Enhancing Algebraic Program Analysis

by

Jason Breck

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

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2020

Date of final oral examination: 07/17/2020

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

Thomas Reps, Professor, Computer Sciences
Aws Albarghouthi, Assistant Professor, Computer Sciences
Loris D'Antoni, Assistant Professor, Computer Sciences
Zachary Kincaid, Assistant Professor, Computer Science, Princeton University
Steffen Lempp, Professor, Mathematics
To my parents, Sandy and Jim
Acknowledgments

My path through graduate school was guided by my adviser, Tom Reps, and throughout the process, my scientific output benefited greatly from his long experience and encyclopedic knowledge of the literature of program analysis. Zak Kincaid—an undeniable genius who’s just beginning his career—was almost like a second adviser to me for several years. I want to thank these two professors, along with the others on my doctoral committee, for their many comments and suggestions along the way.

In computer science, it is common for a dissertation to be composed primarily of previously published material. What is just a little unusual about my case is that almost all of my work was performed as part of a team of researchers, including the work that forms the basis of this dissertation. The research team behind my dissertation work included Thomas Reps, Zachary Kincaid, John Cyphert, Jake Silverman, Ashkan Forouhi Boroujeni, Emma Turetsky, and Stephen Lee. During other research projects, I worked with even more collaborators, including Aditya Thakur, Antoine Miné, Loris D’Antoni, and Qinheping Hu. I would be remiss if I did not thank these very talented collaborators who share in the credit for this research.

During several “engagement” meetings in Washington, D.C., I got to work side-by-side with a different team that included, at various times, Quentin Carbonneaux, John Cyphert, Antonio Flores Montoya, Julien Henry, Lucja Kot, and Alexey Loginov. The experience of close collaboration was enjoyable every time. I want to thank the team for their camaraderie and good spirits during our strange little adventures.

I was incredibly fortunate to study alongside a collection of peers who were always willing to share their kindness, good advice, and sense of humor with each new student who joined the program. More than anything else, I am grateful to them for the many opportunities to commiserate about the everyday indignity and grief of graduate study. I want to thank Zhicheng Cai, Stephen Lee, and Michael Vaughn for all the conversations over the years. I hope that, once or twice at least, I managed to make their burdens just a little lighter, as they did for me. I also need to thank the students who gave me great memories at social events, like Calvin Smith (for the conference-adjacent adventures in Italy and Portugal), and Sam Drews (for hosting PL band), just to name a few.
I want to thank my “older brothers” in the research group, Aditya Thakur, Venkatesh Srinivasan, and Tushar Sharma, for the advice they passed on to me: I followed some of it; I wish I had followed even more of it; and, at least I passed on the rest of it to those who came after me.

Like a crawl through a long and toxic tunnel, a graduate-school career is often darkest at its middle point; but, for me, that was also when a glimmer of light and redemption appeared: halfway through my studies, the Stork of Academia brought two “younger brothers,” John Cyphert and Jordan Henkel, into our research group. From that point on, I was sustained by real friendships with John and Jordan that were far and away the best part of my time in Madison. I wish them the best in graduate school and beyond.

I come from a family that always made education a top priority. I am very grateful for the sacrifices that they made, and for the context of relative privilege around them, without which I would not have reached this point. I want to thank Art and Elaine Breck for making me feel at home out here in the Central Time Zone, especially when my parents couldn’t easily come to visit me. I want to thank my brother, Eric, who I have been looking up to, more than anybody else, for my whole life. I have followed his example in so many ways, including the attainment of a C.S. PhD, and my life is so much the greater for it. Most of all, I want to thank my parents, who are the best parents that anybody could possibly ask for. Their encouragement, wisdom, and love saw me through this time in my life, and words can never express my gratitude for everything that they have done for 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 ONR under grants N00014-17-1-2889 and N00014-19-1-2318. and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from WARF.
Contents

Contents iv

List of Tables vii

List of Figures viii

Abstract x

1 Introduction 1

1.1 Conventional Approaches to Program Analysis 2
1.2 Algebraic Program Analysis 5
1.3 Compositional Recurrence Analysis 6
1.4 Overview of the Results in the Dissertation 9
1.5 Thesis Organization 15

2 Background 16

2.1 Recurrence Relations 16
2.2 Compositional Recurrence Analysis 17
2.3 Algebraic Program Analysis 18
2.4 Newtonian Program Analysis 22

3 Interprocedural Compositional Recurrence Analysis 25

3.1 Introduction 25
3.2 Problem Statement 27
3.3 Technical Details 28
3.4 Implementation and Experiments 46
3.5 Related Work 50
Bibliography 163
List of Tables

3.1 Column 2 shows the breakdown of programs into ones with valid and invalid assertions. “X/Y” means that X is the number of programs containing valid assertions and Y is the number containing invalid assertions. The total number of programs is X + Y. The two columns for each tool show the running time (in seconds) and the number of assertions proved. In each row, the fastest time and the greatest number of assertions proved are shown in bold font. ................................................................. 47

4.1 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. ................................................................. 88

5.1 Column 2 shows the actual asymptotic bound for each benchmark program. Columns 3-4 show the asymptotic complexity of the bounds determined by CHORA and ICRA. Column 5 gives the source of the benchmark as well as the published bound from that source. “n.b.” indicates that no bound was found. For each benchmark, only one other tool’s bound is shown, even if more than one such tool is capable of finding a bound. ................................................................. 120

5.2 Five analysis tools, along with the results of assertion-checking experiments using the benchmarks shown in Fig. 5.5. A ✓ indicates that the tool was able to prove the assertion within 900 seconds, and an X indicates that it was not. We also show the time required to analyze each benchmark. ............................. 123
## List of Figures

1.1 Matrix multiplication using Strassen’s Algorithm ........................................... 2
1.2 An example program and its CFG. ................................................................. 6
1.3 A cubic-time loop nest ................................................................. 8
1.4 The main contributions of this dissertation, as compared with prior techniques. ......... 10
1.5 An execution of the Fibonacci function ..................................................... 13

3.1 An implementation of the Fibonacci function .............................................. 36
3.2 $\text{factor}_X(E)$ ............................................................................. 39
3.3 A recursive-descent parser. ................................................................. 41
3.4 (a) Recursive procedure to show upper and lower resource bounds; (b) `speed_popl10_simple_single_2.c` from C4B. ............................................................. 49

4.1 Four examples: (a) cubic-time loop nest; (b) binary search; (c) rotation in the $x, y$ plane; (d) absence of information flow. ........................................... 55
4.2 Recurrences created for the nested loops in Fig. 4.1(a). ..................................... 56
4.3 Inference rules for implied inequalities .......................................................... 67
4.4 A loop, along with its transition formula and transition wedge ............................ 72
4.5 Translation function $\mathcal{T}_n : F \rightarrow R$. In Eqn. (4.10) $K \subseteq \mathbb{N}$ is some finite index set. .......................................................... 79

5.1 Example program `subsetSum`. The diagram on the right shows a timeline of a height $(h+1)$ execution of `subsetSumAux`. $b_2(h+1)$ is related to the increase of $n\text{ticks}$ between the pre-state (label 1) and the post-state (label 6). $b_2(h)$ is related to the increase of $n\text{ticks}$ between (2) and (3) and also between (4) and (5), i.e., between the pre-states and post-states of height-$h$ executions. ........................................... 99
5.2 Source code for the non-linearly recursive procedure differ, and a depiction of a possible tree of recursive calls differ. The number inside each vertex indicates the value of the parameter n at that execution of differ. The minimum depth M at which a base case occurs in this recursion tree is 3. The upper region of the tree, which includes all vertices at depth less than or equal to M, is shown with bold outlines.

5.3 Results of running CHORA and four other tools on the SV-COMP19 recursive directory of benchmarks. Each point indicates a benchmark containing assertions that a tool proved to be true, and the amount of time taken by that tool on that benchmark.

5.4 Source code for three programs from the SV-COMP suite: Ackermann01, RecHanoi01, and McCarthy91.

5.5 Source code for three non-linearly recursive programs containing assertions.

6.1 Loop with periodic behavior, and its associated matrix recurrence.

6.2 The CHC program of Ex. 6.1 after being translated into a weighted control-flow graph. Bold names represent mutable (local or global) variables, and non-bold names represent existentially quantified temporaries. Each edge is weighted with a call or with a statement consisting of an assignment and possibly also an assumed constraint. Notice that each CHC c becomes one path p in the graph, and the constraint of c appears as an assumed constraint on the final edge of p.

6.3 A modified version of the usual recursive Fibonacci procedure, obtained by splitting the procedure into two procedures in such a way that the length of a root-to-leaf path in the recursion tree may be constrained more tightly than in the original procedure.
Abstract

Many programs have important functional-correctness properties that are most naturally expressed as sophisticated mathematical relationships between numerical variables. In addition to functional-correctness properties, many programs have important numerical properties that characterize their worst-case usage of resources such as time or memory. Proving either of those kinds of properties is a challenge for automated-reasoning systems, because of both theoretical difficulties (e.g., the undecidability of non-linear integer arithmetic) and practical difficulties. Classical approaches to these problems include template-based methods, as well as iterative, fixed-point-finding analyzers based on abstract interpretation and abstraction refinement.

This dissertation applies an alternative framework, called algebraic program analysis, to the problem of proving numerical properties of programs. In this framework, the main steps of the analysis of a program are encapsulated by an algebraic structure (i.e., a carrier set and a collection of operations), and results are obtained by evaluating expressions that are constructed from the operations of that algebraic structure. More specifically, this dissertation builds upon an instance of that framework called Compositional Recurrence Analysis (CRA), in which the loops of a program are analyzed by finding and solving recurrence relations. This dissertation explores several lines of research that enhance CRA.

The first line of research produced a new analysis technique that we call Interprocedural Compositional Recurrence Analysis (ICRA). ICRA applies recurrence-solving in a uniform way to both (i) loops, and (ii) linearly recursive procedures, i.e., procedures that make at most one recursive call along any path through the procedure body. Furthermore, ICRA analyzes non-linearly recursive procedures using a technique that borrows ideas from Newton’s method and Gauss-Jordan elimination. Experiments show that ICRA, when applied to a collection of assertion-checking tasks, has broad overall strength compared with several state-of-the-art software model checkers.

The second line of research improves upon ICRA’s analysis of non-linear mathematical relationships, allowing it to find invariants that include polynomial, exponential, and logarithmic relationships between program variables. The enhanced version of ICRA handles a wider class of recurrence relations (i.e., C-finite recurrences). One component of this enhancement is a new abstract domain called the wedge abstract domain,
which is a modified version of the abstract domain of polyhedra in which the polyhedra have additional dimensions that correspond to non-linear terms over program variables. The operations of the wedge domain use Gröbner-basis methods for handling polynomial equations, and a collection of inference rules for handling non-linear inequalities. Another component of the enhancement is a new recurrence solver based on the operational calculus. Experiments show that the enhanced version of ICRA is able to prove non-linear assertions, and bounds on a program’s usage of resources, such as time or memory.

The third line of research improves upon ICRA’s ability to analyze non-linearly recursive procedures, such as divide-and-conquer algorithms. This enhancement combines two streams of ideas from the literature on generating numerical invariants, namely: (1) template-based methods, and (2) recurrence-based methods. The new analysis technique uses a new kind of template, which we call a hypothetical summary, in which the unknowns are functions, and the analyzer finds and solves recurrence constraints on those unknowns. Experiments show that this analysis technique is effective at proving assertions and finding resource-usage bounds for non-linearly recursive procedures. For instance, it is able to show that (i) the time taken by merge-sort is $O(n \log(n))$, and (ii) the time taken by Strassen’s algorithm is $O(n^{\log_2(7)})$. 
Chapter 1

Introduction

Software bugs are everywhere. They lead to great economic costs, and sometimes even to loss of life. Moreover, some bugs are security vulnerabilities that can be exploited by adversaries to take control of computer systems or steal confidential information. Given that software bugs are so important, it is natural to ask for a way to find bugs or to verify their absence.

This dissertation describes program-analysis techniques that can prove numerical properties of programs, i.e., properties of the values of numerical program variables or auxiliary quantities, such as the amount of time, memory, or other resources used by a program. Such analysis techniques have several applications that are related to proving the absence of software bugs. One application involves proving that a program meets a provided specification: for example, we may wish to prove that an array-index variable is always less than the length of the array, that a variable in the control program of a mechanical device (e.g., an aircraft) stays within a safe range, that a variable does not undergo arithmetic overflow, or that the program does not divide by zero.

Another application is finding bounds on a program’s usage of resources in terms of the program’s inputs. Notably, this second application, which is called resource-bound analysis, has important security implications, because a program’s resource usage is relevant to two kinds of security vulnerabilities: (i) algorithmic-complexity vulnerabilities, in which a program uses resources excessively in response to some inputs, and (ii) side-channel leaks, in which confidential information is revealed by the program’s resource usage (e.g., the time that a server takes to respond to a request.)

To motivate the specific techniques of this dissertation, we now describe a challenging problem that the techniques are able to solve. We wish to automatically analyze the running time of an implementation of Strassen’s Matrix-Multiplication Algorithm [Strassen, 1969]. The algorithm takes as input two $n \times n$ matrices $A$ and $B$, and multiplies them to produce a matrix $C$ by using a divide-and-conquer strategy. The matrices $A$
\[ M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \]
\[ M_2 = (A_{21} + A_{22})B_{11} \]
\[ M_3 = A_{11}(B_{12} - B_{22}) \]
\[ M_4 = A_{22}(-B_{11} + B_{21}) \]
\[ M_5 = (A_{11} + A_{12})B_{22} \]
\[ M_6 = (-A_{11} + A_{21})(B_{11} + B_{12}) \]
\[ M_7 = (A_{12} - A_{22})(B_{21} + B_{22}) \]

\[ C_{11} = M_1 + M_4 - M_5 + M_7 \]
\[ C_{12} = M_2 + M_4 \]
\[ C_{12} = M_3 + M_5 \]
\[ C_{22} = M_1 + M_3 - M_2 + M_6 \]

Figure 1.1: Matrix multiplication using Strassen’s Algorithm

and B are divided into half-size blocks and those blocks are multiplied recursively to obtain the four half-size blocks comprising C, as shown in the following equation:

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix} =
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}.
\]

A naïve divide-and-conquer algorithm would perform eight half-size matrix multiplications \((A_{ij}B_{jk})\) for \(i, j, k \in \{1, 2\}\), and would therefore have running time that is \(O(n^{\log_2 8}) = O(n^3)\). By contrast, Strassen’s Algorithm obtains the same result while performing only seven half-size matrix multiplications, as shown in Fig. 1.1, and therefore it has running time that is \(O(n^{\log_2 7})\).

The algorithm has several features that make it challenging to analyze automatically. For example, it has a complicated control-flow structure that involves recursion, conditional branches, and nested loops; also, the property to be proved—namely, the fact that the running time is \(O(n^{\log_2 7})\)—is mathematically sophisticated. Despite these challenges, in §5.5, we show that our analysis techniques are sufficient to prove a bound on the running time of a C implementation of Strassen’s algorithm.

1.1 Conventional Approaches to Program Analysis

In this section, we set the stage for the remainder of the dissertation by giving a high-level overview of some conventional approaches to program analysis. We also discuss several of the challenging aspects of program-analysis problems that motivated the design of both new and old approaches.

Fundamental limitations on program analysis were established in some of the earliest research on the subject, dating back to Turing’s work on the halting problem [Turing, 1936]. Turing demonstrated that even the problem of determining whether a program halts is undecidable. That is, it is impossible to design a program analyzer that, given any program \(P\) as input, returns true if \(P\) eventually halts and false otherwise. Moreover, this limitation is not specific to the halting problem: Rice’s Theorem tells us that all non-trivial semantic properties of programs are likewise undecidable [Rice, 1953]. However, these theoretical results do
not imply that program analysis cannot be useful; rather, the results imply an important property that every
analyzer will have, namely, the analyzer will sometimes fail to definitively prove or disprove the property of
interest.

The conventional approaches to program analysis that are most relevant to this dissertation are based upon
*abstracts* of programs, that is, simplifications of a program that keep some details of the program’s behavior
while ignoring others. The standard framework for building program analyses using such abstractions is
called *abstract interpretation* [Cousot and Cousot, 1977]. In the context of abstract interpretation, we use the
term *concrete state* to refer to a snapshot of program execution in which each program variable or memory
location has one particular value. Rather than working with concrete states directly, abstract interpretation
works with *abstract states* (also called *abstract values*), which are descriptors that may represent multiple
concrete states at once. An example of an abstract state is \(0 \leq x \leq 2 \land 5 \leq y < \infty\); this abstract state
represents all concrete states where the program variable \(x\) is between 0 and 2, and the program variable \(y\) is
at least 5. By using abstract states, an analyzer can “run the program in aggregate,” and reason about all
reachable states of the program, even though the analyzer may consider a relatively small number of abstract
states and perform a relatively small amount of computational work.

Abstract interpretation is a framework for constructing program analyzers. Each instance of the framework
uses some *abstract domain*, which is a set of abstract states along with a collection of operations that work
with elements of the set. Usually, the elements of an abstract domain are abstract states that describe the
associated set of concrete states in a particular constrained manner. For example, each element of the *abstract
domain of intervals* describes the value of each program variable as lying within some interval. Usually, there
are some sets of concrete states that cannot be described exactly by any of the elements of the abstract domain.
For example, consider a pair of concrete states in which the integer variable \(x\) is 0 in the first state and 2 in
the second state. The smallest interval that contains both values is \([0, 2]\), but this interval also contains 1.
Therefore, to describe that set of two concrete states using the abstract domain of intervals, we need to use an
abstract state that corresponds to a *superset* of the desired set of concrete states. This example illustrates the
loss of precision—which we generally wish to minimize—that results from the use of abstract states, rather
than concrete states.

An analysis based on abstract interpretation consists of a series of steps that ultimately obtains, for each
program location \(\ell\), an abstract state that represents a superset of all concrete states that the program may
reach at \(\ell\). After obtaining those abstract states, the analyzer may use them to prove properties of the program.
For example, suppose that, at some particular program location, the analyzer obtains an abstract state that
describes \(x\) as lying within the interval \([0, 2]\). This abstract state is sufficient to prove that \(x\) can be safely used
to index into an array of size 3. However, this state is neither sufficient to prove, nor to disprove, that it is safe
to divide by \( x \), because the abstract state is consistent with concrete states where \( x \) is zero and concrete states where \( x \) is non-zero. In essence, abstract interpretation sidesteps the problem of undecidability because it sometimes returns indefinite results, as in the case just mentioned.

Program analysis methods, including those based on abstract interpretation, need to be able to contend with several kinds of control-flow structures that appear in programs, such as conditional branches, loops, and (possibly recursive) procedure calls. In classical abstract interpretation, an operation called join is used to bring together the abstract states at control locations where different execution paths come together (e.g., the location after an if-statement when the then-branch and else-branch come together.) Often, using the join operation results in some imprecision, because the operation must produce an element of the abstract domain that encompasses all of the concrete states described by the abstract states that are being joined together. This loss of precision is illustrated by the earlier example in which we determined the smallest interval containing 0 and 2. In this case, the join of \([0, 0]\), which describes the set \(\{0\}\), and \([2, 2]\), which describes the set \(\{2\}\), is \([0, 2]\), which describes the set \(\{0, 1, 2\}\). In contrast, \(\{0\} \cup \{2\} = \{0, 2\}\).

The analysis of loops is a particularly challenging part of program analysis. In classical abstract interpretation, loops are analyzed using a process that iteratively propagates abstract states around each loop until those abstract states converge. During such an analysis, a join operation is used every time an abstract state is propagated along a loop back-edge to the loop head. In some cases, this kind of iterative loop analysis will take a very long time to converge, or will never converge at all. To mitigate the problem of convergence, some analyses apply an additional operation called widening to the abstract states obtained by successive rounds of the analysis. Widening operations, which often work by deliberately reducing precision, ensure that the analyzer makes a sufficiently large amount of progress toward convergence on each round. These join and widening operations are important sources of imprecision in the classical analysis of a loop.

Programs that consist of multiple procedures pose yet more challenges for program analyzers. In general, a procedure is called in more than one calling context in a program, and if the analysis of the program is to obtain precise results, the analyzer should distinguish these different contexts to some extent. There exist program-analysis frameworks for handling this problem of calling context, such as the frameworks of [Sharir and Pnueli, 1981]. One general approach is to consider abstractions of program-state transformers, rather than abstractions of program states. By working with transformers, which are abstractions that are sensitive to the context in which they are applied, the analyzer can distinguish the behaviors of any given procedure in different contexts. Then, the analysis of a multi-procedure program can be accomplished by an iterative process in which abstract transformers are propagated through the program until the transformers converge to a sound over-approximation of the program’s behavior. A particularly challenging case is the analysis of programs containing recursive procedures. When analyzing recursive procedures, the analyzer faces a
problem analogous to the problem posed by loops, namely that an iterative analysis might not converge, and various methods that are designed to ensure convergence may produce results that are very imprecise.

The analysis of numerical properties poses another set of challenges, apart from the challenges of analyzing control-flow structures. Consider the problem of analyzing the amount of time or other resources used by a typical program. Many programs have polynomial, exponential, or logarithmic resource usage. However, reasoning about such non-linear mathematical relationships is difficult. A specific challenge is that the theory of non-linear integer arithmetic is undecidable, although many algorithms are known, such as the algorithms typically employed by computer-algebra systems. Because of these challenges, many past program-analysis developers have chosen to focus on decidable theories, such as the theory of linear integer arithmetic, and associated abstract domains, such as the abstract domain of (convex) polyhedra, in which an abstract state (or state transformer) is a conjunction of inequations over linear arithmetic. However, the restriction to linear arithmetic puts many problems, such as those of resource-bound analysis, out of scope for those analyses.

1.2 Algebraic Program Analysis

The dissertation concerns an approach to static analysis called algebraic program analysis [Tarjan, 1981], which uses an algebraic structure consisting of a carrier set and a collection of operators, to encapsulate many of the main steps of the analysis. The elements of the carrier set, which we call summaries, are representations of the behavior of fragments of a program; in other words, each summary is a representation of the program-state transformation performed by a program fragment (i.e., it is an abstraction of a program-state transformer.) At the outset of such an analysis, we assume that we have summaries for the elementary instructions of the program, such as assignments to variables. During the analysis, the goal is to find summaries of larger fragments of the program, which are composed of elementary instructions and various control-flow structures, such as loops, branches, and procedure calls.

To obtain summaries for those larger fragments, we use algebraic operators that correspond to control-flow structures. For example, conditional branches (i.e., if-then-else statements) in a program correspond to instances of an algebraic operator called combine ($\oplus$); loops (i.e., for and while statements) correspond to instances of an iteration operator ($\ast$); and, sequential composition of a statement $S_1$ with a statement $S_2$ (i.e., “$S_1; S_2$”), corresponds to an occurrence of an operator called extend ($\otimes$). Using these operators, we can construct and evaluate algebraic expressions that build up summaries of larger program fragments from summaries of their parts. We illustrate this process using an example program.

Fig. 1.2 shows a program and its control-flow graph. The elementary instructions of the program appear as the labels on the edges of the control-flow graph. We wish to construct a summary of all possible paths
Figure 1.2: An example program and its CFG.

through the while loop, or in other words, all possible paths from the control location labeled $X_0$ in the control-flow graph, to the control location labeled $X_3$. We can use the aforementioned three algebraic operators to build up an expression $R$ that represents all such paths:

$$R \overset{\text{def}}{=} \left[ i > 0 \otimes (x \leftarrow x + i \oplus y \leftarrow y - i) \otimes i \leftarrow i - 1 \right]^{\ast} \otimes i \leq 0.$$

Note that $R$ may be viewed as a regular expression over the alphabet of instructions that recognizes every\(^1\) path from $X_0$ to $X_3$. Tarjan’s path-expressions method [Tarjan, 1981] may be used to find such a regular expression for every node in a control-flow graph. Given interpretations for $\otimes$, $\oplus$, and $\ast$ as operators on summaries, we may then evaluate the path expressions to obtain summaries of the corresponding paths.

We now proceed to discussing the particular instance of the framework of algebraic program analysis that laid the foundation for the research described in this dissertation.

1.3 Compositional Recurrence Analysis

Compositional Recurrence Analysis (CRA) is an instance of algebraic program analysis that was originally introduced in [Farzan and Kincaid, 2015]. In this section, we give a high-level summary of CRA; more technical explanations of CRA and algebraic program analysis are given in Chapter 2.

CRA is applicable to programs that have a finite set $x$ of integer-valued program variables. A state of such a program is an assignment of values to the variables in $x$. A transition formula $\varphi(x, x')$ is a formula over $x$ plus a set of primed copies $x'$, representing the values of the program variables before and after executing a program fragment, respectively. For example, the instruction $x := x + i$ in Fig. 1.2 may be associated with the transition formula $x' = x + i \land y' = y \land i' = i$, indicating that variable $x$ increases by $i$, while $y$ and $i$ are unchanged. For any program fragment of interest, CRA can be used to find a summary, in the form of a transition formula, that over-approximates the transition relation (i.e., the relation on pre-state/post-state pairs) of that fragment.

\(^1\) Because $R$ contains an occurrence of the $\ast$ operator, it recognizes paths that go around the while loop any number of times; however, $R$ includes assume statements ($i > 0$ and $i \leq 0$) that are derived from the condition of the while loop; these statements allow us to give an interpretation of $R$ that takes into account the loop conditions.
To apply the operator $\oplus$ to two transition formulas $\varphi_1$ and $\varphi_2$, we take their disjunction $\varphi_1 \lor \varphi_2$. To apply the operator $\odot$ to $\varphi_1$ and $\varphi_2$, we compute the relational composition of the two formulas. That is, we replace the post-state vocabulary $x'$ of $\varphi_1$ with $x''$; we replace the pre-state vocabulary $x$ of $\varphi_2$ with $x'''$; and we then conjoin the two formulas and existentially quantify $x''$, which yields $\exists x''. \varphi_1[x''/x'] \land \varphi_2[x''/x]$. The iteration operator $*$ over-approximates the reflexive transitive closure of a transition formula by finding and solving recurrence relations. To explain the algorithm that CRA uses to evaluate $*$, we return to the example program shown in Fig. 1.2.

To compute a transition formula for the loop in Fig. 1.2, we begin by computing a transition formula that represents one iteration of the body of that loop, and then we apply CRA’s iteration operator. The transition formula for the loop body is as follows:

$$\varphi_{body}: (i > 0 \land ((x' = x + i \land y' = y) \lor (x' = x \land y' = y - i)) \land i' = i - 1)$$

The iteration operator of CRA is based on extracting and solving recurrence relations that are entailed by the loop-body formula. In this respect, the recurrence-based analysis that CRA applies to loops is different from many past approaches, in the sense that CRA’s analysis is non-iterative: whereas past approaches often worked by iteratively propagating an abstract state through a loop until convergence, CRA uses recurrence-solving to “leap” from a recurrence to a closed-form solution. Recurrence extraction is accomplished by using a symbolic abstraction procedure (such as the one from [Farzan and Kincaid, 2015], which appears in this document as Alg. 6) to find a convex polyhedron that over-approximates the loop-body formula. Then, recurrence equations and inequations can be read off from the constraint representation of that polyhedron. Closed-form solutions are computed by solving the recurrence relations symbolically. For the loop body $\varphi_{body}$, the iteration operator computes the following six recurrences and closed forms:

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Closed form</th>
<th>Recurrence</th>
<th>Closed form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i' = i - 1$</td>
<td>$i_{(k)} = i_{(0)} - k$</td>
<td>$y' \geq y - i$</td>
<td>$y_{(k)} \geq y_{(0)} - k(k-1)/2 - ki_{(0)}$</td>
</tr>
<tr>
<td>$x' \geq x$</td>
<td>$x_{(k)} \geq x_{(0)}$</td>
<td>$x' \leq x + i$</td>
<td>$x_{(k)} \leq x_{(0)} + k(k-1)/2 + ki_{(0)}$</td>
</tr>
<tr>
<td>$y' \leq y$</td>
<td>$y_{(k)} \leq y_{(0)}$</td>
<td>$x' - y' = x - y + i$</td>
<td>$x_{(k)} - y_{(k)} = x_{(0)} - y_{(0)} + k(k-1)/2 + ki_{(0)}$</td>
</tr>
</tbody>
</table>

(where $i_{(k)}$ denotes the value of $i$ on the $k$th iteration of the loop). Finally, the iteration operator introduces an existentially quantified non-negative variable $k$ (representing the iteration count of the loop), and conjoins the closed form of every recurrence:

$$\varphi_{body} : \exists k. k \geq 0 \land (i' = i - k) \land (x' \geq x) \land (y' \leq y) \land (y' \geq y - k(k-1)/2 - ki_{(0)})$$

$$\land (x' = x + k(k-1)/2 + ki_{(0)}) \land (x' - y' = x - y + k(k-1)/2 + ki_{(0)})$$

CRA has several advantages relative to classical abstract interpretation. CRA uses a very precise abstract domain, which is based on logical formulas, to characterize the behavior of loop-free, non-recursive fragments of code. By contrast, classical abstract interpretation usually employs an abstract domain that tightly constrains
the kinds of properties that may be proven to hold at any location in the code. In particular, CRA’s abstract domain has a join operation (based on logical disjunction) that loses no precision. In an intraprocedural analysis, CRA loses precision only at loops, when extracting a collection of recurrences that over-approximate the loop body. A small example that illustrates this loss of precision is as follows:

```java
for(int i = 0; i < 2*N; i++) { if (i % 2 == 0) { w = w + 2; } else { /* skip */ } }
```

A precise summary of the above for-loop would imply that \( w' = w + 2N \), because the loop iterates \( 2N \) times, and it alternates between adding \( 2 \) and \( 0 \) to \( w \) on successive iterations. However, CRA extracts an imprecise set of recurrences to characterize the loop body, because the class of recurrences that it is capable of extracting cannot express conditional behavior. Instead, CRA extracts recurrences that characterize each iteration of the loop body using inequations, based on the observation that the loop increases \( w \) by at least \( 0 \) and at most \( 2 \) on each iteration; thus, the recurrences describing \( w \) are: \( w' \geq w \) and \( w' \leq w + 2 \). Then, because the loop iterates exactly \( 2N \) times, CRA obtains the following less-precise loop summary: \( (w' \geq w \land w' \leq w + 4N) \).

Also note that, whereas classical abstract interpretation uses a widening operator during the analysis of a loop, CRA does not. This feature is an advantage of CRA’s approach, because in a classical analysis, the use of a widening operator can make the outcome of the analysis more difficult to explain or predict. For example, widening can lead to a situation where an improvement to the precision of some intermediate analysis results would lead to a loss of precision in the final results. Thus, CRA’s avoidance of widening provides a degree of robustness to its analysis.

However, CRA also has several kinds of limitations. Several limitations pertain to the class of recurrence relations that CRA uses to describe the state transformation performed by the body of a loop. Only recurrence relations that are expressible using linear arithmetic are considered by CRA. Thus, when analyzing a loop nest such as the one shown in Fig. 1.3, a problem arises.

```java
for(int a = 0; a < N; a++)
  for(int b = 0; b < N; b++)
    for(int c = 0; c < N; c++)
      ticks++;
```

Figure 1.3: A cubic-time loop nest

CRA begins by analyzing the inner loop, and then it proceeds to analyze the middle loop, and it produces precise summaries in both cases. However, the summary that CRA produces for the middle loop involves a quadratic increase to the `ticks` variable. Therefore, the body of the outer loop cannot be precisely described using linear arithmetic alone. The recurrences that CRA does extract during the analysis of the outer loop are very imprecise, and as a result, CRA ultimately obtains no upper bound on the increase to the `ticks` variable that results from executing the loop nest.

A further limitation is that CRA does not handle recurrences in which a program variable undergoes

\(^2\)The algebraic operator \( \oplus \) is interpreted as the join operator of CRA’s abstract domain.
multiplication (e.g., \(x := 2 \times x\)) or division (e.g., \(x := x / 2\)) on each iteration of a loop. For that reason, CRA is unable to characterize precisely the running time of a binary-search loop, which has logarithmic complexity. Thus, one challenge for the research described in this dissertation was to achieve more-precise results when analyzing a binary-search loop, or a loop nest like the one shown in Fig. 1.3, by extending CRA to handle recurrences, and produce closed-form solutions, that involve multiplication, division, and non-linear arithmetic, such as polynomials, exponentials, and logarithms.

Another important limitation of CRA relates to its analysis of recursive procedures. One of the most well-known applications of recurrence-solving in computer science is the analysis of the running time of recursive algorithms that is typically presented in an undergraduate algorithms course. CRA is not capable of performing that kind of analysis because it uses recurrence-solving to analyze loops, but not recursion. At a conceptual level, CRA may be seen as a purely intraprocedural analysis. However, the original implementation of CRA does have the capability to analyze recursive procedures by using an iterative technique that employs a widening operation to ensure convergence. This technique often produces imprecise results: many algorithms can be written using either a loop or a recursive procedure, and the original implementation of CRA will typically produce less-precise results when analyzing the recursion-based version. Thus, another challenge for the research described in this dissertation was to achieve more-precise results when analyzing recursive procedures by analyzing loops and recursion in a uniform way.

1.4 Overview of the Results in the Dissertation

I worked for several years as a member of a team of collaborators that included, at various times, Thomas Reps, Zachary Kincaid, John Cyphert, Jake Silverman, Ashkan Forouhi Boroujeni, Emma Turetsky, and Stephen Lee. This collaboration produced a series of papers that described enhancements of CRA [Kincaid et al., 2017, 2018; Breck et al., 2020b; Cyphert et al., 2019; Kincaid et al., 2019]. These enhancements, along with future work along the same lines, are the main subject of this dissertation. The goal of these enhancements was to modify CRA so that it produced more precise program-analysis results when applied to two key application domains.

The first domain to which we have applied these techniques is the checking of functional-correctness properties in programs. Specifically, we have created program-analysis tools based on CRA that can check whether an assertion in a program always holds when control reaches that assertion. CRA-based analyses can be applied to these tasks by computing a transition formula that is a summary of all paths through the program that start at the entry of the main procedure and end at the point of the assertion. If this transition formula is strong enough to imply that the asserted condition holds, then the analysis has demonstrated
Conventional Analysis of Recursion

-- Conventional abstract interpretation uses an iterative analysis method (with widening) for recursive procedures
-- CRA is essentially an intraprocedural analysis that represents loops, but not recursion, using recurrences

CRA-Based Analysis of Loops

-- Tarjan-style analysis (regular expression = a language of paths)
-- Abstract domain of formulas; precise for loop-free fragments
-- "**" operator evaluated by (i) extracting and solving linear recurrences; (ii) converting solutions to a formula
-- Summaries may include polynomials

(Ch. 3) Recursive Procedures (ICRA)

-- Linear recursion may be represented using tensored regular expressions
-- Linear recursion gives rise to recurrences indexed by recursion height
-- Non-linear recursion handled with a hybrid approach

(Ch. 4) Non-Linear Arithmetic

-- New wedge abstract domain for reasoning about non-linear arithmetic
-- Extraction of non-linear recurrences
-- Solver for C-finite recurrences based on operational calculus
-- Summaries may include polynomials, exponentials, and logarithms

(Ch. 5) Non-Linear Recursion (CHORA)

-- New analysis method for non-linear recursion that combines template-based and recurrence-based techniques
-- Non-linear recursion gives rise to recurrences indexed by the height of a tree of recursive calls

Tool Support

-- Two program-analysis tools (ICRA and CHORA) implementing the methods of Chs. 3-4 (ICRA) and Chs. 3-5 (CHORA)
-- Evaluation of the tools on a variety of assertion-checking and resource-bound-analysis micro-benchmarks

Figure 1.4: The main contributions of this dissertation, as compared with prior techniques.
that the assertion holds in every execution of the program. The second domain to which we have applied these techniques is the automatic generation of bounds on the usage of time, memory, or other resources by a program. (These are also called complexity bounds.) CRA-based analyses can be applied to these problems by computing summaries of the procedures of the program and then using these summaries to derive bounds on the resource usage of each procedure in terms of the parameters to the procedure. We investigated enhancements of CRA that allow it to find logically stronger summaries of program fragments, and thereby achieve more-precise final results in a wide variety of cases. More specifically, in the domain of functional-correctness properties, the goal of the enhancements was to prove that a larger number of assertions hold, and in the domain of resource-usage bounds, the goal was to prove tighter bounds on resource usage.

The first common thread of the research in this dissertation is the application of CRA-based techniques to programs that have complicated control-flow structures, such as recursive procedures, and loops in which the loop body contains multiple control-flow paths. In many cases, there is a natural way to apply recurrence-solving to characterize such programs, but the original version of CRA fails to find the recurrences because of the complex control flow used by the program. Enhanced versions of CRA can obtain more-precise results in such cases by exploiting the algebraic nature of the analysis. The key idea of such enhancements is that, after a regular expression is constructed, but before the expression is evaluated, we can use algebraic laws to justify a rearrangement of the expression into a distinct, but semantically equivalent, form. In some cases, this rearrangement introduces new algebraic operators into the expression. Finally, the rearranged expression is evaluated, and more-precise results are obtained. Note that this recurrence-based approach to program analysis is able to compute precise summaries of loops and recursion without having an iterative analysis process; in this respect, the approach is very different from conventional abstract interpretation, in which summaries of iterative program structures were computed using successive approximation.

A second common thread of the research is the enhancement of CRA in a way that allows it to work with more sophisticated recurrences, such as matrix recurrences, or matrix recurrences with non-rational eigenvalues, or recurrences that are expressed using non-linear mathematical operations. This kind of enhancement immediately leads to more precise analysis results. Furthermore, enhancements that work with more sophisticated recurrences can be applied not only to the analysis of loops, but also to the analysis of more complicated control-flow structures, such as recursive procedures, if they are used in combination with the other enhancements of CRA that we have studied.

In the remaining sub-sections of this chapter, we give an overview of the research work described in subsequent chapters of the dissertation.
1.4.1 Recursive Procedures (ICRA)

In Chapter 3, which is based on [Kincaid et al., 2017], we describe an enhancement of CRA that allows it to achieve a more precise analysis of recursive procedures. We call the resulting algorithm Interprocedural CRA (ICRA). ICRA is especially precise when analyzing linearly recursive procedures, that is, procedures in which any path through the procedure body makes at most one recursive call. In contrast, a non-linearly recursive procedure is one that has at least one path through its body along which more than one recursive call occurs.

One of the key ideas of ICRA is to introduce a tensor operator into the algebraic structure that is used during the analysis. The tensor operator is used to represent the body of a linearly recursive procedure, which can be decomposed into two fragments: (1) the part of the procedure that executes before the recursive call, and (2) the part of the procedure that executes after the recursive call. Observe that, when a linearly recursive procedure is called, the pre-recursion fragment executes some number \( K \geq 0 \) of times, then the base case of recursion executes, and then the post-recursion fragment executes \( K \) times. Because these two fragments execute the same number of times, each procedure-entry state that occurs “on the way down” to the base case should be paired with a corresponding procedure-exit state that occurs “on the way back up” from the base case. Then, recurrences can be used to describe the sequence of such state pairs. In this way, when analyzing linearly recursive procedures, ICRA uses recurrence-solving to relate the program states at different recursion depths. By contrast, non-linearly recursive procedures, which sometimes make more than one recursive call, cannot be decomposed into two fragments in the same way; for this reason, ICRA analyzes them using a different technique.

Research into ICRA began with the goal of creating a version of CRA that used a program-analysis framework called Newtonian Program Analysis via Tensor Product (NPA-TP) [Reps et al., 2016]. However, during the course of the research, we discovered that CRA did not meet all of the criteria to be used with that framework, because: (i) CRA has infinite ascending chains, and thus NPA-TP[CRA] would not be guaranteed to terminate; and, (ii) CRA does not have an effective equivalence procedure, and thus it is not possible, in general, to ascertain whether an NPA-TP[CRA] analyzer has reached a fixpoint. For these reasons, we created a new variant of NPA-TP that we call NPA-TP-GJ, which has some similarities to Gauss-Jordan elimination.

Our experiments with ICRA show that it has broad overall strength compared with several state-of-the-art software model checkers on a collection of assertion-checking tasks. Also, we show that ICRA is able to generate resource-usage bounds for a collection of programs in which the resource usage is a linear function.

---

3 Warning: We use the term “non-linear” in two different senses: non-linear recursion and non-linear arithmetic. Even for a loop that uses linear arithmetic, non-linear arithmetic may be required to state a loop invariant. Moreover, arithmetic expressions in the programs that we analyze are not limited to linear arithmetic: variables can be multiplied. The two uses of the term “non-linear” are essentially unrelated, and which term is intended should be clear from context. Chapter 4 discusses techniques for handling non-linear arithmetic, and Chapter 5 discusses techniques for handling non-linear recursion.
of the program’s input parameters.

1.4.2 Non-Linear Recursion (CHORA)

In Chapter 5, which is based on [Breck et al., 2020b], we describe another enhancement of ICRA that allows it to obtain more precise results when applied to non-linearly recursive procedures, such as the naive recursive Fibonacci function, or a divide-and-conquer algorithm.

When a non-linearly recursive procedure is called, the resulting set of executions of the procedure can be visualized as a tree of recursive calls (see Fig. 1.5), in which the leaf vertices are the base cases of recursion, and the children of any non-leaf vertex \( v \) are the executions of the procedures that are directly called by \( v \). The enhancement described in Chapter 5 allows the analyzer to find and solve recurrences that relate the program states at different vertices in such a tree of recursive calls. The idea is to classify the vertices of such a tree by their height, and find recurrences that relate the program states at height \( h \) to the program states at height \( h + 1 \).

The technique described in Chapter 5 can also be seen as the confluence of two streams of ideas in the literature on generating numerical invariants, namely: (1) template-based methods, and (2) recurrence-based methods. A template-based method begins with a template that contains unknown quantities, and finds invariants that match the template by extracting and solving constraints on the unknowns. A disadvantage of template-based methods is that they require fixing the set of terms that may appear in an invariant in advance. This disadvantage is particularly prominent for non-linear invariant generation, because the user must supply maximum degrees on polynomials, bases for exponents, etc.

On the other hand, recurrence-based methods are able to find sophisticated non-linear mathematical relations, including polynomials, exponentials, and logarithms, because such relations arise as the solutions to recurrences. However, a disadvantage of past recurrence-based invariant-generation methods, including CRA, is that they are primarily designed to handle a linear sequence of program states (or state pairs) such as the sequence of states that occur during the execution of a loop, or the sequence of state pairs that occur during the execution of a linearly recursive procedure. However, it is not obvious how to apply such techniques to the tree-shaped collection of states that occur during the execution of a non-linearly recursive procedure.

In Chapter 5, we combine these two approaches and obtain a technique that uses templates in which the unknowns are functions rather than numbers, and the constraints on the unknowns are recurrences. The
technique synthesizes invariants involving polynomials, exponentials, and logarithms, even in the presence of arbitrary control-flow, including any combination of loops, branches, and (possibly non-linear) recursion. We implemented this analysis inside a tool called CHORA. In our experiments, we tested CHORA’s abilities to prove assertions and generate resource-usage bounds. In particular, CHORA can prove that (i) the time taken by merge-sort is $O(n \log(n))$, and (ii) the time taken by Strassen’s algorithm is $O(n^{\log_2(7)})$.

1.4.3 Non-Linear Mathematical Relationships

As described in §1.3, several important limitations of CRA pertain to its handling of non-linear arithmetic. In Chapter 4, which is based on [Kincaid et al., 2018], we describe extensions of (I)CRA that allow it to more precisely handle non-linear mathematical relationships, including polynomials, exponentials, and logarithms, both in recurrences and the solution to those recurrences. We provide a collection of algorithms that allow the analyzer to handle such non-linear relationships. The original CRA used operations from the abstract domain of convex polyhedra when extracting recurrences from a transition formula. In Chapter 4, we introduce a new abstract domain that we call the wedge domain, which is an extension of the abstract domain of polyhedra in which the polyhedra have additional dimensions that represent non-linear terms over program variables. The wedge-domain operations use Gröbner-basis methods for analyzing polynomial equalities over program variables, and a collection of inference rules for reasoning about non-linear inequalities. We also describe a recurrence-solving technique based on operational calculus [Berg, 1967], which allows (I)CRA to handle a much wider class of recurrences.

In our experiments, we apply the techniques just described to a collection of assertion-checking and resource-bound-generation problems, and we achieve promising results.

1.4.4 Further Enhancements of CRA

In Chapter 6, we discuss several further enhancements of CRA. We begin by describing two papers with which I was somewhat less involved, although I was a co-author on these papers. In §6.1, we summarize [Cyphert et al., 2019], which improves the precision of CRA when analyzing programs that have loops in which there are multiple paths through the loop body. In §6.2, we summarize [Kincaid et al., 2019], which improves the precision of CRA when working with a certain class of matrix recurrences by finding alternative closed-form solutions to those recurrences that can be handled more precisely by subsequent steps of the analysis.

Next, we describe three avenues for future research related to CRA. In §6.3, we discuss some possible approaches for applying CRA to the problem of refuting, rather than proving, program properties. In §6.4,
we describe a way of applying the ideas of the previous chapters to the analysis of constrained-Horn-clause (CHC) programs, which is seeing increased use as a modeling language (or interface language) in static-analysis tools, where multiple programming-language front ends translate programs written in different languages to a common, CHC-based intermediate representation. In the final sections of Chapter 6, we describe a collection of ideas about how the CHORA analyzer might be generalized so as to produce more precise results or analyze a wider variety of program features.

1.5 Thesis Organization

Chapter 2 gives definitions and discusses background related to CRA and algebraic program analysis. Chapter 3 discusses ICRA, which generalizes CRA to handle (linearly) recursive procedures more precisely. Chapter 4 discusses enhancements to (I)CRA that allow it to more precisely handle non-linear mathematical relationships. Chapter 5 discusses CHORA, which extends ICRA to handle non-linear recursion more precisely. Chapter 6 discusses two more papers and future directions. Chapter 7 gives concluding remarks.
Chapter 2

Background

2.1 Recurrence Relations

*C-finite sequences* are a well-studied class of sequences defined by linear recurrence relations, of which a famous example is the Fibonacci sequence. Formally,

**Definition 2.1.** A sequence $s : \mathbb{N} \to \mathbb{Q}$ is *C-finite* of order $d$ if it satisfies a linear recurrence equation

$$s(k + d) = c_1 s(k + d - 1) + \ldots + c_{d-1} s(k + 1) + c_d s(k),$$

where each $c_i$ is a constant.

It is classically known that every C-finite sequence $s(k)$ admits a closed form that is computable from its recurrence relation and takes the form of an exponential-polynomial

$$s(k) = p_1(k)r_1^k + p_2(k)r_2^k + \ldots + p_l(k)r_l^k,$$

where each $p_i$ is a polynomial in $k$ and each $r_i$ is a constant. In the following, it will be convenient to use a different kind of recurrence relation to present C-finite sequences, namely, *stratified systems of polynomial recurrences*.

