larger saturation depths for some of the benchmarks. However, saturation depth of 8 causes AutoBound not to terminate within the time limit in any of the benchmarks. Further data on the cone size and the run-time breakdown as the saturation depth is increased until a 10 minutes timeout appear in § 5.8.

Figure 5.5: Total running time as the saturation depth is increased. Missing points indicate a time greater than the timeout of 90 seconds. For each benchmark, the depth that suffices for both the upper and lower bounds to be the ones given in Tab. 5.3 is marked by a larger, black marker of the same type as the line plot for the benchmark.
Table 5.2: The size of the cone and run-time breakdown of each examples as the saturation depth is increased until a timeout of 10 minutes. #c-eq and #c-ineq are resp. the number of equalities/inequalities in the generated cone’s ideal/polyhedron; #m is the number of distinct monomials in the inequalities. time is the overall execution time of AutoBound (all times in seconds). csat is the time to saturate the cone. reduce is the time to reduce w.r.t. the cone using local projection. – indicates a timeout.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Saturation depth</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixedPointInt</td>
<td>#c-eq</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>#c-ineq</td>
<td>8</td>
<td>42</td>
<td>162</td>
<td>489</td>
<td>1281</td>
<td>2993</td>
<td>6425</td>
</tr>
<tr>
<td></td>
<td>#m</td>
<td>9</td>
<td>41</td>
<td>131</td>
<td>336</td>
<td>742</td>
<td>1470</td>
<td>2682</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>0.1</td>
<td>0.2</td>
<td>0.4</td>
<td>1.1</td>
<td>4.6</td>
<td>25.3</td>
<td>83.2</td>
</tr>
<tr>
<td></td>
<td>csat</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.7</td>
<td>2.0</td>
<td>6.4</td>
<td>20.8</td>
</tr>
<tr>
<td></td>
<td>reduce</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>0.4</td>
<td>2.5</td>
<td>18.9</td>
<td>62.4</td>
</tr>
<tr>
<td>manualPrice</td>
<td>#c-eq</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>#c-ineq</td>
<td>7</td>
<td>43</td>
<td>163</td>
<td>492</td>
<td>1284</td>
<td>2999</td>
<td>6431</td>
</tr>
<tr>
<td></td>
<td>#m</td>
<td>10</td>
<td>49</td>
<td>168</td>
<td>462</td>
<td>1092</td>
<td>2310</td>
<td>4488</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>0.2</td>
<td>0.3</td>
<td>0.6</td>
<td>2.5</td>
<td>11.4</td>
<td>66.0</td>
<td>338.9</td>
</tr>
<tr>
<td></td>
<td>csat</td>
<td>0.2</td>
<td>0.2</td>
<td>0.5</td>
<td>1.3</td>
<td>4.4</td>
<td>16.7</td>
<td>61.9</td>
</tr>
<tr>
<td></td>
<td>reduce</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>1.2</td>
<td>7.0</td>
<td>49.3</td>
<td>277.0</td>
</tr>
<tr>
<td>manualPriceMonotone</td>
<td>#c-eq</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>#c-ineq</td>
<td>8</td>
<td>53</td>
<td>218</td>
<td>712</td>
<td>1999</td>
<td>5001</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>#m</td>
<td>11</td>
<td>60</td>
<td>228</td>
<td>690</td>
<td>1782</td>
<td>4092</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>0.2</td>
<td>0.4</td>
<td>0.8</td>
<td>4.3</td>
<td>47.9</td>
<td>283.1</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>csat</td>
<td>0.2</td>
<td>0.3</td>
<td>0.6</td>
<td>2.3</td>
<td>11.3</td>
<td>54.1</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>reduce</td>
<td>0.0</td>
<td>0.0</td>
<td>0.1</td>
<td>2.0</td>
<td>36.5</td>
<td>229.0</td>
<td>–</td>
</tr>
<tr>
<td>tokent</td>
<td>#c-eq</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>#c-ineq</td>
<td>12</td>
<td>90</td>
<td>815</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>#m</td>
<td>12</td>
<td>71</td>
<td>288</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>0.2</td>
<td>0.3</td>
<td>2.3</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>csat</td>
<td>0.2</td>
<td>0.3</td>
<td>1.2</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>reduce</td>
<td>0.0</td>
<td>0.1</td>
<td>1.0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>nirn</td>
<td>#c-eq</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>#c-ineq</td>
<td>21</td>
<td>322</td>
<td>4057</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>#m</td>
<td>21</td>
<td>209</td>
<td>1351</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>0.4</td>
<td>1.0</td>
<td>65.7</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>csat</td>
<td>0.4</td>
<td>0.9</td>
<td>9.6</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>reduce</td>
<td>0.0</td>
<td>0.1</td>
<td>56.1</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

5.8 Detailed Statistics from Saturation Depth Experiment

The cone size and run-time breakdown of AutoBound as the saturation bound is increased in each benchmark, until a timeout of 10 minutes.

5.8.1 Bound Quality

Tab. 5.3 presents the bounds we obtain for the benchmarks with saturation depth 3, as well as the bounds generated manually by a verification expert (where available). + indicates that AutoBound has generated a bound we deem as uninformative. As explained in § 5.4, there are potentially multiple optimal upper-bounds.
Table 5.3: Comparison of the bounds that AutoBound generates for input term \( t \) (saturation depth=3) and the bounds curated by a human expert. * indicates a result that is too large to include in the table. ed shows the (second component of the) effective-degree of the bound (the first component is 0).

<table>
<thead>
<tr>
<th>Benchmark name</th>
<th>t</th>
<th>AutoBound</th>
<th>ed</th>
<th>Expert</th>
<th>ed</th>
</tr>
</thead>
<tbody>
<tr>
<td>elastic</td>
<td>( x - y )</td>
<td>( \frac{a}{e} + 1 ) ( \geq -1 )</td>
<td>2</td>
<td>( \frac{a}{e} + 1 ) ( \geq 0 )</td>
<td>2</td>
</tr>
<tr>
<td>fixedPointInt</td>
<td>( \frac{a}{b} )</td>
<td>( a ) ( \geq a - \frac{d}{e} )</td>
<td>1</td>
<td>( a ) ( \geq a - \frac{d}{e} )</td>
<td>1</td>
</tr>
<tr>
<td>manualPrice</td>
<td>( \text{price}(1) )</td>
<td>( \text{startPrice} ) ( \geq \text{minimalPrice} )</td>
<td>1</td>
<td>( \text{startPrice} ) ( \geq \text{minimalPrice} )</td>
<td>1</td>
</tr>
<tr>
<td>manualPriceMonotone</td>
<td>( \text{price}(t_2) - \text{price}(t_1) )</td>
<td>( 0 ) ( \leq 0 )</td>
<td>0</td>
<td>( 0 ) ( \leq 0 )</td>
<td>0</td>
</tr>
<tr>
<td>token</td>
<td>( \text{balanceJoined} - \text{balanceSplit} )</td>
<td>( \frac{1}{2} ) ( - \frac{1}{2} \times \text{funds} )</td>
<td>1</td>
<td>( \frac{1}{2} ) ( - \frac{1}{2} \times \text{funds} )</td>
<td>n.a.</td>
</tr>
<tr>
<td>nirn</td>
<td>( \text{shares1} - \text{shares2} )</td>
<td>( 1.88 ) ( \ast )</td>
<td>5</td>
<td>n.a.</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

Moreover, the technique can discover several upper bounds of the same effective-degree in one run of Alg. 11; in AutoBound we simply return multiple bounds in that case. In the table, this arose in fixedPointInt (5 upper bounds), where it was easy to select the most appealing of them (the other involving additional added terms); and NIRN (2 lower bounds), where both bounds were uninformative, as indicated in the table by *. The latter case we discuss separately, below in § 5.8.1.

**Optimality**

The bounds that AutoBound produces are nearly optimal w.r.t. the effective-degree order. The lower bound for elastic, and the upper bound for manual-price monotonicity, token, and NIRN are all constant, which is always optimal. The upper bounds for fixed point and manual price are also optimal, consisting merely of one variable. (These two results are optimal because there is no valid constant bound; moreover, there is no other bound consisting of a single variable that is valid.) It is harder to evaluate whether the remaining bounds are close to optimal w.r.t. the effective-degree order; however, they have the same effective-degree as the expert bounds, where the latter are available.

**Desirability**

As Tab. 5.3 demonstrates, the bounds that AutoBound computes match or nearly match the bounds produced by a domain expert, and differ in constant or nearly-constant (e.g. \( \frac{1}{b} \)) terms. We attribute these differences to the challenge of inequality reasoning inside a function, which we discussed in § 5.2.

The lower bound for NIRN that AutoBound computes does not seem desirable; it is complex, with an effective degree of 5, four floor terms, and three levels of floor nesting. This result may indicate that a saturation bound larger than 3 is necessary, but we believe that the ultimate cause is inequational assumptions
that are present in the original code but are missing in our model (this was our experience with the token example and the NIRN upper bound).

**Sufficient Saturation Depth**

The smallest saturation bound that suffices to produce the bounds of Tab. 5.3 are marked on Fig. 5.5 using a larger marker of the same type is the other data points of the benchmark. All are below 3, and 2 suffices for some.

**5.9 Related Work**

*Optimization modulo theories.*

Optimization modulo theories is the problem of minimizing (or maximizing) an objective function subject to a feasible region that is defined by a first-order formula [123, 97, 18]. This chapter investigates a variation of this problem, in which the desired result is a term rather than a value. Most work on optimization modulo theories is concerned with linear arithmetic. An exception is Bigarella et al. [18], which—similarly to our work—handles non-linearity by “linearizing” the problem. Bigarella et al. [18] incorporates a linear optimization modulo theories solver into an abstraction-refinement loop with an exact non-linear solver that generates lemmas on demand. In contrast, our method eagerly communicates lemmas between a linear solver and a cooperating (incomplete) non-linear solver (based on cones of polynomials).

*Non-linear abstract domains.* The set of cones can be seen as an abstract domain, in which elements are conjunctions of polynomial inequalities. In this sense, cones are analogous to convex polyhedra, which represent conjunctions of linear inequalities. Unlike the case of convex polyhedra, the concretization function that maps cones to their solutions is not injective; that is, “inequivalent” cones may represent the same set of points (e.g., the cones generated by \{x, y\} and \{x, xy\} are different, but the formulas \(x \geq 0 \land y \geq 0\) and \(x \geq 0 \land xy \geq 0\) represent the same points). The saturation steps (§ 5.5) can be conceived as semantic reduction [35]: each saturation step derives new valid inequalities, while leaving the solutions to those inequalities unchanged.

There are a number of non-linear abstract domains that include semantic-reduction steps akin to cone saturation [87, 10, 63, 28]. The saturation of the wedge abstract domain, presented in Chapter 3, is most similar to our saturation step—it uses a combination of Gröbner-basis, congruence-closure, and polyhedral techniques for saturation. Unlike the wedge domain, this chapter uses a systematic rather than heuristic method for applying the rule that the product of two non-negative polynomials is non-negative. In this
regard, our method is similar to the domain of polynomial inequalities proposed in Bagnara et al. [10], and the domain of polynomial equalities proposed in Colón [33].

Positivstellensatz. In the case of real arithmetic positivstellensatz theorems [108, 121] give completeness results concerning which polynomials are positive given a set of inequality assumptions. They are the non-linear analogue of Farkas’ lemma. For Farkas’ lemma conical combinations by non-negative scalars is sufficient; however positivstellensatz theorems consider an ideal and conical combinations by more general terms; e.g. sum-of-squares polynomials in the case of Putinar [108]. In the Kriving–Stengle positivstellensatz, sum-of-squares polynomial combinations are taken with respect to products of the basis inequalities, which is similar to our process of taking products in § 5.5.3. Chatterjee et al. [29], Feng et al. [52] have both used positivstellensatz theorems with a degree bound in the context of program analysis by reducing to semi-definite programming, and consequently give numerical solutions. Our method does not perform such a reduction and can therefore easily give exact results. However, we lack an analogue of their completeness results.
Chapter 6

Solvable Algebraic Transition Systems

This chapter presents a program-analysis method that generates program summaries involving polynomial arithmetic. Our approach builds on prior techniques that use solvable polynomial maps for summarizing loops. Those techniques are able to generate all polynomial invariants for a restricted class of programs, but cannot be applied to programs outside of this class—for instance, programs with nested loops, conditional branching, unstructured control flow, etc. Prior to the work presented in this chapter, there was a lack of approaches for applying these prior complete methods to the case of general programs. This chapter bridges that gap. Instead of restricting the kinds of programs we can handle, our method abstracts every loop into a model that can be solved with prior techniques, bringing to bear prior work on solvable polynomial maps to general programs. While no method can generate all polynomial invariants for arbitrary programs, our method establishes its merit through a monotonicity result. Informally, monotonicity means that more accurate summaries for sub-parts of a program will yield a more accurate overall summary, i.e., “more information in yields more information out”. More formally, we show our analysis satisfies all of the axioms, including monotonicity, of pre-Kleene algebras (PKAs) (see § 4.3). We have implemented our techniques, and tested them on a suite of benchmarks from the literature. Our experiments indicate our techniques show promise on challenging verification tasks that require non-linear reasoning.

6.1 Introduction

There has been a long history of prior work that automatically generates polynomial invariants. One line of work in this direction seeks to generate all possible polynomial invariants for a restricted class of programs [116, 94, 74, 71, 72]. These complete methods give strong, predictable results; however, there is no obvious way to use such techniques for general programs, which may contain nested loops, branching, and unstructured control flow. Another line of research into the automatic generation of polynomial invariants looks to apply
to general programs; however, such techniques are often heuristic in nature [49, 87] or can only produce limited kinds of invariants [120, 26, 106, 105], e.g., only polynomials up to some degree.

It is impossible to fully bridge the gap between these two lines of research. No method can generate all polynomial invariants for general programs. No method can even generate all linear invariants for general programs [102]. However, the two lines of research raise the question: Can a method that generates polynomial invariants and works for general programs provide some guarantee on predictability?

In this chapter, we present techniques to give a positive answer to the previous question. Our method builds on the algebraic program analysis framework [89] (see § 2.1). Within this framework, summaries are created for larger and larger subprograms in a bottom-up manner. The essential challenge is the summarization of loops. In short, summarizing a loop amounts to over-approximating the reflexive transitive closure of a transition formula that describes the loop body. Once an appropriate loop-summarization technique is constructed, the algebraic framework can then employ the technique as a subroutine in the analysis of whole programs.

The technique for summarizing loops described in this chapter works by abstracting a transition formula describing an arbitrary loop body to an object, which we call a transition ideal. Informally, a transition ideal is a set of polynomial equations describing the transition relation of a loop body. Checking whether a non-linear transition formula (with an integrality predicate) implies a polynomial equation is undecidable for the standard model; however, Kincaid et al. [88] developed a theory, LIRR, for which it is possible to compute all implied polynomial equations. Our work utilizes LIRR to extract transition ideals from loop-body transition formulas. The extraction of transition ideals is complete for LIRR and sound for the standard interpretation. LIRR defines a weaker model of non-linear arithmetic for which a combination of Gröbner basis and polyhedral techniques yield complete reasoning. The gap between LIRR and the standard model is mirrored by the gap between polynomial ideals and all polynomial consequences in the standard model (see Ex. 2.29). However, LIRR is sound with respect to the standard model analogous to Cor. 2.27.

A transition ideal can be generated as a summary of an inner loop, a summary of a program with branches, etc. To summarize the loop, we would like to compute the transitive closure of the extracted transition ideal; however, the dynamics of a transition ideal can be chaotic and difficult to capture. Thus, our key insight is that we need to again abstract the transition ideal to some other object for which we know how to compute invariants. In § 6.4, we show how given an arbitrary transition ideal one can compute a best abstraction as a solvable transition ideal, which we call its solvable reflection. A solvable transition ideal is a transition ideal that contains all of the defining polynomials of at least one solvable polynomial map, a class of polynomial maps that have been used in prior work on complete polynomial-invariant generation [94, 74, 116, 6]. In § 6.5, we show that the method of Kauers and Zimmermann [82] can be generalized to compute the polynomial invariants
of a solvable transition ideal. These resulting polynomial invariants can be translated back to a transition formula, which gives a method for summarizing arbitrary loops. Hence, via the algebraic framework we achieve a method to generate polynomial invariants for programs with arbitrary control-flow.

While our method is not complete for arbitrary programs, we can guarantee that our method is monotone. The exact definition of monotonicity is given in § 6.6; informally though, a program analysis is monotone if “more information in yields more information out”. That is, improving the precision of a code-fragment, e.g., by strengthening the precondition or adding assumptions, necessarily improves the overall analysis result. Our method is monotone because we do not extract just some solvable transition ideal from a loop body, but our method extracts the best solvable transition ideal. Another way to understand this result is that while our method is not complete for general programs, it is complete, in a sense, at every loop. Given a summary for a loop body, we will always compute the solvable transition ideal that most closely approximates it. Thus, in the restricted case of a simple loop whose body is described by a solvable polynomial map, our method is complete; and, in general our method is monotone.

Summarizing, this chapter presents a program analyzer that (1) produces non-linear summaries, (2) works for polynomial programs with arbitrary control-flow, and (3) is monotone. We have implemented our method and our experiments show that it performs comparably to the top performers on the c/ReachSafety-Loops subcategory of the Software Verification Competition.

This chapter makes the following contributions:

1. We introduce transition ideals and solvable transition ideals. We further generalize solvable transition ideals with ultimately solvable transition ideals.

2. We show that transition ideals admit (ultimately) solvable reflections.
   - We present algorithms for computing solvable linear reflections (§ 6.4.1) and ultimately solvable reflections (§ 6.4.2) of transition ideals. Linear reflections correspond to best abstractions with respect to linear simulations.
   - We generalize this method to compute best abstractions with respect to polynomial simulations of bounded degree (§ 6.4.3).

3. We present a complete algorithm for computing all the polynomial invariants of (ultimately) solvable transition ideals.
   - Our summarization algorithm utilizes a sub-algorithm that might be of independent interest. Our sub-algorithm generalizes the technique of Kauers and Zimmermann [82], which computes the
a = x; b = y; p = 1; q = 0;
r = 0; s = 1; c = 0; k = 0;
while (b != 0) {
c = a; k = 0;
while (c >= b) {
c = c − b;
k = k + 1;
}
a = b;
b = c;
p, q = q, p − q * k;
r, s = s, r − s * k;
}
assert(q*x + s*y == 0);
assert(p*x + r*y == a);

Loop body formula \( F(X, X') \)

Transition ideal \( T \in \mathbb{Q}[X, X'] \)

Solvable ideal \( U \in \mathbb{Q}[Y, Y'] \)

Ideal summary \( U^* \in \mathbb{Q}[Y, Y'] \)

Loop body summary \( F(X, X')^* \)

(a) This program implements the extended Euclidean algorithm. The program is a modified version of the verification task egcd2-ll.c (https://github.com/sosy-lab/sv-benchmarks/blob/master/c/nla-digbench/egcd2-11.c). The assignments on lines 11 and 12 are parallel assignments.

(b) Overview of the method

Figure 6.1: A challenging example.

algebraic relations of c-finite sequences of rational numbers, to compute algebraic relations of c-finite sequences over an arbitrary \( \mathbb{Q} \)-algebra (Problem 6.9).

4. An implementation of the combination of the abstraction and transitive closure results yields a monotone program analysis that produces polynomial invariants for polynomial programs.

The rest of the chapter is organized as follows. § 6.2 illustrates the main features of our method on a challenging example. § 6.3 gives background on commutative algebra, polynomial ideals, and solvable polynomial maps. § 6.4 describes the method of extracting (ultimately) solvable transition ideals from arbitrary transition ideals. § 6.5 describes the method of summarizing (ultimately) solvable transition ideals. § 6.6 connects these ideas to transition formulas, and shows how the methods can be integrated into a program analyzer. § 6.7 presents the experimental evaluation. § 6.8 discusses related work.

6.2 Overview

In this section, we present our technique for program verification on the motivating example found in Fig. 6.1a. Two relevant features of this verification task are that (1) the program has a nested loop; and (2) to verify the assertions at the end of the program, an invariant involving non-linear arithmetic is required. This combination of nested loops and non-linear arithmetic presents a significant challenge for existing methods.

Our approach to analyzing programs builds on the framework of algebraic program analysis [89] (see § 2.1). Again we use the domain of transition formulas to summarize programs, and the extend and combine
We say that a transition ideal \( U \) is solvable if it contains a solvable polynomial map \( p \) (a homomorphism \( p : \mathbb{Q}[X] \to \mathbb{Q}[X] \) of a particular form, defined in § 6.3.2), in the sense that \( x' - p(x) \) belongs to \( U \) for all variables \( x \). Such a \( p \) is called a solvability witness for \( U \). For the transition ideal \( T_i \), the extraction step is trivial because \( T_i \) itself is solvable: the function \( p_i \) mapping \( \{ c \mapsto c - b, k \mapsto k + 1, b \mapsto b \} \) (which is affine, a special case of solvable) is a witness. Therefore, the result of the second step of our method is the solvable transition ideal \( U_i = T_i \).

The task of Step 3 of our method (presented in § 6.5) is to “summarize” the ideal \( U_i \), as the transition ideal \( U_i^* = \bigcap_{n=0}^{\infty} U_i^n \). Thinking of \( U_i^n \) as a set of polynomial constraints that hold after \( n \) iterations of \( U_i \), \( U_i^* \) represents the constraint that hold after any number of iterations. The process of computing \( U^* \) from a solvable transition ideal \( U \) is the subject of § 6.5. The basic idea is that we can “solve” the solvable witness \( p \) by deriving a closed-form \( p^n(x) \) for each \( x \in X \). In the case of the inner-loop of Fig. 6.1a, we have \( p_i^n(c) = c - bn, p_i^n(k) = k + n, p_i^n(b) = b \). This solution represents the value of the program variables \( c, k, \) and \( b \) after \( n \) iterations of the loop. We can then obtain polynomial invariants by eliminating \( n \). For our running example, we have \( U_i^* = \langle c' - c + b(k' - k), b' - b \rangle \). Since \( U_i \) happens to coincide with \( \langle c - p_i(c), k - p_i(k), b - p_i(b) \rangle \), computing \( U_i^* \) is essentially the same process as Kovács [94], Kauers and Zimmermann [82]’s complete invariant generation for solvable polynomial maps; § 6.5 shows how these ideas can be extended to solvable transition ideals in general.

