Predictable Inter-Procedural Program Analysis via Almost-Commuting Transition Systems

Nikhil Pimpalkhare

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor Zachary Kincaid

September 2026

Abstract

Program analysis aims to automatically compute properties of programs from their source code. Since the problems that arise are frequently undecidable, the field has focused on developing sound analyses that over-approximate the true properties of a program. Modern program analyzers leverage complex heuristics to make these over-approximations as tight as possible, but these heuristics cause brittle performance: minor updates to codebases can cause large unintended swings in analysis precision.

To help tool users anticipate analysis outcomes, a recent line of work has focused on developing analysis techniques that are robust, and in particular *monotone*: if a program is a refinement of another, its analysis should be at least as precise. Monotonicity is a meta-property of the analysis technique; it gives tool users a way to reason about how certain code changes, like refactoring a loop or deleting a variable, will affect their analysis results, without requiring knowledge of the tool's internal mechanisms. However, development on monotone program analysis had been largely confined to the intra-procedural setting (to analyzing procedures without recursive procedure calls).

This thesis develops a framework for designing monotone inter-procedural program analyses. The key idea is to connect context-free reachability results for abstract machines to program analysis via abstract interpretation: we compute the best abstraction of a program within a given abstraction domain—its reflection—then exactly characterize the input-output behavior of that abstraction. We instantiate this recipe with vector addition systems with resets (VASR), a numerical computational model, establishing techniques to compute the VASR reflection of a program and the context-free reachability relation of a VASR. We extend these results to Lossy VASR, a more expressive variant, producing a program analysis tool which is competitive with the state-of-the-art in verification capabilities while providing formal robustness guarantees. We further investigate Semi-Linear VASRs, showing that

their induced procedure summary can be computed in polynomial time despite exponential abstraction size. Finally, we unify the reachability results across all three classes under almost-commuting transition systems (ACTS), algebraically characterizing the properties that make these context-free reachability problems tractable. ACTS define a rich frontier of abstraction domains for monotone inter-procedural program analysis.

Acknowledgements

I came to Princeton intending to get a master's, hoping to replace the years of college disrupted by the COVID pandemic. I had little idea what I was getting myself into.

I immediately began working with Zachary "Zak" Kincaid, who would advise me for the next 5 years. Zak's passion for mathematics was infectious, and his technical prowess was inspiring. As a mentor, Zak has been encouraging and meticulous; as a collaborator, he has been precise and ambitious. Working with Zak convinced me that a PhD was achievable, and moreover, might even be fun. Zak is the most influential teacher I have had in my life.

I made many friends for keeps in the first year: Jeremiah Coleman, Supantho Rakshit, Maya Gupta, Joy Zheng, Emily Alcazar, and Neil Agarwal. I was particularly close with Eshaan Nichani, Tommy Maldonado, Shai Caspin, and Morgan Nanez. Making friends with so many PhD students normalized the idea of pursuing one myself.

Living with Tommy and Eshaan was the highlight of my years at Princeton. Together, we ran miles, made music, and traveled the world.

The Programming Language group was an excellent academic cohort to be a part of. I learned a tremendous amount from Aarti Gupta, Mae Milano, David Walker, and Andrew Appel. I became good friends with Nicolas Koh, Andrew Johnson, Akash Gaonkar and Josh Cohen.

The later years brought dozens of new friendships with my move to the office at 194 Nassau. To name a few: Christopher Branner-Augmon, Anja Kalaba, Kostas Doumanidis, Natalie Popescu, Raye Kimmerer, Jianan Lu, Mohanna Shahrad, and Liz Austell. It was my honor to be your snack czar. I am extremely grateful towards Mae Milano for originating the move and facilitating this amazing community.

In my fourth year, Zak and I began collaborating with Tom Reps, one of the godfathers of modern program analysis. My discussions with Tom and Zak constantly reminded me about

how much more I can learn about the history and evolution of our field. Tom impressed upon me that finding a way to display a technique visually often aids in development of said technique and its presentation thereafter.

In my last year, I lived with Jeremiah, Joy, and Tommy in the so-called Grad House. I will never forget our family dinners.

Throughout the PhD, I was tremendously supported by a large network of friends from before I got to Princeton: Vikas Sharma, Elle Mahdavi, Nalin Chopra, Surya Gudivada, Michael Pennypacker, David Ness, Scott Nolan, and Chris Burton. Thank you for keeping me connected to the world.

My parents and sister deserve a special mention. Thank you for raising me. Thank you for supporting me. Thank you for keeping me sane. I love you guys.

To the friends I made along the way.

Contents

Chapter 1

Introduction

For a variety of reasons, developers want to answer questions about the behavior of computer programs before they run them. Could this program access private user data? Could this controller reach an unsafe configuration for its physical system? Does a given performance-improving transformation preserve program behavior? These questions are the purview of *program analysis*, a field which aims to automatically compute properties of computer programs from their source code.

A principal application of program analysis is proving the absence of bugs. Software is increasingly used in safety-critical systems in which unexpected behavior can have tremendous consequences. This has led to continued interest in *verified software*, which bundles computer programs with mathematical proofs of correctness and safety. By verifying a property, developers can guarantee that bugs which violate that property do not appear in their software. Program analysis enables the verification of software with minimal developer overhead; given a program and a property of interest, an analysis may be able to fully autonomously prove that the program meets the property.

Program analysis continues to be relevant in the age of artificial intelligence. Recent years have seen the role of the software engineer rapidly transition from program writer to

program checker as language models have improved at generating code. There is real interest in program analysis tools to help humans in this task [52]. Program analysis might be the key to making formal guarantees about the output of language models and verifying that it stays aligned with user intent.

Despite substantial progress in program analysis, a key challenge remains: modern analyzers often lack *predictability*. Minor changes such as refactoring code, reordering statements, or adding annotations can lead to large and unexpected differences in analysis outcomes. This unpredictability is documented across many of the most widely-used commercial program analysis tools. For example, Veracode acknowledges that a change in one part of a codebase can produce new findings in seemingly unrelated parts [32]; users of SonarQube report the same issue [31]. Coverity documents that bug counts can vary between scans of an unchanged codebase [7]. Users of Qodana have noted non-semantic annotations degrading code analysis quality [33]. This unpredictability complicates the use of program analysis tools in practice, as developers cannot reliably anticipate when a code change will cause an analysis to succeed or fail.

The motivation of this thesis is to address this gap by designing predictable program analyses: analyses that make formal guarantees on how program changes affect analysis outcomes, enabling users to anticipate the results of their changes without sacrificing state-of-the-art verification capability. We build on a line of work [21, 62, 66, 16] that achieves such predictability in the intra-procedural setting (programs without procedure calls), and extend these ideas to the inter-procedural setting (programs with procedure calls). This generalization represents a significant challenge, as existing predictable techniques are reliant on program structure that does not exist in the inter-procedural setting. As a result, new theoretical foundations are required to achieve predictable behavior in the presence of procedure calls.

1.1 Classical Approaches to Program Analysis

Many program analyses, particularly those concerned with safety verification, involve reasoning about the possible states that a program can reach. This thesis develops novel reachability results and applies them for program analysis. Many program properties of interest can be formulated as reachability queries.

To illustrate, consider the following program:

```
void copy(int* dst, int* src, int n) {
  for (int i = 0; i < n; i++)
    dst[i] = src[i]; // (*)
}
```

A property that one might want to prove for this program is memory safety: that the program does not access any memory that it is not supposed to. Proving memory safety for this program is equivalent to showing that execution cannot reach the marked statement (*) with a value of i that lies outside the valid bounds of the arrays `src` and `dst`. Reasoning about the reachable set of states at this point involves reasoning about the different paths through the program that could reach this point, as well as the infinite set of possible input values and array lengths that this function could be executed on.

Unfortunately, reasoning precisely about reachability is impossible. Rice's Theorem [57] states that all non-trivial reachability properties are undecidable. This means that for any property, every analysis technique necessarily fails for some programs. Program analyzers resolve this issue by being *over-approximate*: analysis techniques compute an over-approximation of the reachable set of states, either conclusively proving that a program meets a property or being inconclusive (either explicitly by returning unknown or implicitly by not terminating).

```
int add(int m, int n) {
    if (n == 0) {
        return m; }
    if (n > 0) {
        return add(m + 1, n − 1); }
    if (n < 0) {
        return add(m − 1, n + 1); }
    }
```

(a) An implementation of addition.

```
int nodes = 0; int leaves = 0;
void tc1(int n) {
    if (∗) {
        leaves += 1; }
    else {
        nodes += 1;
        tc1((n − 1) / 2);
        tc1((n − 1) / 2);
        }
    }
```

```
int nodes = 0; int leaves = 0;
void tc2(int n) {
    if (n <= 1) {
        leaves += 1; }
    else {
        nodes += 1;
        tc2((n − 1) / 2);
        tc2((n − 1) / 2);
        }
    }
```

(b) Two related tree-counting programs.

Figure 1.1: State-of-the-art verifiers UAutomizer [29] and Korn [19] exhibit unpredictable behavior on both examples. In Figure 1.1a, verifiers can prove $\texttt{add}(m, n) == m + n$ but not $m_1 > m_2 \implies \texttt{add}(m_1, n) > \texttt{add}(m_2, n)$, even though the former implies the latter. In Figure 1.1b, verifiers prove $\texttt{nodes} + 1 == \texttt{leaves} \vee \texttt{nodes} < \texttt{n}$ for `tc1` but not `tc2`, even though the executions of `tc2` are a subset of those of `tc1`.

These over-approximations of the reachable set of states are called invariants. Many sophisticated analysis techniques use complex heuristics in order to guide their search for invariants. The effectiveness of these heuristics depends on intricate algorithms implemented within the tool, and can vary significantly across different implementations of the same technique. As a result, the behavior of the analysis can be difficult to predict.