**Definition 2.2.** A stratified system of polynomial recurrences is a system of recurrence equations over sequences $x_{1,1}, ..., x_{1,n_1}, ..., x_{m,1}, ..., x_{m,n_m}$ of the form

$$\{x_{i,j}(k + 1) = c_{i,j,1}x_{i,1}(k) + \ldots + c_{i,j,n_i}x_{i,n_i}(k) + p_{i,j} \}_{i,j}$$

where each $c_{i,j,1}, ..., c_{i,j,n_i}$ is a constant, and $p_{i,j}$ is a polynomial in $x_{1,1}(k), ..., x_{1,n_1}(k), ..., x_{i-1,1}(k), ..., x_{i-1,n_{i-1}}(k)$. 
Intuitively, the sequences $x_{1,1}, \ldots, x_{1,n_1}, \ldots, x_{m,1}, \ldots, x_{m,n_m}$ are organized into strata ($x_{1,1}$ is the first, $x_{2,1}, \ldots, x_{2,n_1}$ is the second, and so on), the right-hand-side of the equation for $x_{i,j}$ can involve linear terms over the sequences in the $i^{th}$ strata, and additional polynomial terms over sequences of lower strata. It follows from the closure properties of C-finite sequences that each $x_{i,j}$ defines a C-finite sequence, and an exponential-polynomial closed form for each sequence can be computed from a stratified system of polynomial recurrences [Kauers and Paule, 2011]. The fact that any C-finite sequence satisfies a stratified system of polynomial recurrences follows from the fact that a recurrence of order $d$ can be implemented as a system of linear recurrences among $d$ sequences [Kauers and Paule, 2011].

Example 2.3. An example of a stratified system of polynomial recurrences with four sequences $(w, x, y, z)$ arranged into two strata ($(w, x)$ and $(y, z)$) is as follows:

$$
\begin{bmatrix}
  w(k+1) \\
  x(k+1) \\
  y(k+1) \\
  z(k+1)
\end{bmatrix}
= \begin{bmatrix}
  1 & 1 & 3 \\
  0 & 2 & 0 \\
  1 & 0 & 1 \\
  1 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
  w(k) \\
  x(k) \\
  y(k) \\
  z(k)
\end{bmatrix}
+ \begin{bmatrix}
  1 \\
  0 \\
  x(k)^2 + 1 \\
  3w(k) + x(k)
\end{bmatrix}
$$

This system has the closed-form solution

$$
\begin{align*}
w(k) &= w(0) + \frac{2^k - 1}{3} x(0) + k \\
y(k) &= \frac{4^k - 1}{3} x(0)^2 + y(0) + k \\
z(k) &= 3w(0) + \frac{4^k - 3k - 1}{9} x(0)^2 + (2^{k+1} - k - 1)x(0) + ky(0) + z(0) + 2(k^2 - k)
\end{align*}
$$

2.2 Compositional Recurrence Analysis

In this section, we provide some background about Compositional Recurrence Analysis, focusing on how it contrasts with alternative approaches.

Static analysis provides a way to obtain information about the possible states that a program reaches during execution, but without actually running the program on specific inputs. Two important approaches to static analysis are

- abstract interpretation, which conservatively overapproximates a program’s actions so that the analyzer can explore all the program’s reachable states (as well as some unreachable states);
• *symbolic analysis*, which uses formulas to create precise models of a program’s actions, but is usually forced to forgo an exploration that accounts for all reachable states.

*Abstraction refinement* is a widely used technique in static analysis that tailors an abstraction to a property of interest. Abstraction refinement enables analyzers to prove complicated properties, but can also cause them to fail to terminate. An interesting feature of CRA is that it can verify complicated properties *without* using abstraction refinement. CRA is guaranteed to terminate, and in practice it is generally faster than tools based on abstraction refinement. Moreover, it can *generate*—rather than just verify—-invariants, including resource bounds (see §3.4), and non-linear invariants (see Ex. 3.12).

One of the reasons why CRA is interesting is that it represents a completely different point in the design space of static-analysis tools, yet is competitive with software model checkers, in terms of precision, *without* doing any abstraction refinement. (Moreover, the techniques used in software model checkers have had about a 20-year head start over CRA.)

One of the essential differences between CRA and the classical iterative approach to invariant generation is that an iterative program analysis analyzes a loop by “running it in the aggregate”—i.e., by repeatedly running the loop on abstract descriptors that represent collections of states, until converging upon an abstract descriptor that over-approximates all reachable states of the loop. CRA analyzes a loop by finding a transition formula that describes the state change over the course of the loop body, and applies its iteration operator to extrapolate from one iteration to many; there is no repetitive process.

CRA is *compositional* in the sense that it computes the abstract meaning of a program by computing, and then combining, the abstract meanings of its parts.

• At the intraprocedural level, CRA makes use of Tarjan’s path-expression method to compose the meanings of sub-parts via the interpretations of $+, \cdot$, and $^*$ [Farzan and Kincaid, 2015].

• At the interprocedural level, each procedure is analyzed independently of its calling context to produce a summary that is used to interpret calls to the procedure.

2.3 **Algebraic Program Analysis**

In this section, we provide some background about algebraic program analysis, focusing on how it contrasts with alternative approaches.

Algebraic program analysis is an alternative to the classic iterative style of program analysis that takes an algebraic (rather than order-theoretic) approach to approximating repetitive behavior. We illustrate the difference between the algebraic and iterative styles of program analysis with the example shown in Fig. 1.2.
The program’s CFG can also be represented as the following recursive system of equations:

\[
S : \begin{cases}
X_0 = \epsilon + (X_2 \cdot i := i - 1) \\
X_1 = X_0 \cdot i > 0 \\
X_2 = X_1 \cdot (x := x + i + y := y - 1) \\
X_3 = X_0 \cdot i \leq 0
\end{cases}
\]

We can view \( S \) as a grammar for the paths through the program. For each variable (node, non-terminal) \( X_i \), we define \( \text{Paths}(X_i) \) to be the set of paths in the CFG that end at node \( X_i \), or equivalently the set of strings that are generated from non-terminal \( X_i \).

The problem that we are interested in is the classic problem from dataflow analysis of overapproximating the interpretation of all paths that start at the beginning of a program and end at a given program point [Kildall, 1973; Sharir and Pnueli, 1981]. We formalize this problem as follows. Suppose that we are given a space of path properties \( D \) and a function \( \llbracket \cdot \rrbracket : \Sigma \rightarrow D \) that maps each program instruction to a path property representing it (where \( \Sigma \) denotes the set of all program instructions). We suppose that \( D \) is equipped with a sequencing operator \( \otimes \) (with unit \( 1 \)) and a choice operator \( \oplus \) (with unit \( 0 \)). We define an approximation order \( \trianglerighteq \) on \( D \) by defining \( a \trianglerighteq b \) iff \( a \oplus b = a \), and lift \( \llbracket \cdot \rrbracket \) to paths by defining \( \llbracket p_1 \ldots p_n \rrbracket \triangleq \llbracket p_1 \rrbracket \otimes \cdots \otimes \llbracket p_n \rrbracket \). Our goal is

For each variable \( X_i \), compute some path property \( D(X_i) \in D \), such that for each path \( p \in \text{Paths}(X_i) \), \( p \) satisfies the property \( D(X_i) \trianglerighteq \llbracket p \rrbracket \).

In the terminology of abstract interpretation [Cousot and Cousot, 1977], we wish to compute a function \( D \) that abstracts the path semantics of the program within some abstract domain \( D \).

**Definition 2.4.** In the path semantics, the domain of path properties is the set of all path languages: \( D \triangleq 2^{\Sigma^*} \). The operations are: \( L_1 \oplus L_2 \triangleq L_1 \cup L_2; L_1 \otimes L_2 \triangleq \{ p_1 p_2 : p_1 \in L_1 \land p_2 \in L_2 \}; 0 \triangleq \emptyset; \) and \( 1 \triangleq \{ \epsilon \} \). The semantic function \( \llbracket a \rrbracket \triangleq \{ a \} \) maps each instruction \( a \) to \( \{ a \} \).

The classical way to establish that a function \( D \) is a solution to a system of equations \( S \) (in the sense that \( D \) abstracts the path semantics) is to show that \( D \) is a solution to a related system of inequations \( \llbracket S \rrbracket \) over the abstract domain \( D \), where \( \llbracket S \rrbracket \) is obtained by re-interpreting the regular operators appearing in \( S \) with the
corresponding operators in \(D\). For the \(S\) that corresponds to Fig. 1.2, we have

\[
\begin{align*}
\mathcal{S} : & \\
& \\
& \left\{ \\
& \begin{array}{l}
X_0 \uparrow \mathcal{L} (X_2 \odot [i := i - 1]) \\
X_1 \uparrow X_0 \odot [i > 0] \\
X_2 \uparrow (x \odot x := x + i) \odot [y := y - i] \\
X_3 \uparrow X_0 \odot [i \leq 0]
\end{array}
\right\
\end{align*}
\]

We call a solution to \([\mathcal{S}]\) a \textit{post-fixpoint solution} to \(S\). It is easy to see that if \(D\) is a post-fixpoint solution to \([\mathcal{S}]\), then \(D\) overapproximates the path semantics.

In an \textit{iterative} program analysis, a post-fixpoint solution is computed as the limit of a sequence of approximations. Define the \textit{Kleene iteration sequence} \(\langle D_n : \{X_1, X_2, X_3, X_4\} \rightarrow D \rangle_{n \in \mathbb{N}}\) as follows:

\[
\begin{align*}
D_0(X_i) &= 0 \\
D_{n+1}(X_0) &= 1 \oplus (D_n(X_2) \odot [i := i - 1]) \\
D_{n+1}(X_1) &= D_n(X_0) \odot [i > 0] \\
D_{n+1}(X_2) &= D_n(X_1) \odot (x := x + i) \odot [y := y - i] \\
D_{n+1}(X_3) &= D_n(X_0) \odot [i \leq 0]
\end{align*}
\]

When \(D\) has no infinite ascending chains, the sequence \(\langle D^n \rangle_{n \in \mathbb{N}}\) eventually stabilizes, and we may compute \(D\) effectively. If \(D\) fails to satisfy the ascending-chain condition, then we can use a widening operator to ensure convergence [Cousot and Halbwachs, 1978]. Observe that if we take \(D\) to be the \textit{path semantics}, then \(D(X_i) \overset{\text{def}}{=} \bigoplus_n D^n(X_i)\) coincides with \(\text{Paths}(X_i)\).

\textit{Algebraic program analysis} is an alternative to this iterative style. Rather than assuming the ascending-chain condition or a binary widening operator, we assume that the space of program properties is equipped with a unary iteration operator \(\ast\). We compute \(D\) using a two-step process. The first step is to apply Tarjan’s path-expression algorithm [Tarjan, 1981] to compute, for each variable \(X_i\) in the system, a regular expression that recognizes the path language \(\text{Paths}(X_i)\):

\[
\hat{\mathcal{S}} : \left\{ \\
& \begin{array}{l}
X_0 = \text{body}^* \\
X_1 = \text{body}^* \cdot i > 0 \\
X_2 = \text{body}^* \cdot i > 0 \cdot (x := x + i \odot y := y - i) \\
X_3 = \text{body}^* \cdot i \leq 0
\end{array}
\right\
\]

where \(\text{body} \overset{\text{def}}{=} i > 0 \cdot (x := x + i \odot y := y - i) \cdot i := i - 1\). The second step is to evaluate each path expression within \(D\), using the algebraic operators \(\otimes, \oplus, \text{and} \ast\) to interpret the regular expression operators \(\cdot, +, \text{and} \ast\), respectively:
where \([\text{body}] \overset{\text{def}}{=} [i > 0] \bigotimes \begin{pmatrix} x := x + i \\ y := y - i \end{pmatrix} \overset{\text{def}}{=} [i <= 0]. \) Again, if we take \(D\) to be the path semantics (where \(L^* \overset{\text{def}}{=} \bigcup_n L^n, L^0 = \bot, \) and \(L^{n+1} = L^n \otimes L\)), then \(D(X_i)\) coincides with \(\text{Paths}(X_i)\).

The essential difference between the iterative and algebraic approaches is that fixpoint computation is \textit{external} to the abstract domain in the iterative approach, but \textit{internal} in the algebraic approach: the iteration operator is a “built-in” method for solving a particularly simple class of recursive equations—i.e., \(X \mapsto a^*\) is a solution to the single-variable recursive equation \(\{X = 1 \oplus (X \otimes a)\}\). When a domain has infinite ascending chains, the two approaches impose different burdens on the analysis designer. The iterative approach requires designing a (binary) widening operator that ensures convergence by deliberate overapproximation. The algebraic approach requires designing a (unary) \(\star\) operator. The \(\star\) operator offers flexibility, because it need not necessarily be computed using an iterative process. (Although the \(\star\) operator of the path semantics is defined as the limit of an iterative process—i.e., \(L^* \overset{\text{def}}{=} \bigcup_n L^n\)—the analysis designer is free to design a \(\star\) operator that overapproximates the \(\star\) operator of the path semantics however they choose.)

The first step of the algebraic approach can be viewed as merely transforming the specification of the language(s) of paths in the program. For instance, the transformation of \(S\) to \(\hat{S}\) can be thought of as a transformation on grammars in which the language of each nonterminal is preserved. (In the case of \(\hat{S}\), regular operators are used on right-hand sides of productions.) The second step computes a sound (overapproximating) path property for each variable by interpreting its associated right-hand-side within a suitable algebra of path properties.

The method developed in Chapter 3 combines elements of both the iterative and algebraic styles of program analysis. First, the equations defining the paths through a program are transformed (§3.3.3). For \textit{linear} recursive systems, the transformation results in a non-recursive system that can be solved merely by interpreting the right-hand side of each equation within a suitable algebra (as in the algebraic method). For general systems, the transformation results in a recursive system of a restricted form, which can be solved using an iterative method (§3.3.3). Post-fixpoint solutions of the transformed system are generally \textit{not} post-fixpoint solutions of the original system; however, they do overapproximate the path semantics of the original system. The correctness of the approach is captured by the following principle:
**Observation 2.3.1. [Path-Preservation Principle].** A post-fixpoint solution of a transformed (but equivalent) system of equations overapproximates every path of the original equation system. \(\square\)

We use Obs. 2.3.1 in the soundness argument for the program-analysis algorithm presented in §3.3.

### 2.4 Newtonian Program Analysis

One of the original motivations for the research that became Chapter 3 was the idea of combining CRA with another approach to program analysis, called *Newtonian Program Analysis* (NPA) [Esparza et al., 2010]. More specifically, the idea was to combine CRA with a particular form of that approach, called *Newtonian Program Analysis via Tensor Product* (NPA-TP) [Reps et al., 2016]. Essentially, NPA-TP provides a way to solve an interprocedural analysis problem by solving a succession of simpler, intraprocedural analysis problems. In this section, we give a brief introduction to NPA-TP.

As in the previous section, we assume that we are analyzing a program using an abstract domain \(D\) that has operations \(\otimes, \oplus, \ast\), and annihilator \(0\). In this section, however, we also assume that \(D\) supports some additional operations, which we will explain below. Let \(a, b, c, d \in D\), and consider the following equation system, which represents a pair of procedures \(X_1\) and \(X_2\) such that \(X_1\) calls \(X_2\) and \(X_2\) is non-linearly recursive.

\[
X_1 = a \otimes X_2 \\
X_2 = d \oplus b \otimes X_2 \otimes X_2 \otimes c.
\] (2.1)

The Kleene iterates \((\kappa_1^{(i)}\) for \(X_1\) and \(\kappa_2^{(i)}\) for \(X_2\)) for this system can be described as follows. Let \(\kappa_1^{(0)} = 0\) and \(\kappa_2^{(0)} = 0\), and for \(j \in \{1, 2\}\) and \(i \geq 0\), let \(\kappa_j^{(i+1)}\) be derived by substituting \(\kappa_1^{(i)}\) in place of \(X_1\) and \(\kappa_2^{(i)}\) in place of \(X_2\) in the right-hand sides of the equations of Eqn. (2.1). Thus, we have:

\[
\begin{align*}
\kappa_1^{(0)} &= 0 \\
\kappa_1^{(1)} &= 0 \\
\kappa_1^{(2)} &= a \otimes d \\
\kappa_2^{(0)} &= 0 \\
\kappa_2^{(1)} &= d \\
\kappa_2^{(2)} &= d \oplus b \otimes d \otimes d \otimes c \\
\cdots & \cdots
\end{align*}
\] (2.2)

Kleene iteration obtains a solution to the equation system by iterating this substitution process until the sequence converges. Newtonian Program Analysis provides an alternative, and potentially faster, way to obtain a solution to the same equation system, by computing the elements of different sequences \(\nu_1^{(i)}\) and \(\nu_2^{(i)}\). Instead of merely substituting into the original equation system, the analyzer obtains the Newton iterates by using a “model” of that equation system that is linear, in the sense that it does not allow multiplication.
of variables (e.g., $X_2 \otimes X_2$). (Equivalently, one may think of the model as representing a modified set of procedures that are linearly recursive; that is, along any path through the body of any of the model’s procedures, there will be at most one recursive call.) Whereas one may think of Kleene iteration as using substitution into the equation system to obtain $\langle \nu_1^{(i+1)}, \nu_2^{(i+1)} \rangle$ from $\langle \nu_1^{(i)}, \nu_2^{(i)} \rangle$, one may think of Newton iteration as using the linear model to obtain $\langle \nu_1^{(i+1)}, \nu_2^{(i+1)} \rangle$ from $\langle \nu_1^{(i)}, \nu_2^{(i)} \rangle$. Roughly, the advantage of considering the sequence of Newton iterates, rather than the sequence of Kleene iterates, is that the Newtonian sequence typically converges faster, although more computational work is done at each step. The technique is called “Newtonian” because of an analogy with Newton’s method for finding roots of real-valued functions: the essence of both methods is to repeatedly make use of a linear model of the original function.

Newtonian Program Analysis provides an algorithm for transforming the equation system to obtain the linear model equation system. (See [Reps et al., 2016] for the full algorithm.) For the purposes of this discussion, we will show the output of the algorithm on Eqn. (2.1) as a new equation system in terms of variables $Y_1$ and $Y_2$. Roughly, the linear model is constructed as follows: whenever the equation system contains occurrences of variables, the algorithm produces a sum of several terms in which at most one variable occurrence is replaced by a variable of the linear model (e.g., $Y_2$) and all the rest of the occurrences are replaced by with the previous element of the Newton sequence (e.g., $\nu_2^{(i)}$ is used when computing $\nu_2^{(i+1)}$). The linear model for Eqn. (2.1) is shown below:

$$Y_1 = a \otimes \nu_2 \oplus a \otimes Y_2$$
$$Y_2 = d \oplus b \otimes \nu_2 \otimes \nu_2 \otimes c \oplus b \otimes Y_2 \otimes \nu_2 \otimes c \oplus b \otimes \nu_2 \otimes Y_2 \otimes c$$

So far, we have not explained how the linear model equation system is solved at each step of the iterative Newtonian process. In principle, many possible equation-solving methods could be applied to the linear model. However, in this section, we are giving an overview of NPA-TP, which solves the linear model by the use of a tensor operator ($\otimes$). The tensor operator is also used in the following chapter (Chapter 3), where we explain it in more detail, but for now, a high-level overview of the tensor operator will suffice.

The idea of the tensor operator is to split the recursive paths through a linearly recursive procedure into two fragments: (1) the fragment before the recursive call, and (2) the fragment after the recursive call. Consider a procedure $W$ that is described by the equation system $W = (a \otimes W \otimes b) \oplus c$. The recursive part of the procedure body is $(a \otimes W \otimes b)$, and the fragment before the recursive call is $a$, and the fragment after
the recursive call is \( b \). Notice that, when the procedure \( W \) executes, there will always be an equal number of occurrences of \( a \) and \( b \) in matched pairs, e.g., \( c \), or \( a \otimes c \otimes b \), or \( a \otimes a \otimes c \otimes b \otimes b \), and so on. We use the tensor operator to put together the fragments before and after the call (i.e., \( (a \odot b) \)); we then think of the two fragments as a unit that can be repeated any number of times, using a new tensored Kleene-star operator \( (*_T) \).

In [Reps et al., 2016], an abstract domain based on predicate abstraction was used, and in this domain, there is an efficient way of applying the tensor operator to two values, and of evaluating an expression that involves a tensored Kleene-star. Thus, the linear model equation system constructed as part of NPA-TP could be efficiently solved by expressing the linear model using tensors and evaluating the tensored Kleene-star. The original motivation for the research that produced Chapter 3 was to apply NPA-TP with the abstract domain of CRA. Ultimately, it was not possible to apply the NPA-TP framework to CRA, for reasons explained in Chapter 3; however, many of the key ideas, including the tensor operator, did prove to be applicable, and a useful enhancement of CRA was produced by this research.
Chapter 3

Interprocedural Compositional Recurrence Analysis

3.1 Introduction

The original version of CRA (presented in [Farzan and Kincaid, 2015]) is non-uniform: although recursion is a kind of generalized loop construct, the algorithm that CRA uses to summarize loops (based on generating and solving recurrences) is not the same as the one it uses to summarize recursive procedures (which relies on coarse abstraction and widening). For instance, if a loop is re-coded using a tail-recursive procedure, CRA is not able to identify the same numeric invariants in the recursive version that it identifies in the version with an explicit loop.

This chapter addresses the problem of creating a context-sensitive interprocedural version of CRA that handles loops and recursion—including non-linear recursion—in a uniform way. The goal is to create a system that is both

- more resilient to different coding styles—so that if a loop is re-coded to use tail-recursion, the same invariants are identified, and
- more general, by bringing to bear on recursive procedures the methods that CRA uses for generating and solving recurrence equations.

To create such a system, we must deal with the “impedance mismatch” between CRA’s reliance on Tarjan’s path-expression method, which handles regular languages, and the context-free-language underpinnings of context-sensitive interprocedural analysis [Sharir and Pnueli, 1981; Bouajjani et al., 1997; Finkel et al., 1997;
This issue is challenging because at the technical level, the regular-language viewpoint is baked into CRA: CRA’s recurrence-solving step is coupled with interpreting Kleene-star (*).

The inspiration for our work was the recently developed framework for Newtonian Program Analysis via Tensor Product (NPA-TP) [Reps et al., 2016]. At first blush, it appeared that NPA-TP overcomes the impedance mismatch because it provides a way to harness Tarjan’s path-expression method for interprocedural analysis: NPA-TP has the surprising property of converting an interprocedural-analysis problem—i.e., a context-free path problem—into a sequence of regular-language path problems.

For NPA-TP to be applicable, the abstract domain must support a few non-standard operations (i.e., a so-called tensor-product operation and a detensor operation). While the CRA domain can be extended with these non-standard operations (§3.3.2), it fails to satisfy some properties on which NPA-TP relies: (i) CRA has infinite ascending chains, and thus NPA-TP[CRA] would not be guaranteed to terminate; (ii) CRA does not have an effective equivalence procedure, and thus it is not possible, in general, to ascertain whether an NPA-TP[CRA] analyzer has reached a fixpoint.

We address these issues by developing a new framework, which we call NPA-TP-GJ (NPA-TP with Gauss-Jordan elimination, §3.3.3). Whereas Kleene iteration, NPA [Esparza et al., 2010], and NPA-TP do not converge at all when working with an abstract domain that has neither effective equivalence nor the ascending-chain condition, the NPA-TP-GJ algorithm (Alg. 3.20) is able to both detect and enforce convergence.

Our experiments show that NPA-TP-GJ instantiated for Interprocedural CRA (ICRA) has broad overall strength compared with several state-of-the-art software model checkers. (See §3.4.)

ICRA also provides a new, systematic approach to the problem of establishing bounds on resource usage, such as memory used or execution time. Recent work by Hoffmann et al. [Hoffmann et al., 2012a; Carbonneaux et al., 2015a] on establishing resource-usage bounds is able to provide polynomial upper bounds on the resource usage of programs. Hoffmann et al. use ad hoc techniques to analyze procedures, whereas ICRA is based on the classical idea of creating behavior summaries for procedures, including recursive procedures. In the case of resource-bound analysis, the summary for a procedure P characterizes the change in available resources caused by invoking P (and all procedures transitively invoked by P). Moreover, by approaching resource-bound analysis as a problem of finding invariant polynomial inequalities, ICRA is able to obtain lower bounds on the resources used, as well as upper bounds, in a uniform way. (See §3.4.)

**Contributions.** Our work makes three main contributions:

- We extend CRA to create ICRA, a context-sensitive interprocedural version that handles loops and recursion—including non-linear recursion—in a uniform way (§3.3.3).

- We present the NPA-TP-GJ framework, which represents the interface and properties required for our
approach. NPA-TP-GJ is an interprocedural-analysis method that can be used when the abstract domain has infinite ascending chains and does not support effective equivalence (§3.3.3).

- We present the results of experiments with an implementation of ICRA (§3.4). The experiments show that ICRA has broad overall strength, compared with several state-of-the-art software model checkers. ICRA also performs well when applied to the problem of establishing resource-usage bounds.

Outline. §3.2 states the problem that this chapter addresses. §3.3 presents the technical details of our solution. §3.4 describes the implementation of Algorithm NPA-TP-GJ and presents experimental results. §3.5 discusses related work. §3.6 concludes. Proofs can be found in App. A.

3.2 Problem Statement

The problem addressed in the chapter is the following:

Extend CRA to create a context-sensitive interprocedural version (ICRA) that handles loops and recursion—including non-linear recursion—in a uniform way.

Recursion presents an obstacle because the set of paths through a program that uses recursion is not regular: it is context-free [Sharir and Pnueli, 1981; Reps, 1998], which places recursive behavior beyond the scope of what can be analyzed using CRA’s iteration operator.

Recent work on Newtonian Program Analysis via Tensor Product (NPA-TP) [Reps et al., 2016] has provided a crucial piece of the puzzle. NPA-TP uses tensor products (§3.3.2) to extend the algebraic approach to program analysis to linear recursive systems of equations. To solve non-linear recursive equations, NPA-TP uses Newton iteration, an approach pioneered by Esparza et al. [Esparza et al., 2010] that generalizes Newton’s method for numerical analysis. NPA-TP is a hybrid iterative/algebraic approach in which a solution to a system of equations is computed as the limit of a sequence (like in iterative program analysis), but where each iterate is the solution to a linearized model of the equations (which is solved algebraically). However, neither Newton iteration nor Kleene iteration is compatible with CRA:

- The CRA domain has infinite ascending chains, so Newton/Kleene iteration is not guaranteed to converge.

- The problem of determining whether two CRA transition formulas are equivalent is undecidable, so there is no way to tell if Newton/Kleene iteration has converged.
Our algorithm for ICRA, presented in §3.3.3, is based on a new analysis framework, called NPA-TP-GJ. NPA-TP-GJ adopts the idea from NPA-TP of working with a refactored equation system obtained by rearranging expressions using tensor product (§3.3.3). However, other aspects of NPA-TP-GJ are significantly different. For instance, the key step of NPA-TP-GJ has the flavor of Gauss-Jordan elimination: it repeatedly carries out (i) a symbolic variation of NPA-TP’s linearization step (but only applied to a single equation), and (ii) substitution of a closed-form expression for the linearized symbol into the other equations. (See Alg. 3.14.)

### 3.3 Technical Details

#### 3.3.1 Recursive Equation Systems

This chapter shows how CRA can be extended so that it can apply its recurrence-solving techniques to recursive procedures. This section formalizes this problem as the problem of computing a property (within a suitable algebraic structure called a quasi-weight domain) that overapproximates the set of paths defined by a recursive equation system.

**Definition 3.1.** Let $\Sigma$ be an alphabet and let $X = \{X_1, \ldots, X_n\}$ be a finite set of variables. A system of equations $S$ with regular right-hand sides over $\Sigma$ and $X$ consists of one equation for each variable, $X_1 = R_1, \ldots, X_n = R_n$, where each $R_i$ is a regular expression over the alphabet $X \cup \Sigma$. More compactly, we write $S : \{X_i = R_i\}_{i=1}^n$.

Quasi-weight domains are algebraic structures with operations to interpret the operators of regular expressions, the domain of CRA being the motivating example for our interest. The axioms of quasi-weight domains ensure that its operations overapproximate the operations of the path algebra described in Defn. 2.4.

**Definition 3.2.** A quasi-weight domain $D = (D, \equiv, \oplus, \otimes, *, 0, 1)$ is a set $D$ equipped with an equivalence relation $\equiv$, a binary combine operator $\oplus$, a binary extend operator $\otimes$, a unary closure operator $*$, and distinguished elements $0$ and $1$, and that satisfies the following axioms:

1. The equivalence relation $\equiv$ is a congruence with respect to $\oplus$ and $\otimes$ ($a_1 \equiv a_2$ and $b_1 \equiv b_2$ implies $a_1 \oplus b_1 \equiv a_2 \oplus b_2$ and $a_1 \otimes b_1 \equiv a_2 \otimes b_2$). Note that $\equiv$ is not necessarily a congruence with respect to $*$.

2. $\langle D, \oplus, \otimes, 0, 1 \rangle$ is an idempotent semiring “up to equivalence,” meaning that for all $a, b, c \in D$ we have

   a) (Associativity) $a \otimes (b \otimes c) \equiv (a \otimes b) \otimes c$ and $(a \oplus b) \oplus c \equiv a \oplus (b \oplus c)$

   b) (Unit) $a \otimes 1 \equiv 1 \otimes a \equiv a$ and $a \oplus 0 \equiv 0 \oplus a \equiv a$

   c) (Commutativity) $a \oplus b \equiv b \oplus a$
3. The closure operator overapproximates reflexive transitive closure, in the following sense:

a) (Reflexivity) \(1 \preceq a^*\)

b) (Transitivity) \(a^* \otimes a \preceq a^*\) and \(a \otimes a^* \preceq a^*\)

where \(\preceq\) is the natural preorder on \(D\): for any \(a, b \in D\), \(a \preceq b \iff a \oplus b \equiv b\).

Because \(\otimes\) will be used to model sequencing of program actions, there is no assumption that \(\otimes\) is commutative. We assume the following precedences for operators: \(* > \otimes > \oplus\). We also sometimes use \(a \in D\) rather than \(a \in \mathbb{D}\).

Example 3.3. The quasi-weight domain of paths \((2^\Sigma^*, \equiv, \oplus, \otimes, 0, 1)\) is defined using \(\oplus\), \(\otimes\), \(0\), and \(1\) from Defn. 2.4. The equivalence relation \(\equiv\) is equality on languages, and the iteration operation \(L^* \triangleq \bigcup_{i=0}^{\infty} L^i\) is Kleene closure.

Example 3.4. Let \(x\) denote a finite set of program variables. In the quasi-weight domain of CRA over the variables \(x\), the carrier \(D\) is the set of all transition formulas \(\varphi(x, x')\) over the variables \(x\) and primed copies \(x'\). The equivalence relation \(\equiv\) is logical equivalence; the sequencing operator \(\varphi \otimes \psi \triangleq \exists x''.\varphi(x, x'') \land \psi(x'', x')\) is sequential composition; the choice operation \(\varphi \oplus \psi \triangleq \varphi \lor \psi\) is disjunction; the iteration operation \(\varphi^*\) is the defined by extracting recurrences from \(\varphi\) and computing their closed forms as described in §1.3; \(1 \triangleq (x = x')\) is the identity transition; and \(0 \triangleq \text{false}\) is the empty transition.

Note that all of the operations of the CRA quasi-weight domain are effective except the equivalence relation. The equivalence relation is used for developing the underlying theory of NPA-TP-GJ, but does not play a role in any algorithms. The motivation behind the inclusion of an explicit equivalence relation in the definition of quasi-weight domains (rather than quotienting by the equivalence relation to obtain a simpler algebraic structure) is because the iteration operator of CRA may produce inequivalent outputs for equivalent inputs. (Because the theory of non-linear integer arithmetic is undecidable, any non-trivial iteration operator has this deficiency.)

Definition 3.5. Let \(D = (D, \equiv, \oplus, \otimes, *, 0, 1)\) be a quasi-weight domain, \(\Sigma\) be an alphabet, \(X = \{X_1, ..., X_n\}\) be a set of variables, \(\|\| : \Sigma \to D\) be an alphabet interpretation, and \(\sigma : X \to D\) be a variable interpretation. We use \(\|\|_\sigma\) to denote the function that maps any regular expression over \(\Sigma\) and \(X\) to an element of \(D\) by using \(\|\|\) to interpret \(\Sigma\), \(\sigma\) to interpret \(X\), and by using the operations of \(D\) in place of the regular-expression operators \(0, 1, +, -, *, \ast\). If \(R\) contains no variables, we may omit the subscript \(\sigma\).
We use $\| \cdot \|^P$ to denote the obvious alphabet interpretation for the quasi-weight domain of paths (as described in §2.3): for any regular expression $R$, $\|R\|^P$ is the set of paths recognized by $R$. For any system of equations with regular right-hand sides $S : \{ X_i = R_i \}_{i=1}^n$ over $\Sigma$ and $X = \{ X_1, \ldots, X_n \}$, we define $\text{Paths}_S : X \to 2^{\Sigma^*}$ to be the least function (in pointwise set-inclusion order) such that for each $i$, $\text{Paths}_S(X_i) = \|R_i\|_{\text{Paths}_S}$.

We can now formalize the problem of interest.

Let $S : \{ X_i = R_i \}_{i=1}^n$ be a system of recursive equations with regular right-hand sides over an alphabet $\Sigma$ and a set of variables $X = \{ X_1, \ldots, X_n \}$; let $D = (D, \equiv, \oplus, *, \otimes, 0, 1)$ be a quasi-weight domain; and let $\| \cdot \| : \Sigma \to D$ be an alphabet interpretation. A solution to $S$ in $D$ is any map $D : X \to D$ such that for all $i$, for all $p \in \text{Paths}_S(X_i)$, we have $D(X_i) \gtrsim \| p \|$.

In what follows, we will describe a method for solving systems of equations with regular right-hand sides. Our presentation of the method proceeds in two steps: first, we review tensor products, which can be used to solve linear recursive systems [Reps et al., 2016]; second, we present a hybrid iterative/algebraic technique for solving general recursive systems based on tensor products and iteration domains.

### 3.3.2 Tensor Products

Intuitively, quasi-weight domains are well-suited for solving intra-procedural program analysis problems, such as the one described in §2.3. Intra-procedural program analysis problems correspond to left-linear systems of equations (in which each right-hand-side is of the form $\bigoplus_i X_i \cdot R_i$ where $R_i$ does not contain variables). Left-linear systems correspond to regular languages, and the operators of quasi-weight domains correspond to those of regular expressions. For inter-procedural-analysis problems, however, these operations are not sufficient. Just as one cannot describe the set of paths of a recursive procedure using a regular expression, one cannot use the operators of a quasi weight domain to describe the operation of recursive procedures.

Recently, Reps et al. showed that algebraic program analysis can be extended to a linear system of equations by using a tensored domain [Reps et al., 2016]. As a warm-up exercise for §3.3.3, where we present our method for solving an arbitrary system of equations, this section describes a slight variation of the Reps et al. method. Essentially all of the machinery introduced in this section carries over to §3.3.3, but the setting in which it is used here is somewhat less complicated.

Following Obs. 2.3.1, we proceed in two steps. First, we define a way to transform a linear equation system into a different, but equivalent, form (Eqn. (3.1)). The transformation involves the tensor-product domain of paths (Defn. 3.6), and converts a linear equation system with regular right-hand sides into a non-recursive equation system in which the right-hand sides are regular expressions extended with additional operators.
from the tensor-product domain. Second, we describe the properties that a tensor-product domain must satisfy to be used to interpret an extended regular expression, and then describe the ICRA tensor-product domain.

**Definition 3.6.** Let \( \Sigma \) be an alphabet. Define a quasi-weight domain \( D_\tau = (D, \equiv_\tau, \oplus_\tau, \otimes_\tau, \ast_\tau, \emptyset, 1_\tau) \) of tensored paths as follows: a tensored path \( (\mathbf{p}, \mathbf{p}) \) is a pair consisting of one “backward” path \( \mathbf{p} \) and one “forward” path \( \mathbf{p} \); the carrier \( D_\tau \overset{df}{=} 2^{\Sigma^* \times \Sigma^*} \) is the powerset of tensored paths. The \( \otimes_\tau \) operator is coordinate-wise concatenation, with concatenation reversed for backward paths:

\[
T_1 \otimes_\tau T_2 \overset{df}{=} \{(\mathbf{p}_2 \mathbf{p}_1, \mathbf{p}_1 \mathbf{p}_2) : (\mathbf{p}_1, \mathbf{p}_2) \in T_1, (\mathbf{p}_2, \mathbf{p}_1) \in T_2\}
\]

Similarly to the quasi-weight domain of untensored paths, \( \equiv_\tau \) is equality, \( \oplus_\tau \) is union, and \( T^{+\tau} \overset{df}{=} \bigcup_i T^i \) where \( T^0 \overset{df}{=} \{(\epsilon, \epsilon)\} \) and \( T^{i+1} = T^i \otimes_\tau T \). The unit of \( \oplus_\tau \) is \( \emptyset_\tau \overset{df}{=} \emptyset \), and the unit of \( \otimes_\tau \) is \( 1_\tau \overset{df}{=} \{(\epsilon, \epsilon)\} \).

The **tensor-product domain of paths** \( \mathcal{T} = (D, D_\tau, \cdot, \diamond) \) consists of the domain of paths \( D \), the domain of tensed paths \( D_\tau \), a tensor operation \( \cdot : D \times D \rightarrow D_\tau \) and a detensor-product operation \( \diamond : D \times D_\tau \rightarrow D \), defined as follows:

\[
L_1 \cdot L_2 \overset{df}{=} \{(p_1, p_2) : p_1 \in L_1, p_2 \in L_2\}
\]

\[
L \diamond T \overset{df}{=} \{pp' : p \in L, (p, p') \in T\}
\]

**Definition 3.7.** Let \( \Sigma \) be an alphabet and let \( X = \{X_1, \ldots, X_n\} \) be a finite set of variables. A (tensored) **extended regular expression** over \( \Sigma \) and \( X \) is an expression generated by the following grammar:

\[
E, F \in \text{Ext}(\Sigma, X) ::= a \in \Sigma \mid X_i \in X \mid 0 \mid 1 \\
\mid E + F \mid E \cdot F \mid E^* \mid E \diamond E_\tau
\]

\[
E_\tau, F_\tau \in \text{Ext}_\tau(\Sigma, X) ::= (E \circ F) \mid 0_\tau \mid 1_\tau \\
\mid E_\tau +_\tau F_\tau \mid E_\tau \cdot_\tau F_\tau \mid E_\tau^* \tau
\]

For any function \( \sigma : X \rightarrow 2^{\Sigma^*} \) and any extended regular expression \( E \), we use \( \llbracket E \rrbracket_\sigma^p \) to denote the set of paths in the language of \( E \). \( \llbracket E \rrbracket_\sigma^p \) is defined as the interpretation of \( E \) using the operations of \( \mathcal{T} \) from Defn. 3.6 in place of the operators \( 0, 1, +, \cdot, \ast, 0_\tau, 1_\tau, +_\tau, \cdot_\tau, \ast_\tau, \circ, \text{ and } \diamond \).

For extended regular expressions \( E \) and \( F \), we write \( E \simeq F \) to denote that for every function \( \sigma : X \rightarrow 2^{\Sigma^*} \), we have \( \llbracket E \rrbracket_\sigma^p = \llbracket F \rrbracket_\sigma^p \).

As a concrete example, \( \{a^ib^i : i \geq 0\} \) is the classic example of a non-regular language. It is not the language of any regular expression, but it is the language of the extended regular expression \( 1 \ast (a \circ b)^* \).
Using extended regular expressions, we can transform a linear system of equations

\[ S : \{ X_i = r_i + s_{i,1} \cdot X_{i,1} + \ldots + s_{i,m_i} \cdot X_{i,m_i} \cdot t_{i,m_i} \}_{i=1}^n \]

over an alphabet \( \Sigma \) and set of variables \( \mathcal{X} = \{ X_1, \ldots, X_n \} \) as follows. Let \( \mathcal{Z} = \{ Z_1, \ldots, Z_n \} \) be a set of variables (one for each \( X_i \)), and define the following \textit{left-linear} system:

\[
S_{\mathcal{Z}} : \left\{ \begin{array}{l}
Z_i = (1 \circ r_i) +_{\mathcal{Z}} (s_{i,1} \circ t_{i,1}) +_{\mathcal{Z}} \ldots +_{\mathcal{Z}} (s_{i,m_i} \circ t_{i,m_i}) \\
\end{array} \right\}_{i=1}^n
\] (3.1)

Using Tarjan’s path-expression algorithm, we can transform \( S_{\mathcal{Z}} \) into an equivalent \textit{non-recursive} system of equations \( \hat{S}_{\mathcal{Z}} : \{ Z_i = E_i \}_{i=1}^n \) in which the right-hand sides are extended regular expressions. Finally, define system \( \hat{S} \) to be \( \{ X_i = 1 \circ E_i \}_{i=1}^n \). We have the following path-preservation property:

\textbf{Proposition 3.8.} [Reps et al., 2016] For each \( i \), we have

\[ \text{Paths}_{\hat{S}}(X_i) = \text{Paths}_S(X_i) \].

Tensor-product domains and (I)CRA. We now give a general definition of a \textit{tensor-product domain}. A tensor-product domain is equipped with operations that correspond to the extended-regular-expression operators. The operations are required to satisfy conditions that imply that they approximate the corresponding operators on paths.

\textbf{Definition 3.9.} A \textit{tensor-product domain} \( \mathcal{S} = (\mathcal{D}, \mathcal{D}_T, \odot, \times) \) consists of two quasi-weight domains, \( \mathcal{D} \) and \( \mathcal{D}_T \), along with the following two operations:

- A \textbf{tensor-product operator}, denoted by \( \odot : \mathcal{D} \times \mathcal{D} \to \mathcal{D}_T \), such that for all \( a, b, c, a_1, b_1, a_2, b_2 \in \mathcal{D} \),

\[
\begin{align*}
0 \odot a &\equiv_{\mathcal{S}} a \odot 0 \equiv_{\mathcal{S}} 0 \odot a \\
a \odot (b \odot c) &\equiv_{\mathcal{S}} (a \odot b) \odot_{\mathcal{S}} (a \odot c) \\
(b \odot c) \odot a &\equiv_{\mathcal{S}} (b \odot a) \odot_{\mathcal{S}} (c \odot a) \\
(a_1 \odot b_1) \odot_{\mathcal{S}} (a_2 \odot b_2) &\equiv_{\mathcal{S}} (a_2 \odot a_1) \odot (b_1 \odot b_2)
\end{align*}
\]

and for all \( a_1 \equiv a_2 \) and \( b_1 \equiv b_2 \), we have \( a_1 \odot b_1 \equiv_{\mathcal{S}} a_2 \odot b_2 \).
• A detensor-product operation \( \times : D \times D_T \rightarrow D \), such that for all \( a, b \in D \) and all \( p, q \in D_T \) we have

\[
\begin{align*}
\ a \times (p \oplus_T q) & \equiv (a \times p) \oplus (a \times q) \\
(a \oplus b) \times p & \equiv (a \times p) \oplus (b \times p) \\
(a \times (p \odot_T (b \odot c))) & \equiv b \odot (a \times p) \odot c \\
(a \times (p \odot_T q) & \equiv (a \times p) \times q \\
(a \times 1_T) & \equiv a
\end{align*}
\]

and for all \( a \equiv b \) and \( p \equiv_T q \), we have \( a \times p \equiv b \times q \).

One can check that the tensor-product domain of paths (Defn. 3.6) satisfies the conditions of Defn. 3.9. A second example is the tensor-product domain of CRA, denoted by CRA_T.

**Example 3.10.** Recall from Ex. 3.4 that CRA weights are formulas over some specified set of variables \( x \) and primed copies \( x' \). The weights in CRA_T are formulas over four sets of variables \( x, x', s, s' \). It is instructive to think of such a formula as a set of pairs of transitions \((\langle s, s' \rangle, \langle s, s' \rangle)\). Under this interpretation, the semantic definitions of the \( \odot_T, \odot \), and \( \times \) operators are as follows:

\[
\begin{align*}
T_1 \odot_T T_2 & = \{(s, x') : (s, x') \in T_1 \} \\
R_1 \odot R_2 & = \{(s_1, s_1'), (s_1, s_2') : (s_1, s_1') \in R_1, (s_1, s_2') \in R_2 \} \\
R \times T & = \{(s, x) : (s, x) \in R \land (s, x) \in T \}
\end{align*}
\]

The operations of CRA_T can be carried out syntactically using variable renaming and existential quantification:

\[
\begin{align*}
\varphi_T \odot_T \psi_T & \equiv \exists x, x' \varphi_T [x \mapsto x', x' \mapsto x'] \\
\varphi_T \oplus_T \psi_T & \equiv \varphi_T \lor \psi_T \\
\varphi_T \odot T & \equiv \text{false} \\
1_T & \equiv x = x' \land \bar{x} = \bar{x}' \\
\varphi \odot \psi & \equiv (\varphi [x \mapsto x', x' \mapsto x']) \land (\psi [x \mapsto x', x' \mapsto x']) \\
\varphi \times \psi_T & \equiv \exists x, x' \varphi [x \mapsto x', x' \mapsto x'] \\
& \land \psi_T [x \mapsto x', x' \mapsto x']
\end{align*}
\]

The iteration operator \( *_T \) operates by finding closed forms for recurrences just as described in §1.3, except over a vocabulary of four sets of variables.
Tensor-product domains can be used to evaluate extended regular expressions in the same way that quasi-weight domains can be used to evaluate regular expressions (Defn. 3.5). Let $\mathcal{T} = (\mathbb{D}, \mathbb{D}_\mathcal{T}, \circ, \kappa)$ be a tensor-product domain, $\llbracket \cdot \rrbracket : \Sigma \rightarrow \mathbb{D}$ be an interpretation of an alphabet $\Sigma$, and $\sigma : \mathcal{X} \rightarrow \mathbb{D}$ be an interpretation of a set of variables $\mathcal{X}$. We use $\llbracket \cdot \rrbracket_\sigma$ to denote the function that evaluates an extended regular expression within $\mathcal{T}$ by interpreting the extended-regular-expression operators $0, 1, +, \cdot, *, 0_T, 1_T, +_T, \cdot_T, *, \circ$, and $\kappa$ using their counterparts in $\mathcal{T}$, using $\llbracket \cdot \rrbracket$ to interpret symbols in $\Sigma$, and $\sigma$ to interpret variables. We omit the $\sigma$ subscript for extended regular expressions without variables. The crucial property of tensor-product domains is that evaluation of extended regular expressions within a tensor-product domain overapproximates the path semantics:

**Proposition 3.11.** Let $\Sigma$ be an alphabet, let $\mathcal{T} = (\mathbb{D}, \mathbb{D}_\mathcal{T}, \circ, \kappa)$ be a tensor-product domain, and let $\llbracket \cdot \rrbracket : \Sigma \rightarrow \mathbb{D}$ be an interpretation for the alphabet. For any regular expression $E$ over $\Sigma$ and any path $p \in \llbracket E \rrbracket^p$, we have $\llbracket E \rrbracket \bowtie \llbracket p \rrbracket$.

**Proof:** See App. A.

Together, Props. 3.8 and 3.11 give a complete recipe for applying Obs. 2.3.1 to solve a linear equation system $S$ over a tensor-product domain:

- Transform $S$ to eliminate recursion—while preserving paths—using Eqn. (3.1) and Tarjan’s method.
- Evaluate the right-hand sides of the resulting system within the tensor-product domain.