The final step of our iteration operator (Step 4) is to translate \( U_i^* \) back to a transition formula, \( F(X, X')^@ \). For the inner-loop of Fig. 6.1a, the transition ideal \( U_i^* \) translates to the transition formula \( F_i^@ \) is our summary for the inner loop.
The example analysis of the inner loop of Fig. 6.1a gives the basic outline of how our method analyzes a loop. However, because the body of the inner loop implements a solvable polynomial map, Step 2 of Fig. 6.1b was trivial. To understand the general case when a loop’s body is not described by a solvable polynomial map, consider the outer loop of Fig. 6.1b. Let \( F_o \) be a transition formula describing the outer-loop body, and let \( T_o \) be a transition ideal obtained from \( F_o \). \( T_o \) contains many polynomials that do not represent solvable assignments. For example, because the result of analysis of the inner loop \( F_1^o \) is an approximation of the inner loop, the variable \( k \) is updated non-deterministically in \( T_o \); i.e., there is no \( k' - p \in T_o \) for any polynomial \( p \). Furthermore, \( q \) has a non-linear self-dependence, i.e., \( q' - p + qk' \in T_o \), which is not solvable. These complications mean that we cannot capture the dynamics of the variables of the outer loop using solvable polynomial maps.

However, we can find some terms that evolve predictably. For example, \( b \) and \( c \) are always equal in the post-state, i.e., \( b' - c' \in T_o \), and the sign of \( qr - ps \) flips between the pre-state and post-state, i.e., \((q'r' - p's') + (qr - ps) \in T_o \). The evolution of these terms can be represented with a solvable transition ideal. Fig. 6.2a illustrates how the evolution of these terms for a single loop iteration can be represented by the solvable transition ideal \( U'_o \). Because \( U'_o \) represents terms that are in \( T_o \), the pair \( \langle u, U'_o \rangle \) abstracts

\[
\begin{align*}
\langle d \mapsto qr - ps, e \mapsto b - c \rangle, & (d' + d) = (q'r' - p's') + (qr - ps), \\
\langle d' \mapsto qr - ps, e' \mapsto b - c' \rangle, & (b' - c') \end{align*}
\]

(a) A solvable abstraction, \( \langle u, U'_o \rangle \), of \( T_o \)  
(b) Image of \( U'_o \) under \( \overline{u} \)

Figure 6.2: An abstraction of the outer-loop transition ideal

\( T_o \) in the following way. Let \( Y \) be the set of variables \( \{d, e\} \), and recall that \( X \) is used for the set of program variables. The variable \( d \) in \( U'_o \) represents the polynomial term \( qr - ps \) and the variable \( e \) represents the linear term \( b - c \). This connection between the variables of \( U'_o \) and the terms of \( T_o \) is captured by the simulation \( u : \mathbb{Q}[Y] \to \mathbb{Q}[X] \), the polynomial homomorphism defined by \( u(d) = qr - ps \) and \( u(e) = b - c \). \( \langle u, U'_o \rangle \) is a sound abstraction of \( T_o \) in the sense that \( \overline{u}[U'_o] \) is contained in \( T_o \), where \( \overline{u}[U'_o] \) denotes the image of \( U'_o \) under the homomorphism \( u \) extended to “primed” vocabulary by defining \( \overline{u}(d) = u(d)' \) and \( \overline{u}(e) = u(e)' \).

While \( \langle u, U'_o \rangle \) is an abstraction of \( T_o \), there could be other abstractions of \( T_o \) that are better. For example, there are other polynomial terms that behave predictably that are not in \( \overline{u}[U'_o] \), e.g., \((a's' - c'r') + (as - cr) - (br - cr) \in T_o \). Other abstractions may consider terms that are not captured by \( \langle u, U'_o \rangle \). However, the technique of § 6.4 does not just extract a sound abstraction, but actually extracts a best abstraction, with respect to a class of simulations. We call such a best abstraction a solvable reflection.\(^1\) Informally, a solvable reflection is best in that any other abstraction also abstracts the solvable reflection. In § 6.4.1, we give an algorithm for

\(^1\)The name solvable reflection is derived from the notion of a reflective subcategory in category theory.
producing a solvable reflection with respect to linear simulations. For the case of linear simulations, $\langle v, V \rangle$ is a solvable reflection, with $V = \langle e' \rangle \subseteq Q[e, e']$, $v(e) = b - c$. In other words, capturing the dynamics of the linear term $b - c$ is the best among all possible abstractions of linear terms with solvable transition ideals.

In § 6.4.3, we extend our algorithm for finding linear simulations and give a method for producing solvable reflections with respect to polynomial simulations of a bounded degree. The simulation $u$ from Fig. 6.2 is an example of a degree-2 simulation, i.e., the mapping for the variable $d$ is a degree-2 polynomial. Our extended method is able to produce the solvable reflection, $\langle t, U_o \rangle$, with respect to degree-2 simulations, of $T_o$. $\langle t, U_o \rangle$ is too big to be presented here; however, it necessarily captures more dynamics of the outer loop compared to $\langle u, U_o' \rangle$. Furthermore, for this example, the closure, $U_o^*$, when combined with the program’s initial conditions is strong enough to prove the two assertions at the end of the program, verifying the program in Fig. 6.1a.

The key that makes the overall process monotone is the combination of best abstractions with complete invariant generation for solvable transition ideals. In other words, at every loop we are finding the strongest loop-body invariant that we know how to completely solve. This property leads to the result that our iteration operator is monotone (§ 6.6). Moreover, in the case when the loop body is described by a solvable polynomial map, similar to the case of the analysis of the inner-loop, our method essentially reduces to prior methods. Consequently our method is complete in such a case.

6.3 Additional Chapter Background

6.3.1 Commutative Algebra

For any ideal $I \subseteq Q[X]$ and polynomial $p \in Q[X]$, we denote the set $\{ q : p - q \in I \}$ (equivalently, $\{ p + q : q \in I \}$) as $p + I$. We use $Q[X]/I$ to denote the ring with carrier $\{ p + I : p \in Q[X] \}$, with addition and multiplication lifted to sets.

Define a $Q$-algebra $A$ to be a commutative algebra over $Q$; that is, an algebraic structure that is both a commutative ring and a linear space over $Q$. Examples of $Q$-algebras include $Q$ itself, the field of algebraic numbers $\bar{Q}$, $Q[X]$, and $Q[X]/I$ for any set of variables $X$ and ideal $I \subseteq Q[X]$. For any set of variables $X$, a $Q$-algebra $A$ defines an algebra homomorphism $(-)^A : Q[X] \rightarrow (A^X \rightarrow A)$, where $x^A(v) = v(x)$.$^2$ For any set $S \subseteq A^X$, define the vanishing ideal of $S$ to be

$$I_A(S) \overset{\text{def}}{=} \{ p \in Q[X] : p^A(v) = 0 \text{ for every } v \in S \} .$$

Observe that for any polynomial endomorphism $f : Q[X] \rightarrow Q[X]$ and any $Q$-algebra $A$, $f$ defines a function

\[ Q[X] \rightarrow Q[X] \]

$^2$\(Q[X]\) is the free $Q$-algebra generated by $X$; $(-)^A$ is the “evaluation” function that we get from freeness.
where each $\Theta$.

Then let $f : \mathbb{Q}[x, y] \to \mathbb{Q}[x, y]$ be the polynomial homomorphism defined by $f(x) = 2x$ and $f(y) = y + 1$. Then, $f_A(v) = \{x \mapsto 4, y \mapsto w + 1\}$.

Note that for any $\mathbb{Q}$-algebra $A$, the set of infinite sequences over $A$, $A^\omega$, is also a $\mathbb{Q}$-algebra. The multiplication and addition operations of $A^\omega$ are defined pointwise. Let $0_A$ and $1_A$ be the additive and multiplicative unit of $A$, then $\{0_A\}_{n=0}^\infty$ and $\{1_A\}_{n=0}^\infty$ are the additive and multiplicative unit of $A^\omega$. The scalar multiplication operation of $A^\omega$ is defined as applying the scalar multiplication of $A$ to each element of the infinite sequence.

For a $\mathbb{Q}$-algebra $A$, and a set $G \subseteq A$, we use $span\,(G)$ to denote the smallest subspace of $A$ that contains $G$, and $alg\,(G)$ to denote the smallest $\mathbb{Q}$-subalgebra of $A$ that contains $G$.

### 6.3.2 C-finite Recurrences

Let $A$ be a $\mathbb{Q}$-algebra. A sequence $\{a(n)\}_{n=0}^\infty \in A^\omega$ is **c-finite** if it satisfies a c-finite recurrence. A **c-finite recurrence** has the form:

$$a(n) = c_1a(n - 1) + \cdots + c_da(n - d) \tag{6.1}$$

for constants $c_i \in \mathbb{Q}$, for all $n \geq d$. Given a recurrence of the form from Eqn. (6.1) the **order** of the recurrence is $d$. The **characteristic polynomial** of a c-finite recurrence of the form from Eqn. (6.1) is $p(x) = x^d - c_1x^{d-1} - \cdots - c_{d-1}x - c_d \in \mathbb{Q}[x]$. The Fibonacci sequence, $\{F(n)\}_{n=0}^\infty$, is a classical example of a c-finite sequence over the $\mathbb{Q}$-algebra $\mathbb{Q}$, which satisfies the order-2 recurrence $F(n) = F(n - 1) + F(n - 2)$. The characteristic polynomial of the Fibonacci recurrence is $p_{Fib}(x) = x^2 - x - 1$.

Every c-finite recurrence admits a closed-form as a polynomial-exponential [47]. More specifically, given a recurrence of the form from Eqn. (6.1) with $c_i \in \mathbb{Q}$ and $d$ initial values $a(0), \ldots, a(d-1)$ from some $\mathbb{Q}$-algebra $A$, then

$$a(n) = \sum_{i=1}^{d} \left( z_i(n) + \sum_{j=1}^{d} p_{ij}(n)\Theta_i^n \right) a(i - 1) \tag{6.2}$$

where each $\Theta_i$ is a complex root of the characteristic polynomial of the recurrence, each $p_{ij} \in \mathbb{Q}[x]$, and $z_i(n) : \mathbb{N} \to \mathbb{Q}$ with $z_i(k) = 0$ for any $k \geq d$. More specifically, $z_i(k) = 0$ for any $k$ greater than or equal to the multiplicity of 0 as a root of the characteristic polynomial. If 0 is not a root of the characteristic polynomial then the terms $z_i(k) = 0$ for all $k \in \mathbb{N}$ and can be omitted from the closed-form. Determining such a closed-form from a recurrence is referred to as “solving” the recurrence. The roots of the Fibonacci
characteristic polynomial $x^2 - x - 1$ are $\phi = \frac{1+\sqrt{5}}{2}$ and $\psi = \frac{1-\sqrt{5}}{2}$. Assuming, $F(0) = 0$ and $F(1) = 1$, a solution to the Fibonacci recurrence in the form of Eqn. (6.2) is Binet’s formula $F(n) = \frac{1}{\sqrt{5}}\phi^n - \frac{1}{\sqrt{5}}\psi^n$.

A polynomial endomorphism $f : \mathbb{Q}[X] \to \mathbb{Q}[X]$ is **solvable** if there exists a partition $X = X_1 \cup \cdots \cup X_n$ of $X$ (with $X_i \cap X_j = \emptyset$ for all $i \neq j$) such that for each $X_i$ and each $x \in X_i$, $f(x)$ can be written as $g(X_i) + h(X_1, \ldots, X_{i-1})$, where $g$ is a linear polynomial in the variables $X_i$ and $h$ is a polynomial (of arbitrary degree) in the variables $X_1 \cup \cdots \cup X_{i-1}$.

C-finite recurrences are equivalent to solvable polynomial maps in the sense that each solvable polynomial map $f : \mathbb{Q}[X] \to \mathbb{Q}[X]$ defines $|X|$ c-finite sequences $\{f^i(x_1)\}_{i=0}^{\infty}, \ldots, \{f^i(x_n)\}_{i=0}^{\infty}$, each of order $|X|$ (§ 3.8). Conversely, each c-finite recurrence $a(n) = c_1 a(n - 1) + \cdots + c_d a(n - d)$ can be transformed to a solvable map $f$ over $d$ variables, $a_n, \ldots, a_{n-d}$ as the homomorphism defined by $f(a_n) = c_1 a_{n-1} + \cdots + c_d a_{n-d}$ and $f(a_{n-i}) = a_{n-i+1}$ for $0 < i \leq d$. Due to this equivalence, solvable polynomial maps can effectively be “solved” in the form of Eqn. (6.2) in the same way as c-finite recurrences. Hence the name **solvable** polynomial map.

### 6.3.3 Transition Formulas and Linear Integer/Real Rings

Fix a set of program variables $X$. We use $X' = \{x' : x \in X\}$ to denote a set of “primed” copies of variables in $X$ (presumed disjoint from $X$). We use $(-)'$ to denote the homomorphism $\mathbb{Q}[X] \rightarrow \mathbb{Q}[X']$ that maps each $x$ to its primed copy $x'$.

Within this chapter, we shall assume that transition formulas are expressed in the existential fragment of the language of non-linear mixed integer/real arithmetic (that is, the language of rational constants, addition, multiplication, an order relation, and an integrality predicate). Although this language is undecidable over the **standard** model, Kincaid et al. [88] showed that ground satisfiability is decidable if we allow more general interpretations, namely over linear integer/real rings (**LIRR**). For our purposes, we may think of linear integer/real rings as $\mathbb{Q}$-algebras that satisfy some additional axioms concerning the order relation and integrality predicate, which are not relevant to this dissertation. We will assume **LIRR** as a background theory in the remainder of the chapter, and use $F \models_{\text{LIRR}} G$ to denote that the formula $F$ entails the formula $G$ modulo **LIRR**.

In addition to satisfiability being decidable, there is a procedure [88] for computing the vanishing ideal $I_{\text{LIRR}}(F)$ of a formula $F$ in the existential fragment of the language: the ideal of all polynomials $p$ such that $F \models_{\text{LIRR}} p = 0$. See Ex. 6.24 for an example of $I_{\text{LIRR}}(F)$ from a formula $F$. For any ideal $I$ generated by polynomials $p_1, \ldots, p_n$, we use $F(1)$ to denote the formula $p_1 = 0 \land \cdots \land p_n = 0$. The choice of generators for $I$ is irrelevant in the sense that if two sets of polynomials $P$ and $Q$ generate the same ideal, then $\bigwedge_{p \in P} p = 0$ and $\bigwedge_{q \in Q} q = 0$ are equivalent modulo **LIRR**. Note that $I_{\text{LIRR}}$ and $F$ form a Galois connection: for any
transition formula $F$ and ideal $I$ over the free variables of $F$, we have $F \models_{\text{LIRR}} I(F)$ if and only if $I_{\text{LIRR}}(F) \supseteq I$.

This implies that (1) for formulas $F$ and $G$, if $F \models_{\text{LIRR}} G$, then $I_{\text{LIRR}}(F) \supseteq I_{\text{LIRR}}(G)$, and (2) for ideals $I$ and $J$ if $I \supseteq J$, then $F(I) \models_{\text{LIRR}} F(J)$.

### 6.3.4 Transition Ideals

The main results of this chapter are concerned with *transition ideals*. A *transition ideal* is an ideal in the ring $\mathbb{Q}[X, X']$ for some set of variables $X$. Transition ideals are not tied to the theory LIRR, but can be seen as the vanishing ideals of transition formulas, and their operations can be understood in terms of corresponding operations on transition formulas.

For transition ideals $T_1$ and $T_2$, define

$$T_1 \cdot T_2 \overset{\text{def}}{=} (T_1[X' \mapsto X'']) + T_2[X \mapsto X'']) \cap \mathbb{Q}[X, X'].$$

Equivalently, $T_1 \cdot T_2$ is equal to $I_{\text{LIRR}}(F(T_1) \circ F(T_2))$. For any transition ideal $T$ and natural number $n$, define

$$T^0 = \langle \{x' - x : x \in X\} \rangle \quad T^n \overset{\text{def}}{=} \overbrace{T \cdot \cdots \cdot T}^{n \text{ times}} \quad T^* \overset{\text{def}}{=} \bigcap_{i=0}^{\infty} T^i.$$

Note that, since the polynomials in a transition ideal are interpreted as *constraints*, $T^*$ can be interpreted as the set of constraints that are common to all $T^i$. In particular, if $F$ is a transition formula, then for any $n \in \mathbb{N}$, we have $F^n \models_{\text{LIRR}} F(I_{\text{LIRR}}(F))^*$.

**Example 6.1.** $T = \langle w' - wy, x' - 2x - y^2, y' - y - z, z' - 3z, y - z - 1 \rangle$ is an example of a transition ideal. $T^0 = \langle w' - w, x' - x, y' - y, z' - z \rangle$, $T^1 = T$, and

$$T^2 = \langle w' - wy(y + z), x' - 2(2x + y^2) - (y + z)^2, y' - (y + z) - 3z, z' - 3(3z),$$

$$(y + z) - 3z - 1, y - z - 1 \rangle$$

$$= \langle w' - wy^2 - wz, x' - 4x - 3y^2 - 2yz - z^2, y' - y - 4z, z' - 9z, y - 2z - 1, y - z - 1 \rangle$$

$$= \langle w' - w, x' - 4x - 3, y' - 1, z', y - 1, z \rangle$$

Define the *domain* of $T$ to be $\text{dom}(T) \overset{\text{def}}{=} T \cap \mathbb{Q}[X]$, and the *invariant domain* of $T$ to be $\text{dom}^*(T) = \sum_{n=0}^{\infty} \text{dom}(T^n)$. Informally, if we think of a transition ideal as a set of polynomial equations constraining the transition between a pre-state $X$ and post-state $X'$, then the domain of $T$ is the set of constraints that must be
satisfied by a pre-state in order to have a successor. Pre-states that satisfy the invariant domain of T are those states which have arbitrarily long computations described by T. Given a transition ideal T, \( \text{dom}^*(T) \) can be calculated as follows. By inspection of the definition of \( T^n \cdot T \), we see \( \text{dom}(T^n) \subseteq \text{dom}(T^{n+1}) \) for any \( n \geq 1 \). Therefore, we have the ascending chain of ideals:

\[
\text{dom}(T) \subseteq \text{dom}(T^2) \subseteq \text{dom}(T^3) \subseteq \ldots
\]

By Hilbert’s basis theorem this chain must stabilize at some \( N \). That is, there exists \( N \geq 1 \) such that \( \text{dom}(T^N) = \text{dom}(T^{N+1}) = \text{dom}^*(T) \).

We say that T is **solvable** if there is a solvable \( p : \mathbb{Q}[X] \to \mathbb{Q}[X] \) such that \( x' - p(x) \in T \) for all \( x \in X \); call \( p \) a **solvability witness** for T. We say that T is **ultimately solvable** if \( T + \text{dom}^*(T) \) is solvable.

**Example 6.2.** Recall Ex. 6.1, with \( T = \langle w' - wy, x' - 2x - y^2, y' - y - z, z' - 3z, y - z - 1 \rangle \). \( \text{dom}(T) = \langle y - z - 1 \rangle \). For this example, the invariant domain stabilizes at \( N = 2 \), and so \( \text{dom}(T^2) = \text{dom}^*(T) = \langle y - 1, z \rangle \).

T is not a solvable transition ideal, however, because the dynamics of \( w, w \mapsto wy \) cannot be captured by a solvable polynomial map.

\( T' = \langle x' - 2x - y^2, y' - y - z, z' - 3z, y - z - 1 \rangle \) is a solvable transition ideal recognized by the solvability witness \( p : \mathbb{Q}[x, y, z] \to \mathbb{Q}[x, y, z] \) defined by \( p(x) = 2x + y^2, p(y) = y + z, \) and \( p(z) = 3z \). Note that even though there is a non-linear dependence on the variable \( y \) for the assignment \( p(x) \), \( p \) is still solvable using the variable partition \( \{y, z\} \cup \{x\} \).

\( T \) is an example of an ultimately solvable transition ideal. \( T + \text{dom}^*(T) = \langle w' - w, x' - 2x - 1, y' - 1, z', y - 1, z \rangle \) is solvable, and is recognized by the solvability witness \( q : \mathbb{Q}[w, x, y, z] \to \mathbb{Q}[w, x, y, z] \) defined by \( q(w) = w \), \( q(x) = 2x + 1 \), \( q(y) = 1 \), and \( q(z) = 0 \).

Let \( T \subseteq \mathbb{Q}[X, X'] \) and \( U \subseteq \mathbb{Q}[Y, Y'] \) be transition ideals (possibly over different variables). A polynomial homomorphism \( s : \mathbb{Q}[Y] \to \mathbb{Q}[X] \) is a **simulation** from \( T \) to \( U \) (notice reversal of direction) if for every polynomial \( p \in U \), we have \( \overline{s}(p) \in T \), where \( \overline{s} : \mathbb{Q}[Y, Y'] \to \mathbb{Q}[X, X'] \) denotes the extension of \( s \) to the “doubled” vocabulary, which maps each \( x \in X \) to \( s(x) \) and each \( x' \in X' \) to \( s(x) \). We say that \( s \) is a **linear simulation** if it is linear and a simulation. If \( T \subseteq \mathbb{Q}[X, X'], U \subseteq \mathbb{Q}[Y, Y'], \) and \( V \subseteq \mathbb{Q}[Z, Z'] \) are transition ideals and \( s : T \to U \) and \( t : U \to V \) are simulations, then their composition \( t \circ s \) is a simulation from \( T \) to \( V \) (again noting that simulations go in the opposite direction of polynomial homomorphisms).
6.4 Solvable Reflections

In this section, we show that every transition ideal $T \subseteq \mathbb{Q}[X, X']$ admits a solvable reflection: there is a solvable transition ideal $U$ and a linear simulation $s : T \rightarrow U$ that is a closer approximation of $T$ than any other simulating solvable transition ideal.