Figure 1.1 identifies examples in which state-of-the-art verifiers exhibit counterintuitive and unpredictable behavior. Figure 1.1a shows an implementation of addition; state-of-the-art verifiers can verify that this implementation computes addition, but fail to verify that the function is increasing in its first input. This is surprising because the former property implies the latter and weaker properties should intuitively be easier to prove. Figure 1.1b shows two related tree-traversal programs, `tc1` and `tc2`, which count nodes and leaves, differing only in their branching condition. Intuitively, every execution of `tc2` is also an execution of `tc1` because `tc1` places fewer constraints on when branches can be taken. Then, since the executions of `tc2` are a subset of `tc1`, it seems intuitive that it would be easier to prove properties of `tc2` than `tc1`, as it requires reasoning about fewer executions. However, the figure identifies a property that verifiers can prove for `tc1` but cannot for `tc2`.

Ultimately, these counterintuitive phenomena are a symptom of the fact that program analysis techniques generally do not provide any guarantees on their behavior beyond soundness (and in some cases, termination). However, users expect more from their tools; they want to understand and anticipate the tool's behavior without needing to reason about the underlying algorithm [33, 31]. For program analysis designers, this raises the question: *what behavioral guarantees are attainable while retaining state-of-the-art precision and performance?*

Many program analysis techniques, including the ones described in this thesis, are unified under the framework of *abstract interpretation* [14]. Rather than reasoning precisely about the set of states that a program can reach, abstract interpretation reasons about an

abstraction of that set, retaining some information and discarding the rest. These simplified representations are chosen such that reasoning about reachability is tractable and yields an over-approximation of the behavior of the original program. Moreover, the framework provides formal soundness guarantees, guaranteeing that an analysis never misses any states.

A classic example of an abstract interpretation is *interval analysis*, which abstracts program states by associating each variable with an interval $[a, b]$ approximating the range of values it may assume during any execution. The program is executed over these abstract states, yielding, at each program point, an invariant that maps each variable to an interval over-approximating the values it may assume.

This abstract interpretation encounters complications when handling program constructs such as loops and procedure calls. For example, in interval analysis, a loop that expands an interval on each iteration can cause the abstract state to grow indefinitely, never converging. Analyzers resolve this by employing *widening*, a technique that enforces convergence by extrapolating the limit at select program points to obtain sound over-approximations. In the interval domain, for instance, widening may replace bounds that have changed between iterations with ∞, ensuring the sequence of abstract states stabilizes in finitely many steps.

Widening is non-monotone [15]: a more precise input to the analysis does not necessarily yield a more precise output. In the interval domain, for instance, a smaller interval may be widened to ∞ while a larger interval might retain its original bounds. Consider the following loop:

while $(*)$ {
  if (x == 0) {x = 1}
  else {x = 2}}

If the abstraction for x at the beginning of the loop is the interval $[0, 0]$, then standard interval analysis will widen the interval to $[0, \infty]$ as the interval expands in successive iter-

ations of the loop. However, if the abstraction for x at the beginning of the loop is $[0, 2]$, then standard interval analysis will compute $[0, 2]$ as the state abstraction after the loop. This is counter-intuitive: a more precise precondition leads to a less precise postcondition. Non-monotonicity compounds in larger programs, making the behavior of widening-based analyses difficult to predict.

Monotonicity, then, is a desirable property for a program analysis technique to have: a more precise program should have a more precise analysis. What are the key ingredients in an abstract interpretation which guarantee robustness properties like monotonicity for the resulting analysis?

1.2 Prior Work on Predictable Program Analysis

A recent line of work [62, 65, 66, 16] has aimed to develop new program analysis techniques which are predictable by design. By making behavioral guarantees beyond soundness, these techniques give users mathematical properties with which to reason about how their tools will work without needing to understand the internal algorithms of these tools. In this subsection, we examine the guarantees these techniques provide and the key principles that enable them.

In this thesis, we view a program analysis technique as a function that maps any procedure P to a procedure summary $S(P)$. An advantage of this characterization is a *locality* principle. Formalizing analysis as procedure summarization means that the precision of the summary $S(P)$ is independent of unrelated procedures in the same program and the property being proved. Figure 1.1a illustrated a failure of locality in state-of-the-art verifiers. UAutomizer and Korn verify $\texttt{add}(m, n) = m + n$, yet fail to verify the weaker property $m_1 > m_2 \implies \texttt{add}(m_1, n) > \texttt{add}(m_2, n)$. Since the former implies the latter, the only explanation is that the property being proved is influencing how `add` is analyzed, and therefore that the analysis

Original program

```
int x := 0, y := 0;
while (x < n) {
    x := x + 1;
    y := y + 2;
}
// goal: prove y = 2*n
```

Loop body transition:

$x' = x + 1 \ \wedge \ y' = y + 2$

Closed form after k iterations:

$x^{(k)} = x_0 + k \ \wedge \ y^{(k)} = y_0 + 2k$

Postcondition :

$x = n \ \wedge \ \boxed{y = 2n}$ ✓

Figure 1.2: A counting program abstracted by Compositional Recurrence Analysis (CRA) [21, 4]. CRA computes the best abstraction of the program into a family of affine recurrences and uses a precise summary of the repeated iteration of that recurrence as an over-approximation of the program.

conducted by these tools is non-local. A local analyzer produces the same summary regardless of the target property; so if it can prove a strong property, it can prove any weaker one.

A robustness property of a program analysis technique describes how relations between programs correspond to relations between their analyses. The robustness property that we focus on in this thesis is *monotonicity*, which states that if a procedure P is a refinement of another procedure P', then the summary $S(P)$ should be at least as precise as $S(P')$. This implies, for instance, that strengthening the preconditions of a procedure can only strengthen the resulting summary. Figure 1.1 shows a violation of this property. `tc2` is a refinement of `tc1`: `tc1` branches non-deterministically while `tc2` uses the guard $n \leq 1$, making its behavior a strict subset of `tc1`'s. Nevertheless, UAutomizer and Korn verify the property in the caption for `tc1` but fail on `tc2`. A monotone analyzer produces more precise analyses for more refined programs, and so would be able to prove any property for `tc2` that it could for `tc1`.

Figure 1.2 displays Compositional Recurrence Analysis (CRA) [21, 4], an abstract interpretation-based program analysis technique that can be instantiated to be monotone. CRA computes invariants for loops by abstracting each iteration of the loop as a set of

linear recurrences describing how the next state can be computed from the current. CRA then computes a closed form representing the repetition of the recurrence over an arbitrary number of iterations and uses this as an invariant for the loop. CRA compositionally builds summaries for whole programs, computing summaries of each loop in this manner and composing them to form an end-to-end summary - this approach guarantees the locality of the end-to-end technique.

CRA can be instantiated to be monotone, as illustrated in Figure 1.2, owing to two facets of this instantiation. First, it extracts all linear recurrences of the target form that hold across loop iterations, yielding a best abstraction within its domain. Second, these recurrences admit exact closed-form solutions. CRA exemplifies a more general "best abstraction" recipe for predictable program analyses: (1) the computed abstraction is *best*, in the sense that it is at least as precise as any other abstraction in its class, and (2) the reachability relation of the abstraction—the set of input/output state pairs reachable by executing it—can be computed exactly.

Many robust program analyses [62, 65, 66, 16] have used this best abstraction recipe in order to produce monotone and local intra-procedural program analyses under the Algebraic Program Analysis (APA) framework [36]. Analyses within this framework compute a regular expression describing the set of paths through a program and interpret this expression with a suitable semantic algebra. The nested structure of regular expressions enables complex analyses to be broken up into a series of simpler subproblems. By reasoning about programs compositionally in this way, these techniques guarantee locality. By choosing the most precise abstraction of the program within a given abstract domain, and precisely analyzing it, these techniques guarantee monotonicity.

However, this success has been confined to the intra-procedural setting; that is, to analyzing programs without recursive procedure calls. Recursive procedure calls make the set

of program paths context-free rather than regular[1], so it is not possible to compute a regular expression describing the paths through an inter-procedural program. Regular languages decompose algebraically, allowing APA analyses to reduce a whole-program problem to a structured series of subproblems; context-free languages do not, and inter-procedural reachability must instead be confronted as a single global problem. APA-style analyses therefore lack the tools to handle programs with recursive procedures in a uniform way; to analyze procedure calls, they typically resort to iterative methods that employ widening and are therefore non-monotone.