**Example 3.12.** We demonstrate the use of the CRA tensor-product domain with a recursive multiplication routine.

```plaintext
mul():
    if (y != 0)
        y := y - 1; mul(); m := m + x
```

This program corresponds to the following (linear) system of equations:

$$S: \begin{cases} X = y = 0 \\ + (y \neq 0 \cdot y := y - 1 \cdot X \cdot m := m + x) \end{cases}$$

$S$ can be transformed into a non-recursive system using an extended regular expression:

$$\hat{S}: \begin{cases} X = y = 0 \\ \kappa ((y \neq 0 \cdot y := y - 1 \circ (m := m + x))^y) \end{cases}$$

We can compute a summary for $\text{mul}$ by evaluating the right-hand side of the equation in $\hat{S}$ in the CRA$_T$ domain. The crucial computation is the tensored iteration operation. The input is the following tensored formula, which represents
one iteration of the “loop”:

\[ y \neq 0 \land y' = y - 1 \land m' = m + x \land m', x', y', x' = m, x, y, x \]

<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Closed form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y' = y - 1 )</td>
<td>( y^{(k)} = y^{(0)} - k )</td>
</tr>
<tr>
<td>( m' = m + x )</td>
<td>( m^{(k)} = m^{(0)} + kx^{(0)} )</td>
</tr>
</tbody>
</table>

From this formula, we extract the recurrences shown to the right. The output of the iteration operator is the following tensored formula:

\[ \exists k. k \geq 0 \land \left( y' = y - k \land m' = m + kx \land m', x', y', x' = m, x, y, x \right) \]

Finally, by evaluating the detensor-product operation, we get the following summary for \( \text{mul} \):

\[ \exists k. k \geq 0 \land y' = 0 \land y' = y - k \land m' = m + kx \land x' = x \]

Notice that this formula constrains the number of decrements of \( y \) to be equal to the number of increments of \( m \), which is crucial in showing that \( \text{mul} \) correctly implements multiplication (when \( m \) is equal to 0 in the initial state).

### 3.3.3 Non-Linear Systems

This section describes a method for solving general (non-linear) systems of equations with regular right-hand sides over a tensor-product domain equipped with some additional structure. The method is inspired by NPA-TP, and similarly uses a variation of Kleene iteration in which each iterate is computed by solving a simpler “regularized” model of the original system of equations, obtained by rearranging expressions using tensor product.

Our method overcomes two fundamental challenges:

- Quasi-weight domains may have infinite ascending chains.
- Quasi-weight domains may have undecidable equivalence.

The quasi-weight domain of CRA exhibits both of these problems. Note that Kleene iteration is not an option for solving the original system of equations.

- When there are infinite ascending chains, Kleene iteration may not converge.
fib():
    n := p
    if (n <= 1) r := 1
    else
        p := n-1
        fib()
        t := r; p := n-2
        fib()
        r := r+t

Figure 3.1: An implementation of the Fibonacci function

- When equivalence is undecidable, even if Kleene iteration does converge, there is no way to tell.

However, as we show in the remainder of this section, it is possible to carry out Kleene iteration on the “regularized” model, due to its special properties (see Alg. 3.14). Our solution is motivated by the following observation about the CRA domain, which can be used to enforce and detect convergence for that domain:

The iteration operator of CRA can be “factored” through a simple abstract domain (its iteration domain), which has decidable equivalence and a widening operator. By re-arranging a system of equations into a special form, from a successive-approximation sequence in the CRA domain we can construct a derived sequence in the iteration domain. We can use equivalence checking in the iteration domain to detect convergence of the successive-approximation sequence, and use the widening operator to ensure that both sequences converge in finite time.

We motivate our approach using the CRA domain. The iteration operator of CRA can be thought of as a two-phase process (cf. §1.3): given a formula \( \varphi \) representing the body of a loop, we (1) compute a set of recurrences entailed by \( \varphi \) using an SMT solver, and (2) compute closed forms for the recurrences to use as an approximation of the reflexive transitive closure of \( \varphi \). The loop-body formula \( \varphi \) is expressed in a rich assertion language, which includes disjunction, quantifiers, and non-linear terms. The recurrences computed by phase (1), on the other hand, are relatively simple: each recurrence (e.g., \( x' = x + 1 \), representing that \( x \) is incremented in the loop) is a linear constraint, and so a set of recurrences can be represented by a convex polyhedron. Thus, we can express the iteration operator of CRA as the composition of (i) an abstraction function that computes a polyhedron (representing recurrences) from a transition formula, and (ii) an abstract-closure function that computes a transition formula from a polyhedron by computing closed forms. A key feature of our approach is that the widening operator is defined on the iteration domain—polyhedra—rather than on the domain of CRA itself (i.e., transition formulas over non-linear arithmetic). Polyhedra are a well-studied abstract domain, for which equivalence is decidable and widening operators are readily available. Our implementation uses the NewPolka implementation in APRON [APRON, [n.d.]].
Example 3.13. We use the Fibonacci function to show how our analysis technique exploits the two-phase structure of CRA’s iteration operator. The code for fib is pictured to the right; it uses two global variables, p (representing the parameter) and r (representing the return value), and two local variables, n and t. (Our technique handles local variables using essentially the same technique presented in [Reps et al., 2016, §8], but we omit the details.) The paths through fib are captured by the following equation:

\[
S : \begin{cases}
    X = n := p & \text{if } n \leq 1 \text{ and } r := 1 \\
    + \quad & \text{if } n > 1 \text{ and } p := n - 1 \cdot n \cdot r := r + t
    \end{cases}
\]

where right \(= t := r \cdot p := n - 2 \cdot X \cdot r := r + t\). The first step in solving S is to re-arrange the system to use the tensored iteration operator to model the first recursive call

\[
\hat{S} : \begin{cases}
    X = (n := p \cdot n \leq 1 \cdot r := 1) \\
    \times (n := p \cdot n > 1 \cdot r := r + t)^{*}
    \end{cases}
\]

Unlike the case of linear recursion (e.g., Ex. 3.12), \(\hat{S}\) is still recursive: X appears within the expression “right.” However, \(\hat{S}\) has the property that every variable appears below a star. We say that an occurrence of a variable in a (tensored) extended regular expression is guarded if it appears below a star; a variable with an unguarded occurrence is called free. The significance of \(\hat{S}\) having no free variables is that it allows us to detect and enforce convergence of a successive-approximation process via a “derived sequence” of polyhedra.

We define two sequences by mutual recursion: \(\langle D^k \rangle_{k \in \mathbb{N}}\) is a sequence of transition formulas representing summaries for fib, and \(\langle \beta^k \rangle_{k \in \mathbb{N}}\) is a sequence of polyhedra representing transitions of the “loop body” (appearing below the \(\ast^\circ\)):

\[
\text{body} = n := p \cdot n > 1 \cdot r := r + t.
\]

For simplicity, we only sketch the high-level idea of our approach rather than give the exact sequences that would be computed. The first (underapproximating) summary for fib is \(D^0(X) \overset{\text{def}}{=} 0\). We compute the first loop-body polyhedron \(\beta^1 = \bot\) by substituting \(D^0(X)\) for X in body, evaluating within the CRA algebra, and applying the abstraction function. We compute the second summary for fib by applying the abstract-closure function to \(\beta^1\) (which yields 1), and using the result as the approximation of the loop:

\[
D^1(X) = p \leq 1 \land r' = 1 \land p' = p
\]

The next loop-body polyhedron \(\beta^2\) is computed from \(D^1(X)\) just as \(\beta^1\) was computed from \(D^0(X)\), except that we
additionally perform polyhedral widening using $\beta^1$:

$$\beta^2 = \bot \lor \left( \begin{array}{l}
p' = p - 1 \land 1 < p \leq 3 \land r' = r \\
p' = p - 2 \land r' = r + 1
\end{array} \right)$$

$$= \left( \begin{array}{l}
p' = p - 1 \land 1 < p \leq 3 \land r' = r \\
p' = p - 2 \land r' = r + 1
\end{array} \right)$$

The return value is always incremented by 1 because, according to the summary $D^1(X)$, $\text{fib}$ always returns 1. Notice that the precondition on the parameter to the second call to $\text{fib}$ has been propagated backward to yield the constraint $p \leq 3$; this propagation is a result of projecting out the local variable $n$, which has the additional effect of equating the values of $n$ before and after the recursive call ($n'$ and $n$, respectively). Continuing this process, we compute $D^2$, $\beta^3$, and $D^3$:

$$D^2(X) = \left( \begin{array}{l}
p' = p \leq 1 \land r' = 1 \\
\lor \exists k. k \geq 1 \land r' = k + 1 \land k = p - 1 \land p \leq 3
\end{array} \right)$$

$$\beta^3 = p' = p - 1 \land 1 < p \land r' = r \land r' \geq r + 1$$

$$D^3(X) = \left( \begin{array}{l}
p' = p \leq 1 \land r' = 1 \\
\lor \exists k. k \geq 1 \land r' \geq k + 1 \land k = p - 1
\end{array} \right)$$

When computing $\beta^4$, we find that the sequence has converged: $\beta^4 = \beta^3$. Because every variable is guarded by a Kleene-star, the fact that the sequence of loop-body polyhedra has converged implies that the sequence of function summaries has also converged (Thm. 3.18), and $D^3(X) \equiv D^4(X)$ is an overapproximating summary for $\text{fib}$.

More abstractly, the equation system for $\text{fib}$ consists of a single recursive equation with just one variable:

$$S : \{ X = d + (a \cdot X \cdot b \cdot X \cdot c) \}.$$  \hspace{1cm} (3.2)

By treating the first occurrence of $X$ on the right-hand side of Eqn. (3.2) as a variable and the second occurrence as a symbolic constant, we created an equivalent system of equations where every variable appears below a star

$$\hat{S} : \{ X = (1 \odot d) \kappa (a \odot (b \otimes X \otimes c))^\tau \}.$$  

Thus, $\hat{S}$ has no free variables. The goal is to find a solution for $\hat{S}$ in the transition-formula domain. We define a “derived sequence” $\langle \beta^k \rangle_{k \in \mathbb{N}}$ in the polyhedral domain:

$$\beta^k \overset{\text{def}}{=} \beta^{k-1} \lor \bar{\alpha}_{\text{poly}}(a \odot (b \otimes D^k(X) \otimes c)),$$

where $\bar{\alpha}_{\text{poly}}$ is a heuristic that creates a polyhedron that overapproximates a transition formula [Farzan and Kincaid,
\begin{align*}
\text{factor}_X(a) & \triangleq a + X \times 0_T \\
\text{factor}_X(X_i) & \triangleq \begin{cases} 
0 + X \times 1_T & \text{if } X = X_i \\
X + X \times 0_T & \text{otherwise}
\end{cases} \\
\text{factor}_X(E + E') & \triangleq (F + F') + X \times (F_T + F_T') \\
& \quad \text{where } \text{factor}_X(E) = F + X \times F_T \\
& \quad \text{and } \text{factor}_X(E') = F' + X \times F_T' \\
\text{factor}_X(E \cdot E') & \triangleq (F \cdot F') + \left( X \times \begin{pmatrix} F_T & 1 \\ F_T & 1 \end{pmatrix} ight) \\
& \quad \text{where } \text{factor}_X(E) = F + X \times F_T \\
& \quad \text{and } \text{factor}_X(E') = F' + X \times F_T' \\
\text{factor}_X(E^*) & \triangleq E^* + X \times 0_T \\
\text{factor}_X(E \times E_T) & \triangleq (F \times E_T) + X \times (F_T \cdot E_T) \\
& \quad \text{where } \text{factor}_X(E) = F + X \times F_T
\end{align*}

Figure 3.2: factor_X(E).

2015, §IV & Alg. 2]. Convergence of the derived sequence—i.e., $\beta^k = \beta^{k-1}$ for some $k$—is enforced via the widening operator of the polyhedral domain. When the derived sequence converges, so does the $D$ sequence (in particular, $D^{k+1}(X) \equiv D^k(X)$).

We now show how this process can be carried out in a more abstract setting, for an arbitrary equation system over any number of variables. Following the Path-Preservation Principle (Obs. 2.3.1), we proceed in two steps. First, in §3.3.3, we show how to transform a system of equations into an equivalent system of equations with no free variables. Second, in §3.3.3, we formalize iteration domains (of which polyhedra are one concrete instance), and show how to solve systems of equations without free variables using a tensor-product domain equipped with an iteration domain.

Eliminating free variables

We now show how to transform a system of recursive equations with regular right-hand-sides into an equivalent system of equations where the right-hand sides are extended regular expressions with no free variables. The transformation is intuitively similar to Gauss-Jordan elimination, in that variables are successively eliminated in some order. However, unlike in Gauss-Jordan elimination, only free occurrences of variables are eliminated—and thus our method is only a partial elimination method. Given an equation $X = E$, the variable $X$ is eliminated in two steps:

(i) Rewrite $E$ as $F + X \times F_T$, where $X$ is not free in $F$.

(ii) Replace the equation $X = E$ with the equation $X = F \times F_T$ (in which $X$ is not free by construction), and replace every free occurrence of $X$ within the system of equations with $F \times F_T$. 


Rewriting step (i) is accomplished by the function \( \text{factor}_X(E) \), defined in Fig. 3.2. The input to \( \text{factor} \) is a variable \( X \) and a normal extended regular expression \( E \), which is an extended regular expression such that for every subexpression of \( E \) of the form \( F \times E \), \( E \) is of the form \( F^+ \). The output is an extended regular expression \( F + X \times F \) such that (1) \( E \simeq F + (X \times F) \), (2) \( F \) is normal, and (3) \( X \) is not free in \( F \). The complete free-variable-elimination algorithm is as follows:

**Algorithm 3.14** (Free-Variable Elimination). The input is a system of equations \( S : \{X_i = R_i\}_{i=1}^n \) with regular right-hand sides over an alphabet \( \Sigma \) and a finite set of variables \( X = \{X_1, \ldots, X_n\} \). We transform \( S \) into an equivalent system \( \hat{S} \) with no free variables by eliminating variables one at a time, in a style reminiscent of Gauss-Jordan elimination. We use \( S_k : \{X_i = E_{i,k}\}_{i=1}^n \) to denote the system of equations after \( k \) elimination rounds (we take \( S_0 \) def = \( S \) to be the original system of equations, noting that each regular expression \( R_i \) from the original system of equations is also a normal extended regular expression \( E_{i,0} \)).

1. Repeat for each \( k = 1 \) to \( n \):
   a) Let \( \text{factor}_{X_k}(E_{k,k}) = F + (X_k \times F) \)
   b) Define \( S_{k+1} : \{X_i = E_{i,k+1}\}_{i=1}^n \) by:
      - \( E_{k,k+1} \) def = \( F \times F^+ \)
      - For \( i \neq k \), \( E_{i,k+1} \) is obtained from \( E_{i,k} \) by replacing each free occurrence of \( X_k \) with \( F \times F^+ \).

2. Return \( \hat{S} \) def = \( S_{n+1} \).

**Theorem 3.15.** Given a system of equations \( S : \{X_i = R_i\}_{i=1}^n \) with regular right-hand sides over an alphabet \( \Sigma \) and a finite set of variables \( X = \{X_1, \ldots, X_n\} \), Alg. 3.14 computes a system of equations \( \hat{S} : \{X_i = \hat{E}_i\}_{i=1}^n \) where the right-hand sides are normal extended regular expressions over \( \Sigma \) and \( X \) with no free variables, and which is equivalent to \( S \) in the sense that Paths\(_S\)(\( X_i \)) = Paths\(_\hat{S}\)(\( X_i \)) for all \( X_i \).

**Proof:** See App. A.

Thm. 3.15 shows that the transformation performed by Alg. 3.14 results in an equivalent system of equations. Following Obs. 2.3.1, any post-fixpoint solution to the transformed system overapproximates every path of the original equation system. §3.3.3 shows how to compute such a post-fixpoint solution.

**Example 3.16.** We now illustrate Alg. 3.14 on the recursive-descent parser shown in Fig. 3.3. The initial equation system has the form

\[
E = a \cdot T \cdot (b \cdot E + (c + 0))
\]
\[
T = d \cdot ((e + f \cdot E \cdot (g + 0)) + 0) \cdot (h \cdot T + (i + 0)),
\]
void getNextToken() { tokenCount++;...}
void E() { // E ::= T ('+' E)*
    cost++; T();
    if (currentToken == '+') {
        getNextToken(); E();
    } else if (currentToken == EOF_TOKEN ||
        currentToken == ')') return;
    else abort();
}
void T() { // T ::= (ATOM | '(' E ')') ('*' T)*
    cost++; if (currentToken == ATOM) getNextToken();
    else if (currentToken == '(') {
        getNextToken(); E(); if (currentToken == ')') getNextToken();
        else abort();
    } else abort();
    if (currentToken == '*') {
        getNextToken(); T();
    } else if (currentToken == EOF_TOKEN ||
        currentToken == '+' ||
        currentToken == ')') return;
    else abort();
}

Figure 3.3: A recursive-descent parser.

where 0 = abort(), a = cost++, b = currentToken == '+' · getNextToken(), etc. The equation system simplifies to

\[
E = a \cdot T \cdot (b \cdot E + c) \tag{3.3}
\]
\[
T = d \cdot (e + f \cdot E \cdot g) \cdot (h \cdot T + i). \tag{3.4}
\]

Let E be variable \(X_1\) and T be \(X_2\). Step 1a of Alg. 3.14 applies factor \(E\) to the right-hand side of Eqn. (3.3), to produce a new equation for E. After simplification, we have the following extended regular expression for E:

\[
E = (a \cdot T \cdot c) \kappa((a \cdot T \cdot b) \odot 1)^*\tag{3.5}
\]

Eqn. (3.5) is a non-recursive version of Eqn. (3.3): Eqn. (3.5) says that an E consists of some number—say, \(n\)—of copies of \(a \cdot T \cdot b\), followed by \(a \cdot T \cdot c\), followed by \(n\) copies of \(1\).\(^1\) Step 1b of Alg. 3.14 substitutes the right-hand side of Eqn. (3.5) for the free occurrence of E in Eqn. (3.4), and thus round 1 of step 1 of Alg. 3.14 produces Eqn. (3.5) and

\[
T = d \cdot (e + f \cdot ((a \cdot T \cdot c) \kappa((a \cdot T \cdot b) \odot 1)^*) \cdot g) \cdot (h \cdot T + i).
\]

\(^1\)A similar characterization of E via an ordinary regular expression can be obtained by rewriting Eqn. (3.3) as \(E = (a \cdot T \cdot b) \cdot E + (a \cdot T \cdot c)\), which has the solution \((a \cdot T \cdot b)^n(a \cdot T \cdot c)\).
All occurrences of \( E \) have been eliminated, but three free occurrences of \( T \) and two guarded occurrences of \( T \) remain. The second and final round of Alg. 3.14 eliminates the free occurrences of \( T \), producing

\[
E = a \cdot ((d \cdot e \cdot i) \times (\text{left} + \text{right})^\gamma) \cdot c \times ((a \cdot T \cdot b) \odot 1)^\gamma
\]

\[
T = (d \cdot e \cdot i) \times (\text{left} + \text{right})^\gamma, \quad \text{where}
\]

\[
\text{left} = (a \odot c) \cdot T \cdot ((a \cdot T \cdot b) \odot 1)^\gamma \cdot T \cdot ((d \cdot f) \odot (g \cdot i))
\]

\[
\text{right} = d \cdot (e + f \cdot ((a \cdot T \cdot c) \times ((a \cdot T \cdot b) \odot 1)^\gamma) \cdot g) \cdot h \odot 1.
\]

As desired, all free occurrences of all variables have been eliminated. When the method described in §3.3.3 is used to solve this system, it computes summaries of procedures \( E \) and \( T \). Using the domain of CRA, the summary is strong enough to prove that after parsing completes, \( \text{cost} \leq \text{tokenCount} + 1 \)—that is, the cost of parsing is linear in the number of tokens.

**Solving an equation system with no free variables**

We now show how to solve a system of equations over extended regular expressions with no free variables. The key idea is to equip the tensor-product domain with an *iteration domain*, defined below.

**Definition 3.17.** Let \( \mathcal{D} = (\mathcal{D}, \equiv, \oplus, \otimes, \odot, 1, 0) \) be a quasi-weight domain. An *iteration domain* for \( \mathcal{D} \), \( \mathcal{D}^\ast = (\mathcal{D}^\ast, \leq^\ast, \triangledown, \alpha, \text{cl}) \), is a partially ordered set \((\mathcal{D}^\ast, \leq^\ast)\) along with the following operations.

An *abstraction* operator \( \alpha : \mathcal{D} \to \mathcal{D}^\ast \) and an *abstract-closure* operator \( \text{cl} : \mathcal{D}^\ast \to \mathcal{D} \) such that the following properties hold:

1. For all \( a \in \mathcal{D} \), \( \text{cl}(\alpha(a)) = a^\ast \)

2. For all \( a^\ast, b^\ast \in \mathcal{D}^\ast \) such that \( a^\ast \leq^\ast b^\ast \) we have \( \text{cl}(a^\ast) \leq \text{cl}(b^\ast) \).

A *widening* operator \( \triangledown : \mathcal{D}^\ast \times \mathcal{D}^\ast \to \mathcal{D}^\ast \) such that the following properties hold:

1. For all \( a^\ast, b^\ast \in \mathcal{D}^\ast \), \( a^\ast \triangledown b^\ast \) is an upper bound of \( a^\ast \) and \( b^\ast \) (\( a^\ast \leq^\ast a^\ast \triangledown b^\ast \) and \( b^\ast \leq^\ast a^\ast \triangledown b^\ast \))

2. For every infinite sequence \( \langle a^\ast_r \rangle_{r \in \mathbb{N}} \), the ascending chain \( \langle b^\ast_r \rangle_{r \in \mathbb{N}} \) defined as

\[
b^\ast_1 = a^\ast_1 \quad b^\ast_{r+1} = b^\ast_r \triangledown a^\ast_{r+1}
\]

eventually stabilizes.

Let \( \Sigma \) be an alphabet, let \( \mathcal{X} = \{X_1, \ldots, X_n\} \) be a finite set of variables, and let \( S : \{X_i = E_i\}_{i=1}^n \) be a system of equations in which each \( E_i \) is an extended regular expression over \( \Sigma \) and \( \mathcal{X} \) with no free variables. Let \( \mathcal{T} = (\mathcal{D}, \mathcal{D}_\tau, \odot, \times) \) be a tensor product domain such that both \( \mathcal{D} \) and \( \mathcal{D}_\tau \) are equipped with iteration domains and let \( \llbracket \cdot \rrbracket : \Sigma \to \mathcal{D} \) be an interpretation of the alphabet. We now show how to compute a post-fixpoint
solution $D : X \to D$ to $S$. Operationally, we compute the solution via successive approximation, where we widen and and check convergence at each occurrence of $*$ and $*\tau$.

We define $D$ as the limit of a sequence $\langle D_k : X \to D \rangle_{k \in \mathbb{N}}$: $D^0$ is the constant function $\lambda x. 0$; then, for each $k \geq 1$, define $D_k(X_i) \overset{\text{def}}{=} \text{eval}_k(E_i)$, where

\[
\begin{align*}
\text{eval}_k(a) &= \llbracket a \rrbracket \\
\text{eval}_k(X_i) &= D^{-1}(X_i) \\
\text{eval}_k(E \oplus F) &= \text{eval}_k(E) \oplus \text{eval}_k(F) \\
\text{eval}_k(E \otimes F) &= \text{eval}_k(E) \otimes \text{eval}_k(F) \\
\text{eval}_k(E^*) &= \text{cl} (\text{body}_k(E)) \\
\text{eval}_k(E \ltimes E_{\tau}) &= \text{eval}_k(E) \ltimes \text{eval}_k(E_{\tau}) \\
\text{eval}_k(E \circ F) &= \text{eval}_k(E) \circ \text{eval}_k(F) \\
\text{eval}_k(E_{\tau} \oplus T F_{\tau}) &= \text{eval}_k(E_{\tau}) \oplus T \text{eval}_k(F_{\tau}) \\
\text{eval}_k(E_{\tau} \otimes T F_{\tau}) &= \text{eval}_k(E_{\tau}) \otimes T \text{eval}_k(F_{\tau}) \\
\text{eval}_k(E_{\tau}^*) &= \text{cl}_{\tau} (\text{body}_k(E_{\tau})) \\
\end{align*}
\]

and
\[
\begin{align*}
\text{body}_k(E) &= \text{body}_{k-1}(E) \ominus_{\{D^{-1}\}} \alpha(\llbracket E \rrbracket_{D^{-1}}) \\
\text{body}_k(E_{\tau}) &= \text{body}_{k-1}(E_{\tau}) \ominus_{\tau} \alpha_{\tau}(\llbracket E_{\tau} \rrbracket_{D^{-1}}).
\end{align*}
\]

The values computed by $\text{body}_k$ and $\text{body}_{\tau}_k$ (over all $k$) implicitly define the “derived sequence” within the iteration domain. We say that $S$ stabilizes at $k$ if for every subexpression of the form $E^*$ that appears in any $E_i$, we have

\[
\text{body}_k(E) = \text{body}^{k-1}(E),
\]

and for every subexpression of the form $E_{\tau}^*$ that appears in any $E_i$, we have

\[
\text{body}_{\tau}_k(E_{\tau}) = \text{body}^{k-1}_{\tau}(E_{\tau}).
\]

**Theorem 3.18.** There exists a $k$ such that $S$ stabilizes at $k$, and $D^k$ is a post-fixpoint solution to $S$.

**Proof:** See App. A.

**Example 3.19.** Ex. 3.13 is a high-level overview of how our approach analyzes the Fibonacci function. Here we illustrate
one step of this process in greater detail: computing the third iterate $D^3$ of the system $\hat{S}$. Write the system $\hat{S}$ as

$$\hat{S} : \{ X = \text{base} \times (\text{rec}^{*\gamma}) \}, \text{ where}$$

$$\text{base} \overset{\text{def}}{=} \begin{cases} n := p \cdot n \leq 1 \cdot r := 1 \\ t := r \cdot p := n - 2 \cdot X \cdot r := r + t \end{cases}$$

The value of $D^3(X)$ is computed by evaluating the right-hand-side of the equation:

$$D^3(X) = \text{eval}^3 (\text{base} \times (\text{rec}^{*\gamma}))$$

$$= \text{eval}^3 (\text{base}) \times \text{eval}^3 (\text{rec}^{*\gamma})$$

$$= \text{eval}^3 (\text{base}) \times \text{cl}_\tau (\text{body}^3_\tau (\text{rec})).$$

The expression $\text{base}$ contains no variables, so it can be re-interpreted within the CRA quasi-weight domain:

$$\text{eval}^3 (\text{base}) = \begin{cases} n := p \cdot p \leq 1 \cdot r := 1 \\ t := r \cdot p := n - 2 \cdot X \cdot r := r + t \end{cases}$$

The term $\text{body}^3_\tau (\text{rec})$ refers to the third-iteration loop-body polyhedron (called $\beta^3$ in Ex. 3.13). Following the definition of $\text{body}^3_\tau$ above, we have

$$\text{body}^3_\tau (\text{rec}) = \text{body}^2_\tau (\text{rec}) \vee \tilde{\alpha}_{\text{poly},\tau}([\text{rec}]^{D^2}).$$

where $\text{body}^2_\tau (\text{rec})$ is the second-iteration loop-body polyhedron ($\beta^2$ in Ex. 3.13) and $[\text{rec}]^{D^2}$ is the value obtained by evaluating rec in the tensor-product domain of CRA, using $D^2(X)$ to interpret the variable $X$:

$$[\text{rec}]_{D^2} = \begin{cases} n := p \cdot n > 1 \cdot p := n - 1 \\ t := r \cdot p := n - 2 \cdot D^2(X) \cdot r := r + t \end{cases}$$

$$(n' = p \wedge n' > 1 \wedge p' = n - 1 \wedge r' = r \wedge t' = t)$$

$$= \circ \bigg( \begin{cases} p' = n - 2 \leq 1 \wedge r' = 1 + t' \\ \vee (\exists k. k \geq 1 \wedge k = n - 3 \wedge r' = k + 1 + t' \wedge n \leq 5) \\ n' = p \wedge n' > 1 \wedge p' = n - 1 \wedge r' = r \wedge t' = t \\ \vee (\exists k. k \geq 1 \wedge k = n - 3 \wedge r' = k + 1 + t' \wedge n \leq 5) \\ \vee (\exists k. k \geq 1 \wedge k = n - 2 \wedge r' = 1 + t') \\ \vee (\exists k. k \geq 1 \wedge k = n - 3 \wedge r' = k + 1 + t' \wedge n \leq 5) \\ \vee (\exists k. k \geq 1 \wedge k = n - 2 \wedge r' = 1 + t') \\ \vee (\exists k. k \geq 1 \wedge k = n - 3 \wedge r' = k + 1 + t' \wedge n \leq 5) \\ \vee (\exists k. k \geq 1 \wedge k = n - 2 \wedge r' = 1 + t') \\ \vee (\exists k. k \geq 1 \wedge k = n - 3 \wedge r' = k + 1 + t' \wedge n \leq 5) \\ \vee (\exists k. k \geq 1 \wedge k = n - 2 \wedge r' = 1 + t') \\ \vee (\exists k. k \geq 1 \wedge k = n - 3 \wedge r' = k + 1 + t' \wedge n \leq 5) \\ \vee (\exists k. k \geq 1 \wedge k = n - 2 \wedge r' = 1 + t') \\ \vee (\exists k. k \geq 1 \wedge k = n - 3 \wedge r' = k + 1 + t' \wedge n \leq 5) \end{cases} \bigg)$$
We then project out the local variables $n$ and $t$ and apply the abstraction operator to obtain the following polyhedron:

\[
\tilde{\alpha}_{\text{poly},\mathcal{T}}(\|\text{rec}\|_{\mathcal{D}^2}) = (p' = p - 1 \land 1 < p \leq 5 \land t' = x \land t' \geq r + 1).
\]

Returning to the evaluation of $\text{body}^3_T(\text{rec})$, we apply polyhedra widening to the previous iterate $\beta^2$ and $\tilde{\alpha}_{\text{poly},\mathcal{T}}(\|\text{rec}\|_{\mathcal{D}^2})$:

\[
\text{body}^3_T(\text{rec}) = \beta^2 \lor \tilde{\alpha}_{\text{poly},\mathcal{T}}(\|\text{rec}\|_{\mathcal{D}^2})
= (p' = p - 1 \land 1 < p \land t' = x \land t' \geq r + 1)
\]

Finally, returning to the evaluation of $\mathcal{D}^3(X)$, we have

\[
\mathcal{D}^3(X) = \text{eval}^3(\text{base}) \land c_{\mathcal{T}}(\text{body}^3_T(\text{rec}))
= \begin{pmatrix}
p' = p \leq 1 \land t' = 1 \\
\lor \exists k. k \geq 1 \land t' \geq k + 1 \land k = p - 1
\end{pmatrix}.
\]

Algorithm NPA-TP-GJ. Putting all the pieces together, we now state Algorithm NPA-TP-GJ for solving non-linear systems of equations.

Algorithm 3.20 (NPA-TP-GJ). The input is a system of equations $S : \{X_i = R_i\}_{i=1}^n$ with regular right-hand sides over an alphabet $\Sigma$ and a finite set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$, a tensor-product domain $\mathcal{T} = (\mathcal{D}, \otimes, \mathcal{D}_T, \mathcal{D}_T)$, and an interpretation $\|\cdot\| : \Sigma \to \mathcal{D}$. The output is a mapping $\mathcal{D} : \mathcal{X} \to \mathcal{D}$ such that for all $i$ and all $p \in \text{Paths}_S(X_i)$, $\mathcal{D}(X_i) \supseteq \|p\|$.

1. Use Alg. 3.14 to transform $S$ into a system of equations $\hat{S} : \{X_i = \hat{E}_i\}_{i=1}^n$ in which each $\hat{E}_i$ is an extended regular expression over $\Sigma$ and $\mathcal{X}$ that has no free variables.

2. $k \leftarrow 0$; $D^0 \leftarrow \lambda x.0$

3. Repeat
   a) $k \leftarrow k + 1$
   b) $D^k(X_i) \leftarrow \text{eval}^k(\hat{E}_i)$

   until $\hat{S}$ stabilizes

4. Return $D^k$

   In the degenerate case when $S$ is linear, left-linear, or right-linear, the call on Alg. 3.14 in step (1) returns a solution to $S$ (or, more precisely, a system of equations $\hat{S}$ in which the right-hand-sides do not contain any variables). In this case, the evaluation loop stabilizes on the first pass.
3.4 Implementation and Experiments

NPA-TP-GJ is implemented on top of the WALi [Kidd et al., 2007] system for weighted pushdown systems. In particular, it uses the implementation of Tarjan’s method from the FWPDS solver [Lal and Reps, 2006] of WALi to create the initial equation system with regular right-hand sides from a specification of the problem to be solved as a weighted pushdown system. We instantiated NPA-TP-GJ for ICRA by augmenting the implementation of CRA [Farzan and Kincaid, 2015] (which, in turn, makes use of APRON [APRON, [n.d.]] and Z3 [de Moura and Bjørner, 2008]) with $\odot$ and $\ltimes$ to create a tensor-product domain, and by implementing a subclass of WALi’s tensor-product domain interface that makes appropriate calls to operations of the CRA abstract domain.

The implementation supports programs with multiple procedures, including recursion and mutual recursion. It supports local variables via the method explained in [Reps et al., 2016, §8]. The merge function used when returning from a procedure call is based on [Reps et al., 2016, Eqn. (54)].

Our experiments with ICRA were designed to answer the following questions:

1. How fast is ICRA, compared to other tools?
2. How many assertions can ICRA prove, compared to other tools?
3. How often is ICRA able to prove bounds on a program’s resource usage, compared with C4B [Carbonneau et al., 2015a]?

Our experiments showed that ICRA has broad overall strength, as shown by its aggregate performance on three independently developed benchmark suites.

Timings (with a timeout limit of 300 seconds) were taken on a virtual machine (using Oracle VirtualBox), with a guest OS of Ubuntu 14.04, host OS of Red Hat Enterprise Linux 6, and a 3.2 GHz quad-core Intel Core i5-4570 host CPU.

Assertion Checking. To assess ICRA’s assertion-checking capabilities, we ran it on (i) the benchmarks in the Loops and Recursive subcategories of the Integers and Control Flow category from SV-COMP16 [SVCOMP16, 2016]; (ii) the suite of programs from C4B [Carbonneau et al., 2015b, App. A], annotated with upper-bound resource assertions of the bounds reported by C4B,2 and (iii) three groups of new recursive programs: two groups are modified versions of the SV-COMP16 subcategories loop-lit and loop-new in which all loops were converted into recursive procedures; the third group consists of a few newly-written recursive programs, plus a few recursive programs from the test suite of the CRA static-analysis tool. We compared ICRA’s results against four state-of-the-art software model checkers: CPAchecker [Beyer and Keremoglu, 2011] and Ultimate

---

2 C4B obtains bounds for 34 of the 35 programs. The assertion in the 35th program is the bound from SPEED [Gulwani et al., 2009].
Table 3.1: Column 2 shows the breakdown of programs into ones with valid and invalid assertions. “X/Y” means that X is the number of programs containing valid assertions and Y is the number containing invalid assertions. The total number of programs is X + Y. The two columns for each tool show the running time (in seconds) and the number of assertions proved. In each row, the fastest time and the greatest number of assertions proved are shown in bold font.

<table>
<thead>
<tr>
<th>Benchmark Suite</th>
<th>Total</th>
<th>ICRA #A</th>
<th>UAut. #A</th>
<th>CPA #A</th>
<th>LPI #A</th>
<th>SEA #A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
<td>Time</td>
</tr>
<tr>
<td>recursive</td>
<td>18/7</td>
<td>40.7</td>
<td>7</td>
<td>1952.1</td>
<td>8</td>
<td>1817.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>62.0</td>
<td>0</td>
<td>1334.0</td>
</tr>
<tr>
<td>rec.-simple</td>
<td>36/38</td>
<td>168.7</td>
<td>21</td>
<td>6979.3</td>
<td>28</td>
<td>2760.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>179.5</td>
<td>3</td>
<td>743.8</td>
</tr>
<tr>
<td>Rec. (tot.)</td>
<td>54/45</td>
<td>209.4</td>
<td>28</td>
<td>8931.4</td>
<td>36</td>
<td>4578.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>241.5</td>
<td>3</td>
<td>2077.8</td>
</tr>
<tr>
<td>loop-accel.</td>
<td>19/16</td>
<td>20.8</td>
<td>13</td>
<td>6696.5</td>
<td>7</td>
<td>4565.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4227.7</td>
<td>13</td>
<td>2713.1</td>
</tr>
<tr>
<td>loop-invgen</td>
<td>18/1</td>
<td>53.1</td>
<td>16</td>
<td>1876.2</td>
<td>7</td>
<td>4909.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1282.3</td>
<td>15</td>
<td>506.0</td>
</tr>
<tr>
<td>loop-lit</td>
<td>15/1</td>
<td>316.5</td>
<td>12</td>
<td>2722.9</td>
<td>5</td>
<td>2720.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>444.9</td>
<td>13</td>
<td>305.2</td>
</tr>
<tr>
<td>loops</td>
<td>34/32</td>
<td>209.7</td>
<td>22</td>
<td>3984.1</td>
<td>19</td>
<td>4380.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3356.8</td>
<td>26</td>
<td>1821.5</td>
</tr>
<tr>
<td>loop-new</td>
<td>8/0</td>
<td>304.8</td>
<td>7</td>
<td>2147.9</td>
<td>1</td>
<td>1866.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>929.6</td>
<td>4</td>
<td>302.8</td>
</tr>
<tr>
<td>Loops (tot.)</td>
<td>94/50</td>
<td>904.8</td>
<td>70</td>
<td>17427.6</td>
<td>39</td>
<td>18442.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10241.3</td>
<td>71</td>
<td>5648.6</td>
</tr>
<tr>
<td>C4B</td>
<td>35/0</td>
<td>30.3</td>
<td>30</td>
<td>6156.6</td>
<td>1</td>
<td>7817.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6726.7</td>
<td>0</td>
<td>1867.6</td>
</tr>
<tr>
<td>misc</td>
<td>10/4</td>
<td>76.7</td>
<td>10</td>
<td>492.2</td>
<td>8</td>
<td>334.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>332.2</td>
<td>1</td>
<td>5.3</td>
</tr>
<tr>
<td>rec-loop-lit</td>
<td>15/1</td>
<td>312.7</td>
<td>9</td>
<td>2755.5</td>
<td>3</td>
<td>51.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40.4</td>
<td>0</td>
<td>922.6</td>
</tr>
<tr>
<td>rec-loop-new</td>
<td>8/0</td>
<td>6.2</td>
<td>5</td>
<td>1546.9</td>
<td>2</td>
<td>25.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>19.6</td>
<td>0</td>
<td>905.7</td>
</tr>
<tr>
<td>Misc.-Rec.</td>
<td>33/5</td>
<td>395.6</td>
<td>24</td>
<td>4794.6</td>
<td>13</td>
<td>410.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>392.2</td>
<td>1</td>
<td>1833.7</td>
</tr>
</tbody>
</table>

Automizer [Heizmann et al., 2013], based on predicate abstraction; LPI [Karpenkov et al., 2016], based on policy iteration; and SeaHorn [Gurfinkel et al., 2015], a Horn-clause solver based on IC3. We ran the versions of these tools that were submitted to SV-COMP16.

Tab. 3.1 shows that the answers to questions 1 and 2 are

1. Overall ICRA was faster, sometimes dramatically so. The main reason is that ICRA had many fewer timeouts.

2. ICRA correctly handled more valid assertions than all the other tools except for SeaHorn.

The five tools have different success/failure-versus-timeout profiles, and there are examples on which one tool succeeds quickly but the other tools time out or fail. For instance, ICRA took only 0.73 seconds to prove the assertion in rec-gauss_sum_true-unreach-call (shown below), and was the only tool to do so. This benchmark, from rec-loop-new, is a recursive equivalent of an iterative SV-COMP16 benchmark.
int n, sum, i;
void rec() {
    if (i <= n) { sum = sum + i; i++; rec(); }
}
int main() {
    n = __VERIFIER_nondet_int();
    __VERIFIER_assume(1 <= n && n <= 1000);
    sum = 0; i = 1; rec();
    __VERIFIER_assert(2*sum == n*(n+1));
    return 0;
}

In contrast, for underapprox_true-unreach-call1 (see below), ICRA was not able to establish the assertion, but the four other tools were able to do so.

int main(void) {
    unsigned int x = 0, y = 1;
    while (x < 6) { x++; y *= 2; }
    __VERIFIER_assert(y % 3);
}

We also compared the tools’ performance by computing two numbers \((X, Y)\)—each the geometric mean over a set of benchmarks—of the ratios of the other tool’s times divided by ICRA’s times. \(X\) is computed using all benchmarks; \(Y\) is computed using only benchmarks on which neither ICRA nor the other tool timed out or gave an error. The numbers are Ultimate Automizer: (28.2, 9.03), CPAchecker: (16.5, 6.21), LPI: (6.31, 5.33), and SeaHorn: (0.93, 0.49). The value \(Y = 0.49\) indicates that SeaHorn is roughly twice as fast as ICRA on many examples. Because SeaHorn has a substantial number of timeouts, the value \(X = 0.93\) is a bit misleading: there are a nontrivial number of examples on which ICRA has much better performance; it is a coincidence that the geometric mean ends up being so close to 1.0.

Applications in Resource-Bound Analysis. For the experiments discussed above, the C4B benchmarks were annotated with upper-bound resource assertions. We also explored the use of ICRA for generating upper and lower bounds. (C4B itself can only establish upper bounds.)

The procedure in Fig. 3.4(a) illustrates the application of ICRA to resource-bound analysis. Statements of the form \(\text{tick}(k)\) represent manipulations of a resource such as memory or time, by consuming some of the
void perform(int n) {
    tick(7);
    if(*)
    tick(2);
    if(n>0) perform(n-1);
    tick(-3);
}

void start(int n, int m) {
    int x = 0; int y = 0;
    for (;;) {
        if (x < n) {x++;y++;}
        else if (y < m) {x++;y++;}
        else break;
        tick(1);
    }
}

(a) (b)

Figure 3.4: (a) Recursive procedure to show upper and lower resource bounds; (b) speed_popl10_simple_single_2.c from C4B.

resource (if \(k\) is positive) or freeing some of the resource (if \(k\) is negative).

ICRA can compute terms that upper-bound and lower-bound both the final resource usage and the high-water mark of resource usage.\(^3\) When computing bounds, it is useful to treat positive and negative values of a program variable separately, so ICRA searches for bounds that include terms of the form \(\max(0, x)\) and \(\max(0, y - x)\) (as in [Carbonneau et al., 2015b]), and we use the notation \([0, x]\) and \([x, y]\), respectively, for such terms. For the function perform, ICRA computes the following four bounds, each of which is achieved by following paths of a particular form:

<table>
<thead>
<tr>
<th>Bound</th>
<th>Path (line numbers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{final} \geq 4[0, n] + 4)</td>
<td>1 (\rightarrow) 2 (\rightarrow) 4 (\rightarrow) perform(n-1) (\rightarrow) 5</td>
</tr>
<tr>
<td>(\text{final} \leq 6[0, n] + 6)</td>
<td>1 (\rightarrow) 2 (\rightarrow) 3 (\rightarrow) perform(n-1) (\rightarrow) 5</td>
</tr>
<tr>
<td>(\text{hwm} \geq 7[0, n] + 7)</td>
<td>1 (\rightarrow) 2 (\rightarrow) 4 (\rightarrow) perform(n-1) (\rightarrow) 5</td>
</tr>
<tr>
<td>(\text{hwm} \leq 9[0, n] + 9)</td>
<td>1 (\rightarrow) 2 (\rightarrow) 3 (\rightarrow) perform(n-1) (\rightarrow) 5</td>
</tr>
</tbody>
</table>

In the last two paths, each occurrence of \(\text{tick}(-3)\) encountered at line 5 has no effect on the value of the high-water mark.

On the C4B test suite, ICRA is able to generate upper bounds on resource usage for 28 of the 35 programs, and nontrivial lower bounds on resource usage for 30 of the programs. Whenever ICRA generates an upper bound, it is the same as C4B’s bound (modulo four cases where it was unclear where to materialize C4B’s implicit \(\text{tick}\) call, which caused ICRA’s bound to be one unit different than C4B’s bound). In seven cases, ICRA’s bound is more precise: it generates both C4B’s bound, plus one or more additional incomparable

\(^3\) The result of using ICRA for resource-bound analysis is only sound under the assumption that, for all inputs, the program in question terminates.
constraints. For example, for the procedure shown in Fig. 3.4(b), ICRA generates the following bounds:

\[
\text{final} \leq ||0, n|| + ||n, m|| \quad \text{final} \leq ||0, m|| + ||m, n|| \quad \text{final} \leq ||0, n|| + ||0, m||
\]

C4B gets just the third, which is incomparable to the others.

### 3.5 Related Work

The NPA-TP-GJ algorithm and its instantiation for ICRA rely heavily on prior work on CRA [Farzan and Kincaid, 2015], NPA [Esparza et al., 2010], and NPA-TP [Reps et al., 2016]. In particular, the NPA-TP-GJ algorithm adopts NPA-TP’s tensor-product operation so that various terms in a non-linear equation can be rewritten into the “symbolically” left-linear form

\[
X = F \oplus X \bowtie F_T,
\]

and then further rewritten as

\[
X = F \bowtie F_T^*.
\]

When we first began to think about extending CRA to create a context-sensitive interprocedural version that handles loops and recursion, our initial thought was to use NPA-TP, because NPA-TP provides a way to harness Tarjan’s path-expression method for interprocedural analysis. That property meshes well with the way CRA couples its recurrence-solving step with its reinterpretation of Kleene-star. However, there is a mismatch between CRA and NPA-TP, which presented us with two substantial challenges:

- The CRA domain has infinite ascending chains.
- The problem of determining whether two CRA transition formulas are equivalent is undecidable.

Consequently, if we merely instantiated the NPA-TP framework with the CRA domain, the resulting algorithm would not be guaranteed to converge, and even if it did converge, we would not necessarily know that it had. The design of the NPA-TP-GJ framework was driven by the need to address these challenges for CRA.

ICRA relies on the fact that the logic fragment that CRA employs to solve recurrences supports effective equivalence. That property—along with the widening performed during the \textit{body} subroutine of \textit{eval}, and the stabilization result given in Thm. 3.18—ensures that ICRA will always terminate.

The elimination step in NPA-TP-GJ is related to Gauss-Jordan elimination, in that variables are eliminated in some order. Unlike in Gauss-Jordan elimination, variables are only partially eliminated; so-called “free variables” are eliminated, whereas variables that appear under \(^*\) or \(^*\) are not eliminated. Past work on Gauss-Jordan elimination for semirings or regular expressions includes [Tarjan, 1981], [Gondran and Minoux, 2010, §4.5], [Rote, 1990, §4], [Carré, 1971, §7], and [Backhouse and Carré, 1975, §4].

The power of CRA is due to the ability of the iteration operator to summarize loops. There are a number of related techniques for loop summarization [Gulwani and Zuleger, 2010; Kroening et al., 2008; Biallas
et al., 2012] and abstract loop-acceleration [Leroux and Sutre, 2007; Gonnord and Schrammel, 2014; Jeannet et al., 2014]. (The latter computes the post-image of a loop, rather than a summary.) Loop summarization in CRA is based on computing the closed forms for recurrences satisfied by loop bodies. ICRA harnesses the power of CRA’s loop summarization for computing summaries of recursive procedures by computing abstract transitive closures of tensored transition formulas; i.e., in ICRA, the input and output of the iteration operator are formulas that each have four distinct vocabularies.

Ganty et al. [Ganty et al., 2016] have created a Newton-based analysis tool for recursive programs over the integers. Like ICRA, it repeatedly applies a more basic analyzer to a transformed version of the original program. However, there are several differences in capabilities and approach.

- Their tool computes underapproximations of procedures; ICRA computes overapproximations.
- When used with FLATA [Bozga et al., 2012], their tool is limited to finding summaries that are linear inequalities with at most two variables. ICRA can compute polynomial summaries.
- Their tool performs a transformation that restructures the linearized, possibly recursive, multi-procedure program $P_k$ obtained for Newton-round $k$ into a non-recursive program $P'_k$. The transformation introduces occurrences of havoc and assume so that, during the analysis of $P'_k$, the various parts of $P_k$ are analyzed in an order different from the order in which they would be executed in $P_k$. In contrast, ICRA is based on algebraic program analysis, and relies on the properties of its tensor-product domain to tease out the properties of the original program from the restructured equation system that it creates.