Example 6.3. Fig. 6.3 illustrates a transition ideal $T$ that will be used as a running example throughout this section. One may think of $T$ as the polynomial map

$$f(w, x, y, z) = (z^2, 4x(1 - x), -4x^2 + 3x + y - 1, z + x^2 - 2xy + y^2)$$

(corresponding to the first four generators of $T$) restricted to the domain $w - z^2 - 1$ (corresponding to the fifth). The map $f$ is not solvable, since $x$ exhibits a non-linear self-dependence (in fact, $x \mapsto 4x(1 - x)$ is a logistic map, a famous example in chaos theory that Ulam and von Neumann [136] suggested as the basis of a pseudo-random number generator). If we restrict the transition ideal to the variable $w$, the resulting transition ideal $T_w = \langle w' - w + 1 \rangle$ is solvable, and we can compute a closed form for its $i^{th}$ iterate: $T^i_w = \langle w' - w + i \rangle$. This example is an instance of a solvable abstraction of $T$: $T_w$ is a solvable transition ideal that approximates the dynamics of $T$, and the nature of the approximation is given by the inclusion homomorphism $\mathbb{Q}[w] \rightarrow \mathbb{Q}[w, x, y, z]$ (mapping $w \mapsto w$).

There are other solvable abstractions of $T$ that capture different aspects of $T$’s dynamics. Observe that while it is challenging to understand the dynamics of $x$ or $y$, their difference behaves predictably: $T$ contains the polynomial $(x' - y') - (x - y) + 1$, indicating that the value of $(x - y)$ decreases by 1 at each step. Coincidentally, the dynamics of $(x - y)$ is identical to that of $w$ (both decrease by 1), so this information can be represented as the solvable abstraction $\langle \{w \mapsto x - y\}, T_w \rangle$.

Yet another solvable abstraction $\langle s, U \rangle$ is pictured in Fig. 6.3. This abstraction is more desirable than either $\langle \{w \mapsto w\}, T_w \rangle$ or $\langle \{w \mapsto x - y\}, T_w \rangle$, in the sense that $\langle s, U \rangle$ captures the dynamics of not only $(x - y)$ and $w$, but also $z$. In fact, $\langle s, U \rangle$ is a solvable reflection of $T$, in the sense that any other solvable abstraction of $T$ similarly factors through $\langle s, U \rangle$.

Formally, a solvable reflection of $T \subseteq \mathbb{Q}[X, X']$ (with respect to linear simulations) is a pair $\langle s, U \rangle$ consisting of a transition ideal $U \subseteq \mathbb{Q}[Y, Y']$ (for some set of variables $Y$) and a polynomial homomorphism $s : \mathbb{Q}[Y] \rightarrow \mathbb{Q}[X]$ such that:

1. $s$ is a linear simulation from $T$ to $U$
2. $U$ is solvable
ultimately serve as the basis of our algorithm for computing solvable reflections. For a transition ideal $T \subseteq \mathbb{Q}[X,X']$ and a set of polynomials $P \subseteq \mathbb{Q}[X]$, define
\[
\text{Det} (T, P) \overset{\text{def}}{=} \{ p \in \text{span} (X) : \exists q \in P. p' - q \in T \}
\]
Intuitively, $\text{Det} (T, P)$ represents the set of linear functionals that are “determined up to $P'$”; i.e., $T$ constrains the post-state value of $p \in \text{Det} (T, P)$ to be equal to some $q \in P$. Observe that $T$ is solvable exactly when there is an ascending chain $X_1 \subset X_2 \cdots \subset X_n = X$ of sets of variables (called a stratification of $X$) such that for each $i \in \{1, \ldots, n-1\}$, we have $\text{span} (X_{i+1}) \subseteq \text{Det} (T, \text{span} (X_{i+1}) + \text{alg} (X_i))$. (i.e., for each $x \in X_{i+1}$, we have $x' - p - q \in T$ for some linear polynomial $p \in \text{span} (X_{i+1})$ over the variables $X_{i+1}$ and some polynomial $q \in \text{alg} (X_i)$ over the variables $X_i = X_1 \cup \cdots \cup X_i$).

**Example 6.4.** Consider the solvable transition ideal $U$ in Fig. 6.3b. A solvability witness for $U$ is the polynomial homomorphism $f_U$ that sends $a \mapsto a - 1$, $b \mapsto b + 1$, and $c \mapsto c + a^2$, and the corresponding partition of the variables is $\{a, b\} \cup \{c\}$. The fact that $f_U$ is solvable corresponds precisely to the facts that both $f(a)$ and $f(b)$ belong to $\text{span} (\{a, b\}) + \text{alg} (\emptyset)$, and $f(c)$ belongs to $\text{span} (\{a, b, c\}) + \text{alg} (\{a, b\})$.

We may derive from this solvability witness a stratification $\{a, b\} \subset \{a, b, c\}$ (i.e., the $i$th stratum is the union of the first $i$ cells in the ordered partition). Observe that $\text{Det} (U, \text{span} (\{a, b\}) + \text{alg} (\emptyset)) = \text{span} (\{a, b\})$, and $\text{Det} (U, \text{span} (\{a, b, c\}) + \text{alg} (a, b)) = \text{span} (\{a, b, c\})$. 

\[
\begin{pmatrix}
  w' - z^2 \\
  x' - 4x(1 - x) \\
  y' + 4x^2 - 3x - y - 1 \\
  z' - z - x^2 + 2xy - y^2 \\
  w - z^2 - 1
\end{pmatrix}
\begin{pmatrix}
  a \mapsto (x - y) \\
  b \mapsto w \\
  c \mapsto z
\end{pmatrix}
\begin{pmatrix}
  a' - a - 1 \\
  b' - b + 1 \\
  c' - c - a^2 \\
  b - c^2 - 1
\end{pmatrix}
\begin{pmatrix}
  (x' - y') - (x - y) + 1 \\
  w' - w + 1 \\
  z' - z - (x - y)^2 \\
  w - z^2 - 1
\end{pmatrix}
\]

(a) A transition ideal, $T$ (b) A solvable reflection $\langle s, U \rangle$ of $T$ (c) Image of $U$ under $\mathfrak{s}$

Figure 6.3: A solvable reflection of a transition ideal

3. For any other pair $(v, V)$ satisfying 1 and 2, there exists a unique linear simulation $\hat{v} : U \rightarrow V$ such that $v = \hat{v} ; s$.

§ 6.4.1 describes a procedure for computing solvable reflections. We then extend this result in two ways: (1) § 6.4.2 generalizes from solvable to ultimately solvable reflections, and (2) § 6.4.3 generalizes from linear simulations to polynomial simulations.

### 6.4.1 Computing Solvable Reflections

To begin, we give an alternate characterization of solvable transition ideals, which will serve as the basis of our algorithm for computing solvable reflections. For a transition ideal $T \subseteq \mathbb{Q}[X,X']$ and a set of polynomials $P \subseteq \mathbb{Q}[X]$, define

\[
\text{Det} (T, P) \overset{\text{def}}{=} \{ p \in \text{span} (X) : \exists q \in P. p' - q \in T \}
\]

We may derive from this solvability witness a stratification $\{a, b\} \subset \{a, b, c\}$ (i.e., the $i$th stratum is the union of the first $i$ cells in the ordered partition). Observe that $\text{Det} (U, \text{span} (\{a, b\}) + \text{alg} (\emptyset)) = \text{span} (\{a, b\})$, and $\text{Det} (U, \text{span} (\{a, b, c\}) + \text{alg} (a, b)) = \text{span} (\{a, b, c\})$. 

\[
\begin{pmatrix}
  w' - z^2 \\
  x' - 4x(1 - x) \\
  y' + 4x^2 - 3x - y - 1 \\
  z' - z - x^2 + 2xy - y^2 \\
  w - z^2 - 1
\end{pmatrix}
\begin{pmatrix}
  a \mapsto (x - y) \\
  b \mapsto w \\
  c \mapsto z
\end{pmatrix}
\begin{pmatrix}
  a' - a - 1 \\
  b' - b + 1 \\
  c' - c - a^2 \\
  b - c^2 - 1
\end{pmatrix}
\begin{pmatrix}
  (x' - y') - (x - y) + 1 \\
  w' - w + 1 \\
  z' - z - (x - y)^2 \\
  w - z^2 - 1
\end{pmatrix}
\]

(a) A transition ideal, $T$ (b) A solvable reflection $\langle s, U \rangle$ of $T$ (c) Image of $U$ under $\mathfrak{s}$

Figure 6.3: A solvable reflection of a transition ideal
Furthermore, observe that the simulation \( u : T \rightarrow U \) induces a corresponding structure in \( T \), namely \( \text{Det}(T, \text{span} \{ (x - y), w \}) + \text{alg}(\emptyset) = \text{span} \{ (x - y), w \} \) (where \( \{ (x - y), w \} = \{ u(a), u(b) \} \)) and \( \text{Det}(U, \text{span} \{ (x - y), w, z \}) + \text{alg}( (x - y), w, \}) = \text{span} \{ (x - y), w, z \} \) (where \( \{ (x - y), w, z \} = \{ u(a), u(b), u(c) \} \)). In fact this stratification structure determines the solvable reflection of \( T \) uniquely (up to isomorphism) in the sense that if \( v : \mathbb{Q}[d_1, d_2, d_3] \rightarrow \mathbb{Q}[w, x, y, z] \) is any polynomial homomorphism such that \( \{ v(d_1), v(d_2) \} \) is a basis for \( \text{span} \{ (x - y), w \} \) and \( \{ v(d_1), v(d_2), v(d_3) \} \) is a basis for \( \text{span} \{ (x - y), w, z \} \), then \( (v, v^{-1} [T]) \) is a solvable reflection of \( T \). The essential idea behind the algorithm presented in this section is to calculate this structure by discovering each stratum in turn.

**Theorem 6.5.** Every transition ideal has a solvable reflection.

**Proof.** Let \( T \subseteq \mathbb{Q}[X, X'] \) be a transition ideal. We may calculate the solvable reflection of \( T \) as follows. For each natural number \( i \), we define a linear subspace of polynomials \( S_i(T) \subseteq \text{span} \{ X \} \) as follows:

- \( S_0(T) \overset{\text{def}}{=} \{ 0 \} \).
- For each \( i \geq 0 \), \( S_{i+1}(T) \) is the greatest fixed point of \( S \mapsto \text{Det}(T, S + \text{alg}(S_i(T))) \). Such a fixed point always exists by the Knaster-Tarski fixed point theorem, noting that \( S \mapsto \text{Det}(T, S + \text{alg}(S_i(T))) \) is a monotone operator on the complete lattice of subspaces of \( \text{span} \{ X \} \).

Since \( S_0(T) \subseteq S_1(T) \subseteq \ldots \) is an ascending chain of subspaces of a finite-dimensional space \( \text{span} \{ X \} \), it must stabilize (i.e., there is some \( n \) such that \( S_n(T) = S_{n+1}(T) \)). Call the resulting space \( S^*(T) \).

For any \( i \in \mathbb{N} \), let \( d_i \) be the dimension of \( S_i(T) \). Choose an ordered basis \( p_1, \ldots, p_n \) for \( S^*(T) \) such that for each \( i \), \( p_{1i}, \ldots, p_{di} \) spans \( S_i(T) \) (such a basis may be obtained by choosing an arbitrary basis for \( S_0(T) \) and then extending it to \( S_1(T) \), then extending that basis to \( S_2(T) \), and so on). Let \( Y = \{ y_1, \ldots, y_n \} \) be a set of variables disjoint from \( X \), and let \( u : \mathbb{Q}[Y] \rightarrow \mathbb{Q}[X] \) be the homomorphism that maps \( y_i \mapsto p_i \). Finally, define \( R(T) \overset{\text{def}}{=} \langle u, \overline{u}^{-1} [T] \rangle \). We will show that \( R(T) \) is a solvable reflection of \( T \).

First, we show that \( \overline{u}^{-1} [T] \) is solvable. We construct a solvability witness as follows. For any \( i \leq n \), let \( s(i) \) be the least number such that \( i \leq d_{s(i)} \). For each \( 1 \leq k \leq n \), let \( Y_k = \{ y_i : s(i) = k \} \). For any \( i \) we have \( p_i \in \text{span}(S_{s(i)}(T)) \), and so

\[
p_i \in \text{Det}(T, S_{s(i)}(T) + \text{alg}(S_{s(i)-1}(T)))
\]

It follows that there is some \( q_{i} \in S_{s(i)}(T) \) and \( \hat{q}_{i} \in \text{alg}(S_{s(i)-1}(T)) \) such that \( p_{i} - q_{i} - \hat{q}_{i} \in T \). Since \( u(\text{span}(Y_{i})) = S_{s(i)}(T) \) and \( u(\text{alg}(Y_{1} \cup Y_{2} \cdots Y_{i})) = \text{alg}(S_{s(i)-1}(T)) \), there is some \( t_{i} \in \text{span}(Y_{i}) \) and \( \hat{t}_{i} \in \text{alg}(Y_{1} \cup Y_{2} \cdots Y_{i}) \) such that \( u(t_{i}) = q_{i} \) and \( u(\hat{t}_{i}) = \hat{q}_{i} \). Define \( r : \mathbb{Q}[Y] \rightarrow \mathbb{Q}[Y] \) to be the polynomial homomorphism that sends \( y_i \mapsto t_i + \hat{t}_i \). Observe that \( r \) is a solvability witness for \( \overline{u}^{-1} [T] \); \( r \) is solvable by
construction, and for each \( y_i \) we have

\[
\overline{u}(y'_i - r(y_i)) = \overline{u}(y'_i - t_i - \hat{t}_i) = \overline{u}(y'_i) - \overline{u}(t_i) - \overline{u}(\hat{t}_i) = p'_i - q_i - \hat{q}_i \in T
\]

and so \( y'_i - r(y_i) \in \overline{u}^{-1}[T] \).

Next, we must show that \( R(T) \) is universal. Suppose that \( V \subseteq Q[Z, Z'] \) is a solvable transition ideal, and that \( v : Q[Z] \to Q[X] \) is a linear simulation from \( T \) to \( V \). We must show that there is a homomorphism \( \hat{v} : Q[Z] \to Q[Y] \) such that \( u \circ \hat{v} = v \) and \( \hat{v} \) is a simulation from \( \overline{u}^{-1}[T] \) to \( V \).

It is sufficient to show that for each \( z \in Z \), we have \( v(z) \in S^+(T) \): under this assumption, for each \( z \), \( v(z) \) can be written (uniquely) as a linear combination \( v(z) = a_1 p_1 + \cdots + a_n p_n = u(a_1 y_1 + \cdots + a_n y_n) \), and so we can define \( \hat{v} \) by \( \hat{v}(z) = v(z) \) of \( a_1 y_1 + \cdots + a_n y_n \). It follows that

\[
\overline{v}^{-1}[\overline{u}^{-1}[T]] = (\overline{u} \circ \overline{v})^{-1}[T] = \overline{v}^{-1}[T] \subseteq V,
\]

and so \( \hat{v} \) is a simulation from \( \overline{u}^{-1}[T] \) to \( V \).

Since \( U \) is solvable, there is a partition \( Z_1, \ldots, Z_k \) of \( Z \) and a polynomial homomorphism \( g : Q[Z] \to Q[Z] \) such that for each \( i \) and each \( z \in Z_i \), \( g(z) \) can be written as the sum of a linear term in \( Z_i \) and a polynomial in \( Z_{i+1}, \ldots, Z_{i-1} \). For each \( i \in \{0, \ldots, n\} \), let \( Z_{<k} = \bigcup_{j=1}^{i} Z_j \). We show by induction on \( k \) that for all \( z \in Z_{<k} \), \( v(z) \in S_k(T) \). The base case \( k = 0 \) is trivial: \( Z_{<0} \) is empty. For the inductive step, suppose \( s(Z_{<k}) \subseteq S_k(T) \), and prove \( s(Z_{<k+1}) \subseteq S_{k+1}(T) \). Let \( z \in Z_{<k+1} \). Since \( g \) is a witness to solvability of \( V \), we have \( z' - g(z) \in V \), and since \( v \) is a simulation from \( T \) to \( V \), we have \( v(z' - g(z)) \in T \). Since \( g \) is solvable, we have \( g(z) \in \text{span} \ (Z_{<k+1}) + \text{alg} \ (V(Z_{<k})) \), and so \( v(g(z)) \in \text{span} \ (v(Z_{<k+1})) + \text{alg} \ (v(V(Z_{<k}))) \). Since \( v(z' - g(z)) = v(z)' - v(g(z)) \in T \) and \( v(g(z)) \in \text{span} \ (v(Z_{<k+1})) + \text{alg} \ (v(V(Z_{<k}))) \), we have \( v(z) \in \text{Det} \ (T, \text{span} \ (v(Z_{<k+1})) + \text{alg} \ (v(V(Z_{<k})))) \), and so by the induction hypothesis and monotonicity of \( \text{Det} \) we have \( v(z) \in \text{Det} \ (T, \text{span} \ (v(Z_{<k+1})) + S_k(T)) \). Since this holds for all \( z \in Z_{<k+1} \), we have \( \text{span} \ (v(Z_{<k+1})) \subseteq \text{Det} \ (T, \text{span} \ (v(Z_{<k+1})) + S_k(T)) \), and since \( S_{k+1}(T) \) is defined to be greatest fixedpoint of \( S \mapsto \text{Det} \ (T, S + S_k(T)) \), we have \( \text{span} \ (v(Z_{<k+1})) \subseteq S_{k+1}(T) \), and so for all \( z \in Z_{<k+1} \), \( v(z) \in S_{k+1}(T) \).

The proof of Thm. 6.5 is constructive and gives rise to a procedure for computing solvable reflections of transition ideals, depicted in Alg. 14. The procedure relies on a subroutine for computing \( \text{Det} \), which is given in Alg. 13 (the correctness of which is Lem. 6.6). Otherwise, the procedure follows the steps of the proof directly.

**Lemma 6.6.** Let \( T \subseteq Q[X, X'] \), \( V \subseteq Q[X] \), and \( Q \subseteq Q[X] \) (for some set of variables \( X \)). Then Alg. 13 computes a set of polynomials \( D \) such that \( \text{span} \ (D) = \text{Det} \ (T, \text{span} (V) + \text{alg} (Q)) \).
The algorithm presented in § 6.5 reveals that a weaker condition than solvability is sufficient in order to obtain a more powerful algorithm for generating polynomial invariants for loops. In this section, we answer that question in the affirmative.

6.4.2 Ultimately Solvable Reflections

The algorithm presented in § 6.5 reveals that a weaker condition than solvability is sufficient in order to compute the Kleene closure of a transition ideal, namely the transition needs to be ultimately solvable. This raises the question of whether it’s possible to compute ultimately solvable reflections of arbitrary transition ideals, and thereby obtain a more powerful algorithm for generating polynomial invariants for loops. In this section, we answer that question in the affirmative.

Algorithm 13: Computation of determined functionals

Input: Finite sets of polynomials $T \subseteq \mathbb{Q}[X, X']$, $V \subseteq \mathbb{Q}[X]$, and $Q \subseteq \mathbb{Q}[X]$.
Output: Set of polynomials $D$ such that $\text{span}(D) = \text{Det}(\langle T \rangle, \text{span}(V) + \text{alg}(Q))$.

1. $Y \leftarrow \{y_v : v \in V\};$ /* Introduce a variable $y_v$ for each generator $v \in V$ */
2. $Z \leftarrow \{z_q : q \in Q\};$ /* Introduce a variable $z_q$ for each generator $q \in Q$ */
3. Let $f : \mathbb{Q}[X, Y, Z] \rightarrow \mathbb{Q}[X, X']$ be the map that sends $x \mapsto x'$, $y_v \mapsto v$, $z_q \mapsto q$.
4. $F \leftarrow \text{inv.image}(f, T);$  
5. $G \leftarrow \text{Gröbner basis for } F \text{ w.r.t. } \ll_{X,Y} \text{ for some graded order } \ll;$
6. $D \leftarrow \emptyset;
7. \textbf{foreach } g \in G \text{ do}
8. \hspace{1em} \textbf{if } g = p + q + r \text{ for some non-zero } p \in \text{span}(X), q \in \text{span}(Y), \text{ and } r \in \mathbb{Q}[Z] \text{ then}
9. \hspace{2em} D \leftarrow D \cup \{p\};
10. \textbf{return } D$


First we prove $D \subseteq \text{Det}(\langle T \rangle, \text{span}(V) + \text{alg}(Q))$. Let $p \in D$. Then there is some $q \in \text{span}(Y)$ and $r \in \mathbb{Q}[Z]$ such that $p - q - r \in G$. It follows that $f(p - q - r) = p' - f(q) - f(r) \in \langle T \rangle$. Finally, observe that $f(q) \subseteq f(\text{span}(Y)) = \text{span}(V)$, and that $f(r) \subseteq f(\mathbb{Q}[Z]) = \text{alg}(Q)$, and thus $p \subseteq \text{Det}(\langle T \rangle, \text{span}(V) + \text{alg}(Q))$.

Next we prove $\text{Det}(\langle T \rangle, \text{span}(V) + \text{alg}(Q)) \subseteq \text{span}(D)$. Let $p \in \text{Det}(\langle T \rangle, \text{span}(V) + \text{alg}(Q))$, and suppose that $p$ is non-zero. Then there is some $v \in \text{span}(V)$ and some $q \in \text{alg}(Q)$ such that $p' - v - q \in \langle T \rangle$. Since $v \in \text{span}(V) = f(\text{span}(Y))$, there is some $\hat{v} \in \text{span}(Y)$ such that $f(\hat{v}) = v$. Since $q \in \text{alg}(Q) = f(\mathbb{Q}[Z])$, there is some polynomial $\hat{q} \in \mathbb{Q}[Z]$ such that $f(\hat{q}) = q$. Since $\langle T \rangle$ contains $p' - v - q$, $\langle F \rangle = \langle \text{inv.image}(f, T) \rangle = f^{-1}[T]$ must contain $p' - v - q$.