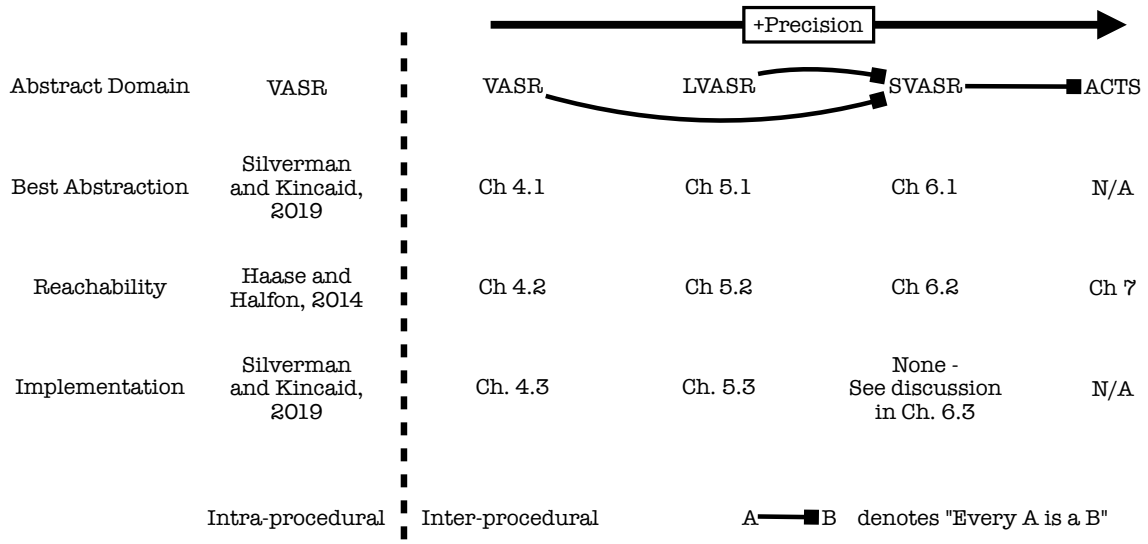
This thesis extends both APA and the best abstraction recipe to the inter-procedural setting by computing the *context-free language* of paths through a program and interpreting it with a program abstraction that is the best of its class, in the sense that this abstraction models the semantics of the program at least as precisely as any other abstraction in its class. The central question of this thesis is which abstract domains admit both best abstractions and tractable inter-procedural reachability analysis. Many of the abstract domains used in APA-style analyses do not have obvious inter-procedural reachability extensions; this thesis begins to answer which ones do. We ask:

> What models of recursive programs admit (1) best abstractions and (2) decidable exact reachability? How can we build practical analyzers from the resulting decision procedures?

1.3 Contributions

The following table diagrams the novel contributions of this thesis, presented in Chapters 4, 5, 6, and 7.

[1]Recursive procedure calls introduce a matched call/return structure that must be tracked along program paths: each call site must be matched with its corresponding return. This set of matching call/return sequences form a context-free language; see Reps et al. [54]



The immediate predecessor of the work in this thesis is a predictable APA-style program analysis technique which abstracts and analyzes programs as *vector addition systems with resets* (VASRs) [62]. Vector Addition Systems are a classical sub-Turing-complete model of computation historically used to model parallel processes and distributed systems. VASRs operate over a set of rational-valued counters; operations optionally reset these counters to zero, then add in some constant. Silverman and Kincaid [62] developed an algorithm that computes the best VASR abstraction of a loop. They leveraged a result by Haase and Halfon [27] that one can compute a logical formula defining the intra-procedural reachability set of a VASR on this abstraction to produce loop summaries. Instead of approximating the set of states at each control point, as with interval analysis, this work approximated the transition step associated with each operation of the program. This technique extracts all "VASR-like" updates that occur within a loop and precisely characterizes what program states are reachable via these updates; this is an over-approximation of the true reachability of the loop. Silverman and Kincaid implemented and evaluated their technique, showing it to

be precise and performant in practice. Their intra-procedural analysis has many robustness properties; namely, it is monotone and local.

Chapter 4 develops a predictable inter-procedural program analysis technique which abstracts and analyzes programs as VASRs. We extend the best abstraction technique of Silverman and Kincaid, which abstracts a single loop, to abstract whole procedures which may contain recursive procedures. We achieve this extension via a divide-and-conquer approach: we compute the best VASR abstraction of each operation in a program and devise a technique to combine abstractions while preserving the optimality of the combination. We additionally extend the result of Haase and Halfon to compute the context-free reachability relation of VASRs, representing an advance in the theory of vector addition systems. Our technique is robust, in that it is monotone and local. We implemented and evaluated our technique, but we found it to be uncompetitive with existing verification tools, largely due to its inability to model conditional branching.

Chapter 5 improves the precision of our analysis technique by extending it to *Lossy VASRs*, a non-deterministic extension of VASRs, while retaining our robustness properties. Best Lossy VASR abstractions are always at least as precise as best VASR abstractions, and are frequently better suited to modeling conditional branching. We implemented and evaluated our technique, showing that our technique computes more precise summaries than existing abstract interpreters and that our tool's verification capabilities are comparable with state-of-the-art software model checkers. Our tool is able to make powerful robustness guarantees without sacrificing precision.

Inspired by this success, we went further by investigating *Semi-Linear VASR* in Chapter 6, an extension which leverages semi-linear sets, a class of integer sets with periodic structure, to further boost the precision of our technique. The size of the best Semi-Linear VASR abstraction of programs is always exponential in the size of the program. This would make analysis prohibitively expensive since a direct reachability analysis of the abstraction

would require exponential time. However, we can exploit symmetries in these abstractions to compute the induced over-approximate reachability relation of the original program in polynomial time, removing a key barrier to this extension being practically useful. We conclude the chapter by identifying remaining obstacles standing in the way of a full implementation.

Our final content chapter, Chapter 7, unites the reachability results of the previous chapters under a single roof. We investigated the algebraic properties of VASRs, LVASRs, and SVASRs that make reachability queries tractable, and uncovered a rich new frontier: almost-commuting transition systems (ACTS). We show that for members of this class the context-free language reachability relation can be effectively represented as a logical formula. We identify several concrete families of this class for which this formula allows us to automatically answer reachability queries and whose state transitions resemble common programming constructs. VASRs and LVASRs are almost-commuting transition systems, and SVASRs are reducible to ACTS (Section 7.3). This work can be viewed as a backend for designing inter-procedural analyses: any new abstraction procedure targeting a suitable class of almost-commuting transition systems can immediately be plugged into our analyzer to produce over-approximate procedure summaries. Together, these contributions support the following thesis statement:

> ACTS reachability expands the frontier of decidable inter-procedural reachability, enabling practical inter-procedural analyzers with provably predictable behavior.

This thesis is organized as follows. Chapter 2 discusses common background and notation. Chapter 3 introduces the core technical recipes for monotone procedure summarization and best abstractions via VAS, a reset-free domain that is simpler to follow. Chapters 4, 5, 6, and7 contain the aforementioned content on VASR, LVASR, SVASR, and ACTS. Chapter 8 describes related work. Chapter 9 concludes.

The contents of Chapters 4 and 5 were published at OOPSLA 2024 [48]. The contents of Chapter 6 were published at RP 2024 [49]. The contents of Chapter 7 were published at POPL 2026 [50].

Chapter 2

Background

This chapter presents common background concepts and notation.

2.1 Program Model

The techniques of this thesis apply the philosophy of Algebraic Program Analysis [36] to the inter-procedural setting. Our aim is to compute procedure summaries, logical formulas over-approximating the input-output behavior of each procedure. We do so by interpreting a representation of the language of paths through a procedure according to an abstract domain. We define our input program model to be a context-free grammar representing the language of paths through a procedure and a transition assignment describing the semantics of each path.

It is widely known that the language of paths through procedures with recursive calls are context-free languages [30], generated by **context-free grammars**. A context-free grammar $G = \langle N, \Sigma, R, n_0 \rangle$ consists of a finite set of nonterminals N, a finite alphabet Σ, a set of production rules $R \subseteq N \times (\Sigma \cup N)^*$, and a designated start symbol $n_0 \in N$. We denote production rule $\langle \alpha, \beta \rangle \in R$ as $\alpha \Rightarrow \beta$. An application of production rule $\alpha \Rightarrow \beta$ replaces a

single occurrence of α with β: $w_1 \alpha w_2 \to w_1 \beta w_2$. The language corresponding to a nonterminal n is the set $\mathcal{L}_G(n) \triangleq \{w \in \Sigma^* : n \to^* w\}$. The language of the grammar $\mathcal{L}(G)$ is the language of its start symbol, $\mathcal{L}_G(n_0)$.

We understand the semantics of a program as a **labeled transition system**. A labeled transition system[1] over a finite set of variables X and a finite alphabet Σ is a pair $T = \left\langle \mathbb{Q}^X, \to_T \right\rangle$ where $\mathbb{Q}^X$ is a state space and $\to_T \subseteq \mathbb{Q}^X \times \Sigma \times \mathbb{Q}^X$ is a labeled transition relation. A program state is written as $\rho \in \mathbb{Q}^X$ and maps each element of $x \in X$ to a rational value $\rho(x)$. We use the following notation for transition systems:

- For a character $s \in \Sigma$, $\rho \xrightarrow{s}_T \rho'$ denotes that $\langle \rho, s, \rho' \rangle$ belongs to transition relation $\to_T$.
- For a word $s_1 \dots s_n \in \Sigma^*$, $\rho \xrightarrow{s_1 \dots s_n} \rho'$ denotes there exist states $\rho_0 \dots \rho_n$ such that $\rho = \rho_0 \xrightarrow{s_1}_T \rho_1 \xrightarrow{s_2}_T \dots \xrightarrow{s_n}_T \rho_n = \rho'$.
- For a language $L \subseteq \Sigma^*$, $\rho \xrightarrow{L}_T \rho'$ denotes that $\rho \xrightarrow{w}_T \rho'$ for some $w \in L$.

Transition formulas are logical formulas which describe a state update and are an expressive way to specify labeled transition systems. In this thesis, we restrict our attention to transition formulas specified in Linear Integer/Rational Arithmetic (LIRA), a logical fragment for which satisfiability is decidable. The syntax of LIRA formulas is given as:

Variables	x	$\in Var$
Constants	c	$\in \mathbb{Q}$
Terms	t	$::= x \mid c \mid c \cdot x \mid t + t'$
Formulas	ϕ	$::= t \leq t' \mid \neg\phi \mid \phi \wedge \phi' \mid \phi \vee \phi' \mid \phi \to \phi' \mid \forall x.\, \phi \mid \exists x.\, \phi$

[1] We restrict our attention to transition systems with state spaces which are finite-dimensional vector spaces over the rationals.