The relationship of ICRA to Ganty et al. is similar to the relationship of Lal et al. [Lal et al., 2008] to Lal and Reps [Lal and Reps, 2009]. The subjects in the two pairs of papers are different—interprocedural analysis of sequential programs and analysis of concurrent programs under a context-switch bound, respectively. However, in each pair, the first paper performs algebraic program analysis using a tensor-product domain, whereas the second uses a code transformation that introduces occurrences of havoc and assume to restructure the code.

Recently, a new iteration scheme was proposed by Meyer and Muskalla [Meyer and Muskalla, 2016], called Munchausen Iteration (MI). MI is based on an iteration step that (a) works on a vector of expressions, and (b) substitutes the current approximant into itself. Both aspects have some similarities with NPA-TP-GJ, but the substitution step in MI is more aggressive. The benefit of MI is that, in principle, MI takes exponentially fewer rounds than NPA; however, Meyer and Muskalla indicate that MI is not yet an algorithm, “. . . so far, we do not know how to extract information from the [structures that represent answers].” Also, when the
ascending-chain condition does not hold, it is not clear how widening would be performed on the symbolic structures that MI employs.

Although the performance of ICRA in the tool comparison reported in §3.4 is encouraging, it is important to note that ICRA is an invariant generator, while CPAchecker, Ultimate Automizer, LPI, and SeaHorn are software model checkers. The standard trade-offs apply. Software model checkers are able to prove challenging program properties by employing abstraction refinement to tailor abstractions to the property at hand, but may refine abstractions indefinitely and fail to terminate. ICRA can be used to compute rich program invariants (rather than just answer verification queries) but is not currently able to compute counter-example witnesses to assertion failures. (We illustrated one use of ICRA’s facility for invariant generation in §3.4 by employing ICRA to compute upper and lower bounds on resource usage.)

### 3.6 Conclusion

This chapter addresses the problem of creating a context-sensitive interprocedural version of CRA that handles loops and recursion—including non-linear recursion—in a uniform way. The method presented here adopts, but adapts in significant ways, ideas used in the NPA-TP framework [Reps et al., 2016].

Our presentation—in terms of the algebraic properties required for the approach—has allowed us to state precisely the interface and properties required for NPA-TP-GJ to work. Although we currently have only one instantiation of NPA-TP-GJ—namely, ICRA—we believe that the algebraic formulation is a valuable contribution. We hope to apply NPA-TP-GJ to programs with loops that manipulate strings or arrays, as well as to the analysis of probabilistic programs (for computing expectations of loops).

The results presented here also offer a number of other opportunities for future work. For instance, it would be valuable to know whether some elimination orders for the Gauss-Jordan-like step of ICRA are better than others (and why). We also have ideas about how to incorporate additional methods for solving recurrence equations in the interpretation of * and **. Finally, it would be desirable to be able to perform witness tracing for ICRA.
Chapter 4

Non-linear Arithmetic

4.1 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 [2004] and Kovács [2008] 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.

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 [Rodríguez-Carbonell and Kapur, 2004; Kovács, 2008]) are capable of.

In this chapter, we present extensions to the numerical-reasoning techniques underlying CRA, and demonstrate that these extensions enable CRA to establish many non-linear numerical invariants. The contributions of the 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 [Reps et al., 2004; Thakur and Reps, 2012], which, given an arbitrary non-linear formula $\varphi$, computes a wedge that over-approximates $\varphi$. (See §4.4.)

- 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 [Kauers and Paule, 2011, §4.2]. (See §4.5.)

- 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 [1967]. 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. (See §4.6.)

Our approach builds upon the recent work of Kincaid et al. [2017], which extended CRA so that it can analyze recursive programs using essentially the same approach that it uses to handle loops.

Organization. §4.2 illustrates the main features of our method via a series of examples. §4.3 presents relevant background material. §4.4 presents the wedge abstract domain. §4.5 describes the method used in our system for extracting a recurrence relation from a wedge. §4.6 presents OCRS. §4.7 presents experimental results. §4.8 discusses related work.

4.2 Overview and Problem Statement

Our system follows in the tradition of CRA [Farzan and Kincaid, 2015] and ICRA [Kincaid et al., 2017] in how it combines symbolic analysis and abstract interpretation:

---

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.
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;
int 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;
    }
}

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

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

- 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 4.1. Consider the loop nest shown in Fig. 4.1(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.
Fig. 4.2 shows the recurrences created for the program in Fig. 4.1(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. 4.2, 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 \( N \times k \). Consequently, the body of the outer loop (at nesting-level 1) has the following non-linear transition formula:

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

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 \( N \times N \), the recurrence-extraction technique of Farzan and Kincaid [2015] 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 \( N \times N \times N \).

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 4.2.** Consider the loop shown in Fig. 4.1(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 \( \text{array\_size} \) 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 inequation.
Next, the recurrence solver finds the solution to this recurrence:

$$\text{steps}^{[k]} = \text{steps}^{[0]} + k \land \text{pos}^{[k]} \geq 2^k \text{pos}^{[0]},$$

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, $\text{pos} < 2(\text{array\_size})$, because otherwise the loop would have exited earlier. We conclude that $\text{steps} < 1 + \log_2(\text{array\_size})$, if $\text{array\_size} > 0$, 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 4.3.** Consider the loop shown in Fig. 4.1(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 $y^{[1]} = -\frac{1}{2}((-1)^1 - i^1)x^{[0]} + \frac{1}{2}((-1)^1 + i^1)y^{[0]} = -i \ast x$, and the relevant part of the formula for $y$ is:

$$y' = 1 \land y' = -i \ast i.$$  \hspace{1cm} (4.1)

If $i$ is interpreted as the imaginary unit, then Eqn. (4.1) 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. (4.1) in real arithmetic with $i$ as a symbolic constant. In real arithmetic, we know that for all $a$, $a \ast a \geq 0$, and thus $y' = -i \ast i \leq 0$, which is inconsistent with $y' = 1$; thus, Eqn. (4.1) is unsatisfiable, which falsely suggests that errorlabel is unreachable.
To avoid problems of this kind, we developed a recurrence solver (described in §4.6) 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, the chapter introduces implicitly interpreted functions (IIFs) (§4.6.4). 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 4.4.** Consider the program shown in Fig. 4.1(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. [2004]. For procedure \( \text{fib} \), we will assume that variable \( n \) is a low-security input, \( \text{high} \) is a high-security input, and the return value of \( \text{fib} \) is a low-security output. The information-flow property that we wish to establish is that \( \text{fib} \)'s return value is unaffected by the value passed in for \( \text{high} \).

Secure information flow is not a safety property: 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 \( P \) can be encoded as a safety property of a more complicated program that consists of \( P \) followed by \( P' \)—a second copy of \( P \) with its variables suitably renamed. By this means, secure information flow can be reduced to an assertion-checking problem in the program \( P; P' \). They call this technique self composition.

In the program above, the self-composition technique is captured by procedure \( \text{main} \), which uses two calls to \( \text{fib} \) rather than two copies of \( \text{fib} \). In \( \text{main} \), the statement \( \text{assume}(n_1 == n_2) \) ensures that the low-security inputs to the two calls on \( \text{fib} \) are arbitrary, but equal. The absence of any constraint on \( \text{high}_1 \) and \( \text{high}_2 \) means that they may have different values in the two calls to \( \text{fib} \). The goal is to prove the assertion on the last line of \( \text{main} \), which claims that the low-security outputs \( \text{obs}_1 \) and \( \text{obs}_2 \) are equal, and hence unaffected by any difference in the values of high-security inputs \( \text{high}_1 \) and \( \text{high}_2 \).

To convince yourself that the assertion does indeed hold, observe that the two while loops in \( \text{fib} \) both compute the \( n^{th} \) Fibonacci number, albeit in slightly different ways. Thus, because the values of \( n_1 \) and \( n_2 \) are equal, \( \text{obs}_1 \) and \( \text{obs}_2 \) will be equal—even though \( \text{high}_1 \) and \( \text{high}_2 \) might differ, which would cause different while loops to be executed during the two calls on \( \text{fib} \).

---

Technically, we are referring to the so-called “termination-insensitive” secure-information-flow problem.
The challenge in analyzing this example is that the return value of $\text{fib}$ is a complicated function of its inputs. Classically, this function would be represented using irrational numbers: the return value of $\text{fib}(n, \text{high})$ is $\frac{1}{\sqrt{5}}((1+\sqrt{5})^n - (1-\sqrt{5})^n)$. As in the previous example (Ex. 4.3), 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 $\text{fib}$ 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 §4.6.3).

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 (§4.4), which encodes non-linear relationships by extending the abstract domain of polyhedra with new dimensions that represent non-linear terms. §4.5 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 [1967]. (See §4.6.)

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 §4.8 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.

4.3 Background

4.3.1 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 $A z \geq b$, 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 $A z \geq b$ is a (convex) polyhedron.

4.3.2 (Non)linear Real Arithmetic with Uninterpreted Functions

Let $\Sigma = (F, ar)$ 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:

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

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

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:

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

We will make use of two different semantics for this logic. A linear structure $L$ consists of an interpretation of each function symbol $f \in \Sigma$ as a function $f^L : \mathbb{R}^{ar(f)} \to \mathbb{R}$; the semantics of numerals, addition, and numeral multiplication are the usual ones. We write $\models_L \phi$ to denote that $L$ is a linear structure that satisfies the sentence $\phi$; write $\phi \models_1 \psi$ if every linear structure that satisfies the sentence $\phi$ also satisfies the sentence $\psi$; and write $\phi \equiv_1 \psi$ if $\phi \models_1 \psi$ and $\psi \models_1 \phi$. A non-linear structure $\mathfrak{N}$ is a linear structure that interprets the designated function symbols in $\Sigma_N$ with their usual interpretation on their domain of definition (i.e., $\times^\mathfrak{N}$ is
multiplication, \( \log_2^\gamma \) 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 is a vocabulary that contains a designated set \( \text{Var} = (x_1, \ldots, x_n) \) of constant symbols and also a disjoint set \( \text{Var}' = (x'_1, \ldots, x'_n) \) 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:

\[
\gamma_N(\phi) \overset{\text{def}}{=} \{(a, a') : \exists \mathcal{M} \phi, a_1 = x^a_1, \ldots, a'_n = x'^a_n\}.
\]

### 4.3.3 Polynomials, Ideals, and Gröbner Bases

We use \( \mathbb{Q}[z_1, \ldots, z_n] \) to denote the ring of polynomials with rational coefficients over the variables \( z_1, \ldots, z_n \). An **ideal** is a set \( I \subseteq \mathbb{Q}[z_1, \ldots, z_n] \) of polynomials over the variables \( z_1, \ldots, z_n \) that contains \( 0 \), is closed under addition, and such that for any \( p \in I \) and any \( q \in \mathbb{Q}[z_1, \ldots, z_n] \), we have \( pq \in I \). Intuitively, one may think of an ideal \( I \) as a set of polynomial equations \( \{ p = 0 : p \in I \} \)—the closure conditions for ideals can be read as simple inference rules: \( 0 = 0 \), if \( p = 0 \) and \( q = 0 \) then \( p + q = 0 \), and if \( p = 0 \) then \( pq = 0 \) for any \( q \). Any set of polynomials \( P \subseteq \mathbb{Q}[z_1, \ldots, z_n] \) generates an ideal \( (P) \), which is the smallest ideal containing \( P \):

\[
(P) \overset{\text{def}}{=} \left\{ \sum_{i=0}^k q_ip_i : k \geq 0, q_i \in \mathbb{Q}[z_1, \ldots, z_n], p_i \in P \right\}
\]

The set of polynomials \( P \) is called a **basis** for \( (P) \).

A Gröbner basis for an ideal is a particular kind of basis that has many computational applications. For a good introduction to Gröbner basis theory, see [Cox et al., 2015]. In the remainder of this section we give a short overview of the properties of Gröbner bases that are needed for the rest of this chapter.

Fix a set of variables \( z_1, \ldots, z_n \). A **monomial** is a product of variables \( m = z_1^{d_1} \cdots z_n^{d_n} \). The sum \( d_1 + \cdots + d_n \) is called the **total degree** of \( m \). A **monomial ordering** \( \preceq \) is a total ordering on monomials such that for any \( m \preceq n \) and any monomial \( v \), we have \( mv \preceq nv \) (some important examples to follow). The **leading term** of a polynomial \( a_1 m_1 + \cdots + a_n m_k \) is the greatest monomial among \( m_1, \ldots, m_k \) (with respect to a given monomial order).

A monomial ordering gives us a way of orienting a polynomial equation \( p = 0 \) into a rewrite rule \( m \rightarrow q \), where \( m \) is the leading term of \( p \) and \( p = am - aq \) for some nonzero \( a \in \mathbb{Q} \). Fixing a monomial ordering, we
may think of a set of polynomials as a rewrite system. A **Gröbner basis** $G$ is a set of polynomials that results in a **confluent** (and terminating) rewrite system—every polynomial can be rewritten to a unique normal form. There is a procedure (Buchberger’s algorithm [Buchberger, 1976], among others) that computes a Gröbner basis for any ideal under any monomial ordering. Given a Gröbner basis $G$ for an ideal $I$ we use $\text{red}_G : \mathbb{Q}[z_1, ..., z_n] \rightarrow \mathbb{Q}[z_1, ..., z_n]$ to denote the function that maps polynomials to their normal forms. Two important properties of the reduction function are:

1. $p \in I$ if and only if $\text{red}_G(p) = 0$, and
2. The leading term of $\text{red}_G(p)$ is $\preceq$ the leading term of $p$.

One monomial ordering of interest is the *degrevlex* (“degree reverse lexicographic”) ordering, which compares monomials first by total degree and then by reverse lexicographic order (the reverse lexicographic part is not essential for our applications—any total order that extends degree ordering will do). Property 2 above implies that if $G$ is a Gröbner basis with respect to degrevlex order, then the degree of $\text{red}_G(p)$ never exceeds the degree of $p$.

Gröbner bases can also be used to project variables out of polynomial systems (similarly to Gaussian elimination for linear systems). Fix a set of variables $X \subseteq \{z_1, ..., z_n\}$ that we wish to eliminate. Any monomial can be written as a product $m = m_X m_{\bar{X}}$, where $m_X$ is a monomial over $X$ and $m_{\bar{X}}$ is a monomial over the remaining variables. We define the *elimination order* w.r.t $X$ as follows: $m \preceq_X n$ if $m_X \preceq_X n_X$ or if $m_X = n_X$ and $m_{\bar{X}} \preceq n_{\bar{X}}$, where $\preceq$ denotes degrevlex order. Property 2 above implies that if $G$ is a Gröbner basis for an ideal $I$ under the elimination order w.r.t. $X$, then if $p$ is free of $X$ variables then so is $\text{red}_G(p)$. Considering $\text{red}_G(p)$ as the canonical representative of the equivalence class of polynomials that reduce to $\text{red}_G(p)$, the latter property means that if any polynomial in the equivalence class is free of $X$ variables (albeit there may be no such polynomial), then so is the representative.

### 4.3.4 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 (§4.4), we use Gröbner bases and congruence closure to reason about non-linear equations, and inference rules to reason about non-linear inequations. In §4.5, we use Gröbner bases to extract non-linear recurrence relations from the body of a loop (represented by a wedge). In §4.6, 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. 4.2, logarithmic relationships are obtained as derived information from exponential relationships (cf. Fig. 4.3).

4.4 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:

\[
w \in \text{Wedge}(\Sigma) := t \geq 0 \lor t > 0 \lor w_1 \land w_2
\]

where \( t \in \text{Term}(\Sigma) \) and has no free variables (§4.3.2)

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: \( w \subseteq_N w' \) iff \( \gamma_N(w) \subseteq \gamma_N(w') \). However, this pre-order is not effective, necessitating the use of a linear variation: \( w \subseteq_L w' \) iff \( w \models_L w' \). The wedge domain's ability to reason about non-linear operations stems from strengthening operations (§4.4.1) 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 4.5.** 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 \( \tau \) admits a term \( t \) if all of \( t \)'s application sub-terms belong to \( \tau \) (including \( t \) itself, if \( t \) is an application term). We say that \( \tau \) admits a wedge if it admits all the terms of that wedge. We say that \( \tau \) is a minimal coordinate system for a wedge \( w \) if \( \tau \) admits \( w \) and no proper sub-list of \( \tau \) admits \( w \).

Given a coordinate system \( \tau = [t_1, ..., t_n] \), we define a linearization function \( \text{lin}_\tau \) that expresses admissible terms and wedges in linear arithmetic over the set of variables \( \{z_1, ..., z_n\} \), where variable \( z_i \) denotes the \( i \)th coordinate of \( \mathbb{R}^n \) (representing the term \( t_i \)).
\[
\text{lin}_\tau(a) \triangleq a, \quad \text{if } a \in \mathbb{Q} \\
\text{lin}_\tau(t_1) \triangleq z_i \\
\text{lin}_\tau(u_1 + u_2) \triangleq \text{lin}_\tau(u_1) + \text{lin}_\tau(u_2) \\
\text{lin}_\tau(k \times u) \triangleq k \times \text{lin}_\tau(u), \quad \text{if } k \in \mathbb{Q}
\]

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

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

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

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

**Example 4.6.** A minimal coordinate system that admits the wedge \( w \triangleq 2 x^2 - y = 0 \land y - 3 x + y \geq 0 \) is \( \tau = [x, y, x^2] \).

The linearization of \( w \) is the polyhedron \( \text{lin}_\tau(w) = -z_2 + 2 z_3 = 0 \land -3 z_1 + 2 z_2 \geq 0 \).

### 4.4.1 Strengthening

Let \( w \) be a wedge. A strengthening of \( w \) is any wedge \( w' \) such that \( w \equiv_N w' \) and \( w' \models_L w \). For example, a strengthening of the wedge \( x = y + 1 \land y^2 \geq x^2 \) is \( x = y + 1 \land x^2 \geq 0 \land x^2 = y^2 + 2 y + 1 \land y \geq 0 \). 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 [Granger, 1992]. 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

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

\( \text{aff}(P) \) 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 \( I(w, \tau) \) to be the ideal generated by \( \text{aff}(P) \) and the
We say that we have Algorithm 1.

Consider the wedge Example 4.7. We will now illustrate Algorithm 1 on the wedge $w = (x \times 2^y = v \land v \leq (2^y + 1) \times y \land x = y \land z \leq 0)$ 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:

$z_1 : v \quad z_2 : x \quad z_3 : y \quad z_4 : 2^z \quad z_5 : x \times 2^z \quad z_6 : 2^y \quad z_7 : (2^y + 1) \times y$

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

$I(w, \tau) \overset{df}{=} \langle z_5 - z_1, z_2 - z_3, z_5 - z_2 z_4, z_7 - (z_6 + 1) z_3 \rangle$.

Definitional equalities

We will now illustrate Algorithm 1 on the wedge $w$. First we compute a (degrevlex) Gröbner basis for $I(w, \tau)$:

$G = \{z_5 - z_1, z_3 - z_2, z_2 z_4 - z_1, z_2 z_6 - z_7 + z_2, z_1 z_6 - z_4 z_7 + z_2 z_4\}$ and set the active coordinates to be $\{4, 6\}$, the ones corresponding to the terms $2^z$ and $2^y$.

First iteration of the outer loop: When processing coordinate 4 in the inner loop, we add the mapping $CC[2^z] := z_4$ to $CC$, indicating that the representative for the term $2^z$ is $z_4$. We then process coordinate 6, and find that the term $(\text{red}_G(2))^{\text{red}_G(z_4)} = 2^z$ 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 $2^y = 2^x$). We add the equation $2^y - 2^x = 0$ to $w$ and compute a Gröbner basis for the ideal generated by $G$ and $B$:

$G = \{z_5 - z_1, z_3 - z_2, z_6 - z_4, z_2 z_4 - z_1, z_7 - z_2 - z_4\}$. 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 $2^z$ and $2^y$ were identified in the previous iteration, the inner loop does not produce any new equations. However, the Gröbner basis $G$ changed in the previous
Algorithm 1: Equational saturation

**Input** : Coordinate system $\tau = \{t_1, \ldots, t_n\}$ and a wedge $w$

**Output**: Equational saturation of $w$ in $\tau$

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

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 $(2^y + 1) \times y = v + x$ to $w$ (line 15). Since $w$ contains the inequation $v \leq (2^y + 1) \times y$ and (now) the equation $(2^y + 1) \times y = v + x$, we have $v \leq v + x$ and thus $0 \leq x$; since $w$ also contains $x \leq 0$, we have $x = 0$. Thus, the underlying polyhedron of $w$ is $z_5 = z_1 \land z_1 \leq z_7 \land z_2 = z_3 \land z_2 = 0$, 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 $v = 0$, $x = 0$, $y = 0$, $x \times 2^y = 0$, $(2^r + 1) \times y = 0$, 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:

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

Inequalities

Let $\tau = \{t_1, \ldots, t_n\}$ 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. 4.3. 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 $\text{lin}_\tau(t)$ 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 $f^{\text{ivl}}$ that approximates $f$ on intervals (e.g., the interval approximation of multiplication is $[a, b] \times [c, d] \approx [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$). The interval constraints in the hypothesis of the rule are computed 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).

### 4.4.2 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 $\models_N$ (it is not, however, a least upper bound with respect to $\models_N$, which is not computable). We compute the pseudo-join $w_1 \sqcup w_2$ by first finding a minimal coordinate system $\tau$ that...
admits both \( w_1 \) and \( w_2 \), strengthening both within \( \tau \) to get wedges \( w'_1 \) and \( w'_2 \), and setting

\[
w_1 \sqcup w_2 \overset{\text{def}}{=} \text{interp}_\tau(\text{lin}_\tau(w'_1) \sqcup_L \text{lin}_\tau(w'_2))
\]

where \( \sqcup_L \) denotes polyhedral join. Observe that due to the strengthening operation, \( w_1 \sqcup w_2 \) is not necessarily an upper bound with respect to \( \models_L \).

**Example 4.8.** Consider the wedges \( y \geq 1 \wedge y^2 \geq x \) and \( 0 \geq x \). A coordinate system admitting both wedges is \([x, y, y^2]\).

Strengthening \( y \geq 1 \wedge y^2 \geq x \) yields \( y \geq 1 \wedge y^2 \geq x \wedge y^2 \geq y \), and strengthening \( 0 \geq x \) yields \( 0 \geq x \wedge y^2 \geq 0 \). The pseudo-join is \( y^2 \geq x \wedge y^2 \geq 0 \). Notice that \( 0 \geq x \not\models_L y^2 \geq x \wedge y^2 \geq 0 \).

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: \( w_1 \sqcap w_2 \overset{\text{def}}{=} w_1 \wedge w_2 \).

**Widening.** In contrast to conventional methods, the program analysis proposed in this chapter makes no use of widening in the analysis of loops: a loop is analyzed by extracting recurrences and computing closed forms. However, ICRA (presented in Chapter 3) does make use of widening when analyzing functions with non-linear recursion (more than one recursive call on some path through the function). 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 \( w_1 \triangledown w_2 \) as

\[
w_1 \triangledown w_2 \overset{\text{def}}{=} \text{interp}_\tau(\text{project}_L(\text{lin}_{\tau_1}(w_1), |\tau|) \triangledown_L \text{project}_L(\text{lin}_{\tau_2}(w_2), |\tau|))
\]

where \( \text{project}_L(P, n) \) denotes the projection of a polyhedron onto its first \( n \) dimensions and \( \triangledown_L \) 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 Algorithm 2. It operates as follows. Let \( w \) be a \( \Sigma \)-wedge and let \( \tau = [t_1, ..., t_n] \) be a minimal coordinate assignment admitting \( w \). If \( t_i \) is a \( \Sigma' \)-term, we mark \( i \) as safe and
Algorithm 2: Projection

**Input**: \( \Sigma \)-wedge \( w \) and sub-vocabulary \( \Sigma' \subseteq \Sigma \)

**Output**: \( \Sigma' \)-wedge \( w' \) with \( w \models_N w' \)

1. \( \tau = [t_1, ..., t_n] \leftarrow \) minimal coord. system admitting \( w \);
2. \( safe \leftarrow \) empty map;
3. for \( i = 1 \) to \( n \) do
   4. if \( t_i \in \text{Term}(\Sigma') \) then
      5. \( safe[i] \leftarrow t_i; \)
   6. repeat
      7. \( \{i_1, ..., i_k\} \leftarrow \text{dom}(safe); \)
      8. \( unsafe \leftarrow \{z_i : i \in \{1, ..., n\} \setminus \text{dom}(safe)\}; \)
      9. \( G \leftarrow \text{Gröbner basis (elim. order \( \preceq_{\text{unsafe}} \)) for } \Pi(w, \tau); \)
   10. for \( i \in unsafe \) do
       11. Let \( t_i = f(s_{j_1}, ..., s_{j_m}); \)
       12. Let \( r_j \leftarrow \text{red}_G(lin_\tau(s_j)) \) for all \( j \);
       13. if \( f \in \Sigma' \) and \( t_j \in \mathbb{Q}[z_{i_1}, ..., z_{i_n}] \) for all \( j \) then
           14. \( s'_i \leftarrow r_j \) with each \( z_{i_k} \) replaced by \( safe[i_k]; \)
           15. \( safe[i] \leftarrow f(s'_1, ..., s'_m); \)
   16. until \( \text{dom}(safe) \cap unsafe = \emptyset; \)
   17. // Polyhedral projection
   18. \( P \leftarrow \text{project}(\text{lin}_\tau(w), \{z_i : i \in \text{dom}(safe)\}); \)
   19. Let \( Az \triangleright b \) be a constraint representation of \( P; \)
   20. \( w' \leftarrow Az \triangleright b \) with each \( z_i \) replaced by \( safe[i]; \)
21. return \( w \)

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 an elimination ordering for \( \{z_i : t_i \text{ is unsafe}\} \). We then traverse the \( unsafe \) terms \( t_i = f(s_{j_1}, ..., s_{j_m}); \) if every coordinate in \( \text{red}_G(lin_\tau(s_j)) \) 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 \( w' \).

### 4.4.3 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 §4.5). 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.

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 (§4.4.1) 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 $w \equiv w'_1 \sqcup \cdots \sqcup w'_n$. 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 Algorithm 3.

**Algorithm 3: 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 $w_I$ be an implicant of $F$ s.t. $\mathcal{L} \models L w_I$;
       // Wedge projection: remove Skolem constants
6   $(w'_I, A_3) \leftarrow$ project$(w_I, \Sigma)$;
       // Wedge pseudo-join
7   $(w, A_\sqcup) \leftarrow w \sqcup w'_I$;
8   $R \leftarrow R \land A_3 \land A_\sqcup \land \neg w_I$;
9 return $w$
```

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 $(w', A)$ consisting of a $\Sigma'$-wedge and a formula $A$ such that $true \models_N A$ and $A \land w \models_L w'$. Similarly, $w_1 \sqcup w_2$ returns a pair $(w, A)$ 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.

### 4.5 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 [Farzan and Kincaid, 2015], 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 (§4.4). A description of how this method goes beyond
that of Farzan and Kincaid [2015] is given in §4.8.

In §4.6, we describe a method for solving recurrence equations of the form

\[
\begin{bmatrix}
    y_1^{[n]} \\
    \vdots \\
    y_k^{[n]}
\end{bmatrix} = A \begin{bmatrix}
    y_1^{[n-1]} \\
    \vdots \\
    y_k^{[n-1]}
\end{bmatrix} + \begin{bmatrix}
    r_1 \\
    \vdots \\
    r_k
\end{bmatrix},
\] (4.2)

(or more succinctly, \( y^{[n]} = Ay^{[n-1]} + r(n) \)), where \( A \in \mathbb{Q}^{k \times k} \) and each \( r_i \) 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 §4.4.3 (Algorithm 3).

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, and \( w_{\text{body}} \) to be a wedge that over-approximates it. Suppose that \( \text{Var} \) is a set of constant symbols that represent program variables, and \( \text{Var'} \) is a disjoint set of “primed copies” of program variables. Suppose \( \Sigma \) is a vocabulary containing \( \text{Var}, \text{Var'}, \) 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 \( \text{Var} \) and the terms \( t_{m+1}, ..., t_{2m} \) correspond to the program variables in \( \text{Var'} \). We use \( x \overset{\text{def}}{=} [z_1, ..., z_m] \) to denote the vector of variables corresponding to \( \text{Var} \) and \( x' \overset{\text{def}}{=} [z_{m+1}, ..., z_{2m}] \) to denote the vector of variables corresponding to \( \text{Var'} \).
while(*) {
    if (*) {
        x := 2x + y + i
    }
    else {
        y := x + 2y + i
    }
    i := i + 1
    j := j + x
}

(a) Example program

Figure 4.4: A loop, along with its transition formula and transition wedge

4.5.1 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 $w_{body} \models L A x' = B A x + c$. 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 $w_{body} \models L A' x' = B' A' x + c'$, then we have $A x' = B A x + c \models L A' x' = B' A' x + c'$). Suppose that $cl$ is a closed-form solution to the system of recurrence equations $y_n = B y_{n-1} + c$ (which matches the form in Eqn. (4.2)), in the sense that $y[n] = B y[n-1] + c \iff y[n] = cl(y[0], n)$. Then the transitive closure of $\phi_{body}$ must entail $\exists n \geq 0. A x' = cl(A x, n)$.

As a motivating example, consider the loop from Fig. 4.4(a) and the associated transition formula and wedge. Observe that (1) the loop is non-deterministic, and there is no exact representation of the closed form of the variables $x$ or $y$, 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$^3$ affine recurrence is

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

(4.3)

(or more simply, $i' = i + 1$ and $(x' + y') = 2(x + y) + i$). Note how the second row of matrix $A$ in Eqn. (4.3) 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

$^3$ “Best” in the sense that it generates the least affine space. The affine space is unique, but its representation as a system of equations is not.
\[ A_1 x' = M_1 x + c_1 \] 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 \( \{v : \exists u. u A_i = v B_i \} \) 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 \( A_i \) by at least 1.

Returning to our running example Fig. 4.4, the first approximation is essentially just the wedge Fig. 4.4(c) itself:

\[
\begin{bmatrix}
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
1' \\
j' \\
x' \\
y'
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 2 & 2 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
1 \\
j \\
x \\
y
\end{bmatrix}
+ \begin{bmatrix}
0 \\
1 \\
0 \\
c_1
\end{bmatrix}
\]

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 \( \{v : \exists u. u A_1 = v B_1 \} \) 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:

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

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. (4.3).

### 4.5.2 Stratified Recurrences

Consider the simple loop \( \text{while}(*) \{ y += x*x; x++; \} \). 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>$y^{[n]} = y^{[0]} + \sum_{i=0}^{n-1} x^{[i]} = y^{[0]} + \sum_{i=0}^{n-1} (x^{[0]} + i)^2 = y^{[0]} + n/6 - n^2/2 + n^3/3 - nx^{[0]} + n^2x^{[0]} + n(x^{[0]}^2)$</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:

$$w_{\text{body}} \models_1 A_0x' = B_0A_0x + c$$
$$w_{\text{body}} \models_1 A_1x' = B_1A_1x + p_1(A_0x)$$
$$\vdots$$
$$w_{\text{body}} \models_1 A_i x' = B_i A_i x + p_i(A_0x, \ldots, A_{i-1}x)$$

where $A_0x' = B_0A_0x + c$ is an affine recurrence and each $p_i(A_0x, \ldots, A_{i-1}x)$ is a polynomial over the terms in $A_0x, \ldots, A_{i-1}x$. For each stratum $j$, the closed form $cl_j$ is computed as the solution to a single matrix recurrence (of the form in Eqn. (4.2)) by substituting closed forms of previous strata into the next:

$$y_0^{[n]} = B_0y_0^{[n-1]} + c \iff y_0^{[n]} = cl_0(y_0^{[0]}, n)$$
$$y_1^{[n]} = B_1y_1^{[n-1]} + p_1(cl_0(y_0^{[n]}, n)) \iff y_1^{[n]} = cl_1(y_0^{[0]}, y_1^{[0]}, n)$$
$$\vdots$$
$$y_i^{[n]} = B_iy_i^{[n-1]} + p_i(cl_0(y_0^{[n]}, n), \ldots, cl_{i-1}(y_0^{[n]}, \ldots, y_{i-1}^{[n]}, n)) \iff y_i^{[n]} = cl_i(y_0^{[0]}, \ldots, y_{i-1}^{[0]}, n)$$

Observing that substitution of $\text{MultiRecTerms}$ into a polynomial yields a $\text{MultiRecTerm}$, this procedure works as long as no $cl_i(y_0^{[0]}, \ldots, y_{i-1}^{[0]}, n)$ 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 §4.5.1), and continuing until reaching a stratum where no new recurrences are found. For each stratum $i$, we compute the recurrence equation

$$A_i x' = B_i A_i x + p_i(A_0x, \ldots, A_{i-1}x)$$

as follows. Let $T = \{t_1, \ldots, t_k\}$ be a set consisting of all linear terms of the form $ax$, where $a$ is a row of one of the matrices $A_0, \ldots, A_{i-1}$; $T$ 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 $(z$ and $r$) variable with its corresponding $(x, x'$, and $t_i)$ term. Then perform the same fixpoint algorithm from §4.5.1 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 $i + 1$.

### 4.5.3 Recurrence Inequations

In §4.5.1 and §4.5.2, 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

$$w_{\text{body}} \models \text{l. } A x' \leq B A x + p(A_0 x, ..., A_k x)$$

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 §4.5.2.

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:

$$\text{cone}(A) \overset{\text{def}}{=} \{ \lambda_1 a_1 + \cdots + \lambda_k a_k : \lambda_1 \geq 0, ..., \lambda_k \geq 0 \}$$

The requirement that $A_i$ and $M_i$ generate the same cone guarantees that there exists a factorization $M_i = B A_i$ 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.

4.6 Operational Calculus Recurrence Solving

4.6.1 Introduction

Operational calculus [J.-G.-M., 1983] 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 [1967, 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 [1967, Ch. II]. The ring of functions \( \mathbb{R} \) is defined as \( \mathbb{R} \triangleq (\mathbb{R}, +, \cdot, 0, 1) \), where \( \mathbb{R} = \mathbb{N} \rightarrow \mathbb{Q} \).

It is often helpful to consider a ring element \( r \in \mathbb{R} \) as an infinite sequence \( \langle r[0], r[1], r[2], \ldots \rangle \). For \( r \in \mathbb{R} \), 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 \( a, b \in \mathbb{R} \), \( a + b \triangleq \lambda i. a(i) + b(i) \), so the additive identity element, \( 0 \), is \( \lambda i.0 = \langle 0, 0, 0, \ldots \rangle \). Ring multiplication is the following convolution-product difference:

**Definition 4.9.** For all \( a, b \in \mathbb{R} \),

\[
    a \cdot b \triangleq \lambda n. \sum_{v=0}^{n} a[v] b[n-v] - \sum_{v=0}^{n-1} a[v] b[n-1-v].
\]

We will often omit multiplication operators, and write \( a \cdot b \) as \( ab \). The multiplicative identity element, \( 1 \), is \( \lambda i.1 = \langle 1, 1, 1, \ldots \rangle \). It is easy to show that multiplication is commutative and that multiplication distributes over addition [Berg, 1967, §7].

The constant elements of \( \mathbb{R} \) are those of the form \( c \triangleq \lambda n. c = \langle c, c, c, \ldots \rangle \). 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: \( c \cdot d = cd \).

\(^4\)Berg uses \( \mathbb{N} \rightarrow \mathbb{C} \), 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 §4.3.2, which requires using \( \mathbb{Q} \).
It is sometimes useful to write Eqn. (4.4) as

\[ a \cdot b = \lambda n \cdot \sum_{\nu=0}^{n-1} a^{[\nu]} (b^{[n-\nu]} - b^{[n-1-\nu]}) + a^{[n]} b^{[0]} \tag{4.5} \]

Id denotes the identity function \( \lambda n. n = \langle 0, 1, 2, \ldots \rangle \). Using Eqn. (4.5), note that for all \( a \in \mathbb{R} \),

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

In other words, 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 \( F \overset{\text{def}}{=} (\mathbb{F}, +, \cdot, -1, 0, 1) \). An element of \( F \) is called an operator. Some operators are elements of \( \mathbb{R} \); however, there are interesting operators that are not members of \( \mathbb{R} \). For instance, the equation

\[ q v = 1 \tag{4.6} \]

has no solution in \( \mathbb{R} \). However, because \( v \neq 0 \), there is an operator \( q \in F \) that solves Eqn. (4.6). Intuitively, \( q \) should be a left-shift operator.

As Berg points out [Berg, 1967, §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. (4.6), is

\[
\begin{array}{cccccccc}
\ldots & -2 & -1 & 0 & 1 & 2 & 3 & \ldots \\
q & = & \ldots & 0 & 1 & 1 & 1 & 1 \ldots 
\end{array}
\]

\(^5\) As is standard, for \( n \geq k \), the **binomial coefficient** \( \binom{n}{k} \) denotes \( \frac{n!}{(n-k)!k!} \). We define \( \binom{n}{0} = 1 \) and, for all \( k > n \), \( \binom{n}{k} = 0 \).
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 $y^{[n+1]}$:

$$ y^{[n+1]} = q \cdot y^{[n]} - (q - 1)y^0. $$ (4.7)

We can depict some of the infinite sequences that appear in Eqn. (4.7) as follows:

<table>
<thead>
<tr>
<th></th>
<th>$\ldots$</th>
<th>$-2$</th>
<th>$-1$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>$\ldots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y^{[n]}$</td>
<td>$\ldots$</td>
<td>0</td>
<td>0</td>
<td>$y^0$</td>
<td>$y^1$</td>
<td>$y^2$</td>
<td>$y^3$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$y^{[n+1]}$</td>
<td>$\ldots$</td>
<td>0</td>
<td>0</td>
<td>$y^1$</td>
<td>$y^2$</td>
<td>$y^3$</td>
<td>$y^4$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$q \cdot y^0$</td>
<td>$\ldots$</td>
<td>0</td>
<td>0</td>
<td>$y^0$</td>
<td>$y^1$</td>
<td>$y^2$</td>
<td>$y^3$</td>
<td>$y^4$</td>
</tr>
<tr>
<td>$(q - 1)y^0$</td>
<td>$\ldots$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$\ldots$</td>
</tr>
</tbody>
</table>

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

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

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

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:

$$ q = \langle 1 || 1, 1, 1, \ldots \rangle \quad (q - 1) = \langle 1 || 0, 0, 0, 0, \ldots \rangle $$

Eqn. (4.7) 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:

$$ y^{[n+m]} = q^m \cdot y^{[n]} - \sum_{\mu=0}^{m-1} q^\mu \cdot (q - 1)y^{[m-1-\mu]]. $$ (4.8)

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. (4.8) can be understood by recognizing that

$$ (q - 1)y^{[m-1-\mu]} = \langle y^{[m-1-\mu]} || 0, 0, 0, 0, \ldots \rangle, $$
\[ T_n(c) = c \]  \hspace{1cm} (4.9)
\[ T_n \left( \sum_{k \in K} g_k \right) = \sum_{k \in K} T_n(g_k) \]  \hspace{1cm} (4.10)
\[ T_n(cf) = cT_n(f) \]  \hspace{1cm} (4.11)
\[ T_n \left( \frac{q - 1}{(q - k)^c} \right) = \binom{n}{k} \]  \hspace{1cm} (4.12)
\[ T_n \left( \frac{q - 1}{q - k} \right) = k^n \]  \hspace{1cm} (4.13)

Figure 4.5: Translation function \( T_n : F \rightarrow R \). In Eqn. (4.10) \( K \subseteq \mathbb{N} \) is some finite index set.

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. (4.7) 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. 4.5 shows the above rule along with other useful translation rules.

4.6.2 Solving First-Order Univariate Recurrences

The properties of the algebra described in §4.6.1 give us a nice algebraic way of solving recurrences of the following form:

\[ y^{[n+1]} = \alpha \cdot y^{[n]} + \sum_i f_i(n) \quad \text{where} \quad f_i(n) = \text{poly}_i(n) \ast \beta_i^n \quad \text{and} \quad \text{poly}_i(n) \in \mathbb{Q}[n] \]  \hspace{1cm} (4.17)
Algorithm 4: 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. \( R \leftarrow \text{Simplify } r \); 
2. \( \text{OpR} \leftarrow \text{Translate } R \text{ into } F \text{ using Eqn. (4.8) and } T^{-1} \); 
3. \( \text{OpR} \leftarrow \text{Solve for } y \text{ in OpR} \); 
4. \( \text{OpR} \leftarrow \text{Perform partial-fraction decomposition on OpR} \); 
5. \( S \leftarrow T_n(\text{OpR}) \); 
6. \( S \leftarrow \text{Simplify } S \); 
7. return \( S \)

(In this section, “\(*\)” denotes ordinary multiplication.) Algorithm 4 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 [2003], 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 2 of Algorithm 4, translates a recurrence of the form of Eqn. (4.17). We can use Eqn. (4.7) to write \( y_{n+1} \) as \( qy_n - (q-1)y_0 \). For the right-hand side of Eqn. (4.17), \( \alpha \cdot y_n \) remains the same, except that the constant \( \alpha \) becomes the constant sequence \( \alpha \). What remains to describe is how we obtain \( T^{-1}_n(\sum_i f_i(n)) \).

As a quick detour, note that for all the rules in Fig. 4.5, 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 \( \{d\}_j \) denotes a Stirling number of the second kind:

\[
\begin{align*}
n^d &= \sum_{j=0}^d \binom{d}{j} \binom{n}{j} \\
\end{align*}
\]  

(4.18)

To perform the translation \( T^{-1}_n(\sum_i f_i(n)) \), first note that by Eqn. (4.10), we can pull the summation out. We now seek a characterization of \( T^{-1}_n(\text{poly}_i(n) * \beta_i^n) \). First, consider rewriting \( \text{poly}_i(n) \) using Eqn. (4.18), where \( d = \text{deg(\text{poly}_i(n))} \).

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

We can use the general transformation

\[
T^{-1}_n \left( \binom{n}{c} k^{n-c} \right) = \frac{q-1}{(q-k)^{c+1}}
\]
to see that every \( f_i(n) \) can be translated into the following form in the operational-calculus algebra:

\[
T^{-1}_n\{ f_i(n) \} = \sum_{j=0}^{d} \frac{\gamma_{i,j}(q-1)}{(q - \beta_i)^{c_j}},
\]

where \( \gamma_{i,j} \) is an appropriate constant. Thus, after Step 2 Algorithm 4 produces an equation of the form

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

The algorithm continues, solving the above equation for \( y_n \) in Step 3.

\[
y(n) = \frac{q - 1}{q - \alpha} y(0) + \sum_{i} \frac{\gamma_i(q-1)}{(q - \alpha)(q - \beta_i)^{c_i}}
\]

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

Thus, after Step 3, we need to transform all the summands to a form that has a translation rule in Fig. 4.5. We can accomplish this goal by performing partial-fraction decomposition on all the summands (Step 4).

\[
\frac{q - 1}{q - \alpha} y(0) + \sum_{i} \frac{a_k}{(q - b_k)^{c_k}}
\]

where \( a_k \) and \( b_k \) are constants resulting from partial-fraction decomposition, and every \( c_k \) is a non-negative integer. After Step 4, the right-hand side of the equation will be a sum of terms, each of which has a translation rule in Fig. 4.5 that can be applied to translate the term back into standard algebra (Step 5). Finally, in Step 6, the algorithm removes binomials and simplifies the result.

The preceding four paragraphs have shown that Algorithm 4 obtains a closed-form solution to any recurrence of the form given in Eqn. (4.17). The steps used in Algorithm 4 never get stuck; i.e., operations such as partial-fraction decomposition in Step 4 can always be performed. In other words, we have proven the following theorem:

**Theorem 4.10.** Algorithm 4 is complete for the class of recurrences given in Eqn. (4.17).

**Some Univariate Examples**

We now illustrate Algorithm 4 via some examples.

**Example 4.11.** Consider the following recurrence relation:

\[
y^{n+1} = y^n + n^3
\]
This recurrence is already simplified, so we proceed to Step 2 of Algorithm 4. By Eqn. (4.7), we have

\[ y_{n+1} = q \cdot y_n - (q - 1)y_0 \]

The algorithm rewrites \( n^3 \) using the binomial transform to yield the following expression:

\[ n^3 = \binom{n}{1} + 6 \binom{n}{2} + 6 \binom{n}{6} \]

Thus, by Eqn. (4.12) the original recurrence problem is translated to the following equation in \( \mathcal{T} \):

\[ q \cdot y_n - (q - 1)y_0 = y_n + \frac{1}{q - 1} + \frac{6}{(q - 1)^2} + \frac{6}{(q - 1)^3} \]

Step 3 solves this equation for the desired variable. In this case, we seek a closed solution for \( y_n \).

\[ q \cdot y_n - y_n = (q - 1)y_0 + \frac{1}{q - 1} + \frac{6}{(q - 1)^2} + \frac{6}{(q - 1)^3} \]

\[ (q - 1)y_n = (q - 1)y_0 + \frac{1}{q - 1} + \frac{6}{(q - 1)^2} + \frac{6}{(q - 1)^3} \]

\[ y_n = y_0 + \frac{1}{(q - 1)^{\frac{3}{2}}} + \frac{6}{(q - 1)^3} + \frac{6}{(q - 1)^4} \]

In this example, partial-fraction decomposition does not change the current expression. Step 5 applies \( \mathcal{T}_n \) to the right-hand side and a final simplification occurs, yielding

\[ y_n = \mathcal{T}_n \left( \frac{y_0}{(q - 1)^{\frac{3}{2}}} + \frac{6}{(q - 1)^3} + \frac{6}{(q - 1)^4} \right) \]

\[ = \mathcal{T}_n \left( y_0 + \frac{1}{(q - 1)^{\frac{3}{2}}} + \frac{6}{(q - 1)^3} + \frac{6}{(q - 1)^4} \right) \]

\[ = y_0 + \frac{n}{2} + 6 \binom{n}{3} + 6 \binom{n}{4} \]

\[ = y_0 + \frac{1}{2} n^2 - \frac{1}{2} n^3 - 3n^2 + 2n + \frac{1}{4} n^4 - \frac{3}{2} n^3 + \frac{11}{4} n^2 - \frac{3}{2} n \]

\[ = y_0 + \frac{1}{4} n^4 - \frac{1}{2} n^3 + \frac{1}{4} n^2 \]

**Example 4.12.** Consider the following example, which generates an exponential closed form.

\[ y^{n+1} = 2y^n + 3^n \]
Step 2 generates the following equation in $F$:

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

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

By Eqn. (4.11), Eqn. (4.9), Eqn. (4.10), Eqn. (4.13), and Eqn. (4.7).

After solving for $y^{[n]}$, Step 3 produces

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

Step 4 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 5 translates the resultant expression back to standard algebra and the algorithm performs a final simplification.

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

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

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

4.6.3 Solving First-Order Matrix Recurrences

We now give details about solving what we call matrix recurrences. Consider a vector of recurrence variables indexed by $n+1$, $x^{[n+1]} = \left[ x_1^{[n+1]}, x_2^{[n+1]}, \ldots, x_k^{[n+1]} \right]^t$; their copies, indexed by $n$, $x^{[n]} = \left[ x_1^{[n]}, x_2^{[n]}, \ldots, x_k^{[n]} \right]^t$; a matrix $A$ with entries in $\mathbb{Q}$; and a vector $b = \left[ f_1(n), f_2(n), \ldots, f_k(n) \right]^t$ in which each $f_i(n)$ is a RecTerm as given in §4.5. We seek a closed-form solution for $x^{[n]}$ in the recurrence

$$x^{[n+1]} = Ax^{[n]} + b \tag{4.19}$$

Let us consider how to extend the method from §4.6.1 and §4.6.2 to solve such an equation.

Left-Hand Side of Eqn. (4.19). We seek a rule for the vector $x^{[n+1]}$ that is analogous to Eqn. (4.7). If we consider applying Eqn. (4.7) to each of the entries of $x^{[n+1]}$, we have the following:
\[
\mathbf{x}^{[n+1]} = \begin{bmatrix}
q \cdot x_1^{[n]} - (q - 1)x_1^{[0]} \\
\vdots \\
q \cdot x_k^{[n]} - (q - 1)x_k^{[0]}
\end{bmatrix} = \begin{bmatrix}
q \cdot x_1^{[n]} \\
\vdots \\
q \cdot x_k^{[n]}
\end{bmatrix} - \begin{bmatrix}
(q - 1)x_1^{[0]} \\
\vdots \\
(q - 1)x_k^{[0]}
\end{bmatrix} = \begin{bmatrix}
q \cdot x_1^{[n]} - (q - 1)x_1^{[0]} \\
\vdots \\
q \cdot x_k^{[n]} - (q - 1)x_k^{[0]}
\end{bmatrix} - \begin{bmatrix}
q \cdot x_1^{[0]}(q - 1) \\
\vdots \\
q \cdot x_k^{[0]}(q - 1)
\end{bmatrix} = q\mathbf{Ix}^{[n]} - x^{[0]}(q - 1)
\]

(4.20)

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

Right-Hand Side of Eqn. (4.19). We now consider rewriting the right-hand side of Eqn. (4.19). In other words, we seek to know \(\mathcal{T}_n^{-1}(\mathbf{A}\mathbf{x}^{[n]} + \mathbf{b})\), where we interpret \(\mathcal{T}_n^{-1}\) as being applied to each row of the vector \(\mathbf{A}\mathbf{x}^{[n]} + \mathbf{b}\).

By Eqns. (4.10) and (4.11), we have

\[
\mathcal{T}_n^{-1}(\mathbf{A}\mathbf{x}^{[n]} + \mathbf{b}) = \mathbf{Ax}^{[n]} + \mathcal{T}_n^{-1}(\mathbf{b}).
\]

(4.21)

By Eqns. (4.20) and (4.21), we can rewrite the right-hand side of Eqn. (4.19) as follows:

\[
q\mathbf{Ix}^{[n]} - (q - 1)x^{[0]} = \mathbf{Ax}^{[n]} + \mathcal{T}_n^{-1}(\mathbf{b})
\]

\[
q\mathbf{Ix}^{[n]} - \mathbf{Ax}^{[n]} = x^{[0]}(q - 1) + \mathcal{T}_n^{-1}(\mathbf{b})
\]

(4.22)

Eqn. (4.22) 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. (4.19) can be expressed as follows:

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

(4.23)

Algorithm 5 gives a procedure that implements Eqn. (4.23), where some additional details have been given in Steps 5, 6, and 7. Unlike the univariate case, Algorithm 5 may not always return a polynomial or exponential
Algorithm 5: Matrix Operational Calculus Recurrence Solving

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( b \leftarrow \text{Simplify} \ b );</td>
</tr>
<tr>
<td>2</td>
<td>( \text{Opb} \leftarrow x_0(q - 1) + T_n^{-1}(b) );</td>
</tr>
<tr>
<td>3</td>
<td>( D \leftarrow \text{Symbolically calculate} \ (qI - A)^{-1}; )</td>
</tr>
<tr>
<td>4</td>
<td>( V \leftarrow D \ast \text{Opb}; )</td>
</tr>
<tr>
<td>5</td>
<td>( V \leftarrow \text{Convert every entry in} \ V \text{ to the form} \ p/q, \text{ where} \ p \text{ and} \ q \text{ are polynomials in} \ Q[q]; )</td>
</tr>
<tr>
<td>6</td>
<td>( V \leftarrow \text{for every} \ p/q \in V, \text{ factor} \ q_i \text{ into its irreducible factors in} \ Q[q]; )</td>
</tr>
<tr>
<td>7</td>
<td>( V \leftarrow \text{for every} \ v_i \in V, \text{ perform partial-fraction decomposition on} \ v_i );</td>
</tr>
<tr>
<td>8</td>
<td>( S \leftarrow \text{for every} \ v_i \in V, \text{ replace} \ v_i \text{ with} \ T_n(v_i); \text{ if no translation applies to a term} \ \tau, \text{ replace} \ \tau \text{ with} \ IIF(\tau); )</td>
</tr>
<tr>
<td>9</td>
<td>( S \leftarrow \text{for every} \ s_i \in S, \text{ simplify} \ s_i );</td>
</tr>
<tr>
<td>10</td>
<td>( \text{return} \ S )</td>
</tr>
</tbody>
</table>

A closed form for the class of matrix recurrences defined by Eqn. (4.19). In particular, note that all translation rules in Fig. 4.5 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. 4.5, we must be able to find all roots of the polynomials in the denominator in Step 6. 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. 4.3) is inadmissible in non-linear real arithmetic. Thus, to mitigate these difficulties, Step 6 finds all rational roots for the polynomial.

However, the restricted scope of Step 6 means that at Step 8 Algorithm 5 can have rational terms that have no corresponding translations in Fig. 4.5. When that happens, Algorithm 5 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 6 for which Fig. 4.5 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. 4.14; for a demonstration of the utility of IIFs, see Ex. 4.4.
4.6.4 Matrix Examples

Example 4.13. Consider the following matrix recurrence:

\[
\begin{pmatrix}
  x^{[k+1]} \\
  y^{[k+1]}
\end{pmatrix} =
\begin{bmatrix}
  1 & 1 \\
  -2 & 4
\end{bmatrix}
\begin{pmatrix}
  x^{[k]} \\
  y^{[k]}
\end{pmatrix} +
\begin{pmatrix}
  0 \\
  1
\end{pmatrix}
\]

For this example, we have

\[
b = \begin{bmatrix} 0 & 1 \end{bmatrix}^t \quad x^{[0]} = \begin{bmatrix} x^{[0]} & y^{[0]} \end{bmatrix}^t \quad T_n^{-1}(b) = \begin{bmatrix} 0 & 1 \end{bmatrix}^t
\]

The additive vector, \(b\) in this recurrence is already simplified, and the translation, \(T_n^{-1}(b)\), 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

\[
\begin{pmatrix}
  q - 1 & -1 \\
  2 & (q - 4)
\end{pmatrix}^{-1} =
\begin{pmatrix}
  \frac{q-4}{q^2-2q+6} & \frac{1}{q^2-2q+6} \\
  \frac{-2}{q^2-2q+6} & \frac{q-4}{q^2-2q+6}
\end{pmatrix}
\]

After Steps 4 and 5, \(V\) equals the following vector:

\[
\begin{bmatrix}
  \frac{(q^2-5q+6)x^{[0]}+(q-1)y^{[0]}+1}{q^2-2q+6} \\
  \frac{(q^2-2q+1)y^{[0]}-(q-1)2x^{[0]}+q-1}{q^2-2q+6}
\end{bmatrix}^t
\]

The term \(q^2 - 5q + 6\) can be factored into \((q-3)(q-2)\), so Steps 6 and 7 produce

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

Finally, Steps 8 and 9 apply \(T_k\) to each entry of the above vector to obtain closed-form expressions for both \(x^{[k]}\) and \(y^{[k]}\).

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

\[
y^{[k]} = (2x^{[0]} - y^{[0]} - 1)2^k + (-2x^{[0]} + 2y^{[0]} + 1)3^k
\]
Example 4.14. Consider the following matrix recurrence:

\[
\begin{bmatrix}
y^{[k+1]} \\
z^{[k+1]}
\end{bmatrix} =
\begin{bmatrix}
1 & 1 \\
-9 & 1
\end{bmatrix}
\begin{bmatrix}
y^{[k]} \\
z^{[k]}
\end{bmatrix}
\]

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 1–5 of Algorithm 5, we are left with the following equation:

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

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 7 (partial-fraction decomposition) yields

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

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

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

There is no direct translation for the remaining terms. Thus, at this point Algorithm 5 creates IIFs for each term involving \(IIF(\cdot)\) to use in the solutions returned to the wedge domain. In the example above, the returned values are \(1 - 9IIF\left(\frac{1}{q^2-2q+10}\right)(k)\) for \(y^{[k]}\) and \(-9IIF\left(\frac{q}{q^2-2q+10}\right)(k) + 9IIF\left(\frac{1}{q^2-2q+10}\right)(k)\) for \(z^{[k]}\).

### 4.7 Experiments

Our techniques are implemented in a tool called ICRA, which uses Z3’s UFLRA solver [de Moura and Bjørner, 2008], and Apron’s NewPolka polyhedron domain [Jeannet and Miné, 2009]. 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:
Table 4.1: 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.

<table>
<thead>
<tr>
<th>Benchmark Suite</th>
<th>Total #A</th>
<th>ICRA Time</th>
<th>UAut. #A</th>
<th>CPA Time</th>
<th>SEA Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOLA</td>
<td>46</td>
<td>123.5</td>
<td>33</td>
<td>1571.9</td>
<td>2004.1</td>
</tr>
<tr>
<td>functional</td>
<td>21</td>
<td>77.9</td>
<td>11</td>
<td>732.8</td>
<td>1155.7</td>
</tr>
<tr>
<td>relational</td>
<td>10</td>
<td>8.1</td>
<td>10</td>
<td>475</td>
<td>605.0</td>
</tr>
<tr>
<td>Total</td>
<td>77</td>
<td>209.5</td>
<td>54</td>
<td>2777.7</td>
<td>3762.8</td>
</tr>
</tbody>
</table>

- 46 programs are from a suite used to test the invariant-generating tool HOLA [Dillig et al., 2013].
- 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 §4.2. The Fibonacci program in §4.2 is an example of a program in this category. The other nine programs were obtained from [Antonopoulos et al., 2017; Barthe et al., 2011; Terauchi and Aiken, 2005].

We compared ICRA on these 77 programs against three state-of-the-art software model checkers: Ultimate Automizer [Heizmann et al., 2013] from SV-COMP16 and CPAchecker [Beyer and Keremoglu, 2011] from SV-COMP17, both based on predicate abstraction; and SeaHorn [Gurfinkel et al., 2015] 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. 4.1 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. [2014], two examples are from §4.2 (Ex. 4.1 and Ex. 4.2), and one additional program is described below.

Example 4.15. 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:
We compared ICRA on these 8 programs against two other resource-bound analysis tools, CoFloCo [Flores-Montoya and Hähnle, 2014] and PUBS [Albert et al., 2013]. 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.

4.8 Related Work

**Non-Linear Abstract Domains.** There is a significant amount of prior work for reasoning about uninterpreted function symbols [Gulwani et al., 2004; Chang and Leino, 2005; Gange et al., 2016] and non-linear arithmetic [Müller-Olm and Seidl, 2004; Colón, 2004; Bagnara et al., 2005b; Gulavani and Gulwani, 2008]. A common theme in this work, including our own, is to introduce new dimensions in some relational abstract domain (such as polyhedra [Cousot and Halbwachs, 1978] or octagons [Miné, 2001]) 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 [2005]) 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 [Chang and Leino, 2005]. Colón [2004] 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. [2005b] generalize this idea, extending polyhedra to reason about polynomial inequalities. Gulavani and Gulwani [2008] 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. 1) infers equalities using congruence closure (like Chang and Leino [2005]) and Gröbner-basis techniques (which generalize [Colón, 2004]); the wedge domain deduces inequalities by applying a set of inference rules (like Gulavani and Gulwani [2008]); the Product rule generalizes the degree-bounded products of Bagnara et al. [2005b].

In abstract-interpretation terms, such techniques are using semantic reductions [Cousot and Cousot, 1979] 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 [Granger, 1992].

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 (§4.4.3) (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 [Sankaranarayanan et al., 2004b; de Oliveira et al., 2016; Srikanth et al., 2017]. Srikanth et al. [2017] 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 [Ammarguellat and Harrison, 1990; Boigelot and Wolper, 1994; Finkel and Leroux, 2002; Rodríguez-Carbonell and Kapur, 2004; Bagnara et al., 2005a; Kovács, 2008; Albert et al., 2008; Blanc et al., 2010; Ancourt et al., 2010; Bozga et al., 2010; Jeannet et al., 2014; Farzan and Kincaid, 2015; de Oliveira et al., 2016; Kincaid et al., 2017; Humenberger et al., 2017].

This chapter builds upon previous work on compositional recurrence analysis (CRA) [Farzan and Kincaid, 2015] and its interprocedural extension ICRA, by presenting techniques that improve (I)CRA’s ability to
generate non-linear invariants. In particular, [Farzan and Kincaid, 2015]'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 this chapter and Farzan and Kincaid’s algorithm is that we extract and solve \textit{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 [2015], the key operation used for extracting recurrences is symbolic abstraction [Reps et al., 2004; Thakur and Reps, 2012]. For more about symbolic abstraction in program analysis, see Thakur [2014]; Reps and Thakur [2016].

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 §4.6 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 Rodriguez-Carbonell and Kapur [2004] accepts the same class of stratified recurrences as our algorithm (§4.5.2), 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. [2017]. For example, their technique can find closed forms that include factorials, whereas ours cannot. The methods of Rodriguez-Carbonell and Kapur [2004] and Humenberger et al. [2017] both search for polynomial equalities, whereas our technique can find invariants that are not polynomial equalities; also, neither Rodriguez-Carbonell and Kapur [2004] nor Humenberger et al. [2017] 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. [2017].