Let $G = \{g_1, \ldots, g_n\}$ be a Gröbner basis for $\langle F \rangle$ with respect to $\ll_{X,Y}$, and let $I$ be the subset of $\{1, \ldots, n\}$ such that $g_i = p_i + q_i + r_i$ for some non-zero linear $p_i \in \mathbb{Q}[X]$, linear $q_i \in \mathbb{Q}[Y]$, and $r_i \in \mathbb{Q}[Z]$, so that $D = \{p_i : i \in I\}$. Since $G$ is a Gröbner basis for $(F)$ with respect to $\ll_{X,Y}$ and $p - \hat{v} - \hat{q} \in \langle F \rangle$, we have $p - \hat{v} - \hat{q} = \sum_{i=0}^{n} c_ig_i$, for some $c_1, \ldots, c_n \in \mathbb{Q}[X, Y, Z]$ with $\text{LM}(c_ig_i) \ll_{X,Y} \text{LM}(p - \hat{v} - \hat{q})$ for each $i$. Since $\ll_{X,Y}$ is using a graded order and $p - \hat{v} - \hat{q}$ is linear in $X$ and $Y$, so must be each $c_ig_i$, and so $c_i \in \mathbb{Q}$ for each $i \in I$. It follows that $p \subseteq \text{span}((p_i : i \in I)) = \text{span}(D)$. \hfill \qed
Let $T$ be a transition ideal. Define a sequence $(t_0, T_0), (t_1, T_1), \ldots$ where $t_0$ is the identity function, $T_0 = T$, and for each $i \geq 0$, $t_{i+1}$ is the simulation component of the solvable reflection of $T_i + \text{dom}^* (T_i)$, and $T_{i+1}$ is $t_{i+1}^{-1} [T_i]$. Since for all $i$, if $t_i$ is not invertible then the dimension of $T_i$ is strictly smaller than $T_{i-1}$, there must be some first index $n$ such that $t_n$ is invertible. Define $R^*(T) \triangleq (u^*, \overline{u}^{-1}[T])$, where $u^* = t_1 \circ \cdots \circ t_{n-1}$.

Lemma 6.7. Let $T$ be a transition ideal. Then $R^*(T)$ is an ultimately solvable reflection of $T$.

Proof. Observe that $\overline{u}^{-1}[T]$ is ultimately solvable, since (by the definition of $u^*$), the solvable reflection of $\overline{u}^{-1}[T] + \text{dom}^* (\overline{u}^{-1}[T])$ is isomorphic to $\overline{u}^{-1}[T] + \text{dom}^* (\overline{u}^{-1}[T])$.

Towards universality, we show that for all $i$, we have

1. $t_0 \circ \cdots \circ t_i$ is a simulation $T \to T_i$.

2. For any ultimately solvable $V$ and simulation $v : T \to V$, there is a unique simulation $v_i : T_i \to V$ such that $v = t_0 \circ \cdots \circ t_i \circ v_i$.

by induction on $i$. The base case $i = 0$ is trivial. For the induction step, suppose that (1) and (2) hold for $i$. By definition, $t_{i+1}$ is the simulation component of a solvable reflection of $R(T_i + \text{dom}^*(T_i))$, and $T_{i+1}$ is $t_{i+1}^{-1}[T_i]$. Thus, $t_{i+1}$ is a simulation from $T_i$ to $T_{i+1}$. Since $t_0 \circ \cdots \circ t_i$ is a simulation $T \to T_i$, it follows that (1) the composition $t_0 \circ \cdots \circ t_i \circ t_{i+1}$ is a simulation from $T$ to $T_{i+1}$. Next, suppose that $V$ is ultimately solvable and $v : T \to V$ is a simulation. By the induction hypothesis, there is a unique simulation $v_i : T_i \to V$ such that $v = t_0 \circ \cdots \circ t_i \circ v_i$. It follows that $v_i$ is also a simulation from $T_i + \text{dom}^*(T_i)$ to $V + \text{dom}^*(V)$.
Since $V$ is ultimately solvable, $V + \text{dom}^*(V)$ is solvable. Since there is some $W_{i+1}$ such that $(t_{i+1}, W_{i+1})$ is a solvable reflection of $T_i + \text{dom}^*(T_i)$, $V + \text{dom}^*(V)$ is solvable, and $v_i : T_i + \text{dom}^*(T_i) \to V + \text{dom}^*(V)$ is a simulation, there is a unique simulation $v_{i+1}$ from $W_{i+1}$ to $V + \text{dom}^*(V)$ such that $t_{i+1} \circ v_{i+1} = v_i$. We have

\[ V \subseteq \overline{\mathbf{w}_{i+1}^{-1}}[T_i] \]

\[ v_i \text{ a simulation } T_i \to V \]

\[ = \overline{t_{i+1} \circ v_{i+1}^{-1}}[T_i] \]

\[ t_{i+1} \circ v_{i+1} = v_i \]

\[ = \overline{v_{i+1}^{-1}}[T_i] \]

\[ = \overline{v_{i+1}^{-1}}[T_i] \]

and thus $v_{i+1}$ is a simulation from $T_{i+1}$ to $V$. For uniqueness, suppose $v_{i+1}'$ is a simulation from $T_{i+1}$ to $V$ with $v = t_0 \circ \cdots \circ t_{i+1} \circ v_{i+1}'$. Since $t_{i+1} \circ v_{i+1}'$ is a simulation $T_i \to V$ with $t_0 \circ \cdots \circ t_{i+1} \circ v_{i+1}' = v$, and $v_i$ is the unique such simulation, we have $v_i = t_{i+1} \circ v_{i+1}'$. Since $v_{i+1}$ is unique such that $v_i = t_{i+1} \circ v_{i+1}$, we have $v_{i+1} = v_{i+1}'$.

Finally we show that $\langle \mathbf{u}^*, \overline{\mathbf{w}^{-1}}[T] \rangle$ is universal. Let $V$ be ultimately solvable, and let $v : T \to V$ be a simulation. By (2), there is a unique simulation $v_{n-1} : T_{n-1} \to V$ such that $v = t_0 \circ \cdots \circ t_{n-1} \circ v_i = \mathbf{u}^* \circ v_i$. Since $T_{n-1} = \overline{\mathbf{u}^{-1}}[T]$, we have the result.

\[ \square \]

### 6.4.3 Polynomial Simulations

Here, we consider a generalization of our definition of (ultimately) solvable reflections, in which the simulation from a transition ideal to its reflection is a polynomial map rather than a linear map.

Let $X$ be a set of variables and let $d \in \mathbb{N}$ be a fixed degree bound. Let $X^{\leq d}$ be the set of monomials of degree at most $d$ (excluding 1), let $Y$ be a set of variables of cardinality equal to that of $X^{\leq d}$, and let $f_{X,d} : Y \to X^{\leq d}$ be a bijection. Observe that if $Z$ is a set of variables and $g : \mathbb{Q}[Z] \to \mathbb{Q}[X]$ is a polynomial homomorphism of degree at most $d$, then there is unique linear polynomial homomorphism $\hat{g}$ such that $g = \hat{g} \circ f_{X,d}$. As a result, we can reduce the problem of computing reflections with respect to bounded-degree polynomial simulations to the problem of computing reflections with respect to linear simulations:

**Lemma 6.8.** Let $T \subseteq \mathbb{Q}[X, X']$ be a transition ideal, and let $d \in \mathbb{N}$ be a fixed degree bound. Suppose that $\langle \mathbf{u}, \mathbf{U} \rangle$ is an (ultimately) solvable reflection of $\overline{\mathbf{u}_{X,d}^{-1}}[T]$. Then $\langle \mathbf{u} \circ f_{X,d}, \mathbf{U} \rangle$ is an (ultimately) solvable reflection of $T$ with respect to degree-$d$ simulations, in the sense that (1) $u \circ f_{X,d}$ has degree at most $d$, (2) $\mathbf{U}$ is solvable, and (3) for solvable transition ideal $V$ and simulation $v$ from $T$ to $V$ of degree at most $d$, there is a unique linear simulation $\hat{v}$ such that $v = \hat{v} \circ u \circ f_{X,d}$.
The problem is to find a basis for the set of polynomials $v$. Let $I$ be a solvable abstraction of $Q$. In this case we have $I = f_\mathcal{A}(X)$. Suppose $V \subseteq Q[Z, Z']$ is an (ultimately) solvable transition ideal, and that $v : T \rightarrow V$ is a degree-$d$ simulation. Then there exists a unique linear simulation $\hat{v} : f_{X,d}^{-1}[T] \rightarrow V$ such that $v = \hat{v} \circ f_{X,d}$. Since $U$ is an (ultimately) solvable reflection of $f_{X,d}^{-1}[T]$, $\hat{v}$ is a linear simulation, and $V$ is (ultimately) solvable, there is a unique linear simulation $w : U \rightarrow V$ such that $w \circ u = \hat{v}$. Finally, observe that $w \circ (u \circ f_{X,d}) = (w \circ u) \circ f_{X,d} = \hat{v} \circ f_{X,d} = v$. 

\[ \square \]

6.5 Kleene Closure of Solvable Transition Ideals

In this section we describe how to compute $T^* = \bigcap_{i=0}^\infty T^i$ when $T$ is either a solvable and ultimately solvable transition ideal. In doing so we introduce a sub-problem of potential independent interest. The sub-problem asks how to find the set of rational polynomials that evaluate to 0 for every position in a $Q$-algebra sequence defined by a solvable polynomial map.

6.5.1 Finding the Relations of a Solvable Map over a $Q$-Algebra

Problem 6.9. Let $\mathcal{A}$ be a $Q$-algebra, let $X$ be a finite set of variables, and let $v \in A^X$. Given a solvable map $f : Q[X] \rightarrow Q[X]$ and basis $I$ such that $\langle I \rangle = \mathcal{I}_\mathcal{A}([v])$, find a basis for $\mathcal{I}_\mathcal{A}(\{ f_i^\mathcal{A}(v) : i \in \mathbb{N} \}) \subseteq Q[X]$.

Intuitively, the solvable map $f$ in Problem 6.9 defines a sequence, $(v, f_1^\mathcal{A}(v), f_2^\mathcal{A}(v), \ldots)$. The goal of the problem is to find a basis for the set of polynomials $p \in Q[X]$ such that

\[
(p^\mathcal{A}(v), p^\mathcal{A}(f_1^\mathcal{A}(v)), p^\mathcal{A}(f_2^\mathcal{A}(v)), \ldots) = (0, 0, 0, \ldots).
\]

The purpose of the ideal $\langle I \rangle = \mathcal{I}_\mathcal{A}([v])$ is to give the set of polynomial relations of the first element of the sequence, $v$. In the case of Problem 6.9 we take $\mathcal{I}_\mathcal{A}([v])$ as a given to encode the relevant information of $\mathcal{A}$ and $v$.

Example 6.10. Let $X = \{x, y\}$, $A = Q[w]/\langle w^2 - 3 \rangle$, and $v = (x \mapsto w + \langle w^2 - 3 \rangle, y \mapsto 2w + 3 + \langle w^2 - 3 \rangle)$. For this case we have $\mathcal{I}_\mathcal{A}([v]) = \langle 2x - y + 3, x^2 - 3 \rangle$. Let $f : Q[x, y] \rightarrow Q[x, y]$ be the polynomial homomorphism defined by $f(x) = 2y$ and $f(y) = 2x$. Then $f$ is a solvable map that defines the following sequence over $A^X$:

\[
\begin{pmatrix}
\{ x \mapsto w + \langle w^2 - 3 \rangle \\ y \mapsto 2w + 3 + \langle w^2 - 3 \rangle \} & \{ x \mapsto 4w + 6 + \langle w^2 - 3 \rangle \\ y \mapsto 2w + \langle w^2 - 3 \rangle \} & \{ x \mapsto 4w + \langle w^2 - 3 \rangle \\ y \mapsto 8w + 12 + \langle w^2 - 3 \rangle \} & \ldots
\end{pmatrix}
\]
It can readily be verified that \( p(x, y) = x^2 - 4xy + y^2 \in I_\text{A} \left( \left\{ f_i^j(v) : i \in \mathbb{N} \right\} \right) \). For instance, take the second (indexing from 0) valuation of the sequence, \( v_2 = \{ x \mapsto 4w + \langle w^2 - 3 \rangle, y \mapsto 8w + 12 + \langle w^2 - 3 \rangle \} \in A^X \). Then \( p^A(v_2) = (4w + \langle w^2 - 3 \rangle)^2 - 4(4w + \langle w^2 - 3 \rangle)(8w + 12 + \langle w^2 - 3 \rangle) + (8w + 12 + \langle w^2 - 3 \rangle)^2 = 144 - 48w^2 + \langle w^2 - 3 \rangle = 144 - 48(3) + \langle w^2 - 3 \rangle = 0 + \langle w^2 - 3 \rangle \).

We can also view Problem 6.9 as defining \( |X| \) c-finite sequences of \( A \) elements, where each sequence is the trajectory of a particular variable. If we take the special case where \( A = \mathbb{Q} \) then the goal is to find the algebraic relations [82] of the \( |X| \) sequences.

**Definition 6.11.** (Modification of Kauers and Zimmermann [82]) Let \( k \) be a field and \( K \) a (commutative) \( k \)-algebra. An algebraic relation over \( k \) among \( a_1, \ldots, a_m \in K \) is an element of the kernel of the \( k \)-algebra homomorphism \( \varphi : k[x_1, \ldots, x_m] \rightarrow K \) that maps \( x_j \) to \( a_j \).

Kauers and Zimmermann [82, Algorithm 2] presents a method to find the set of algebraic relations over \( \mathbb{Q} \) for the case of a set of c-finite sequences over the \( \mathbb{Q} \)-algebra, \( \mathbb{Q}^\omega \); consequently, solving Problem 6.9 for the case where \( A = \mathbb{Q} \). In this section we show how the method of Kauers and Zimmermann [82] can be utilized to solve Problem 6.9 for the case of an arbitrary \( \mathbb{Q} \)-algebra \( A \). First we briefly review the method of Kauers and Zimmermann [82].

At a high-level, the method of Kauers and Zimmermann [82] is, given c-finite sequences \( \{a_1(n)\}_{n=0}^\infty, \ldots, \{a_k(n)\}_{n=0}^\infty \in \mathbb{Q}^\omega \), perform the following:

1. Compute closed-form solutions of each sequence as \( a_i(n) = \sum_{j=1}^{m} p_{ij}(n)\Theta_j^n \) for polynomials \( p_{ij} \in \hat{Q}[n] \) and values \( \Theta_1, \ldots, \Theta_m \in \hat{Q} \).
2. Using the algorithm of Ge [60] compute a basis \( J \subseteq \mathbb{Q}[y_0, y_1, \ldots, y_m] \) for the ideal of algebraic relations over \( \mathbb{Q} \) of the sequences \( \{n\}_{n=0}^\infty, (\Theta_1^n)_{n=0}^\infty, \ldots, (\Theta_m^n)_{n=0}^\infty \in \hat{Q}^\omega \). That is, \( p(y_0, y_1, \ldots, y_m) \in \langle J \rangle \) if and only if \( p(n, \Theta_1^n, \ldots, \Theta_m^n) = 0 \) for \( n \in \mathbb{N} \).
3. Let \( B = \left\{ x_i - \sum_{j=1}^{m} p_{ij}(y_0)y_j : 1 \leq i \leq k \right\} \). Using Gröbner basis elimination techniques compute the ideal \( \langle H \rangle = \langle (B \cup J) \rangle \cap \mathbb{Q}[x_1, \ldots, x_k] \).

The resulting ideal, \( \langle H \rangle \), is the ideal of algebraic relations over \( \mathbb{Q} \) of the sequences \( \{a_1(n)\}_{n=0}^\infty, \ldots, \{a_k(n)\}_{n=0}^\infty \in \mathbb{Q}^\omega \). That is, \( \langle H \rangle \) has the property that \( p \in \langle H \rangle \) if and only if \( p \in \mathbb{Q}[x_1, \ldots, x_k] \) and \( p(a_1(n), \ldots, a_k(n)) = 0 \) for all \( n \in \mathbb{N} \).

**Remark 6.12.** It should be noted that the method presented above as well as in Kauers and Zimmermann [82] only works when the characteristic polynomials of the input recurrences do not have 0 as a root. Kauers and Zimmermann
Algorithm 15: Solve solvable map

**Input:** $\mathbb{Q}$-algebra $A$, valuation $v \in A^X$, solvable map $f : \mathbb{Q}[X] \to \mathbb{Q}[X]$, and basis $I$ for the ideal $I_A (\langle v \rangle)$.

**Output:** A basis for the ideal $I_A (\langle f^n A (v) \rangle : n \in \mathbb{N})$

1. $m \leftarrow |X|$;
2. For each $x_i \in X$ solve the trajectory of $x_i$ as $f^n(x_i) = \sum_{i=1}^{m} (z_{ij}(n)) + \sum_{i=1}^{m} p_{ijk}(n)\Theta_k^n v(x_i)$;
3. For $1 \leq i \leq m$ and $1 \leq j \leq m$ let $a_{ij}(n) = \sum_{k=1}^{m} p_{ijk}(n)\Theta_k^n \in \mathbb{Q}^\omega$;
4. Using Kauers and Zimmermann [82, Algorithm 2] let $J' \subseteq \mathbb{Q} [y_{11}, \ldots, y_{1m}, \ldots, y_{m1}, \ldots, y_{mm}]$ be the ideal of algebraic relations of $\{a_{11}(n + m)\}_n, \ldots, (a_{m1}(n + m))_n, \ldots, (a_{mm}(n + m))_n$;
5. $J \leftarrow$ Basis for the ideal $J' \cap \bigcap_{i=0}^{m-1} ((y_{ij} - (z_{ij}(n)) + \sum_{k=1}^{m} p_{ijk}(n)\Theta_k^n) : 1 \leq i \leq m, 1 \leq j \leq m)$;
6. return $\text{inv.image}(f, I \cup J)$, where $f$ is the map that sends $x_j \mapsto x_1 y_{ij} + \cdots + x_m y_{mj}$.

[82] notes this, and correctly states that this situation can be handled with a pre-processing step. In this section, we are more explicit on how to handle 0 roots.

The main observation that leads to our method for a general $\mathbb{Q}$-algebra $A$ is that the only “new” algebraic relations over $\mathbb{Q}$ for the $A$ sequences must come from the relations of the initial values of the sequence (the ideal $(I) = I_A (\langle v \rangle)$ in the statement of Problem 6.9). This observation leads to our method for solving Problem 6.9, which we present as Alg. 15.

Alg. 15 begins by writing the trajectory of each variable $x_i$ as the closed-form solution $\sum_{i=1}^{m} (z_{ij}(n)) + \sum_{i=1}^{m} p_{ijk}(n)\Theta_k^n v(x_i)$, which is a sum-of-products of the form $a_{ij}(n)v(x_i) + \cdots + a_{mj}(n)v(x_m)$ with each $a_{ij}(n)$ an $\mathbb{Q}^\omega$. Moreover, each $a_{ij}(n)$ is a rational c-finite sequence. However, in general, the sequences $\{a_{ij}(n)\}_n$ might have 0 as a root of their characteristic polynomials—hence the presence of the term $z_{ij}(n)$. These $z_{ij}(n)$ terms mean that the method of [82] cannot be directly applied. The following lemma shows how we can handle this general case.

**Lemma 6.13.** Let $\{a_1(n)\}_n, \ldots, (a_m(n))_n \in \mathbb{Q}^\omega$ such that $a_{1}(n) = z_{1}(n) + a_1(n)$ for $(z_{1}(n))_n$, $a_i(n)_n \in \mathbb{Q}^\omega$. Furthermore, suppose that there exists some $d \in \mathbb{N}$ such that for all $1 \leq i \leq m$, $z_{i}(n) = 0$ for $n \geq d$. Let $J' \subseteq \mathbb{Q} [y_{11}, \ldots, y_{1m}]$ be the ideal of algebraic relations over $\mathbb{Q}$ of $(a_1(n + d))_n, \ldots, (a_m(n + d))_n$. Then $J = J' \cap \bigcap_{i=0}^{m-1} ((y_{ij} - (z_{ij}(n) + a_i(n)) : 1 \leq i \leq m)$ is the ideal of algebraic relations of $(a_1(n))_n, \ldots, (a_m(n))_n$.

**Proof.** We need to show $p \in J$ if and only if $p(a_1(n), \ldots, a_m(n)) = 0$ for $n \in \mathbb{N}$. Let $J'' = \bigcap_{i=0}^{m-1} ((y_{ij} - (z_{ij}(n) + a_i(n)) : 1 \leq i \leq m)$.

$(\implies)$ Let $p \in J$. Then $p \in J'$ and $p \in J''$. Because $p \in J''$ it must be the case that $p = \sum_{i=1}^{m} g_{in}(y_{i} - (z_{i}(n) + a_i(n)))$ for $0 \leq n \leq d - 1$ and some polynomials $g_{in} \in \mathbb{Q}[Y]$. Therefore, $p(a_1(n), \ldots, a_m(n)) = \sum_{i=1}^{m} g_{in}(a_1(n) - (z_{1}(n) + a_1(n))) = 0$ for $0 \leq n \leq d$. For $n \geq d$ we have $a_i(n) = a_i(n)$. Equivalently $a_i(n + d) = a_i(n + d)$ for $n \geq 0$. Because $p \in J'$, $p(a_1(n + d), \ldots, a_m(n + d)) = 0$ for any $n \in \mathbb{N}$.
Therefore, $p(a_1'(n+d), \ldots, a_m'(n+d)) = 0$ for $n \in \mathbb{N}$ and $p(a_1'(n), \ldots, a_m'(n)) = 0$ for $n \geq d$. Thus, $p(a_1'(n), \ldots, a_m'(n)) = 0$ for $n \in \mathbb{N}$.

( $\Leftarrow$ ) Suppose $p(a_1'(n), \ldots, a_m'(n)) = 0$ for $n \in \mathbb{N}$. Then $p(a_1'(n+d), \ldots, a_m'(n+d)) = p(a_1(n+d), \ldots, a_m(n+d)) = 0$ for $n \in \mathbb{N}$, so $p \in \mathcal{P}$. For $0 \leq n < d$, $a_i'(n) = z_i(n) + a_i(n)$, so $p(a_1'(n), \ldots, a_m'(n)) = p(z_1(n) + a_1(n), \ldots, z_m(n) + a_m(n)) = 0$. Let $p(y_1, \ldots, y_m)$ be reduced relative to a Gröbner basis for

$$\langle \langle y_i - (z_i(n) + a_i(n)) : 1 \leq i \leq m \rangle \rangle$$

for some $0 \leq n < d$. Then $p = \sum_{i=1}^{m} g_{in}(y_i - (z_i(n) + a_i(n))) + r$ for some $g_{in}$'s $\in \mathbb{Q}[Y]$ and $r \in \mathbb{Q}$. Thus, we have

$$0 = p(z_1(n) + a_1(n), \ldots, z_m(n) + a_m(n)) = \sum_{i=1}^{m} g_{in}(z_i(n) + a_i(n) - (z_i(n) + a_i(n))) + r = r.$$

Thus, $p \in \langle \langle y_i - (z_i(n) + a_i(n)) : 1 \leq i \leq m \rangle \rangle$ for $0 \leq n < d$. Therefore, $p \in \mathcal{P}''$.