A transition formula over a set of variables X is a satisfiable LIRA formula F whose free variables range over variables X and primed copies X'. The set of all transition formulas over X is written as $\mathbf{TF}(X)$. Given states $\rho, \rho' \in \mathbb{Q}^X$, we write $[\rho, \rho'] \models F$ if F holds when every occurrence of $x \in X$ is replaced with $\rho(x)$ and every occurrence of $x' \in X'$ is replaced with $\rho'(x)$; the intuitive meaning of $[\rho, \rho'] \models F$ is that ρ steps to ρ' along the transition specified by F. For example, the transition formula corresponding to the program operation `x := 2y + 1` is the formula $x' = 2y + 1 \wedge y' = y$ (assuming `x` and `y` are the only variables in the program). We say that a binary relation $R \subseteq \mathbb{Q}^X \times \mathbb{Q}^X$ is **definable** if there exists $F \in \mathbf{TF}(X)$ such that $[\rho, \rho'] \models F$ if and only if $\langle \rho, \rho' \rangle \in R$.

The program model that we work with for the remainder of this thesis is a context-free grammar generating the paths through a program and a **transition assignment** describing the semantics of the program. A transition assignment $\#f : \Sigma \to \mathbf{TF}(X)$ defines a labeled transition system $\left\langle \mathbb{Q}^X, \to_{\#f} \right\rangle$ over alphabet Σ where $\rho \xrightarrow{s}_{\#f} \rho'$ if and only if $[\rho, \rho'] \models \#f(s)$. Given a subalphabet $\Sigma_1 \subseteq \Sigma$, we write $\#f|_{\Sigma_1} : \Sigma_1 \to \mathbf{TF}(X)$ to denote the **restriction** of $\#f$ to Σ_1, defined by $\#f|_{\Sigma_1}(s) = \#f(s)$ for all $s \in \Sigma_1$.

The semantics of a program specified as a context-free grammar G and a transition assignment $\#f$ is understood as the context-free reachability relation $\xrightarrow{\mathcal{L}(G)}_{\#f}$. The programs that are represented by this model can be thought of as numerical C-like programs over global variables. This model is limited in two respects: LIRA cannot express many programming constructs (e.g., memory reads and writes), and we do not model local variables. Analyzing programs using such features would first require abstraction into our model. Despite these limitations, this model is worth studying because many real-world programs can be soundly approximated within it. For instance, an array variable could be modeled as an integer length variable, resulting in a representation suitable for verifying that accesses remain within bounds (as was done in [11]).

Example 2.1.1. This example aims to show how procedures written in a typical programming language can be represented as a context-free language and transition formula mapping. Consider the following integer model of a C program which traverses a binary tree and writes the value of each internal node to an intermediate buffer, flushing the buffer to disk at every leaf.

```
1  int mem_ops, buf;
2  void save_tree() {
3      buf += 1;                          // a
4      if (*) {
5          mem_ops += buf; buf = 0;       // b
6      } else {
7          save_tree();
8          save_tree();
9      }}
```

The global variable `mem_ops` counts the number of integers written to disk and `buf` represents the current length of the intermediate buffer.

A grammar generating the language of paths through this program is:

$$G = \langle \{P\}, \{a, b\}, \{P \Rightarrow aPP, P \Rightarrow ab\}, P \rangle$$

The nonterminal P represents a call to `save_tree`, and the terminals a and b correspond to lines 3 and 5, respectively. The production rules of the grammar identify the two paths through the procedure - either a then b (the base case) or a then two recursive calls to `save_tree`.

The following transition formula mapping $\not\!\!f$ assigns a LIRA formula over the globals $\{\texttt{mem\_ops}, \texttt{buf}\}$ to each terminal character. The two transition formulas are:

$$\not\!\!f(a) \triangleq \begin{pmatrix} \texttt{buf}' = \texttt{buf} + 1 \\ \wedge\, \texttt{mem\_ops}' = \texttt{mem\_ops} \end{pmatrix}$$

$$\not\!\!f(b) \triangleq \begin{pmatrix} \texttt{buf}' = 0 \\ \wedge\, \texttt{mem\_ops}' = \texttt{mem\_ops} + \texttt{buf} \end{pmatrix}$$

Edge a encodes the increment $\texttt{buf}' = \texttt{buf} + 1$ from line 3. Edge b encodes the two operations on line 5: $\texttt{mem\_ops}' = \texttt{mem\_ops} + \texttt{buf}$ and $\texttt{buf}' = 0$.

2.2 Problem Setup

We model a program as a context free language $\mathcal{L}(G) \subseteq \Sigma^*$ interpreted according to a transition assignment $\not\!\!f : \Sigma \to \mathbf{TF}(X)$. We are interested in computing an over-approximation of the $\mathcal{L}(G)$-reachability of the transition system defined by $\not\!\!f$. We represent this over-approximation as a transition formula $F \in \mathbf{TF}(X)$ such that if $\rho \xrightarrow{\mathcal{L}(G)}_{\not\!\!f} \rho'$, then we have that $[\rho, \rho'] \models F$. This summary F can be used to prove properties of the program: if F implies a property P holds, then P holds for the program because F holds for all of its possible executions.

Therefore, we can view our program analysis technique as a function S which maps a context-free grammar G and transition assignment $\not\!\!f : \Sigma \to \mathbf{TF}(X)$ to a **procedure summary** $S(G, \not\!\!f) \in \mathbf{TF}(X)$. The following definitions describe properties that hold for the program analysis techniques presented in this thesis. We go further than soundness: we guarantee that our techniques are monotone and local.

Definition 1. Consider a context-free language $\mathcal{L}(G) \subseteq \Sigma^*$ and a transition mapping $\not\!\!f : \Sigma \to \mathbf{TF}(X)$.

A **sound** program analysis S has the property that:

$$\left(\rho \xrightarrow{\mathcal{L}(G)}_{\mathit{ff}} \rho'\right) \implies [\rho, \rho'] \models S(G, \mathit{ff})$$

Definition 2. Consider a context-free language $\mathcal{L}(G) \subseteq \Sigma^*$ and two transition mappings $\mathit{ff} : \Sigma \to \mathbf{TF}(X)$ and $\mathit{ff}' : \Sigma \to \mathbf{TF}(X)$.

A **monotone** program analysis S has the property that:

$$(\forall s \in \Sigma. \mathit{ff}(s) \models \mathit{ff}'(s)) \implies S(G, \mathit{ff}) \models S(G, \mathit{ff}')$$

Our setup also ensures a form of **locality**: because we represent programs as a pair (G, ff), the summary $S(G, \mathit{ff})$ depends only on the operations that actually appear in the language $\mathcal{L}(G)$. In particular, changes to the behavior of operations that do not appear in $\mathcal{L}(G)$ cannot affect the resulting summary. Locality in this sense is therefore not an additional property to verify, but a consequence of how we have chosen to formalize the problem.

2.3 Linear Abstractions of Transition Systems

This thesis presents techniques for computing the context-free reachability relation for several classes of transition systems and leverages these techniques to compute procedure summaries. Computing the context-free reachability relation of arbitrary transition systems is impossible, even when we require that the transitions are definable as LIRA formulas. We may see that this is true by observing that we can straightforwardly encode a Minsky counter machine [43], a simple Turing-complete model of computation, into LIRA formulas.

Then, how do we use our reachability results to compute procedure summaries of systems within our program model? The answer is that we abstract the program model as another program which lies in a class for which the context-free reachability relation is computable.

We then compute a transition formula defining the context-free reachability relation for this abstraction and use it to compute an over-approximation of the context-free reachability relation of the original program. This process follows the abstract interpretation recipe that lies at the heart of many modern program analyses. In this subsection, we formalize what an abstraction is and how we may use it to compute approximate summaries.

A **linear term** over variables X is a linear combination $\sum_{x \in X} a_x x$ with coefficients $a_x \in \mathbb{Q}$; we denote the set of all such terms $\mathit{LinTerm}(X)$, and this space is a vector space (isomorphic to $X \to \mathbb{Q}$).

An abstraction $\langle f, \mathit{ff}' \rangle$ of a transition system ff consists of another transition system ff' and a simulation f from ff to ff'. In this thesis, we restrict our attention to linear simulations. The standard notion of a linear simulation between labeled transition systems ff over variables X and ff' over variables Y is a linear map $f : \mathbb{Q}^X \to \mathbb{Q}^Y$ such that for all $s \in \Sigma$, if $\rho \xrightarrow{s}_{\mathit{ff}} \rho'$ then $f(\rho) \xrightarrow{s}_{\mathit{ff}'} f(\rho')$. Intuitively, this means that every transition step $\xrightarrow{s}_{\mathit{ff}}$ in ff is simulated by a corresponding step $\xrightarrow{s}_{\mathit{ff}'}$ in ff'.

Since we work with transition systems in which transitions are specified as transition formulas, we adopt an equivalent formulation of linear simulations in terms of substitutions; this is notationally convenient because we can describe how variables of each system relate to each other rather than defining a linear map. A substitution $\sigma : Y \to \mathit{LinTerm}(X)$ uniquely corresponds to a linear map $f : \mathbb{Q}^X \to \mathbb{Q}^Y$ via $f(\rho)(y) = \rho(\sigma(y))$; the two are duals. Given a substitution, $\sigma : Y \to \mathit{LinTerm}(X)$, we use $\bar{\sigma} : Y \cup Y' \to \mathit{LinTerm}(Y) \cup \mathit{LinTerm}(Y')$ to denote its extension to primed variables: $\bar{\sigma}(y') = \sigma(y)'$. Given $F \in \mathbf{TF}(Y)$, we write $F[\bar{\sigma}] \in \mathbf{TF}(X)$ to denote the formula obtained by replacing each variable according to $\bar{\sigma}$. A **linear simulation** between $\mathit{ff} : \Sigma \to \mathbf{TF}(X)$ and $\mathit{ff}' : \Sigma \to \mathbf{TF}(Y)$ is then a substitution $\sigma : Y \to \mathit{LinTerm}(X)$ such that $\mathit{ff}(s) \models \mathit{ff}'(s)[\bar{\sigma}]$ for all $s \in \Sigma$.