```
int x = 1, y = 1; for(n = 0; n < N; n++) { x *= 2; y *= 3; }
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 [Kauers and Paule, 2011, Chapter 4].

**Definition 4.16.** ([Kovács, 2008, Def. 3.2][Kauers and Paule, 2011, §4.2]) A C-finite recurrence is a homogeneous linear recurrence equation with constant coefficients

$$f^{[n+r]} = a_{r-1} f^{[n+r-1]} + \ldots + a_1 f^{[n+1]} + a_0 f^{[n]},$$

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 [2008]: 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 §4.5 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, §4.6 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 (§4.6) as the input to the recurrence extractor (§4.5) is essential for handling nested loops, as illustrated by Ex. 4.1. Although it is possible for the ICRA recurrence extractor to extract recurrences that the solver cannot solve, in practice, the solver presented in §4.6 is well-matched to the class of recurrences that are extracted by the algorithm given in §4.5.

Closed forms produced by the method given in §4.6 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 (§4.4 and §4.5), 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 §4.6 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;
}
```

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

The recurrence-extraction technique from §4.5 would identify the following system of stratified recurrence relations:

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

\[ z^{[n]} = z^{[n-1]} + x^{[n-1]} y^{[n-1]} \]

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. (4.25) 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

\[ x^{[n+1]} - 2x^{[n]} - x^{[n-1]} + 2x^{[n-2]} = 0, \]

(4.26)

which is equivalent to the 3rd-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 $\varphi$ does not entail Eqn. (4.26): $x^{[n+1]}$ and $x^{[n-2]}$ do not belong to the vocabulary of $\varphi$ (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 [Wilf, 1994] are another transform method that can be used to solve recurrence relations [Flajolet and Sedgewick, 2009]. 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: $\sum_{n=0}^{\infty} a_n x^n \ast \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} (\sum_{i=0}^{n} a_i b_{n-i}) x^n$. Thus, the multiplication operator for generating functions is a convolution, whereas the multiplication in operational calculus is a convolution difference (cf. Eqn. (4.4)).

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 $\lambda n c$ is represented by $c = \langle c, c, \ldots \rangle$, whereas the generating function is $\frac{c}{1-x}$. The respective values for the sequence $\lambda n n^2$ are $\frac{2}{(q-1)^2} + \frac{1}{q-1}$ and $\frac{x(x+1)}{(1-x)^3}$.

**Generating functions create simpler terms:** The generating function for the geometric sequence $\lambda n a^n$ is $\frac{1}{1-ax}$, whereas the operational-calculus value is $\frac{q^{-1}}{q^{-1} - a}$.

**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 [1967] discusses recurrence solving in the style of a mathematics textbook, but provides no explicit algorithm. §4.6.2 describes our mechanization of Berg’s approach. §4.6.3 concerns an extension to operational calculus not found in Berg (or elsewhere, as far as we know).

---

6 A 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 5

Non-linear Recursion

5.1 Introduction

A large body of work within the numerical-invariant-generation literature focuses on template-based methods [Colón et al., 2003; Sankaranarayanan et al., 2004a]. Such methods fix the form of the invariants that can be discovered, by specifying a template that contains unknown quantities. Given a program and some property to be proved, a template-based analyzer proceeds by finding constraints on the values of the unknowns and then solving these constraints to obtain invariants of the program that suffice to prove the property. Template-based methods have been particularly successful for finding invariants within the domain of linear arithmetic.

Many programs have important numerical invariants that involve non-linear mathematical relationships, such as polynomials, exponentials, and logarithms. A disadvantage of template-based methods for non-linear invariant generation is that (in contrast to the linear case) there is no “most general” template term, so the user must supply the set of terms that may appear in the invariant.

In this chapter, we present an invariant-synthesis technique that is related to template-based methods, but sidesteps the above difficulty. Our technique is based on a concept that we call a hypothetical summary, which is a template for a procedure summary in which the unknowns are functions, rather than numbers. The constraints that we extract for these functions are recurrences. Solving these recurrence constraints allows us to synthesize terms over program variables that we can substitute in place of the unknown functions in our template and thereby obtain procedure summaries.

Whereas most template-based methods directly constrain the mathematical form of their invariants, our technique constrains the invariants indirectly, by way of recurrences, and thereby allows the invariants to have a wide variety of mathematical forms involving polynomials, exponentials, and logarithms. This aspect
is intuitively illustrated by the recurrences $S(n) = 2S(n/2) + n$ and $T(n) = 2T(n/2) + n^2$: although these two recurrences are outwardly similar, their solutions are more different than one would expect at first glance, in that $S(n)$ is $\Theta(n \log n)$, whereas $T(n)$ is $\Theta(n^2)$. Because the unknowns in our templates are functions, we can generate a wide variety of invariants (involving polynomials, exponentials, logarithms) without specifying their exact syntactic form.

However, recurrence-based invariant-generation techniques typically have disadvantages when applied to recursive programs. Recurrences are well-suited to characterize the sequence of states that occur as a loop executes. This idea can be extended to handle linear recursion—where a recursive procedure makes only a single recursive call: each procedure-entry state that occurs “on the way down” to the base case of the recursion is paired with the corresponding procedure-exit state that occurs “on the way back up” from the base case, and then recurrences are used to describe the sequence of such state pairs. However, non-linear recursion has a different structure: it is tree-shaped, rather than linear, and thus some kind of additional abstraction is required before non-linear recursion can be described using recurrences.

We use the technique of hypothetical summaries to extend CRA and the techniques presented in §3 and §4: hypothetical summaries enable a different approach to the analysis of non-linearly recursive programs, such as divide-and-conquer or tree-traversal algorithms. We show how to analyze the base case of a procedure to extract a template for a procedure summary (i.e., a hypothetical summary). By assuming that every call to the procedure, throughout the tree of recursive calls, is consistent with the template, we discover relationships (i.e., recurrence constraints) among the states of the program at different heights in the tree. We then solve the constraints and fill in the template to obtain a procedure summary. Hypothetical summaries thus provide the additional layer of abstraction that is required to apply recurrence-based invariant generation to non-linearly recursive procedures.

Our invariant generation procedure is both (1) general-purpose, so it is applicable to a wide variety of tasks, and (2) compositional, so the space and time required to analyze a program fragment depends on the size of the fragment rather than the whole program. In contrast, conventional template-based methods are goal-directed (they must be tailored to a specific problem of interest, e.g., a template-based invariant generator for verification problems cannot solve quantitative problems such as resource-bound analysis) and whole-program. The general-purpose nature of our procedure also distinguishes it from recurrence-based resource-bound analyses, which for example cannot be applied to assertion checking.

To evaluate the applicability of our analysis to challenging numerical-invariant-synthesis tasks, we applied it to the task of generating bounds on the computational complexity of non-linearly recursive programs and the task of generating invariants that suffice to prove assertions. Our experiments show that the analysis technique is able to prove properties that ICRA (i.e., the technique of Chapter 3) was not capable of proving,
and is competitive with the output of state-of-the-art assertion-checking and resource-bound-analysis tools.

**Contributions.** Our work makes contributions in three main areas:

1. We introduce an analysis method based on “hypothetical summaries.” It hypothesizes that a summary exists 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.

2. We develop a procedure-summarization technique called height-based recurrence analysis, which uses the notion of hypothetical summaries to produce bounds on the values of program variables based on the height of recursion (§5.4.1). We further develop algorithms that, when used in conjunction with height-based recurrence analysis (§5.4.2 and §5.4.3), yield more precise summaries. Furthermore, we give an algorithm (§5.4.4) that generalizes height-based recurrence analysis to the setting of mutual recursion.

3. The technique is implemented in the CHORA tool. Our experiments show that CHORA is able to handle many non-linearly recursive programs, and generate invariants that include exponentials, polynomials, and logarithms (§5.5). For instance, it is able to 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.

§5.2 presents an example to provide intuition. §5.3 provides background on material needed for understanding this chapter’s results. §5.6 discusses related work.

### 5.2 Overview

The goal of the analyses presented in this dissertation is to find numerical summaries for all the procedures in a given program. For simplicity, however, this section discusses the analysis of a program that contains a single procedure $P$, which is non-linearly recursive and calls no other procedures.

We use the following example to illustrate how our techniques use recurrence solving to summarize non-linearly-recursive procedures.

**Example 5.1.** The function subsetSum (Fig. 5.1) 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. If it finds such a set, it returns the number of elements in the set, and otherwise it returns zero. The recursive function subsetSumAux works by sweeping through the array from left to right, making two recursive calls for each array element. The first call considers subsets that
include the element \(A[i]\), and the second call considers subsets that exclude \(A[i]\). The sum of the values in each subset is computed in the accumulating parameter \(\text{sum}\). When the base case is reached, \text{subsetSumAux} checks whether \(\text{sum}\) is zero, and if so, sets \(\text{found}\) to true. At each of the two recursive call sites, the value returned by the recursive call is stored in the variable \(\text{size}\). After \(\text{found}\) is set to true, \text{subsetSumAux} computes the size of the subset by returning \(\text{size} + 1\) if the subset was found after the first recursive call, or returning \(\text{size}\) unchanged if the subset was found after the second recursive call.

A state of a program is an assignment of integers to program variables. For each procedure \(P\), we wish to characterize the relational semantics \(R(P)\), defined as the set of state pairs \((\sigma, \sigma')\) such that \(P\) can start executing in state \(\sigma\) and finish in state \(\sigma'\). To find an over-approximate representation of the relational semantics of a recursive procedure such as \text{subsetSumAux}, we take an approach that we call height-based recurrence analysis. In height-based recurrence analysis, we construct and solve recurrence relations to discover properties of the transition relation of a recursive procedure. To formalize our use of recurrence relations, we give the following definitions.

We define the height-bounded relational semantics \(R(P, h)\) to be the subset of \(R(P)\) that \(P\) can achieve if it is limited to using an execution stack with a height of at most \(h\) activation records. We define a height-\(h\) execution of \(P\) 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 are defined to be of height 1. Let \(\tau_1, ..., \tau_n\) be a set of polynomials over unprimed and primed program variables, representing the pre-state and post-state of \(P\), respectively. For each \(\tau_k\) we associate a function \(V_k : \mathbb{N} \rightarrow 2^\mathbb{Q}\), such that \(V_k(h)\) is defined to be the set of values \(v\) such that, for some \((\sigma, \sigma') \in R(P, h)\), \(\tau_k\) evaluates to \(v\) by using \(\sigma\) and \(\sigma'\) to interpret the unprimed and primed variables, respectively.

```
int nTicks; bool found;
int subsetSum(int * A, int n) {
    found=false; return subsetSumAux(A,0,n,0);
}
int subsetSumAux(int * A, int i, int n, int sum) {
    nTicks++;
    if (i >= n) {
        if (sum == 0) { found=true; }
        return 0;
    }
    int size=subsetSumAux(A,i+1,n,sum+A[i]);
    if (found) { return size + 1; }
    size=subsetSumAux(A,i+1,n,sum);
    return size;
}
```

Figure 5.1: Example program \text{subsetSum}. The diagram on the right shows a timeline of a height \((h+1)\) execution of \text{subsetSumAux}. \(b_2(h+1)\) is related to the increase of \(n\text{Ticks}\) between the pre-state (label 1) and the post-state (label 6). \(b_2(h)\) is related to the increase of \(n\text{Ticks}\) between (2) and (3) and also between (4) and (5), i.e., between the pre-states and post-states of height-\(h\) executions.
Using subsetSumAux as an example, let $\tau_1 \overset{df}{=} \text{return}'$. Then, $V_1(1)$ denotes the set of values $\text{return}'$ can take on in any base case of subsetSumAux. In this program, $\text{return}'$ is 0 in any base case, and so $V_1(1) = \{0\}$. Now consider an execution of height 2. In the case that found is true, we have that $\text{return}'$ increases by 1 compared to the value that $\text{return}'$ has in the base case. If found is not true then $\text{return}'$ remains the same. In other words, at height-2 executions, $\text{return}'$ takes on the values 0 and 1; i.e., $V_1(2) = \{0, 1\}$. Similarly, $V_1(3) = \{0, 1, 2\}$, and so on. We approximate the value set $V_k(h)$ by finding a function $b_k(h) : \mathbb{N} \rightarrow \mathbb{Q}$ that bounds $V_k(h)$ for all $h$; that is, for any $v \in V_k(h)$, we have $v \leq b_k(h)$. In the case of $\tau_1$, a suitable bounding function $b_1(h)$ is $b_1(h) = h - 1$. The initial step of our analysis chooses terms $\tau_1, ..., \tau_n$, and then for each term $\tau_k$, tries to synthesize a function $b_k(h)$ that bounds the set of values $\tau_k$ can take on.

Note that for a given term $\tau_1$, a corresponding bounding function may not exist. A necessary condition for a bounding function to exist for a term $\tau_1$ is that the set $V_1(1)$ must be bounded. This observation restricts our set of candidate terms $\tau_1, ..., \tau_n$ to only be over terms that are bounded above in the base case. (Specifically, we require the expressions to be bounded above by zero.) For example, $\text{return}' \leq 0$ in the base case, and so $\tau_1 \overset{df}{=} \text{return}'$ is a candidate term. Similarly, the term $\tau_2 \overset{df}{=} n\text{Ticks}' - n\text{Ticks} - 1$ is also bounded above by 0 in the base case, and so $\tau_2$ is a candidate term. There are other candidate terms that our analysis would extract for this example, but for brevity they are not listed here. We discover these bounded terms $\tau_1$ and $\tau_2$ using symbolic abstraction (see §5.3).

Once we have a set of candidate terms $\tau_1, ..., \tau_n$, we seek to find corresponding bounding functions $b_1(h), ..., b_k(h)$. Note that such functions may not exist: just because $\tau_k$ is bounded above in the base case does not mean it is bounded in all other executions. If a bounding function for a term does exist, we would like a closed-form expression for it in terms of $h$. We derive such closed-form expressions by hypothesizing that a bounding function $b_k(h)$ does exist. These hypothetical functions $b_k(h)$ allow us to construct a hypothetical procedure summary $\varphi_h$ that represents a typical height-$h$ execution. For example, in the case of subsetSumAux:

$$
\varphi_h \overset{df}{=} \text{return}' \leq b_1(h) \land n\text{Ticks}' - n\text{Ticks} - 1 \leq b_2(h).
$$

Note that, although $\varphi_h$ assumes the existence of several bounding functions (corresponding to $b_k(h)$ for several values of $k$), the assumptions for different values of $k$ need not all succeed or fail together. That is, if we fail to find a bounding function $b_k(h)$ for some $k$, this failure does not prevent us from continuing the analysis and finding other bounding functions ($b_j(h)$, with $j \neq k$) for the same procedure.

We then build up a height-$(h + 1)$ summary, $\varphi_{h+1}$, compositionally, with $\varphi_h$ replacing the recursive calls. For example, consider the term $\tau_2 = n\text{Ticks}' - n\text{Ticks} - 1$ in the context of Fig. 5.1. Our goal is to create a relational summary for the variable nTicks between labels 1 and 6. We do this by extending a summary for the transition between labels 1 and 2 with a summary for the transition between 2 and 3, namely, our hypothetical summary. Then we extend
that with a summary for the paths between labels 3 and 4, and so on. Between labels 1 and 2, \( n\text{Ticks} \) gets increased by 1. We then summarize the transition between 1 and 3. We know \( n\text{Ticks} \) gets increased by 1 between labels 1 and 2. Furthermore, our hypothetical bounding function \( n\text{Ticks}' - n\text{Ticks} - 1 \leq b_2(h) \) says that \( n\text{Ticks} \) gets increased by at most \( b_2(h) + 1 \) between labels 2 and 3. Combining these summaries, we see that \( n\text{Ticks} \) gets increased by at most \( b_2(h) + 2 \) between labels 1 and 3. \( n\text{Ticks} \) does not change between labels 3 and 4, so the summary between labels 1 and 4 is the same as the one between labels 1 and 3. The transition between labels 4 and 5 is a recursive call, so we again use our hypothetical summary to approximate this transition. Once again, such a summary says \( n\text{Ticks} \) gets increased by at most \( b_2(h) + 1 \) between labels 2 and 3. Extending our summary for the transition between 1 and 4 with this information allows us to conclude that \( n\text{Ticks} \) gets increased by at most \( 2b_2(h) + 3 \) between labels 1 and 5. \( n\text{Ticks} \) does not change between labels 5 and 6. Consequently, our summary for \( n\text{Ticks} \) between labels 1 and 6 is \( n\text{Ticks}' - n\text{Ticks} \leq 2b_2(h) + 3 \). Similar reasoning would also obtain a summary for \( \text{return} \) as \( \text{return}' \leq 1 + b_1(h) \). These formulas constitute our height-\((h+1)\) hypothetical summary, \( \varphi_{h+1} \).

\[
\varphi_{h+1} \overset{\text{def}}{=} \text{return}' \leq 1 + b_1(h) \land n\text{Ticks}' \leq n\text{Ticks} + 2b_2(h) + 3
\]

If we rearrange each conjunct to respectively place \( \tau_1 \) and \( \tau_2 \) on the left-hand-side of each inequality, we obtain height-\((h+1)\) bounds on the values of \( \tau_1 \) and \( \tau_2 \). By definition such bounds are valid expressions for \( b_1(h+1) \) and \( b_2(h+1) \). That is at height-\((h+1)\),

\[
\text{return}' \leq b_1(h) + 1 = b_1(h+1)
\]

\[
n\text{Ticks}' - n\text{Ticks} - 1 \leq 2 + 2b_2(h) = b_2(h+1)
\]

The equations give recursive definitions for \( b_1 \) and \( b_2 \). Solving these recurrence relations give us bounds on the value sets \( V_1(h) \) and \( V_2(h) \), for all heights \( h \).

In §5.4.2, we present an algorithm that determines an upper bound on a procedure’s depth of recursion as a function of the parameters to the initial call and the values of global variables. This depth of recursion can also be interpreted as a stack height \( h \) that we can use as an argument to the bounding functions \( b_k(h) \). In the case of subsetSumAux, we obtain the bound \( h \leq \max(1, 1 + n - i) \). The solutions to the recurrences discussed above, when combined with the depth bound, yield the following summary.

\[
n\text{Ticks}' \leq n\text{Ticks} + 2^h - 1 \land \text{return}' \leq h - 1 \land h \leq \max(1, 1 + n - i)
\]
When subsetSum is called with some array size \( n \), the maximum possible depth of recursion that can be reached by subsetSumAux is equal to \( n \). In this way, we have established that the running time of subsetSum is exponential in \( n \), and the return value is at most \( n \).

5.3 Background

Relational semantics. In the following, we give an abstract presentation of the relational semantics of programs. Fix a set \( \text{Var} \) of program variables. A state \( \sigma : \text{State} \) consists of an integer valuation for each program variable. A recursive procedure \( P \) can be understood as a chain-continuous (and hence monotonic) function on state relations \( \mathcal{F}[P] : 2^{\text{State} \times \text{State}} \rightarrow 2^{\text{State} \times \text{State}} \). The relational semantics \( \mathcal{R}[P] \) of \( P \) is given as the limit of the ascending Kleene chain of \( \mathcal{F}[P] \):

\[
\mathcal{R}(P, 0) = \emptyset \\
\mathcal{R}(P, h + 1) = \mathcal{F}[P](\mathcal{R}(P, h)) \\
\mathcal{R}[P] = \bigcup_{h \in \mathbb{N}} \mathcal{R}(P, h)
\]

Operationally, for any \( h \) we may view \( \mathcal{R}(P, h) \) as the input/output relation of \( P \) on a machine with a stack limit of \( h \) activation records. We can extend relational semantics to mutually recursive procedures in the natural way, by considering \( \mathcal{F}[P] \) to be function that takes as input a \( k \)-tuple of state relations (where \( k \) is the number of mutually recursive procedures).

A transition formula \( \varphi \) is a formula over the program variables \( \text{Var} \) and an additional set \( \text{Var}' \) of “primed” copies, representing the values of the program variables before and after a computation. A transition relation \( \varphi \) can be interpreted as a property that holds of a pair of states \( (\sigma, \sigma') \): we say that \( (\sigma, \sigma') \) satisfies \( \varphi \) if \( \varphi \) is true when each variable in \( \text{Var} \) is interpreted according to \( \sigma \), and each variable in \( \text{Var}' \) is interpreted according to \( \sigma' \). We use \( \mathcal{R}[\varphi] \) to denote the state relation consisting of all pairs \( (\sigma, \sigma') \) that satisfy \( \varphi \). This chapter is concerned with the problem of procedure summarization, in which the goal is to find a transition formula \( \varphi \) that over-approximates a procedure, in the sense that \( \mathcal{R}[P] \subseteq \mathcal{R}[\varphi] \).

A relational expression \( \tau \) is a polynomial over \( \text{Var} \cup \text{Var}' \) with rational coefficients. A relational expression can be evaluated at a state pair \( (\sigma, \sigma') \in \text{State} \times \text{State} \) by using \( \sigma \) to interpret the unprimed symbols and \( \sigma' \) to interpret the primed symbols—we use \( \mathcal{E}[\tau](\sigma, \sigma') \) to denote the evaluation of \( \tau \) at \( (\sigma, \sigma') \).

Intra-procedural analysis. The technique for procedure summarization developed in this chapter makes use of intra-procedural summarization as a sub-routine. We formalize this intra-procedural technique by a function \( \text{PathSummary}(e, x, V, E) \), which takes as input a control-flow graph with vertices \( V \), edges \( E \), entry
Algorithm 6: The convex-hull algorithm from [Farzan and Kincaid, 2015]

Input: Formula of the form $\exists X.\psi$ where $\psi$ is satisfiable and quantifier-free
Output: Convex hull of $\exists X.\psi$

1. $P \leftarrow \bot$
2. while there exists a model $m$ of $\psi$ do
3. Let $Q$ be a cube of the DNF of $\psi$ s.t. $m \models Q$;
4. $Q \leftarrow \text{project}(Q, X)$; /* Polyhedral projection */
5. $P \leftarrow P \sqcup Q$; /* Polyhedral join */
6. $\psi \leftarrow \psi \land \neg P$;
7. return $P$

vertex $e$, and exit vertex $x$, and computes a transition formula that over-approximates all paths in $(V, E)$ between $e$ and $x$. We use $\text{Summary}(P, \varphi)$ to denote a function that takes as input a recursive procedure $P$ and a transition formula $\varphi$, and computes a transition formula that over-approximates $P$ when $\varphi$ is used to interpret recursive calls (i.e., $\mathcal{F}[P](\\mathcal{R}[\varphi]) \subseteq \mathcal{R}[\text{Summary}(P, \varphi)])$. $\text{Summary}(P, \varphi)$ can be implemented in terms of $\text{PathSummary}(e, x, V, E)$ by replacing all call edges with $\varphi$, and taking $(e, x, V, E)$ to be the control-flow graph of $P$.

In principle, any intra-procedural summarization procedure can be used to implement $\text{Summary}(P, \varphi)$; the implementation of our method uses the techniques of Chapter 4.

Symbolic abstraction. We use $\text{Abstract}(\varphi, V)$ to denote a procedure that takes a formula $\varphi$ and computes a set of polynomial inequations over the variables $V$ that are implied by $\varphi$. If $\varphi$ is expressed in linear arithmetic, then a representation of all implied polynomial inequations (namely, a constraint representation of the convex hull of $\varphi$ projected onto $V$) can be computed effectively (e.g., using [Farzan and Kincaid, 2015, Alg. 2], which we show in this chapter as Alg. 6). Otherwise, we settle for a sound procedure that produces inequations implied by $\varphi$, but not necessarily all of them. In our implementation of the techniques of this chapter, we use the wedge-domain symbolic-abstraction algorithm Alg. 3, which is discussed in more detail in §4.4.3. Note that $\text{Abstract}$ may be called on a formula $\varphi$ that is expressed in non-linear arithmetic; to handle that case, our implementation of $\text{Abstract}$ works by using Alg. 3 to compute a wedge $w$ that over-approximates $\varphi$ and then returning the inequations in the constraint representation of the underlying polyhedron of $w$.

5.4 Technical Details

This section gives algorithms for summarizing recursive procedures using recurrence solving. We assume that before these algorithms are applied to the procedures of a program $P$, we first compute and collapse the strongly connected components of the call graph of $P$ and topologically sort the collapsed graph. Our analysis then works on the strongly connected components of the call graph in a single pass, in a topological order of the collapsed graph, by applying the algorithms of this section to recursive components, and applying
intraprocedural analysis to non-recursive components. Moreover, each recursive component can be classified as either linearly recursive or non-linearly recursive. Linearly recursive components can be analyzed using the tensor-based techniques of Chapter 3; then, only non-linearly recursive components need to be analyzed using the techniques described in this section (§5.4).

For simplicity, §5.4.1 focuses on the analysis of strongly connected components consisting of a single recursive procedure $P$. The first step of the analysis is to apply Alg. 7, which produces a set of inequations that describe the values of variables in $P$. Not all of the inequations found by Alg. 7 are suitable for use in a recurrence-based analysis, so we apply Alg. 8 to filter the set of inequations down to a subset that, when combined, form a stratified recurrence. The next step is to give this recurrence to a recurrence solver, which results in a logical formula relating the values of variables in $P$ to the stack height $h$ that may be used by $P$. In §5.4.2, we show how to (i) obtain a bound on $h$ that depends on the program state before the initial call to $P$, and (ii) combine the recurrence solution with that depth bound to create a summary of $P$. In §5.4.3, we discuss how to obtain a certain class of more precise bounds (including lower bounds on the running time of a procedure). In §5.4.4, we show how to extend the techniques of §5.4.1 to handle programs with mutual recursion, i.e., programs whose call graphs have strongly connected components consisting of multiple procedures. In §5.4.5, we discuss an extension of the algorithm of §5.4.4 that handles sets of mutually recursive procedures in which some procedures do not have base cases.

5.4.1 Height-Based Recurrence Analysis

Let $\tau$ be a relational expression and let $P$ be a procedure. We use $V_\tau(P, h)$ to denote the set of values of $\tau$ in a height-$h$ execution of $P$.

$$V_\tau(P, h) \overset{\text{def}}{=} \{E[\tau](\sigma, \sigma') : (\sigma, \sigma') \in R(P, h)\}$$

It consists of values to which $\tau$ may evaluate at a state pair belonging to $R(P, h)$. We call $b_\tau : N \to \mathbb{Q}$ a bounding function for $\tau$ in $P$ if for all $h \in N$ and all $v \in V_\tau(P, h)$, we have $v \leq b_\tau(h)$. Intuitively, the bounding function $b_\tau(h)$ bounds the value of an expression $\tau$ in any execution that uses stack height at most $h$.

The goal of §5.4.1 is to find a set of relational expressions and associated bounding functions. We proceed in three steps. First, we determine a set of candidate relational expressions $\tau_1, ..., \tau_n$. Second, we optimistically assume that there exist functions $b_1(h), ..., b_n(h)$ that bound these expressions, and we analyze $P$ under that assumption to obtain constraints relating the values of the relational expressions to the values of the $b_1(h), ..., b_n(h)$ functions. Third, we re-arrange the constraints into recurrence relations for each of the $b_k(h)$ functions (if possible) and solve them to synthesize a closed-form expression for $b_k(h)$ that is suitable to be used in a summary for $P$. 
Algorithm 7: Algorithm for extracting candidate recurrence inequations

Input : A procedure \( P \), and the associated vocabulary of program variables \( Var \)
Output: A set \( S \) of candidate recurrence inequations for \( P \)

1. \( \beta \leftarrow \text{Summary}(P, \text{false}) \);
2. \( w_{\text{base}} \leftarrow \text{Abstract}(\beta, Var \cup Var') \);
3. \( n \leftarrow \) the number of inequations in \( w_{\text{base}} \);
4. \textbf{foreach} \( k \) in \( 1, ..., n \) \textbf{do}
   5. \hspace{1em} Let \( \tau_k \) be the expression over \( Var \cup Var' \) such that the \( k \text{th} \) inequation in \( w_{\text{base}} \) is \( (\tau_k \leq 0) \);
   6. \hspace{1em} Let \( b_k() \) be a fresh uninterpreted function symbol;
   7. \hspace{1em} \( \phi_{\text{call}} \leftarrow \bigwedge_{k=1}^{n} (\tau_k \leq b_k(h) \land b_k(h) \geq 0) \);
   8. \hspace{1em} \( \phi_{\text{rec}} \leftarrow \text{Summary}(P, \phi_{\text{call}}) \);
   9. \hspace{1em} \( \phi_{\text{ext}} \leftarrow \phi_{\text{rec}} \land \bigwedge_{k=1}^{n} (b_k(h + 1) = \tau_k) \);
   10. \( S \leftarrow \emptyset \);
5. \textbf{foreach} \( k \) in \( 1, ..., n \) \textbf{do}
   6. \hspace{1em} \( w_{\text{ext},k} \leftarrow \text{Abstract}(\phi_{\text{ext}}, b_1(h), ..., b_n(h), b_k[h + 1]) \);
   7. \hspace{1em} \textbf{foreach inequation} \( j \) in \( w_{\text{ext},k} \) \textbf{do}
       8. \hspace{2em} \( S \leftarrow S \cup \{j\} \)
6. \textbf{return} \( S \)

We begin our analysis of \( P \) by determining a set of suitable expressions \( \tau \). If a relational expression \( \tau \) has an associated bounding function, then it must be the case that \( V_\tau(P,1) \) (i.e., the set of values that \( \tau \) takes on in the base case) is bounded above. Without loss of generality, we choose expressions \( \tau \) so that \( V_\tau(P,1) \) is bounded above \textit{by zero}. (Note that if \( V_\tau(P,1) \) is bounded above by \( c \) then \( V_{\tau-c}(P,1) \) is bounded above by zero.) We begin our analysis of \( P \) by analyzing the base case to look for relational expressions that have this property.

Selecting candidate relational expressions. The reason for looking at expressions over program variables, as opposed to individual variables, is illustrated by Ex. 5.1: the variable \( n\text{Tcks} \) has a different value each time the base case executes, but the expression \( n\text{Tcks}' - n\text{Tcks} - 1 \) is always equal to zero in the base case.

With the goal of identifying relational expressions that are bounded above by zero, Alg. 7 begins by extracting a transition formula \( \beta \) for the non-recursive paths through \( P \) by calling \( \text{Summary}(P, \text{false}) \) (i.e., summarizing \( P \) by using \textit{false} as a summary for the recursive calls in \( P \)). Next, we compute a set \( w_{\text{base}} \) of polynomial inequations over \( Var \cup Var' \) (the set of un-primed (pre-state) and primed (post-state) copies of all global variables, along with unprimed copies of the parameters to \( P \) and the variable \textit{return}', which represents the return value of \( P \)) that are implied by \( \beta \) by calling \( \text{Abstract}(\beta, Var \cup Var') \). Let \( n \) be the number of inequations in \( w_{\text{base}} \). Then, for \( k = 1, ..., n \), we rewrite the \( k \text{th} \) inequation in the form \( \tau_k \leq 0 \). In the case of \( \text{Ex. 5.1}, \tau_1 \overset{\text{def}}{=} \text{return}' \) and \( \tau_2 \overset{\text{def}}{=} n\text{Tcks}' - n\text{Tcks} - 1 \) have the property that \( \tau_1 \leq 0 \) and \( \tau_2 \leq 0 \) in the base case.

Note that there are, in general, many sets of relational expressions \( \tau_1, ..., \tau_n \) that are bounded above by zero in the base case. The soundness of Alg. 7 only depends on \( \text{Abstract} \) choosing some such set. Our implementation of \( \text{Abstract} \) uses Alg. 3 (discussed in §4.4.3), and is not guaranteed to choose the set of
relational expressions that would lead to the most-precise results for any given application, e.g., for a given assertion-checking or complexity-analysis problem. Intuitively, in the case that $\beta$ is a formula in linear arithmetic, our implementation of Abstract amounts to using the operations of the polyhedral abstract domain to find a convex hull of $\beta$. Then, each of the inequations in the constraint representation of the convex hull can be interpreted as a relational expression that is bounded above by zero in the base case.

Generating constraints on bounding functions. For each of the expressions $\tau_k$ that has an upper bound in the base case, we are ultimately looking to find a function $b_k(h)$ that is an upper bound on the value of that expression in any height-$h$ execution. Our way of finding such a function is to analyze the recursive cases of $P$ to look for an invariant inequation that gives an upper bound on $V_{\tau_k}(P, h + 1)$ in terms of an upper bound on $V_{\tau_k}(P, h)$. Such an inequation can be interpreted as a recurrence relating $b_k(h + 1)$ to $b_k(h)$.

The remainder of Alg. 7 (Lines (7)–(14)) finds such invariant inequations. The first step is to create the hypothetical procedure summary $\varphi_{\text{call}}$, which hypothesizes that a bounding function $b_{\tau_k}$ exists for each expression $\tau_k$, and that the value of that function at height $h$ is an upper bound on the value of $\tau_k$. $\varphi_{\text{call}}$ is a transition formula that represents a height-$h$ execution of $P$. In Ex. 5.1, $\varphi_{\text{call}}$ is:

$$
\text{return}' \leq b_1(h) \land \text{nTicks}' - \text{nTicks} - 1 \leq b_2(h) \land
b_1(h) \geq 0 \land b_2(h) \geq 0
$$

On line (8), Alg. 7 calls Summary, using $\varphi_{\text{call}}$ as the representation of each recursive call in $P$, and the resulting transition formula is stored in $\varphi_{\text{rec}}$. Thus, $\varphi_{\text{rec}}$ describes a typical height-$(h + 1)$ execution of $P$. In Ex. 5.1, a simplified version of $\varphi_{\text{rec}}$ is given as $\varphi_{h+1}$ in §5.2.

On line (9), the formula $\varphi_{\text{ext}}$ is produced by conjoining $\varphi_{\text{rec}}$ with a formula stating that, for each $k$, $b_k(h + 1) = \tau_k$. Therefore, $\varphi_{\text{ext}}$ implies that any upper bound on $b_k(h + 1)$ must be an upper bound on $\tau_k$ in any height-$(h + 1)$ execution.

Ultimately, we wish to obtain a closed-form solution for each $b_k(h)$. The formula $\varphi_{\text{ext}}$ implicitly determines a set of recurrences relating $b_1(h + 1), ..., b_n(h + 1)$ to $b_1(h), ..., b_n(h)$. However, $\varphi_{\text{ext}}$ does not have the explicit form of a recurrence. Lines (12)–(14) abstract $\varphi_{\text{ext}}$ to a conjunction of inequations that give an explicit relationship between $b_k(h + 1)$ and $b_1(h), ..., b_n(h)$ for each $k$.

Extracting and solving recurrences. The next step of height-based recurrence analysis is to identify a subset of the inequations returned by Alg. 7 that constitute a stratified system of polynomial recurrences (Defn. 2.2). This subset must meet the following three stratification criteria:

1. Each bounding function $b_k(h + 1)$ must appear on the left-hand-side of at most one inequation.
Algorithm 8: Algorithm for constructing a stratified recurrence