Because each $\{a_{ij}'(n)\}_{n=0}^{\infty}$ is c-finite, each $z_{ij}(n)$ has the property that $z_{ij}(n) = 0$ for $n \geq d$, where $d$ is the maximal order of the sequences $\{a_{ij}'(n)\}_{n=0}^{\infty}$. In the case of Alg. 15 we have $d \leq m$. Thus, each $z_{ij}(n) = 0$ for $n \geq m$. Lem. 6.13 shows that $\mathcal{J}$ in Alg. 15 is a basis for the ideal of algebraic relations over $\mathbb{Q}$ of the $\{a_{ij}'(n)\}_{n=0}^{\infty}$ sequences.

We use the next two lemmas to establish the needed property of the return value of Alg. 15 and thus show that Alg. 15 is correct. The desired result of Alg. 15 is the ideal of algebraic relations over $\mathbb{Q}$ of the $A$ sequences, $f^n_A(x_j)$ defined in line (2) of Alg. 15. These sequences are defined as a sum-of-products of rational sequences and constant sequences of the evaluations $\psi(x_i)$. The next lemma (Lem. 6.14) shows that if we want to find the algebraic relations over $\mathbb{Q}$ among arbitrary rational sequences lifted to $A^\omega$ and constant valuations $\{(\psi(x_j))_{i=0}^{\infty} : x_j \in X\}$, it is sufficient to consider the ideals of algebraic relations over $\mathbb{Q}$ among $\{(\psi(x_j))_{i=0}^{\infty} : x_j \in X\}$ and the lifted rational sequences separately. This is what makes Alg. 15 possible: we are given the algebraic relations over $\mathbb{Q}$ among $\{(\psi(x_j))_{i=0}^{\infty} : x_j \in X\}$ as input, and we can calculate the algebraic relations over $\mathbb{Q}$ among the c-finite sequences defined in line (2) using the algorithm of Kauers and Zimmermann [82]. The second lemma (Lem. 6.15) then applies Lem. 6.14 to the specific form of the $\{f^n_A(x_j)\}_{i=0}^{\infty}$ sequences defined in Alg. 15 to establish the correctness of the algorithm.

**Lemma 6.14.** Let $\{a_1(i)\}_{i=0}^{\infty}, \ldots, \{a_m(i)\}_{i=0}^{\infty} \in \mathbb{Q}^\omega$ and let $J \subseteq \mathbb{Q}[y_1, \ldots, y_m]$ be the ideal of algebraic relations over $\mathbb{Q}$ among $\{a_1(i)\}_{i=0}^{\infty}, \ldots, \{a_m(i)\}_{i=0}^{\infty}$. Let $A$ be a $\mathbb{Q}$-algebra with additive unit $0_A$ and multiplicative unit $1_A$. Let
Then \( \varphi(p) = (0_A)_{i=0}^{\infty} \) if and only if \( p \in (I \cup J) \subseteq \mathbb{Q}[X,Y] \).

Proof: (\( \Leftarrow \)) Suppose \( p \in (I \cup J) \). Thus, \( p = g + h \) for some \( g \in (I) \subseteq \mathbb{Q}[X,Y] \) and \( h \in (J) \subseteq \mathbb{Q}[X,Y] \).

Because \( g \in (I), g = \sum_{j=1}^{k_1} f_j g_j \) for some \( f_j \in Q[X,Y] \) and \( g_j \in I \). Because \( g_j \in I = I_A(\{v\}), g_j^A(v) = 0_A \) for each \( j \). Therefore, \( \varphi(g) = \sum_{j=1}^{k_1} \varphi(f_j) \{g_j^A(v)\}_{i=0}^{\infty} = \sum_{j=1}^{k_1} \varphi(f_j)(0_A)_{i=0}^{\infty} = (0_A)_{i=0}^{\infty}. \)

Now we show the same for \( h \). Because \( h \in (J), h = \sum_{j=1}^{k_2} f_j h_j \) for some \( f_j \in Q[X,Y] \) and \( h_j \in J \subseteq \mathbb{Q}[Y] \). Because \( h_j \in J, \varphi(h_j) = \{h_j(\{a_1(i), \ldots, a_m(i)\})(1_A)_{i=0}^{\infty} = (0_A)_{i=0}^{\infty} \) for each \( j \). Therefore, \( \varphi(h) = \sum_{j=1}^{k_2} \varphi(f_j)\varphi(h_j) = \sum_{j=1}^{k_2} \varphi(f_j)(0_A)_{i=0}^{\infty} = (0_A)_{i=0}^{\infty}. \)

Combining the previous paragraphs we have that, because \( \varphi \) is a homomorphism, \( \varphi(p) = \varphi(g) + \varphi(h) = (0_A)_{i=0}^{\infty} + (0_A)_{i=0}^{\infty} = (0_A)_{i=0}^{\infty}. \)

(\( \Rightarrow \)) Suppose \( \varphi(p) = (0_A)_{i=0}^{\infty}. \) Let \( r \) be \( p \) reduced by a Gröbner basis for \( (I \cup J) \) under the elimination order \( \ll_X. \) That is \( p = g_1 + g_1 + r \) for some \( g_1 \in I, g_1 \in J \) and for any other \( g' \in (I \cup J) \) and \( r' \in Q[X,Y] \) with \( p = g' + r', \text{LM}(r') \gg_X \text{LM}(r). \) Because \( \varphi \) is a homomorphism we have \( (0_A)_{i=0}^{\infty} = \varphi(p) = \varphi(g_1 + g_1 + r) = \varphi(g_1) + \varphi(g_1) + \varphi(r) = (0_A)_{i=0}^{\infty} + \varphi(r) = \varphi(r). \) We can write \( r \) by collecting \( X \) monomials with respect to the order \( \gg_X \) as follows

\[
r = m_1^X p_1^Y(y_1, \ldots, y_m) + \cdots + m_k^X p_k^Y(y_1, \ldots, y_m) + p_{k+1}^Y(y_1, \ldots, y_m)
\]

where each \( m_i^X \) is a distinct monomial of \( X \) variables with \( m_i^X \gg_X m_j^X \) for \( j = 2, \ldots, k, \) and each \( p_s^Y \) is a polynomial in \( Q[Y] \).

Observe that there exists some \( i \) such that \( p_i^Y(a_1(i), \ldots, a_m(i)) \neq 0. \) If not, \( \varphi(p_i^Y(y_1, \ldots, y_m)) = \{p_i^Y(a_1(i), \ldots, a_m(i))(1_A)_{i=0}^{\infty} = (0_A)_{i=0}^{\infty}. \) Thus, \( p_i^Y(a_1(i), \ldots, a_m(i)) = 0 \) for \( i \in N, \) and therefore \( p_i^Y \) is an algebraic relation over \( \mathbb{Q} \) among \( (a_1(i))(i=0, \ldots, a_m(i))(i=0, \ldots, \infty) \). Then by definition \( p_i^Y \in J. \) But this contradicts the property that \( r \) is reduced with respect to \( J. \) That is, if \( p_i^Y(y_1, \ldots, y_m) \in (J), \) then there exists a better \( r' \) that does not contain the monomial \( m_i^X. \) Therefore, there is some \( i \) such that \( p_i^Y(a_1(i), \ldots, a_m(i)) \neq 0. \)

Let \( i \) be such that \( p_i^Y(a_1(i), \ldots, a_m(i)) \neq 0. \) We have \( \varphi(r) = (0_A)_{i=0}^{\infty} \) by assumption, and so \( \varphi(r)_{i=0}^{\infty} = 0_A, \)
where

\[ \varphi(r) = (m_1^X)^A(v)p_1^Y(a_1(i), \ldots, a_m(i)) + \cdots + (m_k^X)^A(v)p_k^Y(a_1(i), \ldots, a_m(i)) + p_{k+1}^Y(a_1(i), \ldots, a_m(i)) \]

denotes the \(i\)th element of \(\varphi(r)\). Because \(\varphi(r)_i = 0_A\), we must have

\[ m_1^X p_1^Y(a_1(i), \ldots, a_m(i)) + \cdots + m_k^X p_k^Y(a_1(i), \ldots, a_m(i)) + p_{k+1}^Y(a_1(i), \ldots, a_m(i)) \in I \tag{6.3} \]

Denote the polynomial in Eqn. (6.3) as \(h\). We can rewrite \(r\) as follows

\[ r = \frac{p_1^Y(y_1, \ldots, y_m)}{p_1^Y(a_1(i), \ldots, a_m(i))} h + r' \]

For some \(r'\) containing \(x\) monomials \(m_2^X, \ldots, m_k^X\). However, this (nearly) contradicts the property that \(p\) is reduced with respect to \(I\). That is, \(p = (g_1 + g_j + \frac{p_1^Y(y_1, \ldots, y_m)}{p_1^Y(a_1(i), \ldots, a_m(i))} h) + r'\) with \(g_1 + g_j + \frac{p_1^Y(y_1, \ldots, y_m)}{p_1^Y(a_1(i), \ldots, a_m(i))} h \in (I \cup J)\). Moreover, \(LM(r) \gg_X LM(r')\), because \(r'\) does not contain the monomial \(m_1^X\) and \(m_j^X \gg_X m_1^X\) for \(j = 2, \ldots, k\).

The only way to avoid the contradiction is to have \(m_1^X\) be a constant. Therefore, because \(r\) is reduced with respect to an order that eliminates \(X\) variables and \(LM(r) \in Q[Y]\), we have \(r \in Q[Y]\).

Finally, because \(r \in Q[Y]\) and \(\varphi(r) = (0_A)_{i=0}^\infty\), \(r\) must be an algebraic relation over \(Q\) among \(\{a_1(i)\}_{i=0}^\infty, \ldots, \{a_m(i)\}_{i=0}^\infty\) and therefore \(r \in J\). However, because \(r\) must be reduced with respect to \(J\), \(r = 0\). Thus, \(p = g_1 + g_j\) with \(g_1 + g_j \in I + J\). Therefore, \(p \in (I \cup J)\). \(\square\)

**Lemma 6.15.** Let \(\{a_1m(i)\}_{i=0}^\infty, \ldots, \{a_1m(i)\}_{i=0}^\infty, \ldots, \{a_m(i)\}_{i=0}^\infty, \ldots, \{a_m(i)\}_{i=0}^\infty \in Q^w\) and let \(J \subseteq Q[y_1, \ldots, y_{1m}, \ldots, y_{1m}, \ldots, y_{mm}]\) be the ideal of algebraic relations over \(Q\) among these sequences. Let \(A\) be a \(Q\)-algebra, \(X = \{x_1, \ldots, x_m\}\), \(v \in A^X\), and \(I = I_A(\{v\}) \subseteq Q[X]\). For every \(i \in \mathbb{N}\) let \(w^i \in A^X\) be defined as \(w^i(x_j) = a_{ij}(i)v(x_1) + \cdots + a_{mj}(i)v(x_m)\). Let \(f : Q[X] \to Q[X,Y]\) be the polynomial homomorphism that maps \(x_j \mapsto x_1y_{1j} + \cdots + x_my_{mj}\) for all \(j\). Then \(I_A(\{w^i : i \in \mathbb{N}\}) = f^{-1}(I \cup J)\).

**Proof.** Let \(\varphi : Q[X,Y] \to A^w\) be defined as in Lem. 6.14. Let \(p(x_1, \ldots, x_m) \in Q[X]\). Then \(p(x_1y_{11} + \cdots + \)
\[ x_m y_{1m}, \ldots, x_1 y_{m1} + \cdots + x_m y_{mm} \in \mathbb{Q}[X,Y]. \] Note that

\[
\varphi(p(x_1 y_{11} + \cdots + x_m y_{1m}, \ldots, x_1 y_{m1} + \cdots + x_m y_{mm}))
= p(\varphi(y_{11})\varphi(x_1) + \cdots + \varphi(y_{1m})\varphi(x_m), \ldots, \varphi(y_{m1})\varphi(x_1) + \cdots + \varphi(y_{mm})\varphi(x_m))
= \langle p(a_{i1}(i)v(x_1) + \cdots + a_{im}(i)v(x_m), \ldots, a_{m1}(i)v(x_1) + \cdots + a_{mm}(i)v(x_m)) \rangle_{i=0}^\infty
= \{ p^A(w^i) \}_{i=0}^\infty.
\]

Therefore, by Lem. 6.14, \( \{ p^A(w^i) \}_{i=0}^\infty = 0 \) if and only if \( p(x_1 y_{11} + \cdots + x_m y_{1m}, \ldots, x_1 y_{m1} + \cdots + x_m y_{mm}) \in (I \cup J) \). So we have the following chain

\[
p \in I_A (\{ w^i \} : i \in \mathbb{N})
\iff \{ p^A(w^i) \}_{i=0}^\infty = 0
\iff p(x_1 y_{11} + \cdots + x_m y_{1m}, \ldots, x_1 y_{m1} + \cdots + x_m y_{mm}) \in (I \cup J)
\iff p(x_1, \ldots, x_m) \in f^{-1}(I \cup J)
\]

Combining Lems. 2.65, 6.13 and 6.15 establishes the correctness of Alg. 15.

**Theorem 6.16.** Alg. 15 solves Problem 6.9.

### 6.5.2 Closing Transition Ideals

The result of Alg. 15 produces a polynomial ideal that summarizes the algebraic relations over \( \mathbb{Q} \) of a solvable polynomial map. In this subsection, we show how the result of Alg. 15 can be used to compute \( T^* \) for a (ultimately) solvable transition ideal \( T \). Recall that every solvable transition ideal \( T \) comes with a solvability witness \( p \). The basic idea to compute \( T^* \) is to use Alg. 15 to summarize the algebraic relations over \( \mathbb{Q} \) among the sequences defined by \( p \). However, Problem 6.9, which is solved by Alg. 15, is defined over a \( \mathbb{Q} \)-algebra \( A \). Hence, in this subsection we work to motivate and explain that in order to calculate \( T^* \), the \( \mathbb{Q} \)-algebra we want to instantiate Problem 6.9 with is \( A = \mathbb{Q}[X,X']/\text{dom}^*(T) \).

Before we talk of \( \mathbb{Q} \)-algebras, we make a brief observation of the structure of solvable transition ideal. Intuitively, a solvable transition ideal can be broken into a domain part, containing only unprimed variables, and a transition part. The next lemma formalizes this point.
Lemma 6.17. Let $T \subseteq \mathbb{Q}[X, X']$ be a solvable transition ideal, and let $p : \mathbb{Q}[X] \to \mathbb{Q}[X]$ be a solvable witness for $T$. Then $T^n = \text{dom}(T^n) + \langle \{x'_i - p^n(x_i) : 1 \leq i \leq m\}\rangle$ for $1 \leq n$.

Proof. We prove the lemma by induction on $n$. Let $n = 1$. Because $T$ is solvable, $x'_i - p(x_i) \in T$ for $x_i \in X$. Thus, a Gröbner basis for $T$ with respect to $\ll_X$ is of the form

$$\langle f_1(X), \ldots, f_k(X), x'_i - p(x_i), \ldots, x'_m - p(x_m) \rangle = \text{dom}(T) + \langle \{x'_i - p(x_i) : 1 \leq i \leq m\}\rangle.$$

Now suppose the lemma holds for $n$. We wish to show the lemma holds for $T^{n+1}$.

$$T^{n+1} = T \cdot T^n = (T[X' \to X'']) + T^n[X \to X''] \cap \mathbb{Q}[X, X']$$

$$= (\text{dom}(T)[X' \to X'']) + \langle \{x''_i - p(x_i) : 1 \leq i \leq m\}\rangle$$

$$+ \text{dom}(T^n)[X \to X''] + \langle \{x'_i - p^n(x_i)'' : 1 \leq i \leq m\}\rangle \cap \mathbb{Q}[X, X']$$

$$= (\text{dom}(T) + \text{dom}(T^n)[X \to p(X)]) +$$

$$\langle \{x'_i - p^n(p(x_i)) : 1 \leq i \leq m\}\rangle + \langle \{x''_i - p(x_i) : 1 \leq i \leq m\}\rangle \cap \mathbb{Q}[X, X']$$

$$= \text{dom}(T^{n+1}) + \langle \{x'_i - p^{n+1}(x_i) : 1 \leq i \leq m\}\rangle.$$

\[\square\]

From Lem. 6.17, we see that the iterated behavior of the transition ideal is mostly captured by the iterated behavior of the polynomial witness; what is missing is the $\text{dom}(T^n)$ part. Note that $\text{dom}(T^n)$ is an ideal for each $n$. If we let $A$ be the $\mathbb{Q}$-algebra $\mathbb{Q}[X, X']/I$ for some ideal $I$ we can define a sequence like the one in Problem 6.9 that uses a solvable witness $p$ to transition not only variables but sets of polynomials with respect to $I$. This can be formalized in the language of Problem 6.9 for a solvable transition ideal $T \subseteq \mathbb{Q}[X, X']$ with solvability witness $p : \mathbb{Q}[X] \to \mathbb{Q}[X]$ as follows:

- Let $A = \mathbb{Q}[X, X']/I$.

- Let $\bar{p} : \mathbb{Q}[X, X'] \to \mathbb{Q}[X, X']$ extend $p$ as $\bar{p}(x_i) = x_i$ if $x_i \in X$, and $\bar{p}(x'_i) = (p(x_i))'$ if $x'_i \in X'$.

- Let $v \in A^{X \cup X'}$ be defined as $v(x_i) = x_i + I$ if $x_i \in X$ and $v(x'_i) = p(x_i) + I$ if $x'_i \in X'$.

The question then is what should we take for $I$. We want $I$ to be the polynomials not captured by the iteration of the solvable witness. Thus, from Lem. 6.17 these are the polynomials in $\text{dom}(T^n)$. For Problem 6.9 $I$ needs to be fixed, so we have two obvious options for $I$, the domain of $T$, $\text{dom}(T)$, or the invariant domain.
of \( T, \text{dom}^*(T) \). The next example shows what happens if we use the domain of \( T \) and why that gives us a mismatch for computing \( T^* \).

**Example 6.18.** Consider the solvable transition ideal \( T = \langle y' - y - z, z' - 3z, y - z - 1 \rangle \), with \( \text{dom}(T) = \langle y - z - 1 \rangle = I \). A solvable witness for \( T \) is \( \text{p}(y) = y + z \) and \( \text{p}(z) = 3z \). Let \( A = \mathbb{Q}[X, X']/L, \hat{\text{p}} : \mathbb{Q}[X, X'] \to \mathbb{Q}[X, X'] \) extend \( \text{p} \) as above, and let \( \nu \in A^{X∪X'} \) with \( \nu(y) = y + 1, \nu(z) = z + 1, \nu(y') = y + z + 1 \), and \( \nu(z') = 3z + 1 \). In the language of Problem 6.9, we have \( I_A(\{\nu^i\}) = T \). \( \hat{\text{p}}_A \) defines the following sequence\(^5\):

\[
\begin{align*}
\begin{cases}
y \mapsto y + 1 \\
z \mapsto z + 1 \\
y' \mapsto 2z + 1 + 1 \\
z' \mapsto 3z + 1
\end{cases},
\begin{cases}
y \mapsto y + 1 \\
z \mapsto z + 1 \\
y' \mapsto 5z + 1 + 1 \\
z' \mapsto 9z + 1
\end{cases},
\begin{cases}
y \mapsto y + 1 \\
z \mapsto z + 1 \\
y' \mapsto 14z + 1 + 1 \\
z' \mapsto 27z + 1
\end{cases},
\end{align*}
\]

Taking \( I_A(\{\nu^1\}) \) for each \( i \) of the above sequence is nearly \( T, T^2, T^3, \ldots \). However, \( T^2 = \langle y' - y - z, z' - 3z, y - 1, z \rangle = \langle y' - 1, z', y - 1, z \rangle \), but \( y' - 1 \notin I_A(\{\nu^2\}) \). Moreover, for every \( i \geq 2, I_A(\{\nu^i\}) \neq T^i \).

The essential problem with the previous example is that the domain of \( T \) is not stable for higher iterations of \( T^i \). If instead of \( I = \text{dom}(T) \) in the previous example we used \( I = \text{dom}^*(T) \) then we would have the equality \( I_A(\{\nu^i\}) = T^i \) for \( i \geq 2 \). This observation that equality can be recovered for some \( i \) by using the invariant domain is our key insight for computing \( T^* \). The issue of Ex. 6.18 is fixed using the reasoning in the following lemma.

**Lemma 6.19.** Let \( T \subseteq \mathbb{Q}[X, X'] \) be a solvable transition ideal with solvability witness \( \text{p} \). Let \( N \geq 1 \) be such that \( \text{dom}(T^N) = \text{dom}^*(T) \), and define \( I = \text{dom}^*(T) \). Let \( A \) be the \( \mathbb{Q} \)-algebra \( \mathbb{Q}[X, X']/L \). Let \( \hat{\text{p}} : \mathbb{Q}[X, X'] \to \mathbb{Q}[X, X'] \) be the homomorphism defined by \( \hat{\text{p}}(x_i) = x_i \) and \( \hat{\text{p}}(x'_i) = \text{p}(x_i)' \). Let \( \nu \in A^{X∪X'} \) be the valuation defined by \( \nu(x_i) = x_i + I \) and \( \nu(x'_i) = \text{p}(x_i) + I \). Then for \( 1 \leq N \leq n, T^n = I_A(\{\hat{\text{p}}_A^{-1}(\nu)\}) \). Furthermore, if \( 1 \leq n < N \) then \( T^n \subseteq I_A(\{\hat{\text{p}}_A^{-1}(\nu)\}) \).