We say that abstraction $\langle \sigma, \mathit{ff}' \rangle$ **tracks** term t if there exists some variable y of ff' such that $\sigma(y) = t$. The terms tracked by an abstraction determine what information it retains

about the original system. A best abstraction maximizes the space of tracked terms; we formalize this in Chapter 3.

Formulating simulations as substitutions is convenient for transferring reachability results. When σ is a simulation from $\mathbb{ff}$ to $\mathbb{ff}'$, if $[\rho, \rho'] \models F \iff \rho \xrightarrow{L}_{\mathbb{ff}'} \rho'$, then $F[\bar{\sigma}]$ soundly over-approximates $\xrightarrow{L}_{\mathbb{ff}}$.

We use either a linear map or a substitution to define linear simulations wherever convenient; we adopt the convention that alphabetical characters f, g, h, i denote linear maps and greek letters σ, κ, τ denote substitutions.

Example 2.3.1. Let $X = \{x_1, x_2\}$, $Y = \{y\}$, $\Sigma = \{a\}$, and consider the transition assignments:

$$\mathbb{ff}(a) \triangleq x_1' = x_2 \wedge x_2' = x_1 \qquad \mathbb{ff}'(a) \triangleq y' = y$$

The substitution $\sigma(y) = x_1 + x_2$ is a linear simulation from $\mathbb{ff}$ to $\mathbb{ff}'$: we have $\mathbb{ff}'(a)[\bar{\sigma}] = (x_1' + x_2' = x_1 + x_2)$, and indeed $\mathbb{ff}(a) \models \mathbb{ff}'(a)[\bar{\sigma}]$.

The choice of simulation class determines what terms an abstraction can track. A linear simulation allows the abstract system to track linear terms of the original state space. They are also tractable: because simulations are linear maps, the algebraic questions that arise when computing and combining abstractions reduce to standard problems in linear algebra.

2.4 Category Theory for Abstract Interpretation

As described in Chapter 1, the motivation of this thesis is to design program analyses which have robustness properties guaranteeing that related programs will have related analyses. Our work is presented within an alternative framing of the classic abstract interpretation framework [14] using category theory. The goal of this alternative framing is to do for robust-

ness what the classic abstract interpretation framework does for soundness. In this section, we aim to introduce the category theory concepts that we use via analogy to classic abstract interpretation concepts. For an introduction to the classic abstract interpretation framework, see [14] or [44]; for a formalization of this category theoretic framing on abstract interpretation, see [37]; for more on category theory, see [58].

The following table displays the category theory concepts used in this thesis and their counterparts in typical abstract interpretation. In this section we will formally define each concept, then lend intuition to how they extend their counterpart.

Category Theory	**Abstract Interpretation Analogue**
Category $\mathbf{TS}(\Sigma)$	Concrete domain/lattice
Object of category	Element of domain
Arrow $X \to Y$	Lattice order $X \sqsubseteq Y$
Subcategory $\mathbf{Abs}(\Sigma)$	Abstract domain/lattice
Reflection	Optimal abstraction
Reflective Subcategory	Existence of α in Galois connection
Functor	Translation between lattices
Pushout	Join

Table 2.1: Category-theoretic concepts and their abstract interpretation analogues.

A **category C** consists of:

- A collection of objects $obj(\mathbf{C})$
- For each pair of objects $X, Y \in obj(\mathbf{C})$, a collection $\mathbf{C}(X, Y)$ of arrows
- For each triple of objects $X, Y, Z \in obj(\mathbf{C})$, a composition operator $\circ : \mathbf{C}(X, Y) \to \mathbf{C}(Y, Z) \to \mathbf{C}(X, Z)$
- For each object $X \in obj(\mathbf{C})$, an arrow $1_X \in \mathbf{C}(X, X)$

such that composition is associative, and for each $X, Y \in \mathit{obj}(\mathbf{C})$ and each $f \in \mathbf{C}(X, Y)$ we have $f \circ 1_X = 1_Y \circ f = f$.

Categories can be used as a generalization of lattices from abstract interpretation, in which the objects represent the elements of the lattice and the arrows represent ordering relations. An important category in this thesis is the category $\mathbf{TS}(\Sigma)$ in which the objects are labeled transition systems over the finite alphabet Σ defined by transition assignments and in which the arrows denote linear simulations. This category can be thought of as our concrete domain in abstract interpretation terminology.

A **subcategory D** of category **C** is a category such that:

- its objects $\mathit{obj}(\mathbf{D})$ are contained within $\mathit{obj}(\mathbf{C})$
- for all objects $X, Y \in \mathit{obj}(\mathbf{D})$, we have that $\mathbf{D}(X, Y)$ is contained within $\mathbf{C}(X, Y)$
- all identity arrows and arrow compositions are preserved

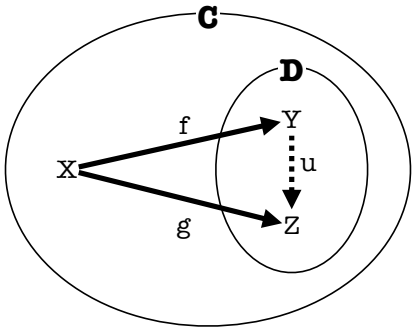
A subcategory **D** of category **C** is full if for each pair of objects $X, Y \in \mathit{obj}(\mathbf{D})$ we have that $\mathbf{D}(X, Y) = \mathbf{C}(X, Y)$. In this thesis, subcategories are always full.

In this thesis, we define multiple subcategories $\mathbf{Abs}(\Sigma)$ of $\mathbf{TS}(\Sigma)$ which are analogous to abstract domains for abstract interpretation. Elements of our domain of abstractions consist of both an abstract object in $\mathbf{Abs}(\Sigma)$ and a simulation (arrow) from the concrete object to the abstract object. Although it is possible to define a lattice structure over such abstractions, reasoning about robustness properties in this setting naturally reintroduces category-theoretic structure. It is useful, instead, to start from a category theory foundation as we do here.

Consider a category **C** and a subcategory **D**. A **D**-reflection of an object $X \in \mathit{obj}(\mathbf{C})$ consists of an object $Y \in \mathit{obj}(\mathbf{D})$ and an arrow $f \in \mathbf{C}(X, Y)$ fulfilling the following universal

property: For any other object $Z \in obj(\mathbf{D})$ and arrow $g \in \mathbf{C}(X, Z)$, there exists an arrow $u \in \mathbf{C}(Y, Z)$ such that $u \circ f = g$ [2].

A reflection is depicted in the following commutative diagram. In such diagrams, nodes represent objects and labeled edges represent arrows. A diagram *commutes* if every directed path between the same pair of nodes composes to the same arrow; here, commutativity means $u \circ f = g$. Dashed arrows denote arrows whose existence is guaranteed by a universal property — in this case, the reflective property.



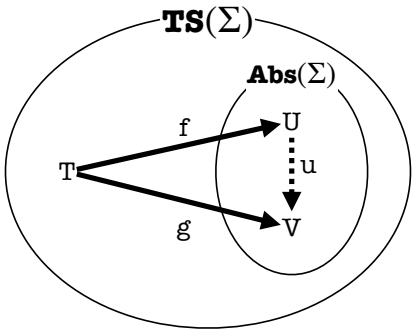
Given a category $\mathbf{C}$ and a subcategory $\mathbf{D}$, we say $\mathbf{D}$ is a reflective subcategory of $\mathbf{C}$ if any object $X \in obj(\mathbf{C})$ has a $\mathbf{D}$-reflection[3]. Reflections generalize the notion of an optimal abstraction from abstract interpretation. The property of a subcategory being reflective is analogous to the existence of an α function in a Galois connection, which computes the most precise abstract element for any concrete element.

We now interpret the notion of reflections in our category of transition systems. Let $\mathbf{Abs}(\Sigma)$ be a subcategory of $\mathbf{TS}(\Sigma)$ for which the context-free reachability relation is computable. Let $T \in obj(\mathbf{TS}(\Sigma))$ be a transition system denoting the semantics of a program of interest. The $\mathbf{Abs}(\Sigma)$-reflection of T is a transition system $U \in obj(\mathbf{Abs})(\Sigma)$ and a linear

[2]The standard definition of a reflection additionally requires that u is unique, but this uniqueness is not important for our purposes. For the remainder of the thesis, we use the weakened definition provided here.

[3]The standard definition requires only that reflections exist; we additionally require that they are computable. Since we use reflections to compute program analyses, a non-constructive existence argument would not suffice.

simulation f from T to U. The universality property in the context of transition systems means that for any other abstract transition system $V \in \mathit{obj}(\mathbf{Abs}(\Sigma))$ and linear simulation g from T to V, there exists a simulation u from U to V such that $u \circ f = g$.



Simulation can be thought of as a decrease in precision; U and V lose precision compared to the original program T, and the universality property states that V loses precision compared to the $\mathbf{Abs}(\Sigma)$-reflection U. It is in this sense that the $\mathbf{Abs}(\Sigma)$-reflection is the best abstraction within the subcategory.