```
Input: A set of candidate inequations \( j_1, ..., j_N \) over the function symbols \( b_1(h), ..., b_n(h), b_1(h+1), ..., b_n(h+1) \)
Output: A set of inequations that form a stratified recurrence
1. Let DefinesBound\[j] be a map from integers to integers;
2. Let UsesBound\[j,k] and UsesBoundNonLinearly\[j,k] be maps that map all pairs of integers to false;
3. \( S \leftarrow \{1, ..., N\} \);
4. foreach \( j \) in \( 1, ..., N \) do
   5. Write \( j \) as \( b_{i_j}(h+1) \leq c_0 + \sum_{i=1}^{m_j} c_i (b_1(h))^{d_{i_1}} \cdots (b_n(h))^{d_{i_n}} \) if \( j \) can be written in that form with \( 1 \leq k \leq n \),
      \( \forall i, c_i \in \mathbb{Q}, \exists i > 0, c_i > 0, \forall i, p, d_{i,p} \in \mathbb{N} \); otherwise let \( S \leftarrow S \setminus \{j\} \) and continue loop;
6. For \( i = 0, ..., m_j \), \( c'_i \leftarrow \max(0, c_i) \);
7. Let \( j' \) be \( b_k(h+1) \leq c'_0 + \sum_{i=1}^{m_j} c'_i (b_1(h))^{d_{i_1}} \cdots (b_n(h))^{d_{i_n}} \);
8. DefinesBound\[j'] := k;
9. foreach \( i \in \{1, ..., m_j\} \) do
   10.   foreach \( p \in \{1, ..., n\} \) do
      11.      if \( c'_i > 0 \land d_{i,p} > 0 \) then UsesBound\[j,p] := true ;
      12.      if \( c'_i > 0 \land d_{i,p} > 0 \land \sum_{q=1}^{n} d_{i,q} > 1 \) then UsesBoundNonLinearly\[j,p] := true ;
13. \( A \leftarrow \emptyset \);
14. repeat
15. \( V \leftarrow S - A \);
16. repeat
17.   foreach \( j \in V \) do
18.      if \( \exists k. \text{UsesBound}[j,k] \land \exists j' \in V. \text{DefinesBound}[j'] = k \) then \( V \leftarrow V \setminus \{j\} \);
19.      if \( \exists k. \text{UsesBoundNonLinearly}[j,k] \land \exists j' \in A. \text{DefinesBound}[j'] = k \) then \( V \leftarrow V \setminus \{j\} \);
20. until \( V \) is unchanged;
21. foreach \( k = \{1, ..., n\} \) do
22.   if \( V \) contains more than one \( j \) such that DefinesBound\[j\] = k then
23.      Arbitrarily choose one such \( j \) to remain in \( V \), and remove all other such \( j \) from \( V \);
24. \( A \leftarrow A \cup V \);
25. until \( V = \emptyset \);
26. return \( \{j' | j \in A\} \)
```

2. If a bounding function \( b_k(h) \) appears on the right-hand-side of an inequation, then \( b_k(h+1) \) appears on some left-hand-side.

3. It must be possible to organize the \( b_k(h+1) \) into strata, so that if \( b_k(h) \) appears in a non-linear term on the right-hand-side of the inequation for \( b_1(h+1) \), then \( b_k(h) \) must be on a strictly lower stratum than \( b_k(h) \).

Alg. 8 computes a maximal subset of inequations that complies with the above three rules.

The next step of height-based recurrence analysis is to send this recurrence to a recurrence solver, such as the one described in §4.6. The solution to the recurrence is a set of bounding functions. Let \( B \) be the set of indices \( k \) such that we found a recurrence for, and obtained a closed-form solution to, the bounding function \( b_k(h) \). Using these bounding functions, we can derive the following procedure summary for \( P \), which leaves the height \( H \) unconstrained.

\[
\exists H. \bigwedge_{k \in B} \lbrack \tau_k \leq b_k(H) \rbrack \tag{5.3}
\]
The subject of §5.4.2 is to find a formula \( \zeta_{P_i}(H, \sigma) \) relating \( H \) to the pre-state \( \sigma \) of the initial call to \( P_i \). The formula \( \zeta_{P_i}(H, \sigma) \) can be combined with Eqn. (5.3) to obtain a more precise procedure summary.

**Soundness.** Roughly, the soundness of height-based recurrence analysis follows from: (i) sound extraction of the recurrence constraints used by CHORA to characterize non-linear recursion; (ii) sound recurrence solving; and (iii) soundness of the underlying framework of algebraic program analysis. The soundness of parts (ii) and (iii) depends on the soundness of prior work (e.g., the soundness of the techniques of Chapter 4). The soundness of (i) is addressed in a detailed proof in App. B. The soundness property proved there is as follows: let \( P_i \) be a procedure to which Alg. 7 and Alg. 8 have been applied to obtain a stratified recurrence. Let \( \{\tau_i\}_{i \in [1,n]} \) be the relational expressions computed by Alg. 7. Let \( B \subseteq [1, n] \) be such that \( \{b_i\}_{i \in B} \) is the set of functions produced by solving the stratified recurrence. We show that each \( b_i \) function bounds the corresponding \( V_{\tau_i}(P, h) \) value set. In other words, for all heights \( h \geq 1 \), for all \( i \in B \), and for all \( v \in V_{\tau_i}(P, h) \) we have \( v \leq b_i(h) \).

### 5.4.2 Depth-Bound Analysis

In §5.4.1, we showed how to find a bounding function \( b_\tau(h) \) that gives an upper bound on the value of a relational expression \( \tau \) in an execution of a procedure \( P_i \) as a function of the stack height (i.e., maximum depth of recursion) \( h \) of that execution. In this section, the goal is to find bounds on the maximum depth of recursion \( h \) that may occur as a function of the pre-state \( \sigma \) (which includes the values of global variables and parameters to \( P_i \)) from which \( P_i \) is called.

For example, consider Ex. 5.1. The algorithms of §5.4.1 determine bounds on the values of two relational expressions in terms of \( h \), namely: \( n\text{Ticks}' \leq n\text{Ticks} + 2^h - 1 \), and \( \text{return}' \leq h - 1 \). The algorithm of this
sub-section (Alg. 9) determines that \( h \) satisfies \( h \leq \max(1, 1 + n - 1) \). These facts can be combined to form a procedure summary for \( \text{SubsetSumAux} \) that relates the return value and the increase to \( n \text{Ticks} \) to the values of the parameters \( i \) and \( n \).

The stack height \( h \) required to execute a procedure often depends on the number of times that some transformation can be applied to the procedure’s parameters before a base case must execute. For example, in Ex. 5.1, the height bound is a consequence of the fact that \( i \) is incremented by one at each recursive call, until \( i \geq n \), at which point a base case executes. Likewise, in a typical divide-and-conquer algorithm, a size parameter is repeatedly divided by some constant until the size parameter is below some threshold, at which point a base case executes. Intuitively, the technique described in this section is designed to discover height bounds that are consequences of such repeated transformations (e.g., addition or division) applied to the procedures’ parameters.

To achieve this goal, we use Alg. 9, which is inspired by the algorithm for computing bounds on the depth of recursion in Albert et al. [2013]. Alg. 9 constructs and analyzes an over-approximate depth-bounding model of the procedures \( P_1, \ldots, P_n \) that includes an auxiliary depth-counter variable, \( D \). Each time that the model descends to a greater depth of recursion, \( D \) is incremented. The model exits only when a procedure executes its base case. In any execution of the model, the final value of \( D \) thus represents the depth of recursion at which some procedure’s base case is executed.

Alg. 9 takes as input a representation of the procedures in \( S \) as a single, combined control-flow graph \((V, E, C)\) having two kinds of edges: (1) weighted edges \((u, \varphi, v) \in E\), which are weighted with a transition formula \( \varphi \), and (2) call edges in the set \( C \). Each call edge in \( C \) is a triple \((u, Q, v)\), in which \( u \) is the call-site vertex, \( v \) is the return-site vertex, and the edge is labeled with \( Q \), representing a call to a procedure \( Q \). We assume that if any procedure \( Q \notin S \) is called by some procedure in \( S \), then \( Q \) has been fully analyzed already, and therefore a procedure summary \( \varphi_Q \) for \( Q \) has already been computed. Each procedure \( Q \) has an entry vertex \( e_Q \), an exit vertex \( x_Q \), and a transition formula \( \beta_Q \) that over-approximates the base cases of \( Q \). Note that \((V, E, C)\) consists of several disjoint, single-procedure control-flow graphs when \( n > 1 \).

On lines (2)–(13), Alg. 9 constructs the depth-bounding model, represented as a new control-flow graph \((V', E', \emptyset)\). The algorithm begins by creating new auxiliary entry vertices \( e'_{P_1}, \ldots, e'_{P_n} \) for the procedures \( P_1, \ldots, P_n \) and a new auxiliary exit vertex \( x' \). The new vertex set \( V' \) contains \( V \) along with these \( n + 1 \) new vertices. Alg. 9 then creates a new integer-valued variable \( D \). For \( i = 1, \ldots, n \), the algorithm then creates an edge from \( e'_{P_i} \) to \( e_{P_i} \), weighted with a transition formula that initializes \( D \) to one, and an edge from \( x_{P_i} \) weighted with the formula \( \beta_{P_i} \), which is a summary of the base case of \( P_i \).

Alg. 9 replaces every call edge \((u, Q, v) \in C\) with one or more weighted edges. Each call to a procedure \( Q \notin \{P_1, \ldots, P_n\} \) is replaced by an edge \((u, \varphi_Q, v)\) weighted with the procedure summary \( \varphi_Q \) for \( Q \). Each call
to some \( P_i \) is replaced by two edges. The first edge represents descending into \( P_i \), and goes from \( u \) to \( e_{P_i} \), and is weighted with a formula that increments \( D \) and havocs local variables. The second edge represents skipping over the call to \( P_i \) rather than descending into \( P_i \). This edge is weighted with a transition formula that havocs all global variables and the variable \( \text{return} \), but leaves local variables unchanged.

The final step of Alg. 9, on line (15), actually computes the depth-bounding summary \( \xi_{P_i}(D, \sigma) \) for each procedure \( P_i \). Because there are no call edges in the new control-flow graph \((V', E', \emptyset)\), intraprocedural-analysis techniques can be used to compute transition formulas that summarize the transition relation for all paths between two specified vertices. For each procedure \( P_i \), the formula \( \xi_{P_i}(D, \sigma) \) is a summary of all paths from \( e_{P_i}' \) to \( x' \), which serves to relate \( D \) to \( \sigma \), which is the pre-state of the initial call to \( P_i \).

The formulas \( \xi_{P_i}(D, \sigma) \) for \( i = 1, ..., n \) can be used to establish an upper bound on the depth of recursion in the following way. Let \( (\sigma, \sigma') \) be a state pair in the relational semantics \( R[P_i] \) of \( P_i \). Then, there is an execution \( e \) of \( P_i \) that starts in state \( \sigma \) and finishes in state \( \sigma' \), in which the maximum\(^1\) recursion depth is some \( d \in \mathbb{N} \). Then there is a path through the control-flow graph \((V', E', \emptyset)\) that corresponds to the path taken in \( e \) to reach some execution of a base case at the maximum recursion depth \( d \). Therefore, if \( d \) is a possible depth of recursion when starting from state \( \sigma \), then there is a satisfying assignment of \( \xi_{P_i}(D, \sigma) \) in which \( D \) takes the value \( d \). The contrapositive of this argument says that, if there does not exist any satisfying assignment of \( \xi_{P_i}(D, \sigma) \) in which \( D \) takes the value \( d \), then it must be the case that no execution of \( P_i \) that starts in state \( \sigma \) can have maximum recursion depth \( d \). In this way, \( \xi_{P_i}(D, \sigma) \) can be interpreted as providing bounds on the maximum recursion depth that can occur when \( P_i \) is started in state \( \sigma \).

Once we have the depth-bound summary \( \xi_P \) for some procedure \( P \), we can combine it with the closed-form solutions for bounding functions that we obtained using the algorithms of §5.4.1 to produce a procedure summary. Let \( B \) be the set of indices \( k \) such that we found a recurrence for the bounding function \( b_k(h) \). We produce a procedure summary of the form shown in Eqn. (5.4), which uses the depth-bound summary \( \xi_P \) to relate the pre-state \( \sigma \) to the variable \( H \), which in turn is used to index into the bounding function \( b_k(h) \) for each \( k \in B \).

\[
\exists H. \xi_P(H, \sigma) \land \bigwedge_{k \in B} [\tau_k \leq b_k(H)] \quad (5.4)
\]

5.4.3 Finding Lower Bounds Using Two-Region Analysis

In this sub-section, we describe two-region analysis, which is an extension of height-based recurrence analysis that is able to prove some stronger conclusions, including non-trivial lower bounds on the running times of

\(^1\) Note that non-terminating executions of \( P_i \) do not correspond to any state-pair \( (\sigma, \sigma') \) in the relational semantics \( R[P_i] \); therefore, such executions are not represented in the procedure summary for \( P_i \) that we wish to construct.
int x; int y;
int differ(int n)
{
    if (n == 0 || n == 1) { x = 0; y = 0; return; }
    differ(nondet()); n - 1 : n - 2;
    int temp = x; // Store x "returned" by first call
differ(nondet()); n - 1 : n - 2;
    x = temp + 1; y = y + 1; // "Return" (temp+1,y+1)
}

Figure 5.2: Source code for the non-linearly recursive procedure `differ`, and a depiction of a possible tree of recursive calls `differ`. The number inside each vertex indicates the value of the parameter `n` at that execution of `differ`. The minimum depth `M` at which a base case occurs in this recursion tree is 3. The upper region of the tree, which includes all vertices at depth less than or equal to `M`, is shown with bold outlines.

In §5.4.1, we discussed height-based recurrence analysis, and showed how it can find an upper bound on the increase to the variable `nTicks` in Ex. 5.1. Now, we consider the application of height-based recurrence analysis to the procedure `differ` shown in Fig. 5.2. `differ` uses the global variables `x` and `y` to (in effect) return a pair of integers. The pair `(x, y)` returned by the procedure is formed from the `x` value returned by the first call and the `y` value returned by the second call, each incremented by one. The base case occurs when the parameter `n` equals zero or one, and at each call site, the parameter `n` is decreased by either one or two. We will apply height-based recurrence analysis and two-region analysis to look for bounds on `x'` and `y'`, and their sum and difference, after `differ` is called with a given value `n`.

For the purposes of the following discussion, we will focus on `x`, but the same conclusions apply to `y`. By applying height-based recurrence analysis to the procedure `differ`, we can prove that the post-state value `x'` is upper-bounded by `n - 1`. At the same time, the analysis also proves a lower bound on `x'` by considering the term `τ1 = -x'`. However, the bounding function `b₁(h)` obtained by height-based analysis is the constant function `b₁(h) = 0`, which yields the trivial lower bound `x' ≥ 0`. As a result, the results of height-based recurrence analysis can only be used to prove that the difference between `x'` and `y'` is at most `n - 1`, which is an over-estimate by a factor of two.

In this sub-section, we extend our formal characterization of the relational semantics of a procedure (given in §5.3) in the following way. We use `V_τ(X)` to denote the set of values that `τ` takes on in a state relation `X`. That is, `V_τ(X) = \{ c[τ](σ, σ') : (σ, σ') ∈ X \}`. We view a procedure `P` as a pair consisting of a state relation `F_{base}[P] ∈ 2^{State × State}` (which gives the relational semantics of the “base case” of `P`) and a (⊥-strict) function `F_{rec}[P] : 2^{State × State} → 2^{State × State}` (which gives the relational semantics of the “recursive case” of `P`), such that `F[P](X) = F_{rec}[P](X) \cup F_{base}[P]` for any state relation `X`. For any natural number `m`, define `F_{rec}[P]^m` to be the `m`-fold composition of `F_{rec}[P]`, and define `F[P]^m` to be the `m`-fold composition of `F[P]`. (Then, for every `m ≥ 1`, `F[P]^{m-1}(F_{base}[P])` is the same as the height-bounded relational semantics `R[P, m]` defined in §5.3.)
Note that $\mathcal{T}_{\text{rec}}[P]^m(\mathcal{T}_{\text{base}}[P])$ corresponds to the state relation that is exactly $m$ “steps” away from $\mathcal{T}_{\text{base}}[P]$, whereas $\mathcal{T}[P]^m(\mathcal{T}_{\text{base}}[P])$ corresponds to a state relation that is inclusive of all state relations between zero and $m$ steps away from $\mathcal{T}_{\text{base}}[P]$. We say that a function $b_\tau : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Q}$ is a lower bound for $\tau$ in $P$ if for all $n, m \in \mathbb{N}$ and all $v \in \mathcal{V}_\tau(\mathcal{T}_{\text{rec}}[P]^m(\mathcal{R}(P, n)))$, we have $b_\tau(m, n) \leq v$. Our goal in this sub-section is to find such lower-bounding functions.

The preceding formal presentation can also be understood using the following intuitive characterization of a tree of recursive calls. We characterize a tree of recursive calls with two parameters: (i) the height $H$, and (ii) the minimum depth $M$ at which a base case occurs. Importantly, the depth-bound analysis of §5.4.2 can be used to obtain bounds on the parameters $H$ and $M$. We define the upper region of $T$ to be the tree that is produced by removing from $T$ all vertices that are at depth greater than $M$. The lower region of $T$ contains all the vertices of $T$ that are not present in the upper region. In general, the lower region is comprised of zero or more trees. (In Fig. 5.2, the upper region is shown with bold outlines.)

The relational semantics of the lower region are given by $\mathcal{R}(P, H - M)$, where $H - M$ is the maximum height of any vertex at depth $M$, which corresponds to the bottom of the upper region. The relational semantics of the upper region are given by $\mathcal{T}_{\text{rec}}[P]^M(\mathcal{R}(P, H - M))$. The idea of our approach is to apply height-based recurrence analysis in the lower region (to summarize $\mathcal{R}(P, H - M)$), and a modified analysis in the upper region (to summarize $\mathcal{T}_{\text{rec}}[P]^M(\mathcal{X})$), and then combine the results to produce a procedure summary for $\mathcal{T}_{\text{rec}}[P]^M(\mathcal{R}(P, H - M))$.

In the lower region, we perform height-based recurrence analysis unmodified (as in §5.4.1) to obtain, for each relational expression $\tau_k$, a bounding function $b_k^L(h)$. The only difference is that we will not evaluate our bounding functions at the height $H$ of the entire tree to find bounds on the value of $\tau_k$ at the root of the tree. Instead, we use the lower-region bounding functions $b_k^L(h)$ to obtain bounds on the value of $\tau_k$ at the height $(H - M)$.

In the upper region, we perform a modified height-based recurrence analysis in which we substitute the notion of upper-region height for the notion of height. The upper-region height of a vertex $v$ at depth $d_v$ in the upper region is defined to be $M - d_v$. Thus, vertices at depth $M$ (i.e., the bottom of the upper region) have upper-region height zero, and the root (at depth 0) has upper-region height $M$. For each $\tau_k$, the upper-region bounding function $b_k^U(h)$ needs to bound $\mathcal{V}_{\tau_k}(\mathcal{T}[P]^M(\mathcal{X}))$. Therefore, in the upper region, we only require the bounding function $b_k^U(h)$ to be a bound on the values that the expression $\tau_k$ can take on at exactly the upper-region height $h_v$, rather than requiring $b_k^U(h)$ to be an upper bound on the values that $\tau_k$ can take on at any height between one and $h$. Consequently, bounding functions $b_k^U(h)$ are not required to be non-decreasing as upper-region height increases.

We make three changes to the algorithms of §5.4.1 to find the bounding functions $b_k^U(h)$ for the upper
region. First, in Alg. 7, on line (7), we remove the conjunct that asserts that the bounding functions are greater than or equal to zero. Second, we change Alg. 8 by removing line (6), so that recurrences are allowed to have a negative constant coefficient. Third, in Alg. 7, we modify line (8) so that the resulting summary formula $\varphi_{rec}$ is a summary of only the recursive paths through the procedure, rather than a summary that includes base cases. Conceptually, the use of a summary that excludes base cases is the most important of the changes. For example, consider a time-complexity analysis of a procedure that has a constant-time base case. If recurrences are extracted from a formula that includes the base case, then the lower-bound recurrence for the time-cost variable must always be consistent with the running time of the base case, and therefore the lower bound on time-cost will be a trivial, constant lower bound. By contrast, if recurrences are extracted from a formula that excludes the base case, then it is possible to have a lower-bound recurrence for time-cost that increases with height.

Analysis results for the two regions are combined in the following way. After analyzing both regions, we have obtained, for several quantities $\tau_k$, closed-form solutions to the recurrences for two bounding functions. $b_L^k(h)$ is the closed form solution for the lower-region bounding function in terms of the height $h$. The upper-region closed-form solution $b_U^k(h', c_U^k)$ is expressed in terms of two parameters: an upper-region-height parameter $h'$, and a symbolic initial condition parameter $c_U^k$ that determines the value of the bounding function when the upper-region-height parameter is zero.

We relate the values of the two bounding functions to one another and to the associated term $\tau_k$ over program variables by constructing the formula given below as Eqn. (5.5). In Eqn. (5.5), bounding functions obtained by height-based analysis of the lower region always equal zero at height one, just as in §5.4.1. By contrast, the initial condition parameter $c_U^k$ for the upper region is specified to be $b_L^k(H - M)$, i.e., the value of the lower-region bounding function evaluated at height $H - M$.

As in §5.4.2, we use, for each procedure $P$, the depth-bound formula $\zeta_P(D, \sigma)$ to bound the tree-shape parameters $H$ and $M$ as a function of the pre-state $\sigma$ of the initial call to $P$. In effect, $\zeta_P(D, \sigma)$ constrains its parameter $D$ to equal the length of some feasible root-to-leaf path in a tree of recursive calls starting from $\sigma$. Thus, we can obtain a sound upper bound on $H$ and a sound lower bound on $M$ by using two copies of $\zeta_P(D, \sigma)$ instantiated with the two shape parameters, because $H$ is upper-bounded by the length of the longest root-to-leaf path in the tree of recursive calls, and $M$ is lower-bounded by the length of the shortest root-to-leaf path.

As in the earlier procedure summary formula Eqn. (5.4) in §5.4.2, $B$ represents the set of indices $k$ such that we obtained bounding functions in both the lower and upper regions. The final procedure summary

---

2We obtain a summary that excludes non-recursive paths by adding an auxiliary flag variable $r$ to the program that indicates whether a recursive call has occurred, and then modifying our internal representation of the procedure so that (i) $r$ is initially false, (ii) $r$ is updated to true when a call occurs, and (iii) $r$ is assumed to be true at the end of the procedure.
produced by two-region analysis is given below as Eqn. (5.5).

\[ \exists H \exists M. M \leq H \land \zeta_P(M, \sigma) \land \zeta_P(H, \sigma) \land \bigwedge_{k \in B} [\tau_k \leq b_k^U(M, b_k^L(H - M))] \] (5.5)

We now consider the application of Eqn. (5.5) to the procedure \textit{Differ} from Fig. 5.2. The two bounded terms related to \( x' \) are \( \tau_1 = -x' \) and \( \tau_2 = x' \). (There are also two terms for \( y' \) that are analogous to those for \( x' \).) The lower-region and upper-region recurrences for these terms are as follows.

\[
\begin{align*}
    b_1^L(h + 1) &= b_1^L(h) & (5.6) \\
    b_1^U(h' + 1) &= b_1^U(h') - 1 & (5.7)
\end{align*}
\]

\[
\begin{align*}
    b_2^L(h + 1) &= b_2^L(h) + 1 & (5.8) \\
    b_2^U(h' + 1) &= b_2^U(h') + 1 & (5.9)
\end{align*}
\]

The closed-form solutions to these recurrences are as follows.

\[
\begin{align*}
    b_1^L(h) &= 0 & (5.10) \\
    b_2^L(h) &= h & (5.12)
\end{align*}
\]

\[
\begin{align*}
    b_1^U(h', c_1^U) &= c_1^U - h' & (5.11) \\
    b_2^U(h', c_2^U) &= c_2^U + h' & (5.13)
\end{align*}
\]

A much-simplified version of the procedure summary that we obtain for \textit{Differ} is:

\[
\frac{n - 1}{2} \leq x' \leq n \land \frac{n - 1}{2} \leq y' \leq n \] (5.14)

The key difference between the upper and lower regions is that Eqn. (5.6) leads to the non-decreasing solution Eqn. (5.10), whereas Eqn. (5.7) leads to the strictly decreasing solution Eqn. (5.11). In the final procedure summary, the initial conditions in the lower region are be specified to equal zero. Nevertheless, the lower-region recurrence solutions can create a non-zero gap between the lower bound (Eqn. (5.10)) on \( x' \) and the upper bound (Eqn. (5.12)) on \( x' \) (when \( h > 0 \)). In the upper region, the solutions Eqn. (5.11) and Eqn. (5.13) represent a lock-step increase in the upper and lower bounds on \( x' \) as \( h' \) increases (because \( -c_1^U + h' \leq x' \leq c_2^U + h' \) for any \( h' \)). However, there can be a gap between the initial condition values \( c_1^U = b_1^L(h) = 0 \) and \( c_2^U = b_2^L(h) = h \).

### 5.4.4 Mutual Recursion

In this section, we describe the generalization of the height-based recurrence analysis of §5.4.1 to the case of mutual recursion. Instead of analyzing a single procedure \( P \), we assume that we are given a set of procedures \( P_1, \ldots, P_m \) that form a strongly connected component of the call graph of some program.
Algorithm 10: Algorithm for extracting candidate recurrence inequations in the case of mutual recursion

**Input**: Procedures $P_1, ..., P_m$, and the associated vocabulary of program variables $Var$

**Output**: For each $i$ in $1, ..., m$, a set $S_i$ of candidate recurrence inequations for $P_i$

1. **foreach** $i$ in $1, ..., m$ **do**
   2. $b_{P_i} \leftarrow \text{Summary}(P_i, \text{false})$
   3. $w_{\text{base}}(P_i) \leftarrow \text{Abstract}(b_{P_i}, Var \cup Var')$
   4. $n_i \leftarrow$ the number of inequations in $w_{\text{base}}(P_i)$
   5. **foreach** $k$ in $1, ..., n_i$ **do**
      6. Let $\tau_{i,k}$ be the expression over $Var \cup Var'$ such that the $k^{\text{th}}$ inequation in $w_{\text{base}}(P_i)$ is $(\tau_{i,k} \leq 0)$
      7. Let $b_{i,k}(\cdot)$ be a fresh uninterpreted function symbol
      8. $\varphi_{\text{call}}(P_i) \leftarrow \bigwedge_{k=1}^{n_i} (\tau_{i,k} \leq b_{i,k}(h) \land b_{i,k}(h) \geq 0)$
   9. **foreach** $i$ in $1, ..., m$ **do**
      10. $\varphi_{\text{rec}}(P_i) \leftarrow \text{Summary}(P_i, \varphi_{\text{call}}(P_1), ..., \varphi_{\text{call}}(P_m))$
      11. $\varphi_{\text{ext}}(P_i) \leftarrow \varphi_{\text{rec}}(P_i) \land \bigwedge_{k=1}^{n_i} (b_{1,k}(h + 1) = \tau_{i,k})$
      12. $S_i \leftarrow \emptyset$
      13. **foreach** $k$ in $1, ..., n_i$ **do**
          14. $w_{\text{ext}}(P_i,k) \leftarrow \text{Abstract}(\varphi_{\text{ext}}(P_i), \{b_{1,1}(h), ..., b_{1,n_i}(h), ..., b_{m,1}(h), ..., b_{m,n_m}(h), b_{1,k}(h + 1))\}$
      15. **foreach inequation $I$ in** $w_{\text{ext}}(P_i,k)$ **do**
          16. $S_i \leftarrow S_i \cup \{I\}$
   17. **return** $S_1, ..., S_m$

Example 5.2. We use the following program to illustrate the application of our technique to mutually recursive procedures. The procedure $P_1$ increments the global variable $g$ in its base case, and calls $P_2$ eighteen times in a for-loop in its recursive case. Similarly, $P_2$ increments $g$ in its base case and calls $P_1$ two times in a for-loop in its recursive case.

```c
int g;

void P1(int n) {
    if (n <= 1) { g++; return; }
    for(int i = 0; i < 18; i++) { P2(n - 1); }
}

void P2(int n) {
    if (n <= 1) { g++; return; }
    for(int i = 0; i < 2; i++) { P1(n - 1); }
}
```

To apply height-based recurrence analysis to a set $S = \{P_1, ..., P_m\}$ of mutually recursive procedures, we use Alg. 10, which is a variant of Alg. 7 that interleaves some of the analysis operations on the procedures in $S$. First, for each procedure $P_i$, lines (2)–(8) of Alg. 10 perform essentially the same operations as lines (1)–(7) of Alg. 7. The result is that, for each procedure $P_i$, we obtain a symbolic summary formula $\varphi_{\text{call}}(P_i)$ and a set of bounded terms $\tau_{i,1}, ..., \tau_{i,n_i}$.

Note that a term $\tau_{i,r}$ that we obtain when analyzing $P_i$ may be syntactically identical to a term $\tau_{j,s}$ that we obtained when analyzing some earlier $P_j$. In such a case, $\tau_{i,r}$ and $\tau_{j,s}$ have different interpretations. For example, when analyzing Ex. 5.2, the two most important terms are $\tau_{i,1} = g' - g - 1$ and $\tau_{2,1} = g' - g - 1$. 

However, $\tau_{1,1}$ represents the increase to $g$ as a result of a call to $P_1$ and $\tau_{2,1}$ represents the increase to $g$ as a result of a call to $P_2$. Our technique will attempt to find distinct bounding functions for these two terms.

On lines (10)–(16) of Alg. 10, which are executed once for each procedure $P_i$, we perform operations that are similar to lines (8)–(14) of Alg. 7. On line (10), however, we use a generalized version of the call to the intraprocedural summarization function $\text{Summary}$. In the general case, each procedure $P_i$ might call every other member of its strongly connected component. To reduce this analysis step to an intraprocedural-analysis problem, we must replace every such call with a summary formula. Therefore, for each $P_i$, the call on the analysis subroutine has the form $\text{Summary}(P_i, \varphi_{\text{call}}(P_1), ..., \varphi_{\text{call}}(P_m))$. $\text{Summary}$ analyzes the body of $P_i$ by replacing each call to some $P_j$ with the formula $\varphi_{\text{call}}(P_j)$. The summary formula thus produced for $P_i$ is denoted by $\varphi_{\text{rec}}(P_i)$.

On line (11), the formula $\varphi_{\text{ext}}(P_i)$ is produced by conjoining $\varphi_{\text{rec}}(P_i)$ with one equality constraint for each of the terms $\tau_{i,1}, ..., \tau_{i,n_i}$. On line (14), the call to $\text{Abstract}$ has the form $\text{Abstract}(\varphi_{\text{ext}}(P_i), (b_{1,1}(h), ..., b_{1,n_i}(h), ..., b_{m,1}(h), ..., b_{m,n_m}(h), b_{i,k}(h+1)))$. That is, we look for inequations that provide a bound on $b_{i,k}(h+1)$, which relates to $P_i$ specifically, in terms of all of the height-$h$ bounding functions for $P_1, ..., P_m$. For example, in Ex. 5.2, we find the constraints $b_{1,1}(h+1) \leq 18b_{2,1}(h) + 17$ and $b_{2,1}(h+1) \leq 2b_{1,1}(h) + 1$.

The next steps of height-based analysis are to find a collection of inequations that form a stratified recurrence, and to solve that stratified recurrence (as in §5.4.1). These steps are the same in the case of mutual recursion as in the case of a single recursive procedure. After solving the recurrence, we obtain a closed-form solution for the subset of the bounding functions $b_{1,1}(h), ..., b_{m,n_m}(h)$ that appeared in the recurrence. Let $B_i$ be the set of indices $q$ such that we found a recurrence for $b_{i,q}(h)$. Then, the procedure summary that we obtain for $P_i$ has the following form:

$$\exists H. \zeta_{P_i}(H, \sigma) \land \bigwedge_{q \in B_i} [\tau_q \leq b_{i,q}(H)] \quad (5.15)$$

In Ex. 5.2, the recurrence that we obtain is:

$$\begin{bmatrix} b_{1,1}(h + 1) \\ b_{2,1}(h + 1) \end{bmatrix} \preceq \begin{bmatrix} 0 & 18 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} b_{1,1}(h) \\ b_{2,1}(h) \end{bmatrix} + \begin{bmatrix} 17 \\ 1 \end{bmatrix}$$

Notice that this recurrence involves an interdependency between the bounding functions for the increase to $g$ in $P1$ and $P2$. Simplified versions of the $g$ bounds found by Chora for $P1$ and $P2$ are $3 \cdot 6^{n-1}$ and $6^{n-1}$,
respectively.

The extension of two-region analysis (§5.4.3) to the case of mutual recursion is analogous to the extension of height-based recurrence analysis. It can be achieved by combining the changes to height-based recurrence analysis described in §5.4.3 with the changes to height-based recurrence analysis described in this sub-section.

For each procedure within a strongly connected component \( S \) of the call graph, the algorithm of §5.4.4 needs to be able to identify a base case (i.e., a set of paths containing no calls to the procedures of \( S \)). Some programs contain procedures without such base cases.

### 5.4.5 Equation Systems With Missing Base Cases

For each procedure within a strongly connected component \( S \) of the call graph, the algorithm of §5.4.4 needs to be able to identify a base case (i.e., a set of paths containing no calls to the procedures of \( S \)). Some programs contain procedures without such base cases, as in the following example.

**Example 5.3.**

```c
void P3(int n) {
    if (n <= 1) { P4(n - 1); P4(n - 1); return; }
    P3(n - 1); P4(n - 1);
}

void P4(int n) {
    if (n <= 1) { cost++; return; }
    P4(n - 1); P3(n - 1);
}
```

Notably, every path through \( P_3 \) makes a call on either \( P_3 \) or \( P_4 \). When Alg. 7 is applied to \( P_3 \), the base case summary \( \beta_{P_3} \) will be the transition formula false, because \( \beta_{P_3} \) is computed in a way that excludes all paths containing calls that are potentially indirectly recursive. Thus, no bounded terms will be found when analyzing \( \beta_{P_3} \). The procedure-summary equation system for these two procedures is shown below as Eqn. (5.17). In Eqn. (5.17), the variables \( P_3 \) and \( P_4 \) stand for the procedure summaries, and \( a \) is the base case of \( P_4 \), i.e. the action that adds one to the global variable \( \text{cost} \).

\[
P_3 = (P_4 \otimes P_4) \oplus (P_3 \otimes P_4) \tag{5.16}
\]

\[
P_4 = a \oplus (P_4 \otimes P_3) \tag{5.17}
\]

We can solve this problem by transforming the equation system in the following manner. For each \( j \in \{1, \ldots, i - 1, i + 1, \ldots, m \} \), create a new procedure-summary variable \( P_{j\backslash\{i\}} \) to represent executions of \( P_j \) that
never result in a call back to $P_i$. Next, replace every call to $P_j$ in the equation for $P_i$ with a call to $(P_j \oplus P_{j\setminus i})$ (so that a call to $P_j$ is allowed to either call back to $P_i$ or not do so). Let the original equation for $P_j$ be $P_j = \text{RHS}$. Then, create an equation for $P_{j\setminus i}$ by replacing $P_i$ with the trivial summary 0 (i.e., \textit{abort}) in \text{RHS}. Applying this transformation to Eqn. (5.17) yields:

$$P_3 = (P_3 \otimes (P_4 \oplus P_{4\setminus 3})) \oplus ((P_4 \oplus P_{4\setminus 3}) \otimes (P_4 \oplus P_{4\setminus 3}))$$
$$P_4 = (P_4 \otimes P_3) \oplus a$$
$$P_{4\setminus 3} = a \oplus (P_{4\setminus 3} \otimes 0) = a$$

Observe that $P_{4\setminus 3}$, considered as a procedure, lies outside of the call-graph strongly-connected-component \{P_3, P_4\}, because it calls neither $P_3$ nor $P_4$. Therefore, $P_{4\setminus 3}$ can be analyzed using the algorithms of this chapter to produce a summary, and we can use that summary when we return to the analysis of \{P_3, P_4\}. Subsequently, when we analyze \{P_3, P_4\}, we find a base case for $P_3$ corresponding to the path $P_{4\setminus 3} \otimes P_{4\setminus 3}$, which corresponds to the action of adding two to cost.

Each time we apply the above transformation, we create $m - 1$ new procedures $P_{j\setminus i}$ for $j \in \{1, \ldots, i - 1, i + 1, \ldots, m\}$. For some equation systems, we must apply this transformation for several such $i$. In the worst case, the transformation can lead to a worst-case increase of $O(2^m)$ in the number of variables in the equation system.

### 5.5 Experiments

Our techniques are implemented as an interprocedural extension of Compositional Recurrence Analysis (CRA) [Farzan and Kincaid, 2015], resulting in a tool we call Compositional Higher-Order Recurrence Analysis (CHORA).

CRA is a program-analysis tool that uses recurrences to summarize loops, and uses Kleene iteration to summarize recursive procedures. We compare CHORA against the ICRA analysis tool, which is the implementation of the techniques of Chapter 3 and Chapter 4. Recall that ICRA uses the technique of NPA-TP-GJ (see §3.3.3) in case of non-linear recursion. CHORA can analyze programs containing arbitrary combinations of loops and branches using CRA. In the case of linear recursion, CHORA uses the same reduction to CRA as ICRA. Thus, in those cases, CHORA will produce results almost identical to those of ICRA. The algorithms of §5.4, which allow CHORA to perform a precise analysis of non-linear recursion, are what distinguish CHORA from prior work. For this reason, our experiments are focused on the analysis of non-linearly recursive programs.
Our experimental evaluation is designed to answer the following question:

Is CHORA effective at generating invariants for programs containing non-linear recursion?

Despite the prominence of non-linear recursion (e.g., divide-and-conquer algorithms), there are few benchmarks in the verification literature that make use of it. The examples that we found are bounds-generation benchmarks that come from the complexity-analysis literature, as well as assertion-checking benchmarks from the recursive subcategory of SV-COMP.

Generating complexity bounds. For our first set of experiments, we evaluate CHORA on twelve benchmark programs from the complexity-analysis literature. This set of experiments is designed to determine how the complexity-analysis results obtained by CHORA compare with those obtained by ICRA and state-of-the-art complexity-analysis tools. We selected all of the non-linearly recursive programs in the benchmark suites from a recent set of complexity-analysis papers [Chatterjee et al., 2019; Carbonneaux et al., 2015a; Kahn and Hoffmann, 2019], as well as the web site of PUBS [Albert et al., 2019], and removed duplicate (or near-duplicate) programs, and translated them to C. Our implementations of divide-and-conquer algorithms are working implementations rather than cost models, and therefore CHORA’s analysis of these programs involves performing non-trivial invariant generation and cost analysis at the same time. Source code for CHORA and all benchmarks can be found in the CHORA repository [Breck et al., 2020a].

To perform a complexity analysis of a program using CHORA, we first manually modify the program to add an explicit variable (cost) that tracks the time (or some other resource) used by the program. We then use CHORA to generate a term that bounds the final value of cost as a function of the program’s inputs. Note that, as a consequence of this technique, CHORA’s bounds on a program’s running time are only sound under the assumption that the program terminates. Throughout the analysis, CHORA merely treats cost as another program variable; that is, the recurrence-based analytical techniques that it uses to perform cost analysis are the same as those it uses to find all other numerical invariants.

The benchmark programs on which we evaluated CHORA, as well as the complexity bounds obtained by CHORA’s analysis, are shown in Tab. 5.1. The first five programs are elementary examples of non-linear recursion. The next seven are more challenging complexity-analysis problems that have been used to test the limits of state-of-the-art complexity analyzers.

We observe that on two benchmarks, karatsuba and strassen, CHORA finds an asymptotically tight bound that was not found by the technique from which the benchmark was taken. For example, the bound obtained by CHORA for karatsuba has the form cost \( \leq 3 \log_2(n) \) which is equivalent to cost \( \leq n^{\log_2(3)} \), and is therefore tighter than the bound using the rational exponent 1.6 cited in [Chatterjee et al., 2019], although the technique from
Table 5.1: Column 2 shows the actual asymptotic bound for each benchmark program. Columns 3-4 show the asymptotic complexity of the bounds determined by CHORA and ICRA. Column 5 gives the source of the benchmark as well as the published bound from that source. “n.b.” indicates that no bound was found. For each benchmark, only one other tool’s bound is shown, even if more than one such tool is capable of finding a bound.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Actual</th>
<th>CHORA</th>
<th>ICRA</th>
<th>Other Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibonacci</td>
<td>$O(\phi^n)$</td>
<td>$O(2^n)$</td>
<td>n.b.</td>
<td>[Albert et al., 2019]: $O(2^n)$</td>
</tr>
<tr>
<td>hanoi</td>
<td>$O(2^n)$</td>
<td>$O(2^n)$</td>
<td>n.b.</td>
<td>[Albert et al., 2019]: $O(2^n)$</td>
</tr>
<tr>
<td>subset_sum</td>
<td>$O(2^n)$</td>
<td>$O(2^n)$</td>
<td>n.b.</td>
<td>[Kahn and Hoffmann, 2019]: $O(2^n)$</td>
</tr>
<tr>
<td>bst_copy</td>
<td>$O(2^n)$</td>
<td>$O(2^n)$</td>
<td>n.b.</td>
<td>[Albert et al., 2019]: $O(2^n)$</td>
</tr>
<tr>
<td>ball_bins3</td>
<td>$O(3^n)$</td>
<td>$O(3^n)$</td>
<td>n.b.</td>
<td>[Kahn and Hoffmann, 2019]: $O(3^n)$</td>
</tr>
<tr>
<td>karatsuba</td>
<td>$O(n^{\log_2(3)})$</td>
<td>$O(n^{\log_2(3)})$</td>
<td>n.b.</td>
<td>[Chatterjee et al., 2019]: $O(n^{1.6})$</td>
</tr>
<tr>
<td>mergesort</td>
<td>$O(n \log(n))$</td>
<td>$O(n \log(n))$</td>
<td>n.b.</td>
<td>[Albert et al., 2019]: $O(n \log(n))$</td>
</tr>
<tr>
<td>strassen</td>
<td>$O(n^{\log_2(7)})$</td>
<td>$O(n^{\log_2(7)})$</td>
<td>n.b.</td>
<td>[Chatterjee et al., 2019]: $O(n^{2.9})$</td>
</tr>
<tr>
<td>qsort_calls</td>
<td>$O(n)$</td>
<td>$O(2^n)$</td>
<td>$O(n)$</td>
<td>[Carbonneaux et al., 2015a]: $O(n)$</td>
</tr>
<tr>
<td>qsort_steps</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>n.b.</td>
<td>[Chatterjee et al., 2019]: $O(n^{2.9})$</td>
</tr>
<tr>
<td>closest_pair</td>
<td>$O(n \log(n))$</td>
<td>n.b.</td>
<td>n.b.</td>
<td>[Chatterjee et al., 2019]: $O(n \log(n))$</td>
</tr>
<tr>
<td>ackermann</td>
<td>$\text{Ack}(n)$</td>
<td>n.b.</td>
<td>n.b.</td>
<td>[Albert et al., 2019]: n.b.</td>
</tr>
</tbody>
</table>

Figure 5.3: Results of running CHORA and four other tools on the SV-COMP19 recursive directory of benchmarks. Each point indicates a benchmark containing assertions that a tool proved to be true, and the amount of time taken by that tool on that benchmark.

[Chatterjee et al., 2019] can obtain rational bounds that are arbitrarily close to $\log_2(3)$. On two benchmarks, CHORA fails to produce an asymptotically tight bound. For example, for qsort_steps, cost tracks the number of instructions, CHORA finds an exponential bound (as does the PUBS complexity analyzer [Albert et al., 2019], which also uses recurrence solving and height-based abstraction), whereas [Chatterjee et al., 2019] finds the optimal $O(n^2)$ bound. On two more benchmarks, CHORA is unable to find a bound. Note that CHORA’s technique for summarizing recursive functions significantly improves upon ICRA’s, which can find only one bound across the suite.

Assertion-checking experiments. Next, we tested CHORA’s invariant-generation abilities on assertion-checking benchmarks. A standard benchmark suite from the literature is the Software Verification Competition (SV-COMP), which includes a recursive sub-category (ReachSafety-Recursive). Within this sub-category, we
```c
int ackermann(int m, int n) {
    if (m == 0) return n + 1;
    if (n == 0) return ackermann(m - 1, 1);
    return ackermann(m - 1, ackermann(m, n - 1));
}

assert(n < 0 || m < 0 || ackermann(m, n) >= 0)

int hanoi(int n) {
    if (n == 1) return 1;
    return 2 * (hanoi(n - 1)) + 1;
}

void applyHanoi(int n, int from, int to, int via) {
    if (n == 0) return;
    counter++;
    applyHanoi(n - 1, from, via, to);
    applyHanoi(n - 1, via, to, from);
}

counter = 0; applyHanoi(n, ...); assert(hanoi(n) == counter)