**Proof.** Consider \( \hat{\text{p}}_A^{-1}(\nu) \) for \( n \geq 1 \). On a variable \( x_i \in X \) we have \( \hat{\text{p}}_A^{-1}(\nu)(x_i) = \nu(x_i) = x_i + I \). For a variable \( x'_i \in X' \) we have

\[
\hat{\text{p}}_A^{-1}(\nu)(x'_i) = (\hat{\text{p}}^{-1}(x'_i))^A(\nu) = \text{p}(\text{p}^{-1}(x_i)) + I = \text{p}^n(x_i) + I.
\]

More succinctly,

\[
\hat{\text{p}}_A^{-1}(\nu) = \begin{cases}
x_i \mapsto x_i + I \\
x'_i \mapsto \text{p}^n(x_i) + I
\end{cases}.
\]

\(^5\)Note that \( y + z + (y - z - 1) = 2z + 1 + (y - z - 1) \)
Note that if \( n \geq N \), \( \text{dom}(T^n) = \text{dom}(T^N) = \text{dom}^*(T) = I \). But if \( n < N \), \( \text{dom}(T^n) \subseteq \text{dom}^*(T) = I \). Thus, if \( n \geq N \) we have

\[
I_A \left( \{ \hat{p}_A^{n-1}(v) \} \right) = \langle \{ x_i - q(x) : q(x) \in x_i + I, 1 \leq i \leq m \} \rangle + \\
\langle \{ x'_i - q(x) : q(x) \in p^n(x_i) + I, 1 \leq i \leq m \} \rangle \\
= I + \langle \{ x'_i - p^n(x_i) : 1 \leq i \leq m \} \rangle \\
= \text{dom}(T^n) + \langle \{ x'_i - p^n(x_i) : 1 \leq i \leq m \} \rangle \\
= T^n
\]

The justification for the last step comes from Lem. 6.17. If \( n < N \) then the second to last equals becomes \( \supseteq \).

\[\square\]

**Corollary 6.20.** Lem. 6.19 also holds for ultimately solvable transition ideals.

**Proof.** Let \( T \) be an ultimately solvable transition ideal. Then by definition \( T + \text{dom}^*(T) \) is solvable. By Lem. 6.19 we have \((T + \text{dom}^*(T))^n \subseteq I_A \left( \{ \hat{p}_A^{n-1}(v) \} \right) \) for \( 1 \leq n < N \), and \((T + \text{dom}^*(T))^n = I_A \left( \{ \hat{p}_A^{n-1}(v) \} \right) \) for \( n \geq N \). It can readily be verified that \((T + \text{dom}^*(T))^n = T^n + \text{dom}^*(T)\) for any transition ideal \( T \). Furthermore, if \( n \geq N \) then \( T^n + \text{dom}^*(T) = T^n \). Therefore, for \( 1 \leq n < N \)

\[
T^n \subseteq T^n + \text{dom}^*(T) = (T + \text{dom}^*(T))^n \subseteq I_A \left( \{ \hat{p}_A^{n-1}(v) \} \right).
\]

For \( n \geq N \),

\[
T^n = T^n + \text{dom}^*(T) = (T + \text{dom}^*(T))^n = I_A \left( \{ \hat{p}_A^{n-1}(v) \} \right).
\]

\[\square\]

**Example 6.21.** Recall Ex. 6.18, but with \( I = \text{dom}(T^2) = \text{dom}(T^3) = \text{dom}^*(T) = \langle y - 1, z \rangle \). \( \hat{p}_A \) defines the following sequence:

\[
\left( \begin{array}{ccc}
  y & 1 + I \\
  z & 0 + I \\
  y' & 1 + I \\
  z' & 0 + I
\end{array} \right) \quad \left( \begin{array}{ccc}
  y & 1 + I \\
  z & 0 + I \\
  y' & 1 + I \\
  z' & 0 + I
\end{array} \right) \quad \left( \begin{array}{ccc}
  y & 1 + I \\
  z & 0 + I \\
  y' & 1 + I \\
  z' & 0 + I
\end{array} \right) \quad \ldots
\]

Informally, Lem. 6.19 states that the long-running relations of a solvable transition ideal \( T \) is exactly captured by \( I_A \left( \{ \hat{p}^i(v) : i \in \mathbb{N} \} \right) \). This is what Ex. 6.21 shows. However, we cannot just take the long-running relations of \( T \) as our summary \( T^* \). This is because for iterations before the invariant domain has stabilized we
do not have equality. However, the invariant domain must stabilize in a finite number of iterations, and can then be recovered via ideal intersection. This leads to the following theorem.

**Theorem 6.22.** Let $T$ be a (ultimately) solvable transition ideal $T \subseteq \mathbb{Q}[X, X']$ with solvability witness $p$. Let $N \geq 1$ be such that $\text{dom}(T^N) = \text{dom}^*(T) = I$. Let $\hat{p} : \mathbb{Q}[X, X'] \to \mathbb{Q}[X, X']$ be the homomorphism defined by $\hat{p}(x_i) = x_i$ and $\hat{p}(x'_i) = p(x'_i)$. Let $\nu \in A^X \cup X'$ be the valuation defined by $\nu(x_i) = x_i + 1$ and $\nu(x'_i) = p(x_i) + 1$. Then

$$\bigcap_{i=0}^{\infty} T_i = \left( \bigcap_{i=0}^{N-1} T_i \right) \cap (I_A (\{ \hat{p}^i (\nu) : i \in \mathbb{N} \})).$$

**Proof.**

$$\left( \bigcap_{i=0}^{N-1} T_i \right) \cap (I_A (\{ \hat{p}^i (\nu) : i \in \mathbb{N} \})) = \left( \bigcap_{i=0}^{N-1} T_i \right) \cap \left( \bigcap_{i=1}^{\infty} I_A (\{ \hat{p}^{i-1} (\nu) \}) \right)$$

$$= \left( \bigcap_{i=1}^{N-1} T_i \right) \cap \left( \bigcap_{i=1}^{N-1} I_A (\{ \hat{p}^{i-1} (\nu) \}) \right) \cap \left( \bigcap_{i=N}^{\infty} I_A (\{ \hat{p}^{i-1} (\nu) \}) \right)$$

$$= T^0 \cap \left( \bigcap_{i=1}^{N-1} T_i \cap (I_A (\{ \hat{p}^{i-1} (\nu) \})) \right) \cap \left( \bigcap_{i=N}^{\infty} I_A (\{ \hat{p}^{i-1} (\nu) \}) \right)$$

$$= T^0 \cap \left( \bigcap_{i=1}^{N-1} T_i \right) \cap \left( \bigcap_{i=N}^{\infty} T_i \right)$$

$$= \bigcap_{i=0}^{\infty} T_i$$

The second to last step is justified by Lem. 6.19. 

The right-hand-side of the equation in Thm. 6.22 is computable. The term $I_A (\{ \hat{p}^i (\nu) : i \in \mathbb{N} \})$ can be computed by Alg. 15 with $I_A (\{ \nu \}) = \text{dom}^*(T) + T$. The term $\left( \bigcap_{i=0}^{N-1} T_i \right)$ is a finite intersection of polynomial ideals which can be computed via Gröbner basis techniques (Lem. 2.71). Asymptotically, the Gröbner basis calculations for computing intersections as well as the Gröbner basis calculations in Alg. 15 dominate the running time, making the overall computation exponential.

**Example 6.23.** Recall $T = \langle y' - y - z, z' - 3z, y - z - 1 \rangle$ from Ex. 6.18.

$$T^* = \langle 3z^2 - 4z' + (z')^2, yz' - yz + z^2 - zz' - z' + z, 2y' - 2y - z' + z \rangle$$

### 6.6 Loop Summarization modulo LIRR

Loop summarization is the problem of computing, for a given transition formula $F$ representing the body of some loop, an over-approximation of the reflexive transitive closure of $F$. This section describes how to
combine the components introduced in the previous two sections to accomplish this task. We prove the key property that our loop summarization procedure is monotone. Finally, we discuss how this procedure can be combined with other summarization techniques to enhance the ability of an algebraic program analyzer to generate non-linear loop summaries.

Our iteration operator takes a four-step approach (pictured Fig. 6.1b). Given an input transition formula \( F \),

1. Compute the transition ideal \( \mathbf{I}_{\text{LIRR}}(F) \) of \( F \) (using the algorithm of Kincaid et al. [88]).

2. Compute a solvable reflection \( \langle t, T \rangle \) of \( \mathbf{I}_{\text{LIRR}}(F) \) (§ 6.4).

3. Compute \( T^* \) (§ 6.5).

4. Calculate the formula corresponding to the image of \( T^* \) under \( t \).

More succinctly, we define an operator \((-)\Simeq: TF \to TF\) to be

\[
F\Simeq \overset{\text{def}}{=} F(t[T^*])
\]

where \( \langle t, T \rangle \) is a solvable reflection of \( \mathbf{I}_{\text{LIRR}}(F) \). Naturally, one may repeat this recipe for defining a loop summarization operator by using ultimately solvable transition ideals (§ 6.4.2) and/or polynomial simulations (§ 6.4.3), and the soundness and monotonicity results that we prove below hold also for these variants.

**Example 6.24.** Consider the transition formula \( F \) below, and its associated transition ideal:

\[
F = \begin{pmatrix}
(k' = k + 1) \land (x' = y) \land (y' = x) \\
(z \geq 0 \land z' = w + z \land w' = x^2) \\
\land \neg((z \geq 0) \land w' = w + z \land z' = x^2)
\end{pmatrix}
\]

\[
\mathbf{I}_{\text{LIRR}}(F) = \begin{pmatrix}
w'x^2 - w'z' + x^2z' - x^2, \\
-w' + w + x^2 - z' + z, \\
x' - y, \\
y' - x, \\
k' - k - 1
\end{pmatrix}
\]

Notice that, while \( F \) employs a rich logical language involving disjunction, negation, and inequalities, its ideal \( \mathbf{I}_{\text{LIRR}}(F) \) is defined by the set of polynomials \( p \) such that \( F \) entails \( p = 0 \). A solvable reflection of \( \mathbf{I}_{\text{LIRR}}(F) \) is \( \langle t, T \rangle \) where \( t \) is the map that sends \( a \mapsto x, b \mapsto y, c \mapsto (w + z), \) and \( d \mapsto k \), and \( T \) is the ideal \( \langle a' - b, b' - a, c' - c - a^2, d' - d - 1 \rangle \). The closure of \( T \) is \( T^* = \langle ab - bb' + (b')^2 - ab', a' + b' - a - b, b^2 d' + a^2 d' - a^2 d - b^2 d - b + ab' + bb' - ab - 2c' + 2c \rangle \). The
(ideal generated by) the image of \( T^* \) under \( \mathcal{I} \) is
\[
\langle xy - yy' + (y')^2 - xy', x' + y' - x - y, \\
y^2k' + x^2k' - x^2k - y^2k - y^2 + xy' + yy' - xy - 2(w' + z') + 2(w + z) \rangle.
\]

Finally, we have \( F^* \overset{\mathcal{I}}{=} xy + (y')^2 = yy' + xy' \land x' + y' = x + y \land y^2k' + x^2k' + xy' + yy' - 2(w' + z') = x^2k + y^2k + y^2 + xy - 2(w + z) \).

**Theorem 6.25** (Soundness). Let \( F \) be a transition formula. For any \( n \in \mathbb{N} \), we have \( F^n \models_{\text{LIRR}} F^* \).

**Proof.** Let \( \langle t, T \rangle \) be a solvable reflection of \( \mathcal{I}_{\text{LIRR}}(F) \). We may show that
\[
\mathcal{I}_{\text{LIRR}}(F^n) \supseteq \mathcal{I}_{\text{LIRR}}(F)^n \supseteq \mathcal{I}[T^n] \supseteq \mathcal{I}[T^*]
\]
by induction on \( n \). The base case \( n = 0 \) is trivial. The induction step follows from the fact that (1) the sequential composition operator for ideals over-approximates the sequential composition operator for transition formulas, and (2) sequential composition for transition ideals preserves simulation.

**Theorem 6.26** (Monotonicity). Let \( F \) and \( G \) be transition formulas. If \( F \models_{\text{LIRR}} G \), then \( F^* \models_{\text{LIRR}} G^* \).

**Proof.** Suppose \( F \models_{\text{LIRR}} G \). Let \( \langle t, T \rangle \) be a solvable reflection of \( \mathcal{I}_{\text{LIRR}}(F) \), and let \( \langle u, U \rangle \) be a solvable reflection of \( \mathcal{I}_{\text{LIRR}}(G) \). Since \( F \models_{\text{LIRR}} G \), we must have \( \mathcal{I}_{\text{LIRR}}(F) \supseteq \mathcal{I}_{\text{LIRR}}(G) \), and thus \( u \) is a simulation from \( \mathcal{I}_{\text{LIRR}}(F) \) to \( U \). Since \( U \) is solvable and \( \langle t, T \rangle \) is a solvable reflection of \( \mathcal{I}_{\text{LIRR}}(F) \), there is a (unique) simulation \( \nu : T \to U \) such that \( u = \nu ; t \). Since \( \nu \) is a simulation, we have \( T^* \supseteq \nu[U^*] \), and so
\[
\mathcal{I}[T^*] \supseteq \mathcal{I} [\nu[U^*]] = (t \circ \nu)[U^*] = \mathcal{I}[U^*]
\]
Since \( F^* = F(\mathcal{I}[T^*]) \) and \( G^* = F(\mathcal{I}[U^*]) \), we have \( F^* \models_{\text{LIRR}} G^* \).

**6.6.1 Modular Design of Loop-Summarization Operators**

The loop summarization operator that is defined in this chapter is designed to compute polynomial invariants. Such invariants are often just a component of a correctness argument for a program—for example, a correctness argument may rely upon reasoning about inequalities, or may require a disjunctive invariant. Our loop summarization operator can be incorporated in a broader invariant generation scheme by using various combinators to combine summarization operators. For instance, the simplest such combinator is a product,
which combines two loop summarization \( \oplus_1 \) and \( \oplus_2 \) into one \( \oplus_1 \times \oplus_2 \) by taking their conjunction:

\[
F^{\oplus_1 \times \oplus_2} \overset{\text{def}}{=} F^{\oplus_1} \land F^{\oplus_2}
\]

Provided that both the \( \oplus_1 \) and \( \oplus_2 \) operators are monotone, then (1) so is their product, and (2) the resulting analysis is at least as precise as either component analysis.

Another kind of summarization combinator is the refinement technique proposed in Chapter 4. This combinator exposes phase structure in loops, and in particular enables a "base" summarization operator that may only generate conjunctive invariants to produce disjunctive invariants. In addition to monotonicity, this combinator requires four additional axioms in order to guarantee that it improves analysis precision. The following proposition states that indeed our summarization operator satisfies these conditions.

**Proposition 6.27.** Let \( F \) be a transition formula. Then the following hold

- (Reflexivity) \( 1 \models_{\text{LIRR}} F^{\oplus} \)
- (Extensivity) \( F \models_{\text{LIRR}} F^{\oplus} \)
- (Transitivity) \( F^{\oplus} \otimes F^{\oplus} \overset{\text{equiv}}{=} \text{LIRR} F^{\oplus} \)
- (Unrolling) For any natural number \( n \), \( (F^n)^{*} \models_{\text{LIRR}} F^{*} \).

**Proof.** Reflexivity, Extensivity, and Transitivity are straightforward. We shall prove unrolling.

Let \( n \) be a natural number. Let \( \langle t, T \rangle \) be a solvable reflection of \( \text{I}_{\text{LIRR}} (F^n) \), and let \( \langle u, U \rangle \) be a solvable reflection of \( \text{I}_{\text{LIRR}} (F) \). Since \( \text{I}_{\text{LIRR}} (F^n) \subseteq \text{I}_{\text{LIRR}} (F^n) \), \( \langle t, T \rangle \) is a solvable reflection of \( \text{I}_{\text{LIRR}} (F^n) \), and \( U^n \) is solvable, there is a (unique) simulation \( v : U^n \to T \) such that \( t \circ v = u \). Since \( (F^n)^{*} = F(t[T^n]) \) and \( (F^n)^{\oplus} = F([U^*]) \), it is sufficient to prove that \( T[T^n] \supseteq \cup [U^*] \). Observe that since \( v : U^n \to T \) is a simulation, we have

\[
T[T^n] = T \left[ \bigcap_{i=0}^{\infty} T^i \right] \supseteq T \left[ \bigcap_{i=0}^{\infty} \cup [U^n] \right] \supseteq T \left[ \bigcup_{i=0}^{\infty} \cup [U^n] \right]
\]

Then since \( t \circ v = u \), we have

\[
T[T^n] \supseteq T \left[ \bigcup_{i=0}^{\infty} U^n \right] = (t \circ v) \left[ \bigcap_{i=0}^{\infty} U^n \right] = v \left[ \bigcap_{i=0}^{\infty} U^n \right] \supseteq v \left[ \bigcup_{i=0}^{\infty} U^i \right] = \cup [U^*] \]

\[\square\]

### 6.7 Experimental Evaluation

We consider two experimental questions concerning our methods for synthesizing loop invariants for general programs:
1. (§ 6.7.2) How do our techniques apply to the task of verifying general programs?

2. (§ 6.7.3) How do our techniques for generating polynomial invariants perform on programs for which other tools guarantee completeness?

In relation to each of these questions we also want to understand the performance, both in terms of accuracy and running time, of using linear simulations as well as polynomial simulations of bounded degree for extracting solvable transition ideals from transition ideals.

### 6.7.1 Experimental Setup

**Implementation.** We implemented the techniques described in this chapter in a tool called **Abstractionator**.

Our implementation relies on

- Chilon and ChilonInv [88], for LIRR operations and generating invariant inequalities, respectively.
- The FGb library [51] for an implementation of the F4 algorithm [50], which we use for computing of Gröbner bases.
- **Flint** [135] for integer lattice computations and **Arb** [79] for numerical polynomial root finding. These operations are required to implement the algorithm of Ge [60] used in Alg. 15.

**Abstractionator** can be configured to use either linear or quadratic simulations, and either solvable or ultimately solvable transition ideals. Our testing revealed that (1) the difference between using solvable and ultimately solvable is negligible (both in success rate and runtime performance), and (2) the cost of naïve computation of the full inverse image $f_{X,2}^{-1}[-]$ for quadratic simulations is prohibitively high. In the following, we report on two configurations of **Abstractionator**: **USP-Lin** is the product of the ChilonInv domain and iteration operator induced by ultimately solvable linear reflections, **USP-Quad** is the product of **USP-Lin** and the iteration operator induced by solvable quadratic simulations with a single stratum (which necessitates only computing the affine polynomials in $f_{X,2}^{-1}[-]$, and is therefore more tractable).

**Environment.** We ran all experiments on a virtual machine (using Oracle VirtualBox), with a guest OS of Ubuntu 22.04 allocated with 8 GB of RAM, using a 4-core Intel Core i7-4790K CPU @ 4.00 GHz. All tools were run with the **BenchExec** [137] tool using a time limit of 300 seconds on all benchmarks.

**Benchmarks.** Our 202 benchmarks programs are sourced from the set of safe benchmarks from the c/ReachSafety-Loops subcategory of the Software Verification Competition (SV-COMP) [16]. We divided our 202 benchmarks into a loops category consisting of 176 programs, and an NLA category consisting of 26

---

*That is, the error location is truly unreachable.*
Table 6.1: Comparison of tools on the loops and NLA benchmarks. T represents the amount of time, in seconds, take by each tool not including timeouts nor out of memory exceptions. The number of timeouts is reported in parentheses. We also experienced out of memory exceptions with VeriAbs which are noted in parentheses. #P represents the number of benchmarks proved correct. The best results in each category is bolded.

<table>
<thead>
<tr>
<th></th>
<th>loops</th>
<th>NLA</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChilonInv</td>
<td>176</td>
<td>26</td>
<td>202</td>
</tr>
<tr>
<td>USP-Lin</td>
<td>974 (5)</td>
<td>38 (1)</td>
<td>1010 (6)</td>
</tr>
<tr>
<td>USP-Quad</td>
<td>1130 (5)</td>
<td>97 (1)</td>
<td>1230 (6)</td>
</tr>
<tr>
<td>CRA</td>
<td>1750 (31)</td>
<td>93 (6)</td>
<td>1840 (37)</td>
</tr>
<tr>
<td>VeriAbs</td>
<td>3430 (55, 2 OOM)</td>
<td>42 (23, 1 OOM)</td>
<td>3470 (78, 3 OOM)</td>
</tr>
<tr>
<td>ULTIMATE Automizer</td>
<td>2270 (51)</td>
<td>247 (17)</td>
<td>2520 (68)</td>
</tr>
</tbody>
</table>

benchmarks. The NLA benchmarks are modified versions of the programs in the nla-digbench set from SV-COMP, intended to evaluate the strength of Abstractionator’s ability to generate non-linear invariants. The nla-benchmark programs from SV-COMP have “proposed invariants” at each loop header, as well as assertions at the end of the programs as post conditions; we obtained the NLA suite by removing these “proposed invariants”. As a result, non-linear invariants must be synthesized in order to prove the post-condition (rather than simply verifying that the proposed invariant is an invariant, and implies the post-condition). The program from Fig. 6.1a is an example of a program in the NLA suite.

Comparison Tools. We have compared our techniques with ChilonInv [88], CRA (Chapter 3), VeriAbs 1.5.1-2 [1], and ULTIMATE Automizer 0.2.3 [69]. ChilonInv and CRA use a similar verification strategy of extracting implied solvable invariants of loop bodies to generate invariants of loops. VeriAbs and ULTIMATE Automizer are high performers at SV-COMP and provide context to the overall results. The strategies of ChilonInv, USP-Lin, and USP-Quad are all monotone algebraic analyses and the refinement technique of Chapter 4 applies. Refinement is guaranteed to improve the precision of these three techniques, and so we have employed refinement in the comparison of these three strategies.