Functors allow us to describe how categories relate to each other. Functors play a role analogous to structure-preserving translations between abstract domains. Given two categories $\mathbf{C}$ and $\mathbf{D}$, a functor F is a map sending every object $X \in \mathit{obj}(\mathbf{C})$ to an object $F(X) \in \mathit{obj}(\mathbf{D})$ and every arrow $f \in \mathbf{C}(X, Y)$ to an arrow $F(f) \in \mathbf{D}(F(X), F(Y))$ such that:

1. For any $f \in \mathbf{C}(X, Y)$ and $g \in \mathbf{C}(Y, Z)$, we have that $F(g \circ f) = F(g) \circ F(f)$
2. For each object $X \in \mathit{obj}(\mathbf{C})$, we have $F(1_X) = 1_{F(X)}$

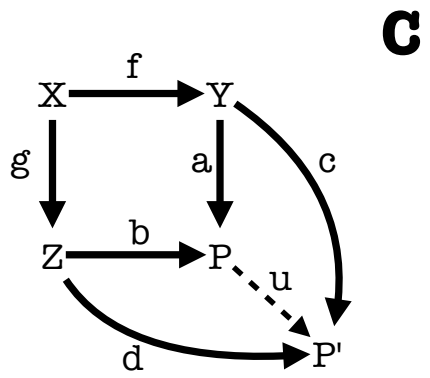
A functor F is **faithful** if for any arrows $f, g \in \mathbf{C}(X, Y)$, if $F(f) = F(g)$ then $f = g$; in other words, F is injective on arrows.

In order to compute reflections, our algorithms make use of **pushouts** (Chapter 3). Pushouts play a role analogous to joins in lattices by constructing a universal way to combine

compatible abstractions. Consider a category $\mathbf{C}$ and three objects $X, Y, Z \in obj(\mathbf{C})$. A pushout of two arrows $f \in \mathbf{C}(X, Y)$ and $g \in \mathbf{C}(X, Z)$ consists of an object $P \in obj(\mathbf{C})$ and two arrows $a \in \mathbf{C}(Y, P)$ and $b \in \mathbf{C}(Z, P)$ such that:

- $a \circ f = b \circ g$
- this answer is **universal**: for any other object $P' \in obj(\mathbf{C})$ and arrows $c \in \mathbf{C}(Y, P')$ and $d \in \mathbf{C}(Z, P')$ such that $c \circ f = d \circ g$, there exists an arrow $u \in \mathbf{C}(P', P)$[4] such that $u \circ a = c$ and $u \circ b = d$.

The diagram for a pushout is pictured below.



2.5 Parikh's Theorem

A key technical tool that we use in order to compute context-free language reachability relations is Parikh's Theorem. The Parikh image of a word is a character count abstraction of it; the Parikh image of the word $w = abbac$ identifies that there are 2 a's, 2 b's, and 1 c but loses all information about the relative ordering of these characters.

[4]Similarly to reflections, the standard definition of a pushout additionally requires that u is unique, but this uniqueness is not important for our purposes. For the remainder of the thesis, we use the weakened definition provided here.

The **Parikh image** [46] of a word $w \in \Sigma^*$ is a function $\pi(w) : \Sigma \to \mathbb{N}$ mapping each symbol $s \in \Sigma$ to the number of occurrences of s in w. The Parikh image of a language L is defined as $\pi(L) \triangleq \{\pi(w) : w \in L\}$. For any context-free grammar $G = \langle N, \Sigma, R, n_0 \rangle$, there is a LIRA formula $\mathit{Parikh}(G)$ that represents the Parikh image of $\mathcal{L}(G)$; its free variables are $\{c_s : s \in \Sigma\}$ and $\mathit{Parikh}(G)[c_s \mapsto m(s)]$ holds if and only if m is the Parikh image of some word $w \in \mathcal{L}(G)$.

The following is a polynomial-time procedure (quadratic in the number of production rules) to compute $\mathit{Parikh}(G)$ from any context-free grammar $G = \langle N, \Sigma, R, n_0 \rangle$. This is a correction to the construction of [63], which has a bug in the connectedness constraints which the formula presented here avoids by being explicit about nonterminals in the underlying graph. The precise details of this construction are not necessary to understand the contributions of this thesis; readers may skip to Chapter 3 without missing anything.

Formula Construction

Our construction is based on a connection between the Parikh image of a grammar and *flows* through a related hypergraph. Our approach follows Verma et al. [63], who adapt a counting technique for regular languages due to Seidl et al. [60]; we give a more complete presentation and correct a bug in their connectedness constraints.

Definition 3. A **directed hypergraph** is a tuple $\langle V, E \rangle$ in which V is a finite set of vertices and E is a set of *hyperedges* $\langle v, S \rangle$ consisting of a source vertex v and a multiset of targets $S : V \to \mathbb{N}$.

Definition 4. A **flow** in a directed hypergraph $\langle V, E \rangle$ is a triple $\langle v, S, f \rangle$ consisting of a source vertex v, a multiset of targets $S : V \to \mathbb{N}$, and a flow mapping $f : E \to \mathbb{N}$.

Consistency formalizes the notion that the flow into each vertex is equal to the flow out (with the exception of the start vertex v, which has an extra unit of incoming flow). A flow $\langle v, S, f \rangle$ is **consistent** if for every $v^* \in V$, we have that:

$$\mathbf{1}[v^* = v] + \sum_{\langle v', S' \rangle \in E} f(\langle v', S' \rangle) S'(v^*) = S(v^*) + \sum_{\langle v^*, S' \rangle \in E} f(\langle v^*, S' \rangle)$$

A flow $\langle v, S, f \rangle$ is **connected** if the following directed hypergraph is connected:

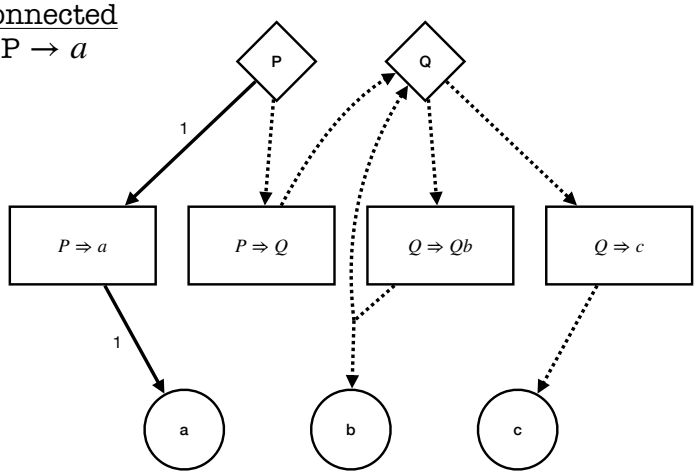
$$\langle \{v' : \langle v', S' \rangle \in E, f(\langle v', S' \rangle) > 0\} \cup \{v : S(v) > 0\}, \{e \in E : f(e) > 0\} \rangle$$

We begin by describing our approach via example, then proceed by formalizing our approach in the following theorem. Consider the context-free grammar:

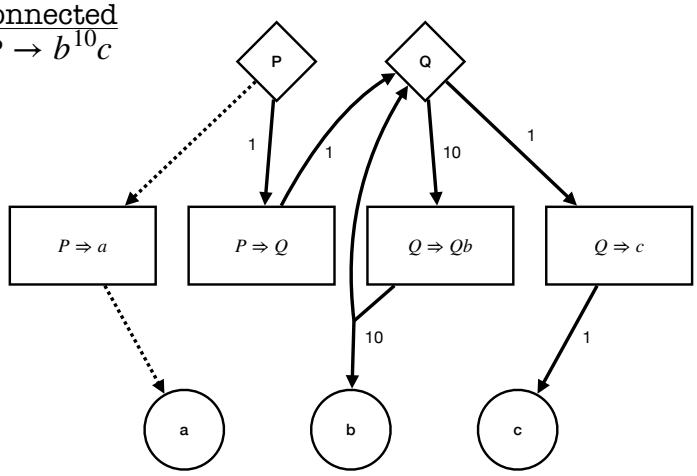
$$G = \langle \{P, Q\}, \{a, b, c\}, \{P \Rightarrow Q, P \Rightarrow a, Q \Rightarrow Qb, Q \Rightarrow c\}, P \rangle$$

We have that $\mathcal{L}(G) = \{a\} \cup \{b^n c : n \in \mathbb{N}\}$. We may associate G with the following directed hypergraph, which introduces nodes for the nonterminals, terminals, and production rules of the grammar and connects them with hyperedges; the nonterminals are connected to the production rules that could possibly consume them and the production rules are connected to the nonterminals and terminals that they produce. All diagrammed flows are consistent. Observe that connected and consistent flows correspond to valid derivations of words in the grammar, while the disconnected flow does not.

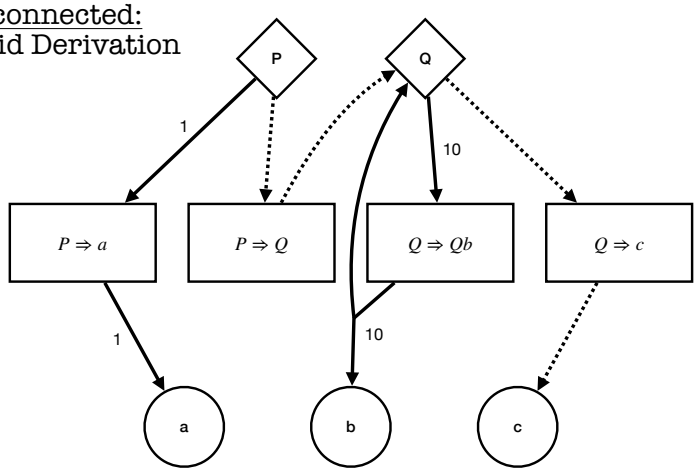
Connected
$\mathrm{P} \to a$



Connected
$\mathrm{P} \to b^{10}c$



Disconnected:
No Valid Derivation



The following theorem formalizes the construction of this directed hypergraph and motivates the construction of our Parikh image formula.