int f91(int x) {
    if (x > 100) return x - 10; else return f91(f91(x + 11));
}

res = f91(x); assert(res == 91 || x > 101 && res == x - 10)
```

Figure 5.4: Source code for three programs from the SV-COMP suite: Ackermann01, RecHanoi01, and McCarthy91

selected the benchmarks in the recursive sub-directory that contained true assertions, yielding a set of 17 benchmarks. We ran CHORA, ICRA, and the top three performers on this category from the 2019 competition: Ultimate Automizer (UA) [Heizmann et al., 2013], UTaipan [Dietsch et al., 2018], and VIAP [Rajkhowa and Lin, 2017]. Fig. 5.3 presents a cactus plot showing the number of benchmarks proved by each tool, as well as the timing characteristics of their runs.

Timings were taken on a virtual machine running Ubuntu 18.04 with 16 GB of RAM, on a host machine with 32GB of RAM and a 3.7 GHz Intel i7-8000K CPU. These results demonstrate that CHORA is roughly an order of magnitude faster for each benchmark than the other tools. UA proved the assertions in 12 out of 17 benchmarks; UTaipan and VIAP each proved the assertions in 10 benchmarks; CHORA proved the assertions in 8 benchmarks; all other tools from the competition proved the assertions in 6 or fewer benchmarks.

While the SV-COMP benchmarks do give some insight into CHORA’s invariant-generation capability, the recursive suite is not an ideal test of that capability, because the suite contains many benchmarks that can be proved safe by unrolling (e.g., verifying that Ackermann’s function evaluated at (2,2) is equal to 7). That is, many of these benchmarks do not actually require an analyzer to perform invariant generation.

We now discuss three benchmarks from the SV-COMP suite that do give some insight into CHORA’s capabilities, in that they are non-linearly recursive benchmarks that require an analyzer to perform invariant-generation. The Ackermann01 benchmark contains an implementation of the two-argument Ackermann function, and the benchmark asserts that the return value of Ackermann is non-negative if its arguments are non-negative; CHORA is able to prove that this assertion holds. The RecHanoi01 benchmark contains a
int quad(int m) {
    if (m == 0) { return 0; }
    int retval;
    do { retval = quad(m - 1) + m } while(*);
    return retval;
}
assert(quad(n) * 2 == n + n * n)

int pow2_overflow(int p) {
    // pow2_overflow is called with 0 ≤ p ≤ 29
    if (p == 0) { return 1; }
    int r1 = pow2_overflow(p - 1);
    int r2 = pow2_overflow(p - 1);
    assert(r1 + r2 < 1073741824);
    return r1 + r2;
}

int height(int size) {
    if (size == 0) { return 0; }
    int left_size = nondet(0, size); // 0 ≤ left_size < size
    int right_size = size - left_size - 1;
    int left_height = height(left_size);
    int right_height = height(right_size);
    return 1 + max(left_height, right_height);
}
assert(height(n) ≤ n)

Figure 5.5: Source code for three non-linearly recursive programs containing assertions.

non-linearly recursive cost-model of the Tower of Hanoi problem, along with a linearly recursive function that doubles its return value and adds one at each recursive call. The assertion in recHanoi01 states that these two functions compute the same value, and CHORA is able to prove this assertion. (The other tools that we tested, namely ICRA, UA, UTaipain, and VIAP, were not able to prove this assertion.) The McCarthy91 benchmark contains an implementation of McCarthy’s 91 function, along with an assertion that the return value of that function, when applied to an argument x, either (1) equals 91, or else (2) equals x − 10. CHORA is not well-suited to prove this assertion because the asserted property is a disjunction, i.e., it describes the return value using two cases, whereas the hypothetical summaries used by CHORA do not contain disjunctions. (ICRA, UA, UTaipan, and VIAP were all able to prove this assertion.)

To further test CHORA’s capabilities, we also manually created three new assertion-checking benchmarks, shown in Fig. 5.5. Because our goal is to assess CHORA’s ability to synthesize invariants, our additional suite consists of recursive examples for which unrolling is an impractical strategy.

quad has a recursive call in a loop that may run for arbitrarily many iterations, and its return value is always n(n + 1)/2. pow2_overflow contains an assertion inside a non-linearly recursive function, and an assumption about the range of parameter values; if the assertion passes, we may conclude that the program is safe from numerical-overflow bugs. The benchmark height asserts that the size (i.e., the number of nodes) of a tree of recursive calls is an upper bound on the height of the tree of recursive calls.
Table 5.2: Five analysis tools, along with the results of assertion-checking experiments using the benchmarks shown in Fig. 5.5. A ✓ indicates that the tool was able to prove the assertion within 900 seconds, and an X indicates that it was not. We also show the time required to analyze each benchmark.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CHORA</th>
<th>ICRA</th>
<th>UA</th>
<th>UTaipan</th>
<th>VIAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>quad</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pow2_overflow</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>height</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The results of our experiments are shown in Tab. 5.2. CHORA is able to prove the assertions in all three programs; ICRA and UTaipan each prove two; UA proves one, and VIAP proves none. Times taken by each tool are also shown in the table. CHORA’s ability to prove the assertion in quad illustrates that it can find invariants even for programs in which running time (and the number of recursive calls) is unbounded. quad illustrates CHORA’s applicability to perform program-equivalence tasks on numerical programs, while pow2_overflow illustrates CHORA’s applicability to perform overflow-checking.

Conclusions. Our main experimental question is whether CHORA is effective at the problem of generating invariants for programs using non-linear recursion. Results from the complexity-analysis and assertion-checking experiment show that CHORA is able to generate non-linear invariants that are sufficient to solve these kinds of problems. In these ways, CHORA has shown success in a domain, i.e., invariant generation for non-linearly recursive programs, that is not addressed by many other tools.

5.6 Related Work

Following the seminal work of Cousot and Cousot [Cousot and Cousot, 1977], most invariant-generation techniques are based on iterative fixpoint computation, which over-approximates Kleene-iteration within some abstract domain. This chapter presents a non-iterative method for generating numerical invariants for recursive procedures, which is based on extracting and solving recurrence relations. It was inspired by two streams of ideas found in prior work.

Template-based methods fix a desired template for the invariants in a program, in which there are undetermined constant symbols [Colón et al., 2003; Sankaranarayanan et al., 2004a]. Constraints on the constants are derived from the structure of the program, which are given to a constraint solver to derive values for the constants. The hypothetical summaries introduced in §5.4.1 were inspired by template-based methods, but go beyond them in an important way: in particular, the indeterminates in a hypothetical summary are functions rather than constants, and our work uses recurrence solving to synthesize these functions.

Of particular relevance to our work are template-based methods for generating non-linear invariants [Kapur, 2004; Sankaranarayanan et al., 2004b; Cachera et al., 2012; Kojima et al., 2016; Chatterjee et al., 2019].
Contrasting with the technique proposed in this chapter, a distinct advantage of template-based methods for generating polynomial invariants for programs with real-typed variables is that they enjoy completeness guarantees [Kapur, 2004; Sankaranarayanan et al., 2004b; Chatterjee et al., 2019], owing to the decidability of the theory of the reals. The advantages of our proposed technique over traditional template-based techniques are (1) it is compositional, (2) it can generate exponential and logarithmic invariants, and (3) it does not require fixing bounds on polynomial degrees a priori. Also note that template-based techniques pay an up-front cost for instantiating templates that is exponential in the degree bound. (In practice, this exponential blow-up can be mitigated [Kojima et al., 2016].)

**Recurrence-based methods** find loop invariants by extracting recurrence relations between the pre-state and post-state of the loop and then generating invariants from their closed forms [Farzan and Kincaid, 2015; Kincaid et al., 2018, 2019; Rodríguez-Carbonell and Kapur, 2004; Humenberger et al., 2017; Kovács, 2008; de Oliveira et al., 2016; Humenberger et al., 2018]. This chapter gives an answer to the question of how such analyses can be applied to recursive procedures rather than loops, by extracting height-indexed recurrences using template-based techniques.

Reps et al. [2017] demonstrate that tensor products can be used to apply loop analyses to linearly recursive procedures. This technique is used in the recurrence-based invariant generator ICRA to handle linear recursion [Kincaid et al., 2017]. ICRA falls back on a fixpoint procedure for non-linear recursion; in contrast, the technique presented in this chapter uses recurrence solving to analyze recursive procedures.

Rajkhowa and Lin [2017] presents a verification technique that analyzes recursive procedures by encoding them into first-order logic; recurrences are extracted and replaced with closed forms as a simplification step before passing the query to a theorem prover. In contrast to this chapter, Rajkhowa and Lin [2017]'s approach has the flexibility to use other approaches (e.g., induction) when recurrence-based simplification fails, but cannot be used for general-purpose invariant generation.

**Resource-bound analysis** [Wegbreit, 1975] is another related area of research. Three lines of recent research in resource-bound analysis are represented by the tools PUBS [Albert et al., 2011], CoFloCo [Flores Montoya, 2017], KoAT [Brockschmidt et al., 2016], and RAML [Hoffmann et al., 2012b]. In resource-bound analysis, the goal is to find an expression that upper-bounds or lower-bounds the amount of some resource (e.g., time, memory, etc.) used by a program. Resource-bound analysis typically consists of two parts: (i) size analysis, which finds invariants that bound program variables, and (ii) cost analysis, which finds bounds on cost using the results of the size analysis. Cost can be seen as an auxiliary program variable, although it is updated in a restricted manner (by addition only), it has no effect on control flow, and it is often assumed to be non-negative. Our work differs from resource-bound analyzers in several ways, ultimately because our
goal is to find invariants and check assertions, rather than to find resource bounds specifically.

The capabilities of our technique are different, in that we are able to find non-linear mathematical relationships (including polynomials, exponentials, and logarithms) between variables, even in non-linearly recursive procedures. PUBS and CoFloCo use polyhedra to represent invariants, so they are restricted to finding linear relationships between variables, although they can prove that programs have non-linear costs. KoAT has the ability to find non-linear (polynomial and exponential) bounds on the values of variables, but it has limited support for analyzing non-linearly recursive functions; in particular, KoAT cannot reason about the transformation of program state performed by a call to a non-linearly recursive function. Typically, resource-bound analyzers also reason about non-terminating executions of a program, whereas our analysis does not. RAML reasons about manipulations of data structures, whereas our work only reasons about integer variables. Originally, RAML only discovered polynomial bounds, although recent work [Kahn and Hoffmann, 2019] extends the technique to find exponential bounds.

The algorithms that we use are different in that we have a unified approach, rather than separate approaches, for analyzing cost and analyzing a program’s transformation of other variables. To perform resource-bound analysis, we materialize cost as a program variable and then find a procedure summary; the summary describes the program’s transformation of all variables, including the cost variable. Recurrence-solving is the essential tool that we use for analyzing loops, linear recursion, and non-linear recursion, and we are able to find non-linear mathematical relationships because such relationships arise in the solutions of recurrences.
Chapter 6

Further Enhancements of CRA

In this chapter, I first summarize two papers that further improve the precision of CRA. Then, I consider some avenues for future research.

Because my level of involvement in the two papers described in this section was less than with the projects described in Chapter 3, Chapter 4, and Chapter 5, I have chosen to summarize these papers rather than including large portions of their text. In §6.1, I summarize [Cyphert et al., 2019], which improves the precision of CRA when analyzing programs that have loops in which there are multiple paths through the loop body. In §6.2, I summarize [Kincaid et al., 2019], which improves the precision of CRA when working with a certain class of matrix recurrences by finding alternative closed-form solutions to those recurrences that can be handled more precisely by subsequent steps of the analysis.

The remaining sections of this chapter describe some ideas for future work, which are primarily my own work. In §6.3, I discuss some possible approaches for applying CRA to the problem of refuting, rather than proving, program properties. In §6.4, I discuss the application of CRA to constrained-Horn-clause programs. In §6.5, I discuss future directions for research that build upon the techniques of Chapter 5.

6.1 Control-flow Refinement

The analysis techniques of the previous chapters summarize a loop by first summarizing the body of the loop, then extracting a recurrence from the summary of the body, and then solving the recurrence. Although the loop-body summary sometimes contains disjunctions (e.g., because there are multiple paths through the loop body, and the loop-body summary is the disjunction of the summaries of those paths), the extracted recurrences cannot contain disjunctions, and for that reason, a significant loss of precision sometimes occurs at the recurrence-extraction step. In terms of algebraic program analysis, the problem occurs when a “∗” operator (representing a loop) is applied to a sub-expression that uses the “⊕” operator (which is used to
combine multiple paths.)

In [Cyphert et al., 2019], we developed a way of refining the regular expression for a multi-path loop to produce a new regular expressions for use in subsequent analysis steps. The key feature of the refinement process is that the precision of the subsequent analysis when using the refined expression is guaranteed not to be worse, and may be better, than that of the original analysis. To obtain this precision guarantee, it is not sufficient to choose an arbitrary regular expression having the same language as (or a strictly smaller language than) the original regular expression. Instead, we prove the precision guarantee by formulating the concept of a pre-Kleene algebra, which is an abstract domain that satisfies a particular set of axioms.

The analysis algorithm works by computing a refinement graph, in which the vertices represent paths through a loop, and there is an edge between two vertices A and B if, on successive iterations of the loop, an execution of the path A may be followed by an execution of the path B. After constructing the refinement graph, the algorithm analyzes the graph to produce the refined regular expression. For example, if it is not feasible for path A to follow path B, then a loop that was originally described by the regular expression \((A \oplus B)^*\) can be described by the refined expression \(A^*B^*\), and the program-analysis results obtained by using the refined expression will often be more precise than the results prior to refinement. Continuing the same example, if it is also true that path B cannot follow path A, then another regular expression \((A^*) \oplus (B^*)\) can be used, and the results obtained by using this regular expression may be even more precise. Both of the expressions just mentioned can be produced by analysis of a refinement graph.

In [Cyphert et al., 2019], we asked the question of what makes one regular expression better than another for the purposes of algebraic program analysis. In other words, we asked how we can predict whether the final results of a program analysis will be more precise (or not) if one regular expression is analyzed rather than another. We described formalisms for representing this question, and gave an algorithm that produces refined regular expressions in a way that allows for guarantees about the precision of analysis results.

To evaluate the technique of [Cyphert et al., 2019], we performed experiments that compared the precision of ICRA's results—with and without refinement of regular expressions—on a collection of assertion-checking micro-benchmark tasks. Using refined regular expressions allowed ICRA to prove approximately 25% more assertions, at the cost of an increase in analysis time of roughly 50%. We concluded that refinement of regular expressions is an important tool for improving the precision of an algebraic program analysis.

### 6.2 Periodic-Rational Spectral Decomposition

CRA analyzes loops by finding recurrences that describe the state transformation performed by the loop body. When there are interdependencies among a program’s variables, the techniques of Chapter 4 may be
used to describe those interdependencies using a matrix recurrence of the form $x^{[k+1]} = Mx^k + b$. When classical methods are used to solve that recurrence, the closed-form solution will involve exponentiation operations in which the eigenvalues of the matrix $M$ appear as the bases of exponentiation. If the eigenvalues are not rational numbers, then CRA cannot incorporate that solution into a transition formula, because the transition formulas used by CRA are required to lie within a logic fragment that contains only rational numbers. However, in some cases, there is an alternative way of expressing the recurrence solution that only uses rational numbers.

$$\textbf{while} \ (+) \ \textbf{do}$$

\begin{align*}
 z_{\text{old}} &= z; \\
 z &= y; \\
 y &= x; \\
 x &= z_{\text{old}}; \\
\end{align*}

Figure 6.1: Loop with periodic behavior, and its associated matrix recurrence.

Consider the program, and associated matrix recurrence, shown in Fig. 6.1. Two of the eigenvalues of the matrix are third roots of unity, which lie outside of the rational numbers. An alternative summary is:

$$\exists K \in \mathbb{Z}. [n = 3K + 0 \land x' = x \land y' = y \land z' = z]$$

$$\lor [n = 3K + 1 \land x' = z \land y' = x \land z' = y]$$

$$\lor [n = 3K + 2 \land x' = y \land y' = z \land z' = x].$$

Eqn. (6.1) uses only rational numbers, and therefore it can be directly incorporated into a CRA transition formula. In [Kincaid et al., 2019], we addressed several problems, including the problem of how to find alternative summaries like Eqn. (6.1) when summarizing loops and recursive procedures. Such summaries can be constructed when each of the eigenvalues of the transition matrix is the $n^{\text{th}}$ root of a rational number for some $n \in \mathbb{N}$. We define the term periodic rational to refer to the set of $n^{\text{th}}$ roots of rational numbers. We describe a decomposition of a matrix that we call the periodic-rational spectral decomposition, which can be used to produce an efficient closed-form solution of a matrix recurrence by identifying the periodic-rational eigenvalues of the associated matrix. The paper also contains several related results, such as a proof of the decidability of the logic fragment in which such closed-form solutions are expressed.

6.3 Property Refutation

Throughout this document, we have described program-analysis techniques that can be used to prove that a property holds in all executions of a program. An important goal for future research is to generalize these
techniques so that they can be used to refute, rather than prove, such program properties. The main challenge of applying these techniques to this goal is that the techniques all work with over-approximations of the transition relation of a program, whereas property refutation requires some form of under-approximation of the transition relation, because it must definitively establish that a property-violating state is reachable.

The simplest way to apply CRA to the problem of property refutation is to run an unmodified CRA-based analysis tool (such as ICRA or CHORA) on a program to obtain an error summary, which over-approximates the transition relation between the entry location of the program and an error location that is reachable from the assertion location when the asserted property does not hold. Then, an SMT solver can be used to find a model of the error summary, which consists of an assignment of values to the free variables of the error summary. In particular, the model will assign some tuple of values $\mathbf{x}$ to the parameters of the main procedure of the program. Then, the program can be instrumented to make an assertion failure detectable, and executed with $\mathbf{x}$ as input. If an assertion failure is detected, then the asserted program property is refuted.

In this approach, the program itself serves as the under-approximation of the program’s behavior. This simple approach is an interesting first step towards to property refutation via CRA, and it might yield useful results in some cases, because CRA’s over-approximate summaries are relatively precise compared with the summaries and invariants produced by other techniques. However, this approach has two main problems.

The first problem is that the summary produced by CRA is an over-approximation, which means that $\mathbf{x}$ does not necessarily correspond to an input that can actually lead to an assertion violation. The second problem is that the program itself may be non-deterministic, or it may take many—perhaps unboundedly many—additional inputs (e.g., because it reads from an input stream) during its execution, in which case the initial inputs $\mathbf{x}$ do not determine the final assertion-violating state of the program. For both of those reasons, running the program with $\mathbf{x}$ as input may not yield an assertion violation even if the assertion-violating state is reachable. Moreover, after such an indeterminate result is observed, it is not clear what the analyzer should do next to pursue the goal of refuting the asserted program property.

6.3.1 Unrolling

A more sophisticated strategy for handling property refutation via CRA would exploit the counter variable that occurs in CRA’s summaries of loops. Such summaries have the form $\exists K \geq 0. \psi(K)$, where $K$ represents a loop counter. If the formula is modified so as to make $K$ a free variable, then a model of $\psi(K)$ will include a particular value $k$, which represents a possible number of times that the loop can be unrolled. That is, an analyzer can use the value $k$ to produce an unrolled version of the loop summary that consists of $k$ sequentially composed copies of the loop body summary. This unrolled summary will often be a more precise
representation of the loop’s behavior for the specific number $k$ of iterations.

When a loop summary $\exists K \geq 0.\psi(K)$ occurs as a sub-formula of an error summary $\varphi$, an unrolled version of the error summary can be obtained and then substituted back in place of $\exists K \geq 0.\psi(K)$ within $\varphi$ to obtain a more precise error summary. This improvement in precision may help to address the problems noted in the previous sub-section. This unrolling strategy may be applied iteratively to nested loops by first unrolling the outermost level of the loop nest, and then unrolling the next-outermost level of the loop nest, and so on. However, because this unrolling strategy chooses $k$ (the number of times to unroll a loop) by inspecting an over-approximate summary $\varphi$, the strategy will sometimes fail because it makes a bad choice of $k$. That is, the strategy may choose a $k$ for which the error is not actually reachable. When the strategy makes a bad choice of $k$, it is not obvious what to do next.

### 6.3.2 Under-Approximate Summaries

An alternative approach to property refutation is to create a new version of CRA that works with under-approximate, rather than over-approximate, summaries. This approach would involve creating under-approximating analogues of the operations that CRA uses to analyze loops and recursive procedures (e.g., an under-approximating interpretation of a Kleene-star). Currently, however, CRA-based analysis tools (ICRA and CHORA) work by transforming C programs into control-flow graphs in which edges are weighted by over-approximate transition formulas. That is, over-approximation occurs during the first step of the analysis performed by such tools, in the transformation of C code fragments into transition formulas.

It would be challenging to replace these over-approximate transition formulas with under-approximate or exact transition formulas, because the formulas would have to contend with various complex features of C’s execution model, such as general pointer-based data structures. One way of dealing with this difficulty would be to integrate a CRA-based analysis tool with a concolic-execution engine that could serve as an under-approximate representation of the execution of a C program.

An alternative approach, which might be a good way to begin exploring property refutation with CRA, would be to change the problem domain. Instead of analyzing C programs, which have relatively complex semantics, we could analyze constrained Horn clauses, which have much simpler semantics. Perhaps after discovering which approaches to property refutation are successful in the domain of CHCs, we could return to the analysis of C programs.
6.4 Horn Clauses

In this section, we describe how to adapt the techniques of CRA to the analysis of constrained Horn clauses. The input to this analysis is a collection of constrained Horn clauses, which we call a CHC program. We present an algorithm that translates a CHC program into a control-flow graph with edges weighted by transition formulas, which may then be analyzed using CRA, along with the various enhancements presented in this document.

We begin by explaining the structure of CHC programs and an analogy between C programs and CHC programs. Each clause in a CHC program is either a fact, a rule, or a query constraint. Clauses consist of occurrences of one or more predicates $P_i$, along with (explicit) constraint formulas $\varphi_i$ in some SMT logic. A fact is a constrained Horn clause having no predicate occurrences in its body; the general form of a fact is:

$$P_i(V') \leftarrow \varphi_i(V').$$

A rule is a constrained Horn clause having one or more predicate occurrences in its body; the general form of a rule is as follows:

$$P_i(V') \leftarrow P_{j_1}(V_1) \land \cdots \land P_{j_k}(V_k) \land \varphi(V', V_1, \ldots, V_k).$$

Predicates in a CHC program are analogous to procedures in a C program. The appearance of each predicate $P_j$ in the body of the rule is the analogue of a call, with “caller” $P_i$ and “callee” $P_j$. Recursive rules within a CHC program may be analyzed in a way that is analogous to the analysis of recursive procedures.

Constrained Horn clauses have an explicit constraint formula as part of each fact or rule. However, they can also have implicit constraints arising from sharing of variables between various arguments in predicate occurrences. The following clauses illustrate implicit constraints:

$$P_1(x, x) \leftarrow (x \geq 0 \land x \leq 5)$$
$$P_2(w, x, y) \leftarrow P_2(x, w, z)$$
$$P_3(x + 1, x + 2) \leftarrow P_3(x - 10, y + y) \land x = y + 5.$$  
$$P_4(x + x + x) \leftarrow P_4(x)$$

Clauses that have implicit constraints can always be replaced by equivalent clauses without implicit constraints, by making the implicit constraints explicit. To perform that transformation, we identify each predicate argument that is a non-trivial term (e.g., $x + 1$) rather than a single variable occurrence (e.g., $x$), and each
predicate argument that occurs more than once (possibly as a subterm) among the predicate arguments of
the clause. We replace each such predicate argument \( t \) with a fresh variable \( v \), and conjoin the equation \( v = t \)
onto the clause’s explicit constraint. For the clauses shown above, the result of this transformation is:

\[
\begin{align*}
P_1(a, b) &\leftarrow (a = x \land b = x \land x \geq 0 \land x \leq 5) \\
P_2(a, b, y) &\leftarrow P_2(c, d, z) \land a = d \land b = c \\
P_3(a, b) &\leftarrow P_3(c, d) \land a = x + 1 \land b = x + 2 \land c = x - 10 \land d = y + y \land x = y + 5. \\
P_4(a) &\leftarrow P_4(b) \land a = x + x + x \land b = x. 
\end{align*}
\]

For the purposes of this document, I assume that constrained Horn clauses have been pre-processed in this
way to make implicit constraints explicit.

Another useful pre-processing step translates any Boolean variables that occur in a CHC program into
corresponding integer-valued variables that are constrained to equal zero or one wherever they appear. This
step is required because CRA, which is based on finding and solving numerical recurrences, only recognizes
numeric variables. (It would be useful to rigorously investigate whether this approach to handling Boolean
variables is good enough, or whether something more precise and efficient is needed.)

We now sketch an algorithm for translating a CHC program into a control-flow graph weighted with
transition formulas. (This algorithm has been implemented within the CHORA analysis tool.) We wish to
translate each predicate occurrence within a rule body into a procedure call. For this purpose, we create a set
of global variables \( g_1, ..., g_M \), where \( M \) is the maximum number of arguments taken by any predicate in the
CHC program. These global variables may be thought of as holding the “return values” of a call. That is, we
will translate each predicate \( P_i(v_1, ..., v_n) \) into a procedure that leaves the values of some tuple \( (v_1, ..., v_n) \) in
the globals \( (g_1, ..., g_n) \) after it executes. For each predicate \( P_i \), we create an entry vertex \( e_i \) and an exit vertex
\( x_i \). Each fact has the following form:

\[
P_i(v_1, ..., v_n) \leftarrow \psi_i(v_1, ..., v_n).
\]

For each such fact, we create an edge from entry to exit (i.e., from \( e_i \) to \( x_i \)) that is weighted with a transition
formula that assumes the constraint \( \psi_i(v_1, ..., v_n) \) and assigns the variables \( v_1, ..., v_n \) to the globals \( g_1, ..., g_n \).

Each rule has the form:

\[
P_i(V') \leftarrow P_j(V_1) \land \cdots \land P_k(V_k) \land \varphi(V', V_1, ..., V_k),
\]
in which the capital letters \( V', V_1, ..., V_k \) represent tuples of variables. To translate each such rule, we create a path starting at \( e_i \), ending at \( x_i \), and consisting of \((2k + 1)\) edges. For each predicate occurrence \( P_{j_m}(V_m) \), we generate a call edge that calls the procedure associated with \( P_{j_m} \), followed by an edge that stores the values of the global variables into a collection of local variables \( V_m \). The final edge of the path is weighted with a transition formula that assumes the constraint \( \varphi(V', V_1, ..., V_k) \) and stores the values of \( V' \) into the global return values \( g_1, ..., g_m \).

After all CHCs have been translated into weighted control-flow graphs, the techniques of (enhanced) CRA can be applied to obtain summaries for all procedures. The CHC program will typically also contain a query clause, which has the form:

\[
\bot \leftarrow P_{j_1}(V_1) \land \cdots \land P_{j_k}(V_k) \land \varphi(V', V_1, ..., V_k).
\]

Predicate occurrences in the body of this query clause can be replaced by their corresponding predicate summaries. If, after that substitution, the body is unsatisfiable, then the analysis has demonstrated that the CHC program as a whole is satisfiable (because, in that case, the head of the query clause, \( \bot \), is not asserted). In this way, the methods of CRA can be used to check the satisfiability of CHC programs.

We now illustrate the transformation of a CHC program into a weighted control-flow graph using the following example.

**Example 6.1.** Consider the following CHC program.

\[
\begin{align*}
add(0, \text{sum}, \text{sum}) & \leftarrow \text{true} \\
add(x + 1, y, \text{sum}) & \leftarrow add(x, y + 1, \text{sum}) \\
mul(0, b, 0) & \leftarrow \text{true} \\
mul(a + 1, b, \text{prod'}) & \leftarrow mul(a, b, \text{prod}) \land add(b, \text{prod}, \text{prod'})
\end{align*}
\]

For simplicity, all variables in this example range over the natural numbers. There are two predicates, each having one fact and one rule. The predicate \( add(x', y', \text{sum'}) \) holds when \( \text{sum'} = x' + y' \). The predicate \( mul(a', b', \text{prod'}) \) holds when \( \text{prod'} = a' \cdot b' \). Each CHC in the above program exhibits implicit constraints. The first step of the translation of
the above program is to make the implicit constraints explicit, which produces the following program:

\[
\begin{align*}
\text{add}(x', y', sum') & \leftarrow x' = 0 \land y' = sum' \\
\text{add}(x', y', sum') & \leftarrow \text{add}(x, y, sum) \land x' = x + 1 \land y' + 1 = y \land sum' = sum \\
\text{mul}(a', b', prod') & \leftarrow a' = 0 \land prod' = 0 \\
\text{mul}(a', b', prod') & \leftarrow \text{mul}(a, b, prod) \land \text{add}(d, e, f) \land a' = a + 1 \land b' = b \land d = b \land e = prod \land f = prod'
\end{align*}
\]

Finally, following the method outlined above, we create two procedures, and we create one path for each CHC. The resulting weighted control-flow graph is shown in Fig. 6.2.

**Figure 6.2:** The CHC program of Ex. 6.1 after being translated into a weighted control-flow graph. Bold names represent mutable (local or global) variables, and non-bold names represent existentially quantified temporaries. Each edge is weighted with a call or with a statement consisting of an assignment and possibly also an assumed constraint. Notice that each CHC \( c \) becomes one path \( p \) in the graph, and the constraint of \( c \) appears as an assumed constraint on the final edge of \( p \).
6.5 Generalizing the CHORA Analyzer

In the remaining sections of this chapter, we give a sketch of a more abstract explanation of the analysis presented in Chapter 5, as implemented in the CHORA analyzer, as well as some suggestions for future research.

The first sub-sections present the main ideas of the CHORA analyzer, using a higher level of abstraction than the original presentation in Chapter 5. This more abstract view may be useful when developing generalizations of CHORA’s techniques in the future. In §6.5.1, we view CHORA as reducing the analysis of a non-linearly recursive program to the analysis of a simple iterative model program, such as a (height-based) recurrence. In §6.5.2, we discuss CHORA in terms of operators on program-fragment summaries, and the properties of those operators. Although the story is far from complete, these sections make some progress towards the goal of having a purely algebraic account of CHORA, or of some generalization of CHORA.

The later sub-sections give some guidance for future research in this area. We describe the analysis of an iterative model as an “outer” layer of abstraction to contrast it with an inner layer of abstraction, which is used to collect together various fragments of a tree of recursive calls and represent them with an abstract value. In the case of CHORA, all fragments of the recursion tree at a given height are grouped together, but it would be possible to decompose a tree into a different set of fragments, and that is the subject of §6.5.3. In §6.5.4, we discuss some preliminary ideas for alternative abstractions of recursion-tree fragments, i.e., alternatives to CHORA’s hypothetical summaries.

6.5.1 The Outer Layer of Abstraction

In this section, we discuss the view of CHORA as reducing the analysis of a non-linearly recursive program to the analysis of a simple iterative model program, which constitutes an outer layer of abstraction.

In CHORA, the iterative model is a recurrence relation over rationals. The analysis to be performed at the outer layer of abstraction consists of recurrence solving. The sequence that is described by the recurrence is indexed by recursion-tree height; that is, elements of that sequence provide representations of sets of recursion-tree fragments of increasing height. Any terminating recursion tree is represented by some element of the sequence, and therefore, after we find a precise characterization of the sequence, we have a complete analysis of the original non-linearly recursive procedure. In the future, however, we might want to replace CHORA’s use of recurrences with some iterative model that would incorporate arrays or other data types, or some very different abstract domain. Such an iterative model might admit a closure operation that is not based on recurrence solving.

The ultimate goal is to describe the state-transformations performed by program fragments, as in many
other program analyses. A novel feature of CHORA is that it introduces a kind of parameterized state transformer called a hypothetical summary, in which the parameters are the values of bounding functions, which are represented by uninterpreted function symbols in the formulas used by CHORA.

There are various subtleties in the formalization of a general notion of parameters. We assume that we have a set $\mathcal{P}$ of parameter transformers (i.e., parameter-transition relations), and we have a set $\mathcal{S}$ of state transformers (i.e., state-transition relations). Hypothetical summaries reside in a set $\mathcal{SP}$ of parameterized state transformers, from which we can obtain a state transformer if we specify an element of $\mathcal{P}$. (For example, CHORA’s parameterized state transformers are transition formulas that contain occurrences of the function symbols $b_i(h)$; the intent is that such formulas will eventually be conjoined with constraints on the function symbols $b_i(h)$.) During CHORA’s analysis, we substitute hypothetical summaries in place of a procedure’s recursive calls and thereby build up a (parameterized) summary of the function body. We make a distinction between the set $\mathcal{H} \subseteq \mathcal{SP}$, which consists of hypothetical summaries themselves, and the containing space $\mathcal{SP}$, which also includes the parameterized state-transformers that are built up during the aforementioned analysis of a function body in which hypothetical summaries are used.

In CHORA, the parameterized state transformers in $\mathcal{SP}$ can be treated just like ordinary state transformers in $\mathcal{S}$ with the addition of auxiliary variables; these auxiliaries behave like immutable global variables. This approach may or may not work in a generalization of CHORA.

6.5.2 CHORA, by way of Operators and Properties

This section is a rough sketch of how we might generalize CHORA in the style of algebraic program analysis, i.e., by focusing on a collection of operators (about which we know various properties) that may be used to build up a summary of a program from summaries of its parts.

In this chapter, we use the conceptual framework of regular-tree expressions, following [Comon et al., 2008, §2.2]. However, it would also be possible to dispense with the regular-tree expressions and give all of the definitions using equation systems or context-free grammars instead. It is not obvious at this point whether regular-tree expressions are the right formalism. The trees recognized by the expressions that we describe in this section have individual program instructions at their leaves. On the one hand, our ultimate interest is in the concrete semantics of programs, and those semantics seem to be conceptually analogous to the yields of the aforementioned trees. If we are only interested in the yields of those trees, rather than their tree structure, then context-free grammars or equation systems may be a more apt formalism than regular-tree expressions. On the other hand, the tree structure of recursion is semantically important in some ways. For example, the passing of a calling procedure’s local variables between the pre-state and the post-state of a call...
is a feature of the semantics that seems to be a good match for a tree-structured formalism. Also, CHORA’s analysis measures the height of nodes in a tree, and that measurement constitutes an important connection to the tree structure of a recursive computation. Finally, properties of (combinations of) regular-tree languages are generally decidable, while those of yields of trees of (combinations of) regular-tree languages are not always decidable. Although we are not sure whether it is helpful or not, for the purposes of this section of the dissertation, we have tentatively chosen to use the formalism of regular-tree expressions.

Non-linear recursion may be represented using regular-tree expressions containing alternation ($\oplus$), substitution points ($\Box_i$), a substitution operation ($\cdot \Box_i$), and the closure of substitution ($\ast \Box_i$). Sometimes (e.g., in cases of mutual recursion) it is necessary to apply subscripts to the three operations that are related to substitution ($\Box_i$, $\cdot \Box_i$, and $\ast \Box_i$). However, in most of the example expressions that we discuss, only a single subscript would be necessary, and in such cases, we omit the subscripts.

Whereas the usual interpretation of a regular-word expression is a set of words, the usual interpretation of a regular-tree expression is a set of trees. The substitution operator $\cdot \Box$ has the following interpretation in terms of sets of trees: $\epsilon \cdot \Box \zeta$ is the set of trees that can be obtained by taking any tree $t$ in $\epsilon$ and, for each occurrence $s$ of a substitution point ($\Box$) in $t$, substituting some tree in $\zeta$ in place of $s$. (When there are multiple occurrences of substitution points in $t$, $\cdot \Box$ is not constrained to substitute the same tree from $\zeta$ at each occurrence.)

An application $\epsilon^{\ast \Box}$ of the tree-star operator is given its usual interpretation in terms of the substitution operator ($\cdot \Box$) as follows.

$$L_\epsilon(0) = \{\Box\}$$
$$L_\epsilon(n + 1) = L_\epsilon(n) \cup (\epsilon \cdot \Box L_\epsilon(n)) \text{ for } n \geq 1$$
$$\epsilon^{\ast \Box} = \bigcup_{n=0}^{\infty} L_\epsilon(n)$$

We use a language of regular-tree expressions that includes (in addition to the usual regular-tree-expression operators listed above) sequencing ($\otimes$) and sequential iteration ($\ast$) operations, and distinguished elements 0 and 1, with the obvious properties.

Then, instead of describing a recursive procedure using an equation:

$$X = b \oplus (s_1 \otimes X \otimes s_2 \otimes X \otimes s_3),$$
we could use a regular-tree expression, as follows:

\[ X = (s_1 \oplus \Box \oplus s_2 \oplus \Box \oplus s_3)^* \cdot \Box \cdot b. \]

In the following presentation, it will be convenient to write expressions to take advantage of the presence of the annihilator (0) and the combine operator (\( \oplus \)), which allow us to write a procedure’s Kleene iterates in a very simple way, as follows. Let \( \epsilon \) be a regular-tree expression representing the body of a procedure, including both the recursive cases and base cases, with recursive calls represented as substitution points. Thus, in the above example \( \epsilon \) is \( b \oplus (s_1 \oplus \Box \oplus s_2 \oplus \Box \oplus s_3) \). Then we have the following:

\[ \text{\( n \)th Kleene iterate} = \epsilon \cdot \Box \epsilon \cdot \Box \cdots \epsilon \cdot \Box \text{0} \] \tag{6.2}

Note that if \( \epsilon \) includes a base case, then Eqn. (6.2) may include a set of recursion trees having “heights” that range from zero to \( n \). A regular-tree expression \( \epsilon \) satisfying \( \epsilon \cdot \Box \text{0} = \epsilon \) may be called non-recursive (i.e., a base case); the non-recursive part of an expression \( \epsilon \) is \( \epsilon \cdot \Box \text{0} \). An expression \( \epsilon \) is purely recursive if \( \epsilon \cdot \Box \text{0} = 0 \).

Operators

In this sub-section, we describe some operators that are abstracted versions of various analysis-steps that CHORA performs.

Hypothetical Summary Extraction. Hypothetical summaries are obtained by applying the operator \( \mathcal{H}[\cdot] : \text{TreeRegex}[S] \rightarrow \mathbb{H} \), which takes as input the procedure body.

Substitution of Parameters. Given a hypothetical summary, we may substitute some parameter values to obtain a state transformer using the operator \( S_{\cdot}[\cdot] : \mathbb{H} \times \mathbb{P} \rightarrow S \).

Extraction of a Parameter-Transformer. Given a function body and a hypothetical summary, CHORA produces a recurrence, which defines some iterable operation on which we know how to perform a closure (i.e., recurrence solving). A generalization of CHORA would replace this extraction operation with a new operator, \( E_{\cdot}[\cdot] : \mathbb{H} \times \text{TreeRegex}[S] \rightarrow \mathbb{P} \) which takes a hypothetical summary and a function body (regex) and extracts a parameter transformer in \( \mathbb{P} \), which is the generalized version of a recurrence.

The abstract description of CHORA that we present in this sub-section also needs to characterize the initial values of parameters, which are related to the initial conditions of recurrences and the base cases of procedures, as we now describe. Recall that, in the initial condition of every recurrence used by CHORA, every bounding function equals zero; furthermore, whenever CHORA produces a hypothetical summary \( h \) for some procedure \( P \), substituting the value zero for each bounding function in \( h \) yields a state transformer that
over-approximates the base case of $P$. In the abstract account of CHORA that we present in this sub-section, we define $p_0 \in P$ to be a parameter transformer that sets every parameter to an appropriate initial value; thus, in the context of CHORA, $p_0$ may be thought of as an assignment statement that assigns zero to each bounding function.

**Basic Properties**

In this sub-section, we describe some basic properties that we want the operators $S_{(\cdot)[\cdot]}$, $E_{(\cdot)[\cdot]}$, and $\mathcal{H}[\cdot]$ and the value $p_0 \in P$ to have.

The first property says that we can over-approximate the base case of a procedure by extracting a hypothetical summary for that procedure and then substituting $p_0$ into that hypothetical summary.

**Property 6.5.1 (Over-Approximating a Base Case Using a Hypothetical Summary).** For any $\varepsilon \in \text{TreeRegex}[S]$ representing the body of a procedure, we want the following:

$$\varepsilon \cdot \Box 0 \subseteq S_{\gamma(\varepsilon)}[p_0].$$

The following property says that we can over-approximate the effects of the *non-linear* substitution operation ($\cdot \Box$) that happens in tree regular expressions (which is “tree-shaped” because it represents the calls in a non-linearly recursive procedure) using a *linear* operation (i.e., applying an extracted parameter-transformer to some $p \in P$) on which CHORA will later perform a transitive closure using recurrence solving.

**Property 6.5.2 (Linear Approximation of Substitution).** For any $\varepsilon \in \text{TreeRegex}[S]$ and $p \in P$,

$$\varepsilon \cdot \Box S_{\gamma(\varepsilon)}[p] \subseteq S_{\gamma(\varepsilon)}[p \otimes P E_{\gamma(\varepsilon)}[\varepsilon]].$$

Ultimately, we want to use the operations and properties defined in this section to analyze tree-stars, which represent the non-linear recursion used by some procedures. The following property says that we use a (parameter-space) word-star to over-approximate a (state-space) tree-star.

**Property 6.5.3 (Linear Approximation of the Closure of Substitution).** For any $\varepsilon \in \text{TreeRegex}[S]$,

$$\varepsilon^* \Box \subseteq S_{\gamma(\varepsilon)}[p_0 \otimes P (E_{\gamma(\varepsilon)}[\varepsilon]^*])].$$

Viewed abstractly, the summarization algorithm of CHORA takes a non-linearly recursive procedure $P$ with body $\varepsilon$ and evaluates the right-hand side of Prop. 6.5.3 (i.e., $S_{\gamma(\varepsilon)}[p_0 \otimes P E_{\gamma(\varepsilon)}[\varepsilon]^*])$ to obtain an over-approximation of the transition relation of $P$. 
A More Detailed View of The Extraction Operator \(E_{(·)}[·]\)

The formalization given in the preceding sections does not provide any details about how the extraction operator \(E_{(·)}[·]\) works. It may be useful to construct a formalization that makes the details of the extraction process more explicit; doing so might make it clearer to the reader how the \(\text{CHORA}\)-style analysis relates to the underlying intraprocedural analysis. In particular, we may wish to formalize the computation of \(E_{\mathcal{G}(ε)}[ε]\) as consisting of two steps.

1. The first step processes the regular expression \(ε\) using an operator \(L_{(·)}[·]\) that takes a hypothetical summary and a regular expression over un-parameterized state transformers, and produces a parameterized state transformer. In the context of \(\text{CHORA}\), this operator corresponds to a lifting of an un-parameterized transition formula to a transition formula that contains occurrences of the function symbols \(b_1(\cdot h)\), which represent \(\text{CHORA}\)'s bounding functions.

   We define \(L_{(·)}[·] : \mathbb{H} \times \text{TreeRegEx}[S] \rightarrow \mathbb{SP}\) as follows:

   \[
   L_{\mathcal{G}(ε)}[ζ ⊗ η] = L_{\mathcal{G}(ε)}[ζ] ⊗_{SP} L_{\mathcal{G}(ε)}[η]
   \]

   \[
   L_{\mathcal{G}(ε)}[ζ ⊕ η] = L_{\mathcal{G}(ε)}[ζ] ⊕_{SP} L_{\mathcal{G}(ε)}[η]
   \]

   \[
   L_{\mathcal{G}(ε)}[□] = \mathcal{H}[ε]
   \]

   \[
   L_{\mathcal{G}(ε)}[ζ^*] = L_{\mathcal{G}(ε)}[ζ]^{*SP}
   \]

   where \(s_{SP} \in \mathbb{SP}\) is a lifted version of \(s \in S\) that is unaffected by the supplied element of \(P\).

2. After applying \(L_{(·)}[·]\), we then apply a second operator \(X[·] : \mathbb{SP} \rightarrow \mathbb{P}\) which performs the actual extraction of an iterative model, i.e., an element of \(P\), so that \(E_{\mathcal{G}(ε)}[ε] = X[L_{\mathcal{G}(ε)}[ε]]\). In other words, the operator \(X[·]\) corresponds to extracting a recurrence from a parameterized transition formula that represents an entire function body.

6.5.3 Alternative Decompositions of a Recursion Tree

In this section, we discuss the possibility of generalizing \(\text{CHORA}\) by exploiting the power of algebraic program analysis to perform syntactic rearrangements of the extended regular expressions representing a procedure. Whereas \(\text{CHORA}\)'s analysis considers a sequence of procedure-summary approximations that represented different subsets of executions indexed by \textit{height}, the following proposed analyses work with subsets of executions bounded by some other index.

\textit{ICRA-Style Decomposition of a Recursion Tree.} Consider a procedure defined by the following equation:

\[
X = b \oplus s_1 \otimes X \otimes s_2 \otimes X \otimes s_3 X \otimes s_4.
\]
Effectively, \textsc{Chora} would analyze $X$ by evaluating the right-hand side of the following equation:

$$X = (b \oplus s_1 \otimes \Box s_2 \otimes \Box s_3 \otimes \Box s_4) \ast \Box \cdot \Box 0.$$ 

Newtonian (or ICRA-style) analysis can be seen as working with a different equation:

$$X = [([b \oplus s_1 \otimes \Box_1 \otimes s_2 \otimes \Box_2 \otimes s_3 \otimes \Box_2 \otimes s_4 \oplus s_1 \otimes \Box_2 \otimes s_2 \otimes \Box_1 \otimes s_3 \otimes \Box_2 \otimes s_4 \oplus \Box_1 \otimes s_1 \otimes \Box_2 \otimes \Box_1 \otimes s_2 \otimes \Box_2 \otimes s_3 \otimes \Box_2 \otimes s_4 \ast \Box_1 \cdot \Box_1] \ast \Box_2 \cdot \Box_2 0]. \quad (6.3)$$

Notice that the right-hand side of Eqn. (6.3) contains two nested tree-star operators. The crucial feature of the inner tree-star ($\ast \Box_1$) is that it exhibits linear recursion, and it can therefore be represented using a tensored star. Roughly speaking, ICRA handles the inner tree-star using a tensored star, while handling the outer tree-star ($\ast \Box_2$) using Kleene iteration with widening.

Define $\zeta$ to be the body of the outer tree star ($\ast \Box_2$) in Eqn. (6.3). Just as Eqn. (6.2) gave the Kleene iterates of a procedure given the procedure’s body $\epsilon$, Eqn. (6.4) gives the quasi-Newtonian “ICRA iterates” of a procedure given the tensor-starred expression $\zeta$ that is derived from a procedure’s body:

$$n^{th} \text{ ICRA iterate} = \zeta \cdot \Box_2 \cdot \Box_2 \cdots \Box \cdot \Box_2 0. \quad (6.4)$$

In theory, it should be possible to perform a \textsc{Chora}-style analysis of the outer tree-star in the above example, rather than evaluating it using Kleene iteration with widening. In such a \textsc{Chora}-style analysis, we would construct a hypothetical summary $\mathcal{H}[\zeta]$ from the base case $b$, plug in $\mathcal{H}[\zeta]$ at each $\Box_2$ substitution point, and finish evaluating $\zeta$ by performing a tensored star at $\ast \Box_2$; finally, we would extract recurrences from the result of evaluating $\zeta$, and solve those recurrences to obtain bounds on the parameters of $\mathcal{H}[\zeta]$. (Note that there is no guarantee that doing so would obtain more precise results than \textsc{Chora} on most examples. It might not even help on any examples.) Such a \textsc{Chora}-style analysis could be combined with a generalization of the depth-bound analysis that finds a bound on the dimension of a recursion tree.

Analyzing the Fibonacci Function More Precisely. \textsc{Chora} finds loose bounds on the return value of the usual recursive Fibonacci function; the bound is roughly $(\sqrt{2})^n = 2^{(\frac{n}{2})} \leq F(n) \leq 2^{n+1}$. This bound is loose because of two factors:

\textit{Factor #1.} Recall that \textsc{Chora} performs height-based and dual-height-based analysis, each of which relies upon bounding the length of a root-to-leaf path in the recursion tree. Let $\ell_n$ be the length of any such path,
int fibprime1(int n) {
    if (n == 0 || n == 1) return 1;
    return fibprime1(n-1) + fibprime2(n-2);
}

int fibprime2(int n) {
    if (n == 0 || n == 1) return 1;
    return fibprime1(n);
}

Figure 6.3: A modified version of the usual recursive Fibonacci procedure, obtained by splitting the procedure into two procedures in such a way that the length of a root-to-leaf path in the recursion tree may be constrained more tightly than in the original procedure.

when starting from an initial value of the procedure’s parameter n. The bound that CHORA obtains on $\ell_n$ is roughly $\frac{n}{2} \leq \ell_n \leq n + 1$. Note the similarity between this path-length bound and the exponents in the return-value bound given before.

*Factor #2.* CHORA does not recognize any difference between the return values of $F(n - 1)$ and $F(n - 2)$; it uses a single quantity to represent the bound on both of those return values at each height. Subsuming both return values under a single bound, which gets added to itself (i.e., doubled) as height goes up, causes CHORA to find an exponential bound having base 2, rather than a base of the golden ratio $\phi$.

There is a generalization of CHORA that would improve upon the above-described analysis by changing CHORA’s view of the recursion tree. In this sub-section, we describe this change as a source-to-source transformation of the input program (or a transformation of the equation system), but a generalized version of CHORA could be designed that would have the effect of performing this transformation automatically when given the original program.

The following equation describes the usual recursive Fibonacci function:

$$F = b \oplus s_1 \otimes F \otimes s_2 \otimes F \otimes s_3. \quad (6.5)$$

Now, consider a modified equation system:

$$F_1 = b \oplus s_1 \otimes F_1 \otimes s_2 \otimes F_2 \otimes s_3. \quad (6.6)$$
$$F_2 = b \oplus F_1.$$

Notice that this modified equation system is reminiscent of the matrix recurrence that ICRA (or CHORA) extracts from an iterative Fibonacci implementation, except that it is defined using an equation system over summary-valued variables, rather than numeric variables. The modification that produced Eqn. (6.6) from Eqn. (6.5) can also applied as a source-to-source transformation; the resulting code is shown in Fig. 6.3.

Perhaps surprisingly, the analysis performed by CHORA on this modified version of the Fibonacci procedure (which we call fibprime) is significantly more precise than CHORA’s analysis of the original procedure. CHORA
can prove an \(O(\phi^n)\) bound\(^1\) (albeit in the form of an implicitly interpreted function) on the return value of \(\text{fibprime}\).

\(\text{CHORA}'s\) bound on \(\text{fibprime}\) is better than its bound on the original Fibonacci procedure because both \textit{Factor #1} and \textit{Factor #2} are different for \(\text{fibprime}\): Notice that the length \(l_n\) of a root-to-leaf path in the recursion tree can be bounded much more tightly; the bound now looks something like \(n \leq l_n \leq n + 2\).

(Why is there no longer a \(\frac{1}{2}\) coefficient on the lower bound? Because when \(\text{fibprime1}\) subtracts two from \(n\), there are two steps down in height before control is returned to \(\text{fibprime1}\), and when it subtracts one from \(n\), there is only one step down in height before control is returned to \(\text{fibprime1}\); thus, the height very closely tracks the value of \(n\).) Also, \(\text{CHORA}\) now gives the return values of the first and second recursive call different bounds, because they are bounds on the values of two different procedures.

### 6.5.4 Alternative Inner Layers of Abstraction

In this sub-section, we discuss one aspect of the hypothetical summaries that may not be obvious. For each parameter (i.e., bounding function) \(b(h)\), with associated relational expression \(\tau\), \(\text{CHORA}\)'s hypothetical summary contains a sub-formula \(\psi_1(\tau, b(h)) \overset{\text{def}}{=} \tau \leq b(h)\). We may wish to consider generalizations of \(\text{CHORA}\) that use some different connective sub-formula \(\psi(\tau, b(h))\) to relate the parameter to the relational expression.

In this sub-section, we discuss a constraint on the choice of such connective sub-formulas. The hypothetical summary is used (as an abstraction of a recursive call) to build up a summary of all paths through the procedure. Along such paths, the hypothetical summary is effectively used once per call. If a recursive call occurs in a loop, there may be unboundedly many calls in a path through the procedure. Furthermore, note that all of these uses of the hypothetical summary share a single vector of parameter values. Let \(\tau_1, \ldots, \tau_n\) be the occurrences of the relational expression \(\tau\) at \(n\) different calls along some path through the procedure. Then, using the hypothetical summary implicitly assumes the following formula, which says that we must always be able to find a single parameter value \(b(h)\) that is consistent with any finite set of \(\tau\) values.

\[
\forall \tau_1, \ldots, \tau_n, \exists b(h). \psi(\tau_1, b(h)) \land \ldots \land \psi(\tau_n, b(h)). \tag{6.7}
\]

In the case that the connective sub-formula \(\psi\) is \(\psi_1\), the formula Eqn. (6.7) holds, because for any sequence of values \(\tau_1, \ldots, \tau_n\), we can always choose any \(b(h)\) value that is greater than or equal to their maximum. Suppose, however, that we tried using \(\psi_2(\tau, b(h)) \overset{\text{def}}{=} \tau = b(h)\) instead. Then, we would not be able to find a single

---

\(^1\) The assertion that \(\text{CHORA}\) can prove, comparing the recursive \(\text{fibprime1}\) to the obvious loop-based computation of the Fibonacci numbers (called \(\text{iterfib}\)) is \(\text{fibprime1}(n) \leq \text{iterfib}(n)\| \text{fibprime1}(n) \leq \text{iterfib}(n + 1)\), due to the analysis of recursion-tree path length only being precise up to an additive factor of 1. Note that, because the summary of \(\text{iterfib}\) involves an implicitly interpreted function, \(\text{CHORA}\) cannot prove that \(\text{iterfib}\) is non-decreasing, which prevents \(\text{CHORA}\) from proving a simplified version of the above assertion in which the first disjunct is removed.
value $b(h)$ that satisfied $\psi_2(\tau_i, b(h)) \land \psi_2(\tau_j, b(h))$ whenever $\tau_i \neq \tau_j$, and therefore the program analysis would be *unsound*: some paths through the procedure would be excluded from consideration because of the hypothetical summary that we used. It may be useful to look for generalizations of CHORA that use different connective sub-formulas $\psi$, but we must make sure that they have the property that Eqn. (6.7) holds.
Chapter 7

Conclusion

In this chapter, we reiterate the main contributions of the research, we discuss some limitations of our techniques, and we offer some concluding remarks.

7.1 Contributions

The main contributions of the research presented in this dissertation are as follows.

- We extended CRA to create ICRA, a context-sensitive interprocedural version that handles loops and recursion—including non-linear recursion—in a uniform way. (See §3.3.3.)

- We presented the NPA-TP-GJ framework, which represents the interface and properties required for our approach. NPA-TP-GJ is an interprocedural-analysis method that can be used when the abstract domain has infinite ascending chains and does not support an effective operation for testing the equivalence of two values.

  Much as NPA-TP resurrects Tarjan’s intraprocedural path-expression method for use in interprocedural analysis, NPA-TP-GJ resurrects another old intraprocedural technique—elimination methods [Ryder and Paul, 1986]—for use in interprocedural analysis. (See §3.3.3.)

- We presented the results of experiments with an implementation of ICRA. The experiments showed that ICRA has broad overall strength, compared with several state-of-the-art software model checkers. ICRA also performs well when applied to the problem of establishing resource-usage bounds on the C4B suite [Carbonneaux et al., 2015a]. (See §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* [Reps et al., 2004; Thakur and Reps, 2012], which, given an arbitrary non-linear formula $\varphi$, computes a wedge that over-approximates $\varphi$. (See §4.4.)

- 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 sequences* [Kauers and Paule, 2011, §4.2]. (See §4.5.)

- 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 [1967]. 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. (See §4.6.)

- We introduced an analysis method based on “hypothetical summaries.” It hypothesizes that a summary exists 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.

- We developed a procedure-summarization technique called height-based recurrence analysis, which uses the notion of hypothetical summaries to produce bounds on the values of program variables based on the height of recursion. (See §5.4.1.)

- We further developed algorithms that, when used in conjunction with height-based recurrence analysis, yield more precise summaries. (See§5.4.2 and §5.4.3.)

- Furthermore, we gave an algorithm that generalizes height-based recurrence analysis to the setting of mutual recursion. (See §5.4.4.)
We implemented height-based recurrence analysis, and its generalizations, in the CHORA tool. Our experiments showed that CHORA is able to handle many non-linearly recursive programs, and generate invariants that include exponentials, polynomials, and logarithms. For instance, it is able to 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. (See §5.5.)

7.2 Limitations

The previous chapter proposed a few future directions for research. In this section, the goal is similar, but we discuss the possibilities for future research from a different perspective, by emphasizing various limitations of the work that we have done so far.

Integer Variables. The techniques of this dissertation only discover facts about the integer-valued variables of a program, and overflow is not considered. Non-integer variables, including arrays and data structures, are not presently analyzed by our techniques. Extending the techniques to handle more data types is arguably the deepest and most important area of future work.

No Refutation of Properties. As discussed in §6.3, the techniques of this dissertation can only prove, but not disprove, that an assertion in a program always holds whenever control reaches the assertion. Thus, our techniques do not provide a way to return counterexamples when an assertion does not always hold.

No Completeness Guarantees. For the analyses described in this dissertation, we have rarely sought to provide guarantees about the completeness of the analysis methods. Thus, if the analyses fail to find a desired result, it is not usually easy to explain why the desired result was not obtained. This failing may cause difficulties when attempting to build tools that use CRA as a backend, because it is not clear how to craft input to CRA in such a way as to guarantee good output from CRA.

Because many problems related to non-linear mathematics are undecidable, it is not possible in general to provide all the completeness guarantees than an analyst might want. As demonstrated in Chapter 4 and Chapter 5, CRA-based analyses are able to prove some sophisticated numerical properties, but their capabilities in this regard are sometimes brittle in practice.

C-Finite Recurrences. In Chapter 4, we describe an analysis pipeline that works with C-finite recurrences. However, there are quite natural recurrences that are outside of that class. Thus, for example, ICRA can precisely analyze a loop having the body $x = 2 \times x$, but not a loop having the body $x = y \times x$, because the recurrence for the latter loop is not C-finite.
Simple Numerical Invariants (e.g., Non-negativity). It is common to have some program variables that are always non-negative. In practice, however, proving non-negativity is somewhat difficult for a CRA-based analysis, and when the analyzer fails to discover this simple property, it will often fail to discover more complex ones. Some other analysis tools solve this problem by working with terms of the form $\max(x, 0)$ or $|x|$, where $x$ is a program variable, so that non-negativity holds by construction of the terms. The analysis tools CRA, ICRA, and CHORA have all included, as the first step of their analysis, an intraprocedural abstract-interpretation step using the domain of intervals, to allow the analyzers to discover some important facts of this kind. However, this analysis pass has various limitations (e.g., it only performs intraprocedural analysis).

Non-Terminating Executions. The analysis techniques presented in this dissertation characterize the transition relations (i.e., the pre-state/post-state relations) of program fragments. One limitation of transition-relation-based analysis is that it ignores non-terminating executions of program fragments, precisely because such executions never reach a post-state.

As a result of our focus on transition relations, whenever we applied our analysis techniques to resource-bound-analysis problems, we made no attempt to characterize the resource usage of non-terminating executions of programs. Therefore, whenever our analysis techniques obtain a bound on the resource usage (e.g., the running time) of a program, the bound applies only to the terminating executions of a program; such a bound can be interpreted as saying that, whenever the program terminates, it will terminate with a usage of resources that satisfies the bound.

However, it would be possible to use the techniques of this dissertation to obtain information about non-terminating executions. For example, one could prove that a program always terminates by (1) annotating the program with a time-cost variable in an appropriate manner, and then (2) finding an upper bound on the value of the time-cost variable at the head of each loop and the entry location of each recursive procedure, in terms of the values of variables at the entry location of the main procedure.

Templates Used By CHORA. It is easy to construct a procedure for which the obvious summary falls outside of the class of summaries that the techniques of Chapter 5 (i.e., the CHORA analyzer) can discover. For example, consider the following program:

```c
int g; void rec(int n, int p) { g += p; if(n == 1) { return; } rec(n - 1, p); rec(n - 1, p); }
```

A precise summary for $rec$ is $g' = g + p(2^n - 1)$, but that summary falls outside the class of summaries that the CHORA analyzer can discover. (Note that, when CHORA finds bounds on a relational expression $\tau$, the bounds have the form $\tau \leq b(h)$, and in the case of the procedure $rec$, the height $h$ equals the parameter $n$.) There are many ways that CHORA could be enhanced so as to find bounds that have a more general form, so that it could find the precise summary for $rec$, although such enhancements would typically make the
analysis more complex by requiring various side conditions to be checked.

If CHORA worked with relational expressions such as $\tau = \frac{g'}{p} - g$, then it could use the bounding function $b(h) = 2^h - 1$. However, CHORA only allows linear relational expressions. Alternatively, CHORA could be modified to find bounding functions that are functions of the pre-state $x_0$ of the initial call to rec, so that bounds have the form $\tau \leq b(h, x_0)$. Then, if CHORA could determine that $p = p_0$ always holds (where $p_0$ is the initial value of $p$), it could use the relational expression $g' - g$ with the bounding function $b(h, x_0) = p_0(2^h - 1)$. However, CHORA only allows bounding functions that are functions of the height $h$ alone. In any case, the procedure rec illustrates that it is not difficult to construct an example that is outside of the capabilities of the current technique.

7.3 Concluding Remarks

In this dissertation, we have investigated CRA, and more generally, the algebraic approach to program analysis.

We enhanced CRA by creating ICRA, which produces more precise results when analyzing recursive procedures, and especially linearly recursive procedures, by introducing a tensor operator and bringing in ideas from Newtonian Program Analysis and an equation-system-rewriting algorithm that is analogous to Gauss-Jordan elimination. We enhanced ICRA’s mathematical reasoning capabilities by introducing the wedge abstract domain, which uses Gröbner-basis methods and a collection of inference rules to find relationships among certain non-linear terms over program variables, along with a recurrence-solving technique based on the operational calculus. We then enhanced ICRA’s handling of non-linear recursion by introducing the CHORA analyzer, which uses hypothetical summaries to extract height-based recurrences.

Throughout this research program, the algebraic framework gave us the capability to consider rearrangements of the structure of a program that are a better fit for recurrence-based analysis, and we used this capability to achieve more precise analysis results. We also showed how to analyze many program features with a unified, recurrence-based approach, and as a consequence of that unified approach, enhancements of the central recurrence-handling facility yielded benefits to the analysis of multiple program features at once.

Using these enhancements, we were able to prove more assertions and prove tighter resource-usage bounds than the original version of CRA. Moreover, the precision of our results compared favorably with state-of-the-art implementations of a variety of conventional approaches. Taken together, these observations suggest that there is a promising future for research into algebraic, compositional, and recurrence-based approaches to program analysis.
Appendix A

Proofs about ICRA

Proposition 3.11. Let $\Sigma$ be an alphabet, let $\mathcal{T} = (\mathcal{D}, \mathcal{D}_\tau, \odot, \times)$ be a tensor-product domain, and let $\llbracket \cdot \rrbracket : \Sigma \to \mathcal{D}$ be an interpretation for the alphabet. For any regular expression $E$ over $\Sigma$ and any path $p \in \llbracket E \rrbracket^P$, we have $\llbracket E \rrbracket \succeq \llbracket p \rrbracket$.

Proof: Extend $\llbracket \cdot \rrbracket$ to tensored paths by defining $\llbracket (p, q) \rrbracket \overset{\text{def}}{=} \llbracket p \rrbracket \odot \llbracket q \rrbracket$. Observe that every operation of $\mathcal{T}$ is monotone with respect to $\preceq$ and $\preceq^\tau$ with the exception of the iteration operators $*$ and $*^\tau$. We prove that for any (tensored) extended regular expression $E$, for all (tensored) paths $q \in \llbracket E \rrbracket^P$, $\llbracket E \rrbracket \succeq \llbracket p \rrbracket$ by induction on $E$.

- Case $a \in \Sigma$: trivial

- Case $E \cdot F$: if $p \in \llbracket E \cdot F \rrbracket^P$, then there is some $p_1 \in \llbracket E \rrbracket^P$ and $p_2 \in \llbracket F \rrbracket^P$ such that $p = p_1p_2$.

$$\llbracket p \rrbracket = \llbracket p_1 \rrbracket \odot \llbracket p_2 \rrbracket \quad \llbracket E \rrbracket \odot \llbracket F \rrbracket \quad \text{IH, Monotonicity}$$

- Case $E_T \cdot_T F_T$: if $(p, q) \in \llbracket E_T \cdot_T F_T \rrbracket^P$, then there is some $(p_1, q_1) \in \llbracket E_T \rrbracket^P$ and $(p_2, q_2) \in \llbracket F_T \rrbracket^P$ such that $p = p_2p_1$ and $q = q_1q_2$.

$$\llbracket (p, q) \rrbracket = \llbracket p \rrbracket \odot \llbracket q \rrbracket \quad \text{Definition}$$

$$\overset{\tau} = \llbracket (p_1, q_1) \odot (p_2, q_2) \rrbracket \quad \overset{\tau} = \llbracket (p_1, q_1) \odot (p_2, q_2) \rrbracket \quad \text{IH, Monotonicity}$$

- Case $E + F$: if $p \in \llbracket E + F \rrbracket^P$, then we must have $p \in \llbracket E \rrbracket^P$ or $p \in \llbracket F \rrbracket^P$. From the induction hypothesis, we
must have \([p] \lesssim [E]\) or \([p] \lesssim [F]\). In either case we have \([p] \lesssim [E] \oplus [F] = [E + F]\). The case \(E \star + F\) is similar.

- Case \(E^*\): if \(p \in [E^*]^p\), then there exists \(p_1, ..., p_n \in [E]^p\) such that \(p = p_1...p_n\). By the induction hypothesis, we have \([p_i] \lesssim [E]\) for all \(i\), and thus \([p] \lesssim [E]^n\). The fact that \([E]^n \lesssim [E^*] = [E]^*\) holds for any \(n\) can be shown by induction on \(n\), using the Reflexivity and Transitivity axioms. The case \(E^*_T\) is similar.

- Case \(E \circ F\): if \((p, \overline{p}) \in [E \circ F]^p\), then \(p \in [E]^p\) and \(\overline{p} \in [F]^p\).

\[
[[p, \overline{p}]] = [p] \circ [\overline{p}] \\
\lesssim [E] \circ [F] \quad \text{IH, Monotonicity} \\
= [[E \circ F]] \quad \text{Definition}
\]

- Case \(E \times F\): if \(p \in [E \times F]^p\), then there is some \(p_{mid} \in [E]^p\) and some \((p, \overline{p}) \in [F]^p\) such that \(p = pp_{mid}\overline{p}\).

\[
[[p]] = [p] \circ [p_{mid}] \circ [\overline{p}] \\
\quad \equiv [[p] \circ ([p_{mid}] \times 1_T) \circ [\overline{p}]] \\
\quad \equiv [p_{mid}] \times (1_T \circ_T ([p] \circ [\overline{p}])) \\
\quad \equiv [p_{mid}] \times ([p] \circ [\overline{p}]) \\
\lesssim [[E] \times [F]] \quad \text{IH, Monotonicity} \\
\equiv [[E \times F]] \quad \text{Definition}
\]

**Lemma A.1.** Let \(\Sigma\) be an alphabet, let \(X\) be a set of variables, let \(E \in \text{Ext}(\Sigma, X)\) be an extended regular expression over \(\Sigma\) and \(X\), and let \(X \in X\) be a variable. Then \(E \simeq \text{factor}_X(E)\).

**Proof:** By structural induction on \(E\).

- Case \(a \in \Sigma\): \(\text{factor}_X(a) = a + (X \times 0_T) \simeq a\).

- Case \(X\): \(\text{factor}_X(X) = 0 + (X \times 1_T) \simeq X\).

- Case \(X_i \in X\) for some \(X_i \neq X\):

\[
\text{factor}_X(X_i) = X_i + (X \times 0_T) \simeq X_i.
\]
• Case $E + E'$: Let $\text{factor}_X(E) = F + (X \times F_T)$ and let $\text{factor}_X(E') = F' + (X \times F'_T)$.

$$\text{factor}_X(E + E') = \begin{pmatrix} (F + F') \\ + (X \times (F_T + T F'_T)) \end{pmatrix}$$

Def'n

$$\simeq \begin{pmatrix} F + (X \times F_T) \\ + F' + (X \times F'_T) \end{pmatrix}$$

Def'n

$$= \text{factor}_X(E) + \text{factor}_X(E')$$

Def'n

$$\simeq E + E'$$

IH

• Case $E \cdot E'$: Let $\text{factor}_X(E) = F + (X \times F_T)$ and let $\text{factor}_X(E') = F' + (X \times F'_T)$.

$$\text{factor}_X(E \cdot E') = \begin{pmatrix} (F \cdot F') \\ + X \times \begin{pmatrix} F_T \cdot F' (1 \cdot F') \\ + T F'_T \cdot F' (E \times 1) \end{pmatrix} \end{pmatrix}$$

Def'n

$$\simeq \begin{pmatrix} (F \cdot F') \\ + (X \times (F_T \cdot F' (1 \cdot F'))))) \\ + (X \times (F'_T \cdot F' (E \times 1)))) \end{pmatrix}$$

Def'n

$$\simeq \begin{pmatrix} (F \cdot F') \\ + ((X \times F_T) \cdot F') \\ + (X \times (F'_T \cdot F' (E \times 1)))) \end{pmatrix}$$

Def'n

$$\simeq \begin{pmatrix} (F + (X \times F_T)) \cdot F' \\ + (X \times (F'_T \cdot F' (E \times 1)))) \end{pmatrix}$$

Def'n

$$\simeq \text{factor}_X(E) \cdot F'$$

$$\simeq (E \cdot F') + (X \times (F'_T \cdot F' (E \times 1))))$$

IH

$$\simeq (E \cdot F') + (E \cdot (X \times F'_T))$$

$$\simeq E \cdot (F' + (X \times F'_T))$$

$$= E \cdot \text{factor}_X(F)$$

Def'n

$$\simeq E \cdot F$$

IH

• Case $E^*$: $\text{factor}_X(E^*) = E^* + (X \times 0_T) \simeq E^*$. 
• Case $E \bowtie E_T$: Let $\text{factor}_X(E) = F + (X \bowtie F_T)$.

\[
\begin{align*}
\text{factor}_X(E \bowtie E_T) &= (F \bowtie E_T) + X \bowtie (F_T \cdot E_T) & \text{Defn} \\
&\simeq (F \bowtie E_T) + (X \bowtie F_T) \bowtie E_T \\
&\simeq (F + (X \bowtie F_T)) \bowtie E_T \\
&\simeq (\text{factor}_X(E)) \bowtie E_T & \text{Defn} \\
&= E \bowtie E_T & \text{IH}
\end{align*}
\]

**Theorem 3.15.** Given a system of equations $S : \{X_i = R_i\}_{i=1}^n$ with regular right-hand sides over an alphabet $\Sigma$ and a finite set of variables $X = \{X_1, \ldots, X_n\}$, Alg. 3.14 computes a system of equations $\hat{S} : \{\hat{X}_i = \hat{E}_i\}_{i=1}^n$ where the right-hand sides are normal extended regular expressions over $\Sigma$ and $X$ with no free variables, and which is equivalent to $S$ in the sense that $\text{Paths}_S(X_i) = \text{Paths}_{\hat{S}}(X_i)$ for all $X_i$.

**Proof:** Recall that Alg. 3.14 constructs a sequence of systems

\[S_0 : \{X_i : E_{i,0}\}_{i=1}^n \cdots S_n : \{X_i : E_{i,n}\}_{i=1}^n\]

with $S_0 = S$ and $S_n = \hat{S}$. It is sufficient to show that for every $k < n$, we have $\text{Paths}_{S_k} = \text{Paths}_{S_{k+1}}$. Recall that $\text{Paths}_{S_k}$ is defined to be the least solution to $S_k$, so if $\sigma$ is any function with $\sigma(X_i) = [E_{i,k}]_\sigma^P$ for all $i$, we must have that $\text{Paths}_{S_k}$ is less than $\sigma$ (in the sense that $\text{Paths}_{S_k}(X_i) \subsetneq \sigma(X_i)$ for all $X_i$). It is thus sufficient to prove that for all $k$ and all $i$, $\text{Paths}_{S_k}(X_i) = [E_{i,k+1}]_{\text{Paths}_{S_k}}^P$ (so that $\text{Paths}_{S_{k+1}}$ is less than $\text{Paths}_{S_k}$) and $\text{Paths}_{S_{k+1}}(X_i) = [E_{i,k}]_{\text{Paths}_{S_{k+1}}}^P$ (so that $\text{Paths}_{S_k}$ is less than $\text{Paths}_{S_{k+1}}$). Furthermore, because $E_{i,k+1}$ is merely a substitution instance of $E_{i,k}$ for all $i \neq k$, it is sufficient to prove the $i = k$ case.

Let $F + X \bowtie F_T = \text{factor}_{X_k}(E_{k,k})$. By definition, $E_{k,k+1}$ is $F \bowtie F_T^+$.

1. Prove $\text{Paths}_{S_{k+1}}(X_k) = [E_{k,k}]_{\text{Paths}_{S_{k+1}}}^P$.

Since $\text{Paths}_{S_{k+1}}$ is the least solution to $S_{k+1}$, we have

\[
\begin{align*}
\text{Paths}_{S_{k+1}}(X_k) &= [E_{k,k+1}]_{\text{Paths}_{S_{k+1}}}^P \\
&= [F \bowtie F_T^+]_{\text{Paths}_{S_{k+1}}}^P,
\end{align*}
\]
Since

\[ F \times F_2^+ \simeq F \times (1_\tau + F_2^+ \cdot \tau F_2) \]
\[ \simeq (F \times 1_\tau) + (F \times F_2^+ \cdot \tau F_2) \]
\[ \simeq F + (F \times F_2^+) \times \tau F_2 \]

we have

\[ \text{Paths}_{k+1}(X_k) = [F \times F_2^+]^P_{\text{Paths}_{k+1}} \]
\[ = [F + (F \times F_2^+) \times \tau F_2]^P_{\text{Paths}_{k+1}} \]
\[ = [F + X_k \times F_2]^P_{\text{Paths}_{k+1}} \]

Finally, since \( F + X_k \times F_2 \simeq E_{k,k} \) (Lemma A.1), we have \( \text{Paths}_{k+1}(X_k) = \text{Paths}_{k+1}(E_{k,k}) \).

2. Prove \( \text{Paths}_k(X_k) = \|E_{k,k+1}\|_{\text{Paths}_k}^P \). By assumption \( \text{Paths}_k(X_k) = \|E_{k,k}\|_{\text{Paths}_k}^P \). Since \( F + X_k \times F_2 \simeq E_{k,k} \) (Lemma A.1), we have \( \text{Paths}_k(X_k) = [F + X_k \times F_2]^P_{\text{Paths}_k} \). Since \( \text{Paths}_k \) is the least solution to \( S_k \), we must have \( \text{Paths}_k(X_k) = \bigcup_i [F \times F_2^+]^P_{\text{Paths}_k} = [F \times F_2^+]^P_{\text{Paths}_k} \) and thus

\[ \text{Paths}_k(X_k) = [F \times F_2^+]^P_{\text{Paths}_k} \]

**Lemma A.2.** Let \( S : \{X_i = E_i\}_{i=1}^n \) be a system of equations where the right-hand sides are extended regular expressions over an alphabet \( \Sigma \) and a finite set of variables \( X = \{X_1, ..., X_n\} \). Let \( \mathcal{F} = (\mathcal{D}, \mathcal{D}_\tau, \odot, \times) \) be a tensor-product domain, and let \( \| \cdot \| : \Sigma \to \mathcal{D} \) be an interpretation for the alphabet. Let \( D : X \to \mathcal{D} \) be any function such that \( D(X_i) \succeq \|E_i\|_D \) for all \( i \). Then \( D \) is a solution to \( S \): for all \( i \) and all \( p \in \text{Paths}_S(X_i) \), we have \( D(X_i) \succeq \|p\| \).

**Sketch of Proof:** Observe that \( \text{Paths}_S \) is the limit of a Kleene iteration sequence:

\[ P^0(X_i) \overset{\text{def}}{=} \emptyset \]
\[ P^{n+1}(X_i) \overset{\text{def}}{=} \|E_i\|_{P_n} \]
\[ \text{Paths}_S(X_i) \overset{\text{def}}{=} \bigcup_{n \in \mathbb{N}} P^n(X_i) \].

It follows that if \( p \in \text{Paths}_S(X_i) \), then there is some \( n + 1 \) such that \( p \in P^{n+1}(X_i) = \|E_i\|_{P_n} \). Thus, it is sufficient to prove that for all \( n \in \mathbb{N} \), for all \( X_i \in X \), and all \( p \in \|E_i\|_{P_n} \), we have \( \|E_i\|_D \succeq \|p\| \) (whence \( D(X_i) \succeq \|p\| \) holds by transitivity of \( \succeq \)).
The proof proceeds by nested induction on \( n \) and \( E_i \). The argument is essentially the same as for Prop. 3.11, except that there is an additional case because \( E_i \) may contain variables. The case for variables follows immediately from the induction hypothesis for \( n \).

**Theorem 3.18.** There exists a \( k \) such that \( S \) stabilizes at \( k \), and \( D^k \) is a post-fixpoint solution to \( S \).

**Proof:** First, we show that \( S \) eventually stabilizes. Let \( E^* \) be any starred subexpression that appears in some \( E_i \) (the case for \( E^*_T \) is similar). We must show that the sequence \( \langle \text{body}^r(E) \rangle_{r \in \mathbb{N}} \) eventually stabilizes. Consider the infinite sequence \( \langle \alpha([E]_{D^r}) \rangle_{r \in \mathbb{N}} \) in \( D^k \). Because \( \triangleright \) is a widening operator, the sequence

\[
\begin{align*}
 b^*_1 &= \alpha([E]_{D^1}) \\
 b^*_{r+1} &= b^*_r \triangleright \alpha([E]_{D^r})
\end{align*}
\]

eventually stabilizes. Finally, observe that the sequence \( \langle b^*_r \rangle_{r \in \mathbb{N}} \) coincides with the sequence \( \langle \text{body}^r(E) \rangle_{r \in \mathbb{N}} \).

Next, we show that if \( S \) stabilizes at \( k \), then \( D^k \) is a post-fixpoint solution to \( S \). We must demonstrate that for all \( i \) and all paths \( p \in \text{Paths}_{S}(X_i) \), we have \( D^k(X_i) \gtrsim \llbracket p \rrbracket \). By Lem. A.2, it is sufficient to prove that \( D^k(X_i) \gtrsim \llbracket E_i \rrbracket_{D^k} \) for all \( i \). The essence of the argument is to show that the following sequence of (in)equalities is valid:

\[
D^k(X_i) = \text{eval}^k(E_i) \gtrsim \llbracket E_i \rrbracket_{D^{k-1}} = \llbracket E_i \rrbracket_{D^k}.
\]

The first equality in the sequence holds by the definition of \( D^k \). The middle inequality we prove by induction on \( E_i \). We prove validity of the last equation by showing that the assumption that \( S \) stabilizes at \( k \) (i.e., the fact that \( \text{body}^k(E) = \text{body}^{k-1}(E) \) for every “loop-body” expression \( E \)) is sufficient to prove that \( D^{k-1} = D^k \).

First, we show that \( \text{eval}^k(E_i) \gtrsim \llbracket E_i \rrbracket_{D^{k-1}} \) holds. We prove that this property holds for all sub-expressions of \( E_i \) by structural induction.

- Case \( a \in \Sigma, X_j \in X \): trivial.
- Case \( E + F, E \cdot F, E \cdot \cdot F, E \cdot \cdot \cdot F, E \cdot \cdot \cdot \cdot F \): from the induction hypothesis and monotonicity.
- Case \( E^* \) (Case \( E^*_T \) is similar): First, observe that by property 1 of the widening operator, we have

\[
\text{body}^{k-1}(E) \triangleright \alpha([E]_{D^{k-1}}) \gtrsim \alpha([E]_{D^{k-1}}).
\]
It follows that:

\[
\text{eval}^k(E^*) = \text{cl}(\text{body}^k(E)) \\
= \text{cl}(\text{body}^{k-1}(E) \lor \alpha([E]_{D^{k-1}})) \\
\geq \text{cl}(\alpha([E]_{D^{k-1}})) \\
= ([E]_{D^{k-1}})^* \\
= [E^*]_{D^{k-1}}
\]

Definition

Second, we prove that \(D^{k-1} = D^k\) holds. Let \(X_i\) be a variable. We have \(D^k(X_i) = \text{eval}^k(E_i)\) by definition, so it is sufficient to prove that \(\text{eval}^k(E_i) = \text{eval}^{k-1}(E_i)\). We prove that this equality holds for all closed sub-expressions of \(E_i\)—i.e., subexpressions without free variables—by structural induction.

- Case \(a \in \Sigma\): trivial.
- Case \(E + F, E \cdot F, E \times F, E \circ F, E \oplus F, E \cdot F, E \cdot F, E \cdot F\): immediately from the induction hypothesis.
- Case \(E^*\) (Case \(E^*\) is similar): Because \(S\) stabilizes at \(k\), we have \(\text{body}^k(E) = \text{body}^{k-1}(E)\). It follows that \(\text{cl}(\text{body}^k(E)) = \text{cl}(\text{body}^{k-1}(E))\), and thus \(\text{eval}^k(E^*) = \text{eval}^{k-1}(E^*)\).
Appendix B

Soundness of Height-Based Recurrence Analysis

In this section, we provide a detailed argument for the soundness of height-based recurrence analysis. We discuss the process of performing a height-based recurrence analysis on some procedure $P$. The sequence of operations in that analysis is as follows. First, Alg. 7 analyzes $P$, and produces as output a set of candidate recurrence inequations. On lines (1)–(6), Alg. 7 also produces a set $\{\tau_i\}_{i \in [1, n]}$ of two-vocabulary relational expressions. Next, Alg. 8 filters down the set of candidate recurrences produced by Alg. 7 to obtain a stratified recurrence that can be solved by a C-finite recurrence solver. Finally, a recurrence solver produces a solution in the form of a set of functions $\{b_i\}_{i \in B}$, where $B = \{i_1, \ldots, i_m\}$ is a subset of the indices $[1, n]$.

In this discussion of soundness, we wish to relate the functions $\{b_i\}_{i \in B}$ that are produced by the analysis to the sets of values $V_\tau(P, h)$ taken on by each relational expression $\tau_i$ at each height $h$, which have the following definition in terms of the relational semantics given in §5.3:

$$V_\tau(P, h) \overset{\text{def}}{=} \{\mathcal{E}[\tau](\sigma, \sigma') : (\sigma, \sigma') \in R(P, h)\}.$$  

We use $V_\tau(P, h)$ to prove a height-relative soundness property of our procedure summaries, which contain the height $h$ as an explicit parameter. The fact that the summaries contain an explicit representation of height means that they can be made more precise by conjoining them to the depth-bound summaries computed in §5.4.2.

The goal of this section is to prove the following soundness theorem.

**Theorem B.1.** Let $P$ be a procedure to which Alg. 7 and Alg. 8 have been applied to obtain stratified recurrence. Let $\{\tau_i\}_{i \in [1, n]}$ be the relational expressions computed by Alg. 7. Let $B \subseteq [1, n]$ be such that $\{b_i\}_{i \in B}$ is the set of functions produced by solving the stratified recurrence. Then, the following statement holds: $\forall h \geq 1. \bigwedge_{i \in B} \forall v \in V_\tau_i(P, h). v \leq b_i(h)$. 
We will prove Thm. B.1 by induction on the height $h$. However, before the main inductive argument, we will provide some definitions, and discuss the properties of Alg. 7, the recurrence-extraction algorithm Alg. 8, and the set of functions $\{b_i\}_{i \in B}$. (Note that the Alg. 8 referred to in this appendix is not the same as the Alg. 3 that appears in the conference version [Breck et al., 2020b] of this document, which appears as the depth-bounding algorithm Alg. 9 in this technical report version.)

Define a feasible trace of a procedure $P$ to be a finite list of pairs of control locations and program states, starting at the entry location of $P$, ending at the exit location of $P$, in which all the state transitions are consistent with the semantics of $P$, and all calls are matched by returns. Note that this definition only considers finite (i.e., terminating) traces of $P$, which is useful because our ultimate goal is to find procedure summaries that over-approximate a procedure’s pre-state-post-state relation, and a procedure only has a post-state when it terminates. (As described below, we will also discuss a modified version of $P$ called $\hat{P}$, and we will consider the feasible traces of $\hat{P}$ to be only those that meet some additional constraints.) Furthermore, we define the feasible traces of $P$ up to height $h$ to be those feasible traces that have a recursion depth not exceeding $h$. For the following soundness proof, we define invariants of $P$ to be properties that hold in all feasible traces of $P$.

As explained above, the main function of Alg. 8 is to filter down the set of candidate recurrence inequations produced by Alg. 7, to obtain a subset that constitute a stratified recurrence. At the end of that process, the inequations are changed into equations so as to obtain the maximal solution to the set of inequations. The output of Alg. 8 is a stratified recurrence that can be written as:

$$\bigwedge_{i \in B} b_i(h + 1) = p_i(b_i(h), \ldots, b_m(h)),$$

in which each $p_i(x_1, \ldots, x_m)$ is a polynomial in the variables $x_1, \ldots, x_m$.

All coefficients in the polynomials $\{p_i\}_{i \in B}$ are non-negative, including the constant coefficients, as a result of line (6) of Alg. 8, which drops terms having negative coefficients from the polynomial inequations that are given as input to Alg. 8, thereby weakening the inequations. Aside from line (6), all other steps of Alg. 8 serve only to filter down the set of candidate recurrences. Because of the dropping of terms having negative coefficients on line (6), the polynomials $p_i$ may differ from the corresponding polynomials that appeared in the input to Alg. 8; for each $i \in B$, we denote by $p_i'$ the corresponding polynomial in the input, before terms having negative coefficients were dropped. We refer to the candidate inequations involving the $p_i'$ polynomials as the selected candidate inequations, because they are the ones that are selected by Alg. 8 for inclusion in the stratified recurrence (after their terms having negative coefficients are dropped).

During the recurrence-solving phase, the zero vector is used as the initial condition of the recurrence.
Thus, the set of functions that occur as the solution to the recurrence satisfy $\forall i \in B, b_i(1) = 0$. Because all polynomials $p_i$ have only non-negative coefficients, the functions $\{b_i(h)\}_{i \in B}$ are non-negative and non-decreasing for $h \geq 1$.

Our final digression before proving Thm. B.1 is a discussion of Alg. 7. Alg. 7 operates by manipulating formulas that include a set of function symbols named $\{b_i(h)\}_{i \in \{1..n\}}$ and $\{b_i(h+1)\}_{i \in \{1..n\}}$. The names of these symbols are the same as those of the corresponding functions that are derived by recurrence solving; however, in this proof, we will use separate names for the symbols manipulated by Alg. 7 and the corresponding functions. Instead of $b_i(h)$ we will refer to the symbol $x_i$, and instead of $b_i(h+1)$, we will refer to the symbol $y_i$. Furthermore, although Alg. 7, as written, manipulates formulas that are augmented with these additional symbols, it will be convenient in the following argument to take an alternative, but equivalent, view, according to which Alg. 7 analyzes a modified version of the procedure $P$ called $\hat{P}$ that is obtained by making three changes to $P$.

First, $\hat{P}$ is augmented with a set of immutable auxiliary variables named $x_1, ..., x_n, y_1, ..., y_n$. Second, we impose a constraint on the feasible traces of $\hat{P}$, namely that the pre-state $\sigma$ and post-state $\sigma'$ of any trace $t$ of $\hat{P}$ must satisfy the constraint $\bigwedge_{i=1}^{n} (y_i = E^T \tau_i (\sigma, \sigma'))$, or else $t$ is not considered to be a feasible trace of $\hat{P}$. This constraint is the equivalent of the formula-manipulation performed by line (9) of Alg. 7.

Third, the recursive call sites in $P$ are replaced with control-flow edges that havoc their post-state $\sigma'$ and execute $\text{assume}(\phi_{\text{call}})$. That is, the feasible executions of these control-flow edges of $\hat{P}$ are all those in which the pre-state $\sigma$ and post-state $\sigma'$ of the control-flow edge satisfy $\bigwedge_{i=1}^{n} (E^T \tau_i (\sigma, \sigma') \leq x_i \land x_i \geq 0)$. The effect of replacing call edges of $P$ in this way is equivalent to that of the formula-manipulation performed by line (7) of Alg. 7.

Having described the above constraints on feasible executions of $\hat{P}$, we can now succinctly describe the output of Alg. 7: each of the candidate recurrence inequations returned by Alg. 7 is an invariant of $\hat{P}$, i.e., a property that holds in all feasible executions of $\hat{P}$. The soundness of these invariants follows from the soundness of the underlying program-analysis primitives used by Alg. 7. For the proof of Thm. B.1, the crucial invariant of $\hat{P}$ is the conjunction of the selected candidate inequations:

$$\{y_i \leq p_i(x_{i_1}, ..., x_{i_m})\}_{i \in B}.$$  \hfill (B.1)

We now begin the inductive proof of Thm. B.1.

**Proof.** The base case of the proof corresponds to a height value of 1, which in turn corresponds to executions
of the base case of procedure $P$. At height 1, we must show:

$$\bigwedge_{i \in B} \forall v \in V_{\tau_i}(P, 1). v \leq b_i(1),$$

which holds because each relational expression $\tau_i$ was constructed to be bounded above by zero in the base case, and each bounding function $b_i$ evaluates to zero at height 1.

The inductive step of the proof is as follows. The inductive hypothesis states that, for some $h$,

$$\bigwedge_{i \in B} \forall v \in V_{\tau_i}(P, h). v \leq b_i(h),$$

and the goal is to prove that

$$\bigwedge_{i \in B} \forall v \in V_{\tau_i}(P, h + 1). v \leq b_i(h + 1).$$

Let $t$ be any feasible trace of $P$ at height up to $h + 1$. We will show that we can modify the trace $t$ to produce a new trace $\hat{t}$, and we then prove that $\hat{t}$ is a feasible trace of $\hat{P}$.

To construct $\hat{t}$, first modify $t$ to add the immutable auxiliary variables $x_1, \ldots, x_m, y_1, \ldots, y_m$ to each program state in $t$. We choose the values of these auxiliary variables as follows. Let $\sigma$ and $\sigma'$ be, respectively, the initial and final states of $t$. For each $i \in [1, n]$, set $y_i$ to be the result of evaluating the relational expression $\tau_i$ using the state pair $(\sigma, \sigma')$, that is, $y_i = E[\tau_i](\sigma, \sigma')$. Define the outermost recursive calls in the feasible trace $t$ to be the recursive calls to $P$ that do not occur inside any other recursive call to $P$. Any feasible trace $t$ is of finite length, and therefore $t$ contains some finite number of outermost recursive calls. Thus, the set $R_{i,t}$ of values taken on by $\tau_i$ evaluated at the pre-state/post-state pairs of each outermost recursive call in $t$ is a finite set, and therefore we may define $M_{i,t} \overset{\text{def}}{=} \max(0, \max(R_{i,t}))$ to be the maximum value of $\tau_i$ occurring at any outermost recursive call in $t$. Now set each of the auxiliary variables $x_i$ as follows:

$$x_i = \begin{cases} b_i(h) & \text{if } i \in B \\ M_{i,t} & \text{otherwise} \end{cases}$$

Finally, modify $\hat{t}$ by collapsing all of the intermediate steps of each outermost recursive call in $t$ into a single state transition, so as to match the replacement of recursive-call edges of $P$ with their corresponding edges in $\hat{P}$.

We now argue that $\hat{t}$ meets all the necessary constraints to be considered a feasible trace of $\hat{P}$. The constraint on the initial and final state of $\hat{t}$ is that $\bigwedge_{i=1}^n (y_i = \tau_i)$, which holds by construction. The constraint at each outermost recursive call of $t$ is that, if $(\sigma, \sigma')$ are, respectively, the pre-state and the post-state of the
call, then $\bigwedge_{i=1}^{n} [\mathcal{E}][\tau_i](\sigma, \sigma') \leq x_i \wedge x_i \geq 0$. For $i \notin B$, $x_i = M_{i,t}$, and each such $M_{i,t}$ satisfies the constraint by construction.

For $i \in B$, $x_i = b_i(h)$. We must show that $x_i$ is greater than or equal to $\mathcal{E}][\tau_i](\sigma, \sigma')$ and also greater than or equal to zero. Each $x_i$ is non-negative because each $b_i(h)$ is non-negative. By hypothesis, $t$ is a feasible execution trace of $P$ at height $h + 1$. Thus, each outermost recursive call in $t$ corresponds to an execution of $P$ of height at most $h$. (Note that, if a trace is at height exactly $h + 1$, one of its recursive calls must be at height exactly $h$, but the others may be at any height between 1 and $h$ (inclusive).) Thus, the inductive hypothesis,

$$\bigwedge_{i \in B} \forall v \in \mathcal{V}_{\tau_i}(P, h). v \leq b_i(h),$$

implies that the constraint relating $x_i$ to the value of $\tau_i$ is met at each call. We conclude that the relevant constraints on $\hat{t}$ are met, and therefore $\hat{t}$ is a feasible trace of $\hat{P}$.

As noted above, the output Alg. 7 is a set of invariants of $\hat{P}$, i.e., properties that hold in all feasible traces of $\hat{P}$. One such property is the conjunction of the selected candidate inequations shown in Eqn. (B.1). Let $i \in B$. We conclude that the $i$th selected candidate inequation holds in $\hat{t}$:

$$y_i \leq p'_i(x_i, \ldots, x_{i_m})$$

Let $\sigma$ and $\sigma'$ be the initial and final states of $\hat{t}$. By the construction of $\hat{t}$, we know that $\mathcal{E}[\tau_i](\sigma, \sigma') = y_i$. Thus,

$$\mathcal{E}[\tau_i](\sigma, \sigma') \leq p'_i(x_i, \ldots, x_{i_m})$$

By the construction of $\hat{t}$, we know that, for each $k \in B$, $x_k = b_k(h)$. Thus,

$$\mathcal{E}[\tau_i](\sigma, \sigma') \leq p'_i(b_i(h), \ldots, b_{i_m}(h))$$

As noted above, each $b_i(h)$ is non-negative for any $h \geq 1$. Thus, because $p'_i$ and $p_i$ are being evaluated at non-negative arguments, and $p_i$ was derived from $p'_i$ by dropping negative coefficients, we conclude that $p'_i(b_i(h), \ldots, b_{i_m}(h)) \leq p_i(b_i(h), \ldots, b_{i_m}(h))$ and therefore,

$$\mathcal{E}[\tau_i](\sigma, \sigma') \leq p_i(b_i(h), \ldots, b_{i_m}(h)). \quad (B.2)$$

The right-hand side Eqn. (B.2) matches the right-hand side of the defining recurrence for $b_i$, i.e.,
\[ p_i(b_i(h), \ldots, b_{i_m}(h)) = b_i(h + 1). \]

Thus,

\[ E[\tau_i](\sigma, \sigma') \leq b_i(h + 1). \]

Because we have shown this inequation to hold for each \( i \in B \), we conclude that

\[ \bigwedge_{i \in B} E[\tau_i](\sigma, \sigma') \leq b_i(h + 1). \] (B.3)

Recall that \( \sigma \) and \( \sigma' \) are the initial and final states of \( \hat{t} \), and that, by the construction of \( \hat{t} \), these are also the initial and final states of the original trace \( t \) of \( P \). But \( t \) was an arbitrary feasible execution trace of \( P \) of height up to \( h + 1 \), and therefore we conclude that Eqn. (B.3) holds if \( (\sigma, \sigma') \) are the initial and final states of any feasible execution trace of \( P \) of height up to \( h + 1 \). Thus,

\[ \bigwedge_{i \in B} \forall v \in V_{\tau_i}(P, h + 1). v \leq b_i(h + 1), \]

and the proof is complete. \( \square \)
Bibliography


M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. 2014. Alternating runtime and size complexity analysis of integer programs. In TACAS.


P. Cousot and R. Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In POPL.


A. Lal and T. Reps. 2006. Improving Pushdown System Model Checking. In CAV.


Pritom Rajkhowa and Fangzhen Lin. 2017. VIAP - Automated System for Verifying Integer Assignment Programs with Loops. In SYMASC.


S. Sankaranarayanan, H. Sipma, and Z. Manna. 2004a. Constraint-Based Linear-Relations Analysis. In SAS.


SVCOMP16 2016. 5th Int. Competition on Software Verification (SV-COMP16). https://sv-comp.sosy-lab.org/2016/