6.7.2 How Do Our Techniques Perform on a Suite of General Verification Tasks?

Tab. 6.1 gives the results of running each tool on the program verification benchmarks. Theoretically, in terms of precision, we have ChilonInv ≤ USP-Lin ≤ USP-Quad. However, practically this does not consider timeouts. Due to the increased power of USP-Lin and USP-Quad we would expect in terms of time taken ChilonInv ≤ USP-Lin ≤ USP-Quad, and this is what we see reflected in Tab. 6.1. In our experiments we found
USP-Lin to outperform ChilonInv in both the loops category and the NLA category in terms of programs verified, at the expense of additional running time. Theoretically, USP-Quad is stronger than ChilonInv and USP-Lin; however, the extra power comes at a price of running time. As can be seen from Tab. 6.1 USP-Quad performed worse on the loops category compared with ChilonInv and USP-Lin because of the number of timeouts. However, due to its strong non-linear reasoning capability, USP-Quad outperformed all the other tools on the difficult NLA benchmarks.

Theoretically, USP-Lin and USP-Quad are incomparable with the other tools. On one hand CRA’s recurrence extraction procedure is weaker than the methods in this chapter. However, CRA is also able to produce invariants involving exponential and polynomial terms, whereas the techniques in this chapter are only able to produce invariants involving polynomial terms. VeriAbs is a portfolio of many different techniques, such as bounded model checking and k-induction. ULTIMATE Automizer implements a trace abstraction algorithm. We note that while USP-Lin outperformed VeriAbs and ULTIMATE Automizer on the loops category, VeriAbs and ULTIMATE Automizer have additional capabilities such as the ability to produce counterexamples in the case when an assertion does not hold. This capability is outside the scope of USP-Lin and USP-Quad. Nevertheless, we find USP-Lin to be quite competitive on our benchmark suite. It outperformed all other tools on the loops category except for CRA, where it is behind by only 5 examples. Moreover, because of the success of USP-Quad on the NLA suite we find that powerful techniques that generate polynomial invariants are required to verify interesting programs found in the literature.

### 6.7.3 How Do Our Techniques Compare with Prior Methods for Complete Generation of Polynomial Invariants?

In this subsection, we consider how our method for generating polynomial invariants (which works on general programs) compares with the method presented by Humenberger et al. [74] (which is complete, but applies to a more limited class of programs). The method of Humenberger et al. [74] is implemented in a tool called ALIGATOR. Both our methods of linear simulations as well as polynomial simulations are complete for loops whose bodies are described by a solvable polynomial map. ALIGATOR is also complete for such loops. However, the completeness result of Humenberger et al. [74] also extends to multi-path loops, where each branch is described by a solvable

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>USP-Lin</th>
<th>USP-Quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>egcd.c</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>fermat2.c</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>lcm2.c</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>divbin.c</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>prodbin.c</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>dijkstra.c</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
polynomial map (e.g., a loop of the form while(*){ if (*) A else B }, where A and B are described by solvable polynomial maps). On such an example, Aligator will produce all polynomial invariants of the loop, but Abstractionator cannot make the same guarantee. At the level of a loop we abstract the loop body to a solvable transition ideal. In the case of while(*){ if (*) A else B }, we create a solvable transition ideal that abstracts both A and B, which is strictly weaker than considering A and B separately as in Humenberger et al. [74].

We investigated how our USP-Lin and USP-Quad implementation perform on multi-path loops for which Aligator is complete, but our techniques are incomplete. A direct practical comparison between USP-Lin, USP-Quad, and Aligator is challenging because they take different formats as input. However, a subset of 6 programs from the multi-path benchmark suite of Aligator are applicable for our tool\footnote{Aligator and Abstractionator treat integer division differently}. All of these 6 programs are found in the NLA suite discussed in § 6.7.2. More detailed results of running USP-Lin and USP-Quad on these six examples can be found in Tab. 6.2. The completeness result of Humenberger et al. [74] applies to these 6 programs, so given enough time Aligator would be able to verify all 6 of them. As can be seen from Tab. 6.2 USP-Lin, is unable to verify any of the 6 programs; however, USP-Quad is able to verify 4 of the 6. For the other 2 programs, the reason USP-Quad is unable to succeed is because those examples perform integer division in a situation in which no round-off occurs. In these programs this property is essentially encoded with an exponential invariant, which is outside the capabilities of USP-Quad. From the results of Tab. 6.2 we conclude that while the class of loops for which our technique is complete is a subset of Aligator’s, we can still generate most of the invariants needed to prove correctness.

6.8 Related work

Polynomial Abstractions of Loops. The algorithm in § 6.4 for computing the solvable reflection of a transition ideal can be seen as both a refinement of the method from Chapter 3 for extracting a solvable polynomial map from a transition formula and a generalization of Zhu and Kincaid [140]’s algorithm for computing deterministic affine reflections. Contrasting with Chapter 3, the method of this chapter is guaranteed to find a best abstraction as a solvable transition ideal, which is essential to prove monotonicity of our analysis. The method of Chapter 3 instead finds a “best” affine recurrence. One way solvable transition ideals generalize affine recurrences is that they can describe domain information. This generalization is necessary in order to produce a reflection (and consequently lead to monotonicity). The issue with the affine recurrences from Chapter 3 is essentially captured by Proposition 1 of Zhu and Kincaid [140]. Contrasting with Zhu and
Kincaid [140], our algorithm consumes and produces transition ideals, which generalize the affine relations considered in that work.

Amrollahi et al. [6] considers the problem of abstracting polynomial endomorphisms by solvable polynomial maps. The technique presented in § 6.4 is more general in the sense that it operates on transition ideals rather than polynomial endomorphisms. A polynomial endomorphism \( p : \mathbb{Q}[X] \to \mathbb{Q}[X] \) can be encoded as a transition ideal, generated by the polynomials \( \{x' - p(x) : x \in X\} \), in which case the algorithm in § 6.4.1 computes a solvable transition ideal (from which we may recover a solvable polynomial map—that is, our procedure serves the same purpose as of Amrollahi et al. [6] for the inputs considered in that work). Moreover, our procedure provides a precision guarantee: it finds solvable reflections of transition ideals.

For example, consider the loop below (left) along with its solvable reflection (right)

\[
\text{while } * \text{ do } \begin{cases} x := x + z^2 + 1; \\ y := y - z^2; \\ z := z + (x + y)^2 \end{cases} \quad \langle \{a \mapsto x + y, b \mapsto z\}, (a' - a - 1, b' - b - a) \rangle \quad \text{Solvable reflection}
\]

While the technique in [6] is able to identify the first polynomial in the reflection (corresponding to the update \( (x + y) := (x + y + 1) \)) it cannot find the second \( (z' := z + (x + y)^2) \), since there is a non-linear dependence of \( z \) upon the “defective” variables \( x \) and \( y \) whose dynamics cannot be described by a solvable polynomial map.

Frohn et al. [58] considers another related problem: \textit{given a polynomial endomorphism } \( p \), \textit{is there a polynomial automorphism } \( f \) \textit{such that } \( f^{-1} \circ p \circ f \) \textit{is solvable?} The procedure in § 6.4 can also be used to solve this problem: if \( \langle t, T \rangle \) is the solvable reflection of \( p \), then such an \( f \) exists (namely, \( t \)) exactly when the ambient dimension of \( T \) is equal to that of \( p \) (and \( T \) has real eigenvalues). § 6.4 generalizes this result in the sense that, (1) we operate on transition ideals rather than polynomial endomorphisms and (2) should the answer to the decision problem be “no”, we may still compute an abstraction of \( p \).

Complete polynomial invariant generation. Kovács [94], Humenberger et al. [74], Hrushovski et al. [72, 71], Rodríguez-Carbonell and Kapur [116] are complete methods for generating polynomial invariants on limited program structures. Our method matches the completeness results of these works on single loops whose bodies are described by solvable polynomial maps; however, the completeness result of each of these works extend beyond simple loops whose bodies are described by a solvable polynomial map.

Hrushovski et al. [71, 72] present a method that is complete for generating polynomial invariants for affine programs where all branching represents non-deterministic choice. Our methods have no issue analyzing such programs. Moreover, our method can also reason about programs with polynomial assignments as
well as branching with conditionals. However, even though our method can reason about general affine programs, we can only guarantee completeness in the case of a loop whose body is described by a solvable polynomial map.

Kovács [94] presents complete polynomial invariant generation for $P$-solvable loops. These are loops, with no branching, whose bodies have either Gosper-summable or c-finite assignments. As stated in § 6.3.2, c-finite sequences are equivalent to solvable polynomial maps, and so our technique matches Kovács [94] in that regard. However, while we always extract a solvable transition ideal from a loop, solvable transition ideals are not powerful enough to capture certain Gosper-summable examples. Thus, while our method is monotone on such examples, it does not guarantee completeness. Humenberger et al. [74] extends Kovács [94] to the case of multi-path loops where each branch has a body with Gosper-summable or c-finite recurrence assignments. In the Gosper-summable case the comparison is the same as Kovács [94]. In the c-finite case we are also not complete; however, we experimentally compare with Humenberger et al. [74] in this case in § 6.7.3. In either the case of Kovács [94] or Humenberger et al. [74] they cannot make a completeness guarantee for programs having branching with conditionals or programs with arbitrary loop nesting.

Rodríguez-Carbonell and Kapur [116], Rodríguez-Carbonell and Kapur [117] present a complete method for the case of a single multi-path loop where each branch has a body described by a c-finite recurrence. This matches the c-finite case of Humenberger et al. [74], except Rodríguez-Carbonell and Kapur [116], Rodríguez-Carbonell and Kapur [117] have an additional restriction on the eigenvalues of the c-finite recurrences (corresponding to the $\Theta_i$ variables of Eqn. (6.2)). For Rodríguez-Carbonell and Kapur [116], Rodríguez-Carbonell and Kapur [117] the eigenvalues are required to be positive and rational. We have no such restriction and so our method generalizes Rodríguez-Carbonell and Kapur [116], Rodríguez-Carbonell and Kapur [117] in the case of a simple loop where the body is described by a c-finite recurrence. However, their completeness result goes beyond our capability in the case of a multi-path loop with positive rational eigenvalues.

Template Based Methods. Another method for generating polynomial invariants is to reduce the problem to constraint solving by supposing that the invariant takes the form of some parameterized template, and solving for the parameters [91, 106, 120, 26, 105, 29, 61]. These methods have the benefit of being able to handle problems with arbitrary control flow. Furthermore, they are often complete for generating invariants that fit the given template. Many template methods consider all polynomials up to some bounded degree. In such cases when the desired polynomial is within the degree bound, template based methods have the potential to generate invariants for general programs that our method would theoretically miss. In contrast, our method does not require a degree bound. Even for linear simulations, there is no bound on the degree of
the invariant our method calculates.

Monotone algebraic program analysis. There has been a recent line of work that uses the framework of algebraic program analysis to develop program analyses with monotonicity guarantees [126, 141, 140, 88]. In particular, Kincaid et al. [88] proposes a monotone loop summarization algorithm based on the theory of linear integer/real rings. Our technique is complementary in the sense that our method computes stronger invariant polynomial equations than Kincaid et al. [88], but cannot synthesize invariant polynomial inequalities.
Chapter 7

Other Contributions

7.1 Periodic-Rational Spectral Decomposition

Chapter 3 describes a process of summarizing loops via extraction and solving of matrix recurrences—that is, recurrences of the form $x^{[n+1]} = Ax^{[n]} + b$ for a rational matrix $A$ and vector of c-finite recurrences $b$ over $n$. Determining a closed-form solution for a matrix recurrence requires factoring the characteristic polynomial of $A$. If the characteristic polynomial is fully factored into linear terms, each linear term will generate an exponential term, with the corresponding root of the polynomial the base of the exponential term.

The choice of which field to use when factoring the characteristic polynomial generates a challenging design decision. If a closed-form solution involving only polynomials and exponentials is required, then one has to factor over an algebraically closed field, such as the complex numbers $\mathbb{C}$. However, this approach requires a system that can perform expensive manipulations of algebraic numbers, and, more critically, automated reasoning about complex exponentiation is notoriously difficult. Evidence of this difficulty is the open status of the decidability of Skolem’s problem [107]. In essence, Skolem’s problem is whether, given a recurrence of the form $x^{[n+1]} = Ax^{[n]}$, as well as appropriate initial conditions, is there an $n$ where $x^{[n]}$ has a zero entry.\footnote{This recurrence as well as the other matrix recurrences in this section have no additive vector $b$, in contrast with the recurrences in Chapter 3. However, every recurrence of the form $x^{[n+1]} = Ax^{[n]} + b$, with $b$ a vector of c-finite recurrences in $n$, can be transformed to a recurrence of the form $x^{[n+1]} = A'x^{[n]}$ by adding dimensions to $A$.} Given an instance of Skolem’s problem, one can exactly capture the dynamics of $x^{[n]}$ using polynomial and exponential terms over $\mathbb{C}$, that is, $x^{[n]} = f(n)$ for some vector of polynomial-exponentials in $n$. However, because $f(n)$ can involve complex exponentiation, $f(\cdot)$ can cross the $x$-axis an infinite number of times. Answering Skolem’s problem requires determining if one of these crossings occurs at an integer value.

Due to these complications of reasoning over complex arithmetic, in Kincaid et al. [87] (Chapter 3), we made the choice to factor the characteristic polynomial of $A$ over $\mathbb{Q}$. This choice yielded a system that reasons...
over purely rational numbers. On the other hand, we had to introduce the notion of implicitly interpreted functions (IIFs) to handle the case when the characteristic polynomial of the recurrence could not be fully factored over \( Q \). Moreover, IIFs are treated as uninterpreted functions outside the recurrence solver, and thus the wedge domain can perform limited reasoning with these terms. In Kincaid et al. [86], we made the observation that the solution to some matrix recurrences can be expressed in rational arithmetic even when the characteristic polynomial of the recurrence contains irreducible factors.

To illustrate the idea, consider the following matrix recurrence:

\[
\begin{bmatrix}
  x[n+1] \\
  y[n+1]
\end{bmatrix} =
\begin{bmatrix}
  0 & 1 \\
  -1 & 0
\end{bmatrix}
\begin{bmatrix}
  x[n] \\
  y[n]
\end{bmatrix}
\]  
(7.1)

The eigenvalues of the matrix in Eqn. (7.1) are \( i \) and \( -i \), where \( i \in C \) is the imaginary unit. Thus, the following “standard” closed-form solution for Eqn. (7.1) involves complex exponentiation:

\[
x[n] = x[0] \frac{i^n + (-i)^n}{2} + y[0] \frac{(-i)^n + i(-i)^n}{2} \\
y[n] = x[0] \frac{(-i)^n + i(-i)^n}{2} + y[0] \frac{i^n + (-i)^n}{2}
\]

Let \( B = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \). \( B \) has the relevant property that while \( B \) has complex eigenvalues, \( B^2 \) has strictly rational eigenvalues. This observation can then be leveraged to obtain a closed-form solution for Eqn. (7.1) over rational arithmetic. To see how, observe that if we have \( x[n+1] = Ax[n] \), then \( x[n] = A^n x[0] \). In other words, finding a solution to a matrix recurrence can be transformed to a symbolic matrix-exponentiation problem.

Note that, for any \( p > 1 \), we have the following equality

\[
A^n = A^{(n \mod p)} (A^p)^{\lfloor \frac{n}{p} \rfloor}
\]  
(7.2)

Eqn. (7.2) gives an alternative method to symbolically compute a matrix \( A \) to some power \( n \). Given a fixed \( p \), \( A^{(n \mod p)} \) only takes on finitely many different values; therefore, calculating \( A^n \) can be reduced to calculating the matrix \( A^p \) to some power \( \lfloor \frac{n}{p} \rfloor \). Moreover, if \( A^p \) has strictly rational eigenvalues for some \( p \), then the symbolic exponentiation of \( A^p \) will contain only rational exponentiation.

For example, as stated previously, the matrix \( B \) from Eqn. (7.1) has complex eigenvalues; however, \( B^2 \) has

\(^2\text{Equivalently, roots of the characteristic polynomial of the recurrence}\)
only rational eigenvalues. Thus, we can use Eqn. (7.2) to symbolically exponentiate \( B \) as follows:

\[
\begin{bmatrix}
0 & 1 \\
-1 & 0
\end{bmatrix}^n = \begin{bmatrix}
0 & 1 \\
-1 & 0
\end{bmatrix}^{(n \mod 2)} \begin{bmatrix}
0 & 1 \\
-1 & 0
\end{bmatrix}^2 \left\lfloor \frac{n}{2} \right\rfloor
\]

\[
= \begin{bmatrix}
0 & 1 \\
-1 & 0
\end{bmatrix}^{(n \mod 2)} \begin{bmatrix}
-1 & 0 \\
0 & -1
\end{bmatrix} \left\lfloor \frac{n}{2} \right\rfloor
\]

\[
= \begin{cases}
\begin{bmatrix}
(-1)^{\left\lfloor \frac{n}{2} \right\rfloor} & 0 \\
0 & (-1)^{\left\lfloor \frac{n}{2} \right\rfloor}
\end{bmatrix} & \text{if } n \text{ is even} \\
\begin{bmatrix}
0 & (-1)^{\left\lfloor \frac{n}{2} \right\rfloor} \\
-(-1)^{\left\lfloor \frac{n}{2} \right\rfloor} & 0
\end{bmatrix} & \text{if } n \text{ is odd}
\end{cases}
\]

(7.3)

From Eqn. (7.3) we can write a solution to the original recurrence as

\[
n \mod 2 \equiv 0 \implies (x[n] = x[0](-1)^{\left\lfloor \frac{n}{2} \right\rfloor} \land y[n] = y[0](-1)^{\left\lfloor \frac{n}{2} \right\rfloor}) \land \\
n \mod 2 \equiv 1 \implies (x[n] = y[0](-1)^{\left\lfloor \frac{n}{2} \right\rfloor} \land y[n] = -x[0](-1)^{\left\lfloor \frac{n}{2} \right\rfloor})
\]

In Kincaid et al. [86], we developed a method to solve matrix recurrences \((x^{[n+1]} = Ax^{[n]} \) based on the principle that it is possible for rational matrices with complex eigenvalues to have purely rational eigenvalues when raised to some power \( p \), and this fact can be leveraged to produce closed forms in the logic of rational arithmetic. A key result of that work was that it is possible to bound such a power \( p \) with an exponential expression based on the size of the matrix. In other words, let \( A \in \mathbb{Q}^{n \times n} \). If there is some \( k \) such that \( A^k \) has purely rational eigenvalues, then there is a period, \( p \), and exponential expression \( e(-) \) such that \( p \leq e(n) \) and \( A^p \) has purely rational eigenvalues. With such a bound, one can then produce closed-form solutions in rational arithmetic by testing whether \( A^p \) has rational eigenvalues for \( p = 1 \) up to \( e(n) \). If such a \( p \) is found, Eqn. (7.2) can then be used to produce a formula for the closed-form solution that requires at most exponential space. If no such \( p \) is found, then there is no power \( k \) such that \( A^k \) has only rational eigenvalues, and such a matrix recurrence is then outside the scope of the method.

However, in Kincaid et al. [86] we improve upon the exponential-space result by showing that it is possible to capture the dynamics of a matrix recurrence, \( x^{[n+1]} = Ax^{[n]} \) where \( A^p \) has only rational eigenvalues for some \( p \), using a polynomial-space formula. Moreover, this polynomial-space formula can be produced with a polynomial-time algorithm. Suppose that \( A \) is a rational matrix with \( A^k \) having purely rational eigenvalues. The idea is that while the smallest \( p \) for which \( A^p \) has all rational eigenvalues may be exponential, \( A \) can
be made up of smaller periodic blocks. We show that the periods of these smaller blocks are bounded from above by the polynomial $n^3$ ([86, Lemma 5.3]). In other words, the smallest $p$ such that $A^p$ has all rational eigenvalues may be exponential; however, $A^r$ may have some rational eigenvalues with $r \leq n^3 < p$. The characterization of these periodic blocks is referred to as periodic rational spectral decomposition. Computing a periodic rational spectral decomposition is similar to the exponential method of testing $A^p$ for $p = 1$ to $e(n)$, except that instead of checking whether $A^p$ has all rational eigenvalues, we check whether $A^p$ has some rational eigenvalues for $p = 1$ to $n^3$. Similar to Eqn. (7.2) the exponential behavior of $A$ can be characterized using rational arithmetic by characterizing the exponential behavior of the periodic blocks.

Computing a periodic rational spectral decomposition has not only the advantage of being polytime, but also has the advantage of being able to partially characterize matrices $A$ for which there is no $p$ such that $A^p$ has purely rational eigenvalues. That is, even though $A^p$ may never have all rational eigenvalues for any $p$, $A^r$ may have some rational eigenvalues for some $r$. Moreover, these periodic sub-blocks can be used to partially capture the dynamics, i.e., with an abstraction, of the exponentiation of the matrix $A$. Thus, the method in Kincaid et al. [86] can still be useful even in the case when $A^p$ never has strictly rational eigenvalues. In other words, given an arbitrary matrix $A$, our algorithm can produce an abstraction $B$ such that $B^p$ has purely rational eigenvalues for some $p$. Furthermore, it is shown that this abstraction is a best abstraction.

The main motivation for this work was to express closed-form solutions of matrix recurrences using rational numbers. Rational numbers are simpler to manipulate and therefore a simpler analysis system results. In Kincaid et al. [86], we went further and showed that determining the satisfiability of the resulting closed-forms using rational numbers is decidable, which implies that Skolem’s problem is decidable for matrices $A$ where $A^p$ has all rational eigenvalues for some $p$. As hinted at previously, the difficulty of the general case is that when a closed form $f(n)$ has complex exponentiation, $f(\_)$ can cross the x-axis an infinite number of times. However, if $f(n)$ has only rational (or real) exponentiation, then eventually one of the exponential terms dominates, and so $f(\_)$ can only cross the x-axis a finite number of times. From this insight, we gave a decision procedure for the logic of closed forms, which is of mostly theoretical interest.

Contributions

Recapping, Kincaid et al. [86] provides the following contributions:

- A polytime algorithm for producing a periodic rational spectral decomposition of a rational matrix. Moreover, this algorithm can be applied to any rational matrix resulting in a best abstraction.

- A method for producing a closed-form solution of a matrix recurrence from a periodic rational spectral decomposition. If the original matrix $A$ has the property that $A^p$ has rational eigenvalues for some
p, then the closed form exactly captures the exponential behavior of A. Otherwise the closed form approximates the exponential behavior of A.