Theorem 1. Let $G = \langle N, \Sigma, R, n_0 \rangle$ be a context-free grammar. Consider the directed hypergraph $\langle N \cup \Sigma \cup R, E \rangle$, in which:

$$E \triangleq \{\langle n, \{\langle n, \beta \rangle\}\rangle : \langle n, \beta \rangle \in R\} \cup \{\langle \langle n, e_1 \dots e_n \rangle, \{e_1, \dots, e_n\} \rangle : \langle n, e_1 \dots e_n \rangle \in R\}$$

For any $m : \Sigma \to \mathbb{N}$, we have that $m \in \pi(\mathcal{L}(G))$ if and only if there exists a consistent and connected flow $\langle n_0, S, f \rangle$ such that $S(s) = m(s)$ for all $s \in \Sigma$ and $S(e) = 0$ for all $e \in N \cup R$.

Proof. ($\Rightarrow$) Assume that a function $m : \Sigma \to \mathbb{N}$ is in $\pi(\mathcal{L}(G))$. Then, there exists some word $w \in \mathcal{L}(G)$ such that $m = \pi(w)$. There exists a connected and consistent flow $\langle n_0, S, f \rangle$ meeting the conditions above in which f maps each production rule r to its number of usages $f(r)$ in the derivation of w, and each nonterminal n to the number of times $f(n)$ it was introduced by the right hand side of one of those production.

($\Leftarrow$) We proceed by induction on $\sum_{n \in N} \sum_{\langle n, S \rangle \in E} f(\langle n, S \rangle)$. For the base case, when this sum equals 1, the flow corresponds to a single path from n_0 to a production rule whose right-hand side consists entirely of terminals. The derivation is then a single application of this rule.

For the inductive step, there always exists a production rule incident to n_0 with positive flow, such that deleting one unit of flow from n_0 to this rule and one unit from each edge exiting the rule yields several smaller consistent and connected flows. By the inductive hypothesis, each admits a derivation; combining these derivations under the selected production rule yields the full derivation. □

Connectedness requires the following helper lemma to be easily encoded. The connectedness constraint in [63] is vacuously satisfiable — their distance variables can all be set to zero

and satisfy their constraints, allowing disconnected flows to be models of their formula. Our formula avoids this pitfall by treating nonterminals uniformly with terminals in the flow.

Lemma 1. Let $\langle V, E\rangle$ be a directed hypergraph and let $\langle v, S, f\rangle$ be a consistent flow. Then $\langle v, S, f\rangle$ is connected if and only if there exists a mapping $d : V \to \mathbb{N}$ such that $d(v) = 1$ and for every edge $\langle w, S'\rangle \in E$ with $f(\langle w, S'\rangle) > 0$ and $w \neq v$, there exists an edge $\langle u, S''\rangle \in E$ with $f(\langle u, S''\rangle) > 0$ and $d(u) < d(w)$.

Proof. ($\Rightarrow$) If the flow is connected, define $d(w)$ as the distance from v to w in the induced hypergraph. For every edge $\langle w, S'\rangle \in E$ with $f(\langle w, S'\rangle) > 0$ and $w \neq v$, w has an immediate predecessor u on the shortest path from v for which $d(u) < d(w)$.

($\Leftarrow$) Suppose d exists satisfying the condition above. Suppose, for purposes of obtaining a contradiction, that the flow is not connected. Let C be the set of vertices of the induced hyper-graph that do not have a path to v, and let $w \in C$ minimize $d(w)$ over all vertices in C. By flow consistency, w must have some outgoing flow. Then, by the assumptions of the lemma, there must be some incoming edge $\langle u, S''\rangle$ such that $S''(w) > 0$ and such that $d(u) < d(w)$. However, this contradicts the minimality of w. Therefore, the flow is connected. □

We are now fully equipped to define our formula. Consider a context-free grammar $G = \langle N, \Sigma, R, n_0\rangle$ and the associated directed hypergraph $\langle N \cup \Sigma \cup R, E\rangle$ described in Theorem 1. Define variables c_s for all $s \in \Sigma$ (representing the flow targets S), variables f_e for all $e \in E$ (representing our flow map f), and variables d_v for all $v \in N \cup \Sigma \cup R$ (representing the map

d from Lemma 1). We define:

$$\texttt{Consistency} \triangleq \quad \forall s \in \Sigma. \quad \sum_{\langle v', S' \rangle \in E} f_{\langle v', S' \rangle} S'(s) = c_s$$

$$\wedge \ \forall v \in N \cup R. \quad \mathbf{1}[v = n_0] + \sum_{\langle v', S' \rangle \in E} f_{\langle v', S' \rangle} S'(v) = \sum_{\langle v, S' \rangle \in E} f_{\langle v, S' \rangle}$$

$$\texttt{Connectedness} \triangleq \quad \forall \langle w, S' \rangle \in E. \quad w \neq n_0 \wedge f_{\langle w, S' \rangle} > 0$$

$$\implies \bigvee_{\substack{\langle u, S'' \rangle \in E \\ S''(w) > 0}} d_u < d_w$$

$$\wedge \ d_{n_0} = 1$$

$$\mathit{Parikh}(G) \triangleq \exists \{f_e : e \in E\}, \{d_v : v \in \Sigma \cup N \cup R\}. \quad \texttt{Consistency} \wedge \texttt{Connectedness}$$

Chapter 3

Foundations

This section introduces the reader to the formalisms used in the rest of the thesis. Specifically, it aims to introduce:

1. our end-to-end recipe for computing procedure summaries
2. the use of Parikh's Theorem to compute context-free reachability relations
3. a formalized divide-and-conquer approach to computing reflections

To elaborate, the first section formalizes the general steps of our analysis techniques and how they guarantee the robustness properties of monotonicity and locality. The next two sections describe an inter-procedural analysis technique that has these robustness properties and uses *vector addition systems* (VAS) as its abstract domain. We begin by showing that Parikh's Theorem enables us to compute the context-free reachability relation of VAS, and continue by showing that we can compute the most precise VAS abstraction, or *VAS reflection*, of any program. We finally investigate the category theoretic foundations underpinning our technique for computing reflections, yielding a framework that simplifies the descriptions of computing reflections in later chapters.

3.1 Analysis Recipe

In this thesis, we present program analyses that are sound, monotone, and local. This section describes how we do so. Given a context-free grammar G describing the control structure of a program and a transition assignment $\not\!\!f : \Sigma \to \mathbf{TF}(X)$, we aim to compute a formula $F \in \mathbf{TF}(X)$ over-approximating the $\mathcal{L}$-reachability relation of $\not\!\!f$, as described in Section 2.2.

Recall that $\mathbf{TS}(\Sigma)$ refers to the category in which the objects are labeled transition systems over Σ that are defined with a transition mapping; in other words, the category of our input programs. Each program analysis presented in this thesis defines a subcategory $\mathbf{Abs}(\Sigma)$ of $\mathbf{TS}(\Sigma)$, and computes summaries via the following common recipe:

1. We compute the $\mathbf{Abs}(\Sigma)$-reflection $\langle f, T \rangle$ of $\not\!\!f$, consisting of a labeled transition system $T = \left\langle \mathbb{Q}^Y, \to_T \right\rangle \in \mathit{obj}(\mathbf{Abs}(\Sigma))$ and a simulation f from $\not\!\!f$ to T.

2. We compute a transition formula F precisely defining the $\mathcal{L}(G)$ reachability relation of T:
$$\langle \rho, \rho' \rangle \models F \iff \rho \xrightarrow{\mathcal{L}(G)}_T \rho'$$

3. Letting σ be the substitution corresponding to f, we take $F[\bar{\sigma}]$ to be an over-approximation of the $\mathcal{L}(G)$-reachability of $\not\!\!f$.

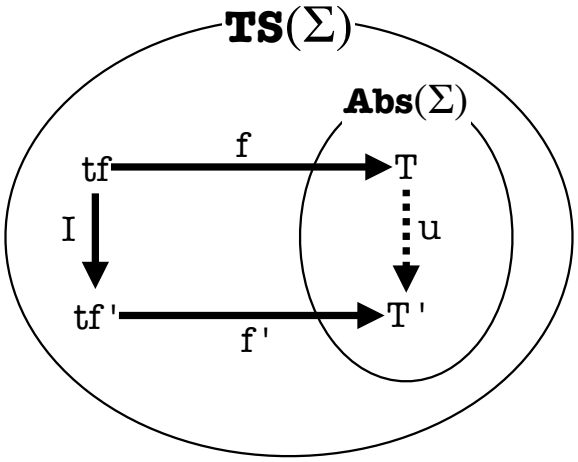
Theorem 2. Consider a program defined by context-free language $\mathcal{L}(G) \subseteq \Sigma^*$ and transition assignment $\not\!\!f : \Sigma \to \mathbf{TF}(X)$. Program analyses following the above recipe are sound and monotone.

Proof. First, we may observe that this recipe always produces sound analyses:

$$\begin{aligned} \rho \xrightarrow{\mathcal{L}(G)}_{\not\Vdash} \rho' &\implies f(\rho) \xrightarrow{\mathcal{L}(G)}_{T} \rho' \\ &\implies \langle f(\rho), f(\rho') \rangle \models F \\ &\implies [\rho, \rho'] \models F[\bar{\sigma}] \end{aligned}$$

Now, we can observe that this recipe always produces monotone analyses. Consider transition assignments $\not\Vdash : \Sigma \to \mathbf{TF}(X)$ and $\not\Vdash' : \Sigma \to \mathbf{TF}(X)$ such that $\not\Vdash(s) \models \not\Vdash'(s)$ for all $s \in \Sigma$. Then, the identity function I is a simulation from $\not\Vdash$ to $\not\Vdash'$.