- A decision procedure showing the resulting closed forms belong to a decidable fragment of logic.

### 7.2 Compositional Higher-Order Recurrence Analysis

The previous examples of this thesis have only been examples of intra-procedural analysis, i.e., non-recursive programs. However, via the techniques of Kincaid et al. [85], the analysis techniques of Chapters 3, 4 and 6 can be directly applied to linearly recursive programs. Linear recursive procedures make at most one recursive call along each path of the procedure. A procedure is non-linearly recursive if it contains a path with more than one recursive call along that path. The procedure $\text{subsetSumAux}$ in Fig. 7.1 is an example of a non-linearly recursive procedure. An application of resource-bound analysis (rba) would be to automatically establish an upper bound on the increase of the $n\text{Ticks}$ variable based the size of the input array $n$. Unfortunately, while the techniques of Kincaid et al. [85] can be applied to non-linear procedures, those techniques are limited in the setting of non-linear recursion; hence, the techniques of Kincaid et al. [85] are unable to bound the increase of the $n\text{Ticks}$ variable for the program in Fig. 7.1.

```plaintext
1
2 int nTicks; bool found;
3 int subsetSum(int *A, int n) {
4    found = false;
5    return subsetSumAux(A, 0, n, 0);
6  }
7
8 int subsetSumAux(int *A, int i, int n, int sum) {
9    nTicks++;
10   if (i >= n) {
11      if (sum == 0) { found = true; }
12      return 0;
13   }
14   int size = subsetSumAux(A, i + 1, n, sum + A[i]);
15   if (found) { return size + 1; }
16   size = subsetSumAux(A, i + 1, n, sum);
17   return size;
18 }
```

Figure 7.1: $\text{subsetSum}$ takes an array $A$ of $n$ integers, and performs a brute-force search to determine whether any non-empty subset of $A$’s elements sums to zero. $\text{subsetSumAux}$ is an example of a non-linearly recursive procedure.

In Breck et al. [24], we introduced a new technique to address the deficiencies of prior techniques on non-linearly recursive procedures. We described this technique as combining template-based methods with our prior experience of recurrence analysis. Hence, the title of the paper *Templates and Recurrences: Better Together*. 
The execution of a non-linearly recursive procedure can be modeled as a tree, where each node of the tree represents a call to the procedure. Base cases are represented by the leaves of the tree, and the initial call is represented by the root of the tree. This recursion-tree model of the execution of a procedure gives a natural notion of recursion height of an execution. In Breck et al. [24], we defined a height-$h$ execution of a procedure $P$ to be any execution of $P$ that uses a stack height of at most $h$, or, in other words, an execution of $P$ having recursion depth no more than $h$. Base cases were defined to have height 1. Ultimately, a sound summary, say $\alpha$, of a procedure, $P$, will over-approximate the execution of $P$ for any height $h$. In Breck et al. [24], our summaries took the form $\alpha = \exists h \in \mathbb{N}. \beta(h)$, which allowed us to create summaries that directly depend on the recursion height $h$. In other words, our strategy was to concoct a summary $\beta(h)$ that over-approximates any height-$h$ execution of the procedure. Moreover, the summary $\beta(h)$ can be a function of $h$ even if $h$ is not directly a parameter of $P$. Our strategy for constructing such a $\beta(h)$ was to synthesize a set of hypothetical summaries in two steps:

1. Using a template, create a summary $\beta(h)$ such that $\beta(1)$ over-approximates any height-1 execution, i.e. a base case of the procedure.

2. Suppose that $\beta(h)$ is an over-approximate summary for any height-$h$ execution. Using the procedure $P$ as well as $\beta(h)$ as a summary for the recursive calls, extract and solve a recurrence for $\beta(h+1)$.

A key insight of Breck et al. [24] was that a recurrence solution for $\beta(h)$ essentially provides an inductive argument for the soundness of $\beta(h)$ for any $h$.

To illustrate the technique, consider the $\text{subsetSumAux}$ procedure of Fig. 7.1. Our hypothetical summaries consist of a conjunction of inequalities of the form $\tau_i(x,x') \leq b_i(h)$, where $\tau_i$ is a polynomial over the program variables $x$ and their primed copies $x'$, and $b_i : \mathbb{N} \to \mathbb{Q}$. The first step to create a sound over-approximation is to find polynomial terms that are bounded from above by a constant in every height-1 execution, i.e., in the procedure’s base cases. $\text{subsetSumAux}$ has only one base case, and in that case there are many polynomial terms that are bounded from above. Our technique finds many of them, but for presentation purposes we focus on the polynomials $\tau_1 \overset{\text{def}}{=} \text{return}$ and $\tau_2 \overset{\text{def}}{=} \text{nTicks} - \text{nTicks} - 1$. $\tau_1$ and $\tau_2$ are both bounded from above by the constant 0. From this information, we can create the hypothetical summary $\tau_1 \leq b_1(h) \land \tau_2 \leq b_2(h)$ with $b_1(1) = b_2(1) = 0$. In general, such polynomial terms and constants can be found automatically using the wedge symbolic-abstraction algorithm (Chapter 3, Alg. 5). For the “inductive step” of the analysis, we suppose that there are functions $b_1(h)$ and $b_2(h)$, such that $\tau_1 \leq b_1(h)$ and $\tau_2 \leq b_2(h)$ for any height-$h$ execution. We then consider a height-$h+1$ execution. Furthermore, in our analysis of a height-$h+1$ execution we can use our hypothetical summary for the recursive calls. For example, consider how the variable $\text{nTicks}$ increases in a height-$h+1$ execution, assuming that $\text{nTicks}' - \text{nTicks} - 1 \leq b_2(h)$.
for some yet to be determined $b_2(h)$. We have from procedure entry to the first recursive call that $nTicks$ increases by 1. Because our hypothetical summary states that $nTicks$ can increase by at most $b_2(h) + 1$ through the first recursive call, we conclude that $nTicks$ can increase at most $b_2(h) + 2$ from procedure entry to just after the first recursive call in a height-$h + 1$ execution. Similarly, because our hypothetical summary states that $nTicks$ can increase by at most $b_2(h) + 1$ through the second recursive call, we conclude that $nTicks$ can increase at most $2b_2(h) + 3$ from procedure entry to just after the second recursive call in a height-$h + 1$ execution. Written another way, assuming that $nTicks' - nTicks - 1 \leq b_2(h)$ holds in any height-$h$ execution, we have that $nTicks' - nTicks - 1 \leq 2b_2(h) + 2$ in any height-$h + 1$ execution. This reasoning can give a recursive definition for $b_2(h)$. That is, $b_2(h + 1) = 2b_2(h) + 2$ and $b_2(1) = 0$. By induction, a function $b_2$ that satisfies this recurrence will be a sound over-approximation for $\tau_2$ for any height-$h$ execution. A solution to this recurrence is $b_2(h) = 2^h - 2$. Thus, $nTicks' - nTicks - 1 \leq 2^h - 2$ for any height-$h$ execution. A similar analysis establishes that in any height-$h + 1$ execution, $return' \leq b_1(h) + 1$, and so $return' \leq h - 1$ for any height-$h$ execution. From this height-base recurrence analysis, we can conclude $\exists h \in \mathbb{N}. \text{return}' \leq h - 1 \land nTicks' \leq nTicks + 2^h - 1$ is a sound summary for $\text{subsetSumAux}$. Unfortunately, the resulting summary in the previous paragraph is not at all useful. Because of the placement of the quantifier, the resulting formula simplifies to true. The issue is that we have not discovered any relationship between the possible height values $h$ and the procedure’s input parameters. Thus, the method of Breck et al. [24] has a depth-bound analysis phase that attempts to bound the overall recursion depth in terms of the input parameters. I do not give the details of the depth-bound analysis phase here, but instead mention the result of this phase for the procedure $\text{subsetSumAux}$. In Breck et al. [24], we note that the depth-bound analysis determines that $h \leq \max(1, 1 + n - i)$ holds. Combining this result with the height-based recurrence analysis, we have

$$\exists h \in \mathbb{N}. h \leq \max(1, 1 + n - i) \land return' \leq h - 1 \land nTicks' \leq nTicks + 2^h - 1.$$ 

From this summary, it can be seen that $\text{subsetSum}$ has running time bounded from above by $2^n + 1$ and return value bounded from above by $n$.

Contributions

Recapping, Breck et al. [24] provides the following contributions:

- An analysis method based on “hypothetical summaries.” A hypothetical summary hypothesizes an over-approximation of a particular form, using uninterpreted-function symbols to stand for unknown
expressions. Analysis is performed to obtain constraints on the function symbols, which are then solved to obtain a summary.

- A procedure-summarization technique called height-based recurrence analysis, which uses the notion of hypothetical summaries to bound values of program variables based on recursion height.

- A depth-bound analysis phase that is able to bound the recursion height of a procedure based on the input parameters of the program.

- An implementation of the combined techniques in a tool called CHORA. CHORA is able to handle many non-linearly recursive programs, and generate summaries that include exponentials, polynomials, and logarithms. Experimental highlights show that CHORA is able to automatically show that (i) the time taken by merge-sort is $O(n \log(n))$, (ii) the time taken by Strassen’s algorithm is $O(n^{\log_2(7)})$, and (iii) an iterative function and a non-linearly recursive function that both perform exponentiation are functionally equivalent.
Chapter 8

Conclusion

In this chapter, we recap the main contribution of this dissertation, discuss some limitations, and give some final concluding remarks.

8.1 Contributions

The main contributions presented in this dissertation are as follows:

§ 3.4 We presented the wedge abstract domain, a numerical abstract domain capable of reasoning about non-linear arithmetic. Just as convex polyhedra represent properties in the conjunctive fragment of linear arithmetic, wedges represent properties in the conjunctive fragment of non-linear arithmetic (including polynomials, exponentials, and logarithms). The deductive power of wedges is due to polyhedral and Gröbner-basis techniques, congruence closure, and simple inference rules for non-linear functions. The key operation supported by the domain is symbolic abstraction [110, 134], which, given an arbitrary non-linear formula \( \varphi \), computes a wedge that over-approximates \( \varphi \).

§ 3.5 We presented a semantics-based algorithm for extracting recurrence relations that are entailed by a loop-body formula. The algorithm is based on first over-approximating the loop body by a wedge, and then using techniques from linear algebra to extract recurrences from the wedge. The algorithm can extract recurrences involving non-linear arithmetic and inter-dependent program variables; the class of recurrences that can be extracted by this algorithm corresponds to C-finite recurrences.

§ 3.6 We presented an algorithm, OCRS, that is able to solve these recurrences, and find closed-form solutions that include polynomials, exponentials, and logarithms. OCRS is based on an automated and enhanced form of the discrete operational calculus of Berg [15]. Classically, the closed forms of C-finite sequences involve algebraic irrational or algebraic complex numbers, but OCRS avoids non-rational numbers by
using what we call *implicitly interpreted functions*. Each implicitly interpreted function is associated with a term in the logic of OCRS that exactly characterizes the function, but outside of the recurrence solver (and in particular, within the wedge domain) an implicitly interpreted function is treated as an uninterpreted function symbol.

§ 4.3 We introduced the structure of pre-Kleene algebras to characterize algebraic analyses. The terms of pre-Kleene algebras are the same as the terms of regular expressions. Because algebraic analyses use regular expressions to represent analysis problems, the theory of pre-Kleene algebras can be used to compare the analysis precision of two different algebraic program-analysis problems, by reinterpreting the defining regular expressions in the theory of pre-Kleene algebras.

§ 4.4 We leveraged the theory of pre-Kleene algebras to develop an algorithm to automatically refine a regular expression, that originally specifies how a loop is to be analyzed, creating another regular expression that is guaranteed to yield at least as good an analysis result compared with analyzing the original expression.

§ 4.5 Using the algorithm for refining loops, we developed an algorithm that refines arbitrary program-analysis problems. We implemented our general refinement algorithm and found that the technique had substantial impact. Our experiments showed that algebraic analyses with refinement were able to prove approximately 25% more benchmarks correct at the expense of approximately 50% more running time, when compared with analyses that did not use refinement.

§ 5.1 We introduced the *optimal symbolic-bound synthesis problem* (OSB). An instance of the OSB problem over a formula $\phi$, asks to find a symblic-bound $t^*$ for a term $t$, such that (i) $\phi$ implies that $t^*$ upper-bounds $t$, and (ii) $t^*$ is optimal with respect to some term-desirability order.

§ 5.4 We introduce a *local-projection* procedure for polyhedra. Local projection can be seen as an instantiation of model-based projection [92]. In the case of polyhedra, local projection provides a lazy way of projecting out variables—which, in practice, can be more efficient compared with a full projection method, such as Fourier-Motzkin elimination (§ 5.4.2). We further showed how our local-projection method can be used to solve instances of the OSB problem over polyhedra and “dimensional orders.”

§ 5.4.3 We introduced the notion of a *cone of polynomials*. A cone of polynomials consists of a polynomial ideal, representing known equations, and a polyhedral cone, representing known inequalities. We showed how to reduce a polynomial term $t$ with respect to a cone of polynomials, $C$, yielding an upper-bound $t^*$ that is optimal with respect $C$ and a monomial order.
§ 5.5 We gave a heuristic method for extracting an implied cone of polynomials from a non-linear formula, $\phi$. Our method for extracting allows for additional function symbols in formula $\phi$, such as floor and division. We then showed how our extraction method can be used in conjunction with our method for reducing a polynomial by a cone of polynomials to yield a method that addresses the OSB problem for non-linear formulas.

§ 6.3 We introduce solvable transition ideals to serve as an intermediate representation for summarizing loops.

§ 6.4 We gave a method for computing a reflection of a transition ideal as a solvable transition ideal. That is, our method takes in a set of polynomial equations, representing invariants of a loop body, and produces an abstraction of the loop body as a solvable transition ideal. Furthermore, our method produces a best abstraction (with respect to a class of simulations) of a transition ideal as a solvable transition ideal.

§ 6.5 We gave an algorithm for computing the Kleene closure of a solvable transition ideal. In other words, the algorithm takes in a solvable transition ideal $U$, and computes the polynomial invariants that are common to each $U^n$, where $U^n$ represents the n-fold composition of $U$. Our closure method makes use of a sub-algorithm that generalizes the method of Kauers and Zimmermann [82], which computes the algebraic relations over $\mathbb{Q}$ of a set of c-finite sequences of rational numbers, to compute algebraic relations over $\mathbb{Q}$ of a set of c-finite sequences of an arbitrary $\mathbb{Q}$-algebra.

§ 6.6 We show how the combination of (i) computing a reflection of a transition ideal as a solvable transition ideal, and (ii) computing the Kleene closure of the resulting solvable transition ideal, yields a program-summarization method that is monotone for polynomial programs.

8.2 Limitations and Potential Future Directions

Here some subtle limitations are noted. Some possible future directions for the work presented in this dissertation would be the creation of methods and techniques to address the limitations discussed below.

Lack of support for non-numeric invariants. The example analyses presented in this dissertation produce numeric invariants. The bulk of the research of this dissertation is designed with the goal of increasing the expressiveness of—and the precision of—numeric invariants returned by program analyzers. However, numeric invariants may not always be the best model to represent program behaviors. For example, most programs use arrays or other more complex objects in memory during their execution. The techniques presented in this dissertation can still analyze such programs by essentially abstracting away the array accesses. However, this approach means that our techniques are not powerful enough to verify many
programs, when correctness requires some reasoning about more complex data-structures compared with simple numeric variables.

Aside from the lack of support for complex data-structures, when our techniques produce numeric invariants, we assume that the values of variables range over “mathematical numbers,” such as the integers or the reals. In reality, program variables range over computational representations of numbers. For example, given an integer variable in a program, in our methods we assume that variable can range over the mathematical integers. In a computer, that variable must be stored in some finite representation, and thus can, for example, \textit{overflow}; however, the possibility of overflow is not captured by our encoding.

\textit{No support for producing counter-examples.} In Chapters 3, 4 and 6, we experimentally compared our techniques with top performers on the Software Verification Competition (SV-COMP) benchmarks. Those experiments showed that our techniques are competitive on those benchmarks. However, the other comparison tools often have the capability to \textit{refute} a property of a given program. That is, when an assertion inside a program does not hold, many other techniques have the potential to provide a concrete counterexample that demonstrates the invalidity of the assertion. Our techniques employ abstraction, and thus when we are unable to prove an assertion it could be because the assertion is invalid, or it could be that the abstraction is simply too imprecise. Each assertion check that our technique makes corresponds to a satisfiability check between our abstraction and the assertion. When an assertion does not pass, the satisfiability check returns a valid model of our abstraction that violates the assertion. It remains an open problem whether one could make use of this model in the search or construction of a true counterexample.

\subsection{Potential Future Directions}

\textit{More powerful monotone iteration operators.} In Chapter 3, we gave a method for producing non-linear invariants, which can contain polynomial, exponential, and logarithmic terms. A downside was that the resulting method is not monotone. This issue was partially rectified by the method of Chapter 6, where we adapted the techniques from Chapter 3 to create a monotone analysis that is able to produce non-linear invariants. However, the method of Chapter 6 is only able to produce invariants containing polynomial terms, but not exponential or logarithmic terms. The methods are fairly close, and so this raises the question of why can’t the method of Chapter 6 be upgraded using the techniques of Chapter 3 to produce more expressive invariants. Part of the issue would be the resulting logic used to specify monotonicity. In the case of Chapter 6, we used \texttt{LIRR} [88] as the underlying logic, but \texttt{LIRR} does not currently have support for exponential or logarithmic terms. It is not immediately obvious how to add these additional terms to \texttt{LIRR}. At a high-level, the issue is that while polynomial arithmetic is difficult to reason about, it is relatively easy to specify the semantics. In
contrast, exponential and logarithmic terms are both difficult to reason about and difficult to formally specify.

In Chapter 6, our claim of monotonicity was

$$\text{If } F \models_{\text{LIRR}} G \text{ then } F^\oplus \models_{\text{LIRR}} G^\oplus.$$ 

The “star” in that method only produces polynomial equations as invariants. That is $F^\oplus$ consists of polynomial equations. However, the dynamics of $F$ also potentially specify many exponential invariants as well. In fact, in the course of computing $F^\oplus$, we actually compute those exponentials in Alg. 15. But because, LIRR is only able to reason about polynomials, rather than utilizing the full closed-form summary we must project exponentials out in our final closed form. In other words, the issue is that our reasoning about the closed forms we would potentially produce is not strong enough. For more information, see the history of the related problem known as “Tarski’s high-school algebra problem”.

Thus, one potential way forward would be to try and develop an extension of LIRR that handles exponential terms. It would be easy to include exponentials in LIRR as uninterpreted functions. However, we would most likely need some additional reasoning power in order to show monotonicity. The essential challenge of this direction is to play the balance of coming up with a set of axioms for exponentials that are strong enough to reason about the closed-forms our iteration operator produces, but at the same time be weak enough in order for us to design an effective method around.

One key result is the algorithm of Kauers and Zimmermann [82] which allows us to compute the algebraic relations of exponential sequences. This algorithm would most likely need to be combined with other simple exponential axioms, such as $x^a \cdot x^b = x^{a+b}$, and a more robust method for applying inference rules as compared with the wedge domain. In combination, we would have a reasonable method for producing and reasoning about exponentials. What would then be required is to define a logic akin to LIRR for which that method would yield complete reasoning.

More Complicated Refinements. The refinement method of Chapter 4 refines general path expressions by refining expressions at each loop level. This means that the method of Chapter 4 is limited to what type of refinements it can make. On the other hand if we had a general decision procedure for pre-Kleene algebras, we could potentially design much more exotic refinements. That is, suppose we had a method to determine whether $\models_{\text{PKA}} R' \leq R$ for arbitrary expressions $R$ and $R'$. If we had such a method, we would be free to take an expression $R$ and perform exotic rewrites to some potentially better $R'$. We could then check whether $\models_{\text{PKA}} R' \leq R$. If the answer is yes, then we know that the analysis of $R'$ is no worse than the analysis of $R$.

Due to the simplicity of axioms of pre-Kleene algebras (all the axioms are equations) such a reasonable decision procedure most likely exists. So a potential future direction would be to find such a decision
procedure and then design a method that either rewrites an expression $R$ with respect to the decision procedure, or design a guess and check method that generates possible refinements and then checks the refinement using the new decision procedure.

More powerful methods for the OSB problem. Another possible future direction would be adopting our reduction techniques of Chapter 5 to more specific settings. For example, our polyhedron reduction algorithm works for conjunctive linear formulas. We suspect that a combination of polyhedral reduction with a convex hull procedure could be used to give a complete and efficient solution to the OSB problem for arbitrary linear formulas.

In this context the convex hull is the analogue of our heuristic saturation step. In the case of convex hull a completeness argument can be made, which combined with our reduction algorithm would give a completeness result for linear formulas. However, the convex hull is often a very expensive method. We suspect that our polyhedral reduction algorithm could be combined with a convex hull procedure to yield a much more efficient method. The idea is that our reduction method projects polyhedra down to a much smaller dimension, where convex hull operations become more feasible.

8.3 Concluding Remarks

In this dissertation, we presented results aimed at increasing the expressiveness and precision of compositional program analyses. Towards the goal of increasing expressiveness, we presented methods for producing invariants that contain non-linear terms. Through our experiments, we found that these more expressive invariants have applications in bounding the resource usage of many popular algorithms (see § 3.7 and 7.2). Furthermore, our experiments also showed how non-linear invariants are also useful in establishing the functional correctness of programs (see § 3.7, 5.7 and 6.7).

During this program of research, the goal emerged of producing analyses that are more robust. Towards this goal, we explored the property of monotone program analyses. Specifically, we examined how monotonicity could be used to make compositional analyses more precise. Furthermore, we ultimately united these three topics to produce a compositional analysis that is monotone for polynomial programs and produces non-linear invariants.

In spite of the theoretical difficulties of non-linear arithmetic, the results of this dissertation show not only the utility of producing non-linear invariants, but also that we do not necessarily have to sacrifice analysis robustness when addressing a more challenging setting. We hope that these results encourage more work that tackles difficult theoretical settings, but also encourages a thoughtful examination of the new techniques developed, with the goal of establishing the meta-level properties of the resulting analyses.
Bibliography