Let $\langle f, T \rangle$ and $\langle f', T' \rangle$ be $\mathbf{Abs}(\Sigma)$-reflections of $\not\Vdash$ and $\not\Vdash'$ respectively. Since $\langle f, T \rangle$ is a reflection, there exists a simulation u from T to T' making the following diagram commute:



We know that the arrow u exists in the above diagram because $f' \circ I$ is an arrow from $\not\Vdash$ to T' and $\langle f, T \rangle$ is a reflection. Then, since u is a simulation from T to T', we have that if $f(\rho) \to_T f(\rho')$ then $u(f(\rho)) \to_{T'} u(f(\rho'))$.

Then, as in step (2) of the recipe, let $F \in \mathbf{TF}(X)$ and $F' \in \mathbf{TF}(X)$ be formulas such that:

$$[\rho, \rho'] \models F \iff \rho \xrightarrow{\mathcal{L}(G)}_{T} \rho' \qquad [\rho, \rho'] \models F' \iff \rho \xrightarrow{\mathcal{L}(G)}_{T'} \rho'$$

Let σ and σ' be the substitutions corresponding to linear maps f and f'. We may now conclude the monotonicity property:

$$\begin{aligned}
\langle \rho, \rho' \rangle \models F[\bar{\sigma}] &\implies f(\rho) \xrightarrow{\mathcal{L}(G)}_T f(\rho') \\
&\implies u(f(\rho)) \xrightarrow{\mathcal{L}(G)}_{T'} u(f(\rho')) \\
&\implies f'(\rho) \xrightarrow{\mathcal{L}(G)}_{T'} f'(\rho') \\
&\implies [\rho, \rho'] \models F'[\bar{\sigma'}]
\end{aligned}$$

□

The remainder of this chapter will describe a program analysis technique which instantiates this recipe with the abstraction subcategory of *vector addition systems* (VAS). First, in Section 3.2, we convince ourselves that step (2) is possible for this abstraction class: for any VAS, we may compute a transition formula F precisely defining its $\mathcal{L}(G)$-reachability relation. Then, in Section 3.3, we describe how we achieve step (1) - how we compute the VAS reflection of any LIRA transition assignment. We generalize our technique for computing VAS reflections in Section 3.4, extracting a category-theoretic recipe that will be used in future chapters.

3.2 VAS Reachability via Parikh's Theorem

Vector Addition Systems are a class of transition systems that have historically been used to model concurrent systems, and in which each operation adds a fixed offset to each variable independently [34]. VAS are classically defined over counters in $\mathbb{N}$, but here we work with a rational-valued extension inspired by the integer-valued extension of Haase and Halfon [27].

Definition 5. A **VAS transition** over a set of variables Y is a transition formula in $\mathbf{TF}(Y)$ of the form:

$$\bigwedge_{y \in Y} y' = y + a_y$$

where $a_y \in \mathbb{Q}$ for every $y \in Y$.

Definition 6. A **Rational Vector Addition System** over a set of variables Y and finite alphabet Σ is a transition assignment $\mathcal{V} : \Sigma \to \mathbf{TF}(Y)$ in which $\mathcal{V}(s)$ is a VAS transition for every $s \in \Sigma$. We write $\mathit{Offset}(\mathcal{V}, s, y)$ for the rational a_y in $\mathcal{V}(s)$.

A notable feature of VAS transitions is that they commute under sequencing. The sequential composition of two VAS transitions $\mathcal{V}(s)$ and $\mathcal{V}(s')$ is the same net increment regardless of order: $\mathit{Offset}(\mathcal{V}, s, y) + \mathit{Offset}(\mathcal{V}, s', y)$ for all $y \in Y$. Because of this, the $\mathcal{L}(G)$-reachability of any VAS depends only on how many times each symbol appears in a word, not on the order in which they appear. This is precisely the information captured by the Parikh image of $\mathcal{L}(G)$, making it the natural tool for computing the $\mathcal{L}(G)$-reachability relation of VAS.

Theorem 3. Let $\mathcal{V}$ be a VAS over a set of variables Y and finite alphabet Σ. Let $\mathcal{L}(G) \subseteq \Sigma^*$ be a context-free language. In polynomial time, we can compute a transition formula $F \in \mathbf{TF}(\Sigma)$ such that:

$$\langle \rho, \rho' \rangle \models F \iff \rho \xrightarrow{\mathcal{L}(G)}_{\mathcal{V}} \rho'$$

Proof. We may define F as:

$$F \triangleq \exists \{c_s : s \in \Sigma\} \left(\mathit{Parikh}(G) \wedge \bigwedge_{y \in Y} y' = y + \sum_{s \in \Sigma} \mathit{Offset}(\mathcal{V}, s, y) c_s \right)$$

($\Rightarrow$) Observe that if $\langle \rho, \rho' \rangle \models F$, then there exists some valuation of $\{c_s : s \in \Sigma\}$ satisfying the rest of the formula; let $m : \Sigma \to \mathbb{Z}$ be that valuation. By the definition of $\mathit{Parikh}(G)$, there

exists some word $w \in \mathcal{L}(G)$ such that $m = \pi(w)$, i.e., $m(s) = \pi(w)(s)$ for all $s \in \Sigma$. Since c_s is instantiated with value $m(s)$, we have $\sum_{s\in\Sigma} \mathit{Offset}(\mathcal{V}, s, y)c_s = \sum_{s\in\Sigma} \mathit{Offset}(\mathcal{V}, s, y)\pi(w)(s)$ for all $y \in Y$. This is precisely the increment applied by the composition of VAS transitions along w. Therefore, we have that $\rho \xrightarrow{\mathcal{L}(G)}_{\mathcal{V}} \rho'$.

($\Leftarrow$) If $\rho \xrightarrow{\mathcal{L}(G)} \rho'$ then there exists some $w \in \mathcal{L}(G)$ such that $\rho \xrightarrow{w} \rho'$. Then we have that F holds, instantiating c_s with value $\pi(w)(s)$ for all $s \in \Sigma$: $\mathit{Parikh}(G)$ holds because $w \in \mathcal{L}(G)$ and the rest holds by the commutativity of VAS operations. □

Example 3.2.1. Let $\Sigma = \{a, b\}$, $Y = \{y\}$, and consider the VAS $\mathcal{V}$ defined by:

$$\mathcal{V}(a) \triangleq y' = y + 1 \qquad \mathcal{V}(b) \triangleq y' = y - 2$$

and the context-free language $\mathcal{L}(G)$ generated by:

$$G = \langle \{P\}, \{a, b\}, \{P \Rightarrow aPb, P \Rightarrow a\}, P \rangle$$

which is the language $\{a^{n+1}b^n : n \geq 0\}$. The Parikh image of $\mathcal{L}(G)$ is characterized by $\mathit{Parikh}(G) \triangleq c_a = c_b + 1 \wedge c_b \geq 0$. By the theorem, the $\mathcal{L}(G)$-reachability of $\mathcal{V}$ is:

$$F \triangleq \exists c_a, c_b \, (c_a = c_b + 1 \wedge c_b \geq 0 \wedge y' = y + c_a - 2c_b)$$

Theorem 3 shows us that we can precisely summarize the behavior of a VAS over a context-free language, making it a suitable class of abstractions for our program analysis recipe. The key idea involved was to exploit the commutativity of VAS operations and use the Parikh image of a context-free language

Chapters 4, 5, and 6 describe abstract domains related to vector addition systems that are suitable for program analysis. These other domains are not, however, commutative. We solve their associated reachability problems by translating them to a related problem that is fully commutative and using the Parikh Image analogously to the above construction. This pattern yields a generalized recipe for using Parikh Images to analyze a certain class of non-commutative transition systems, and this is the subject of Chapter 7.

3.3 VAS Reflections of Programs

This section describes a technique to compute the VAS reflection of a program, which is the most precise VAS abstraction of it for program analysis.

Example 3.3.1. Let $\Sigma = \{a, b\}$, $X = \{x_1, x_2\}$, and consider the transition assignment $\not\!\!f$ defined by:

$$\not\!\!f(a) \triangleq x_1' = x_2 + 1 \wedge x_2' = x_1 + 1 \qquad \not\!\!f(b) \triangleq x_1' = x_1 - 1 \wedge x_2' = x_2 + 1$$

This transition assignment is not a VAS; observe that $\not\!\!f(a)$ is not a VAS transition. However, we can define a VAS abstraction of it that tracks the sum $x_1 + x_2$. Let $Y = \{y\}$ and define the VAS $\mathcal{U}$ by:

$$\mathcal{U}(a) \triangleq y' = y + 2 \qquad \mathcal{U}(b) \triangleq y' = y$$

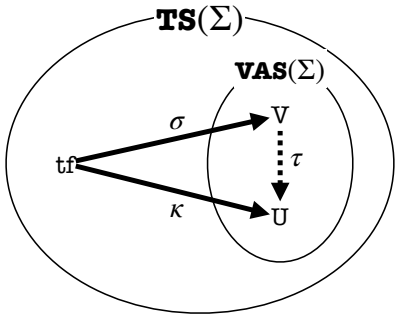
The substitution $\kappa(y) = x_1 + x_2$ is a linear simulation from $\not\!\!f$ to $\mathcal{U}$: for each $s \in \Sigma$, one can verify that $\not\!\!f(s) \models \mathcal{U}(s)[\bar{\kappa}]$. This example highlights one of the benefits of defining our

abstractions with linear simulations; even when no individual variable can be abstracted we can still extract relationships between variables.

$\mathcal{U}$ is not the most precise VAS abstraction of $\#$: we can also track the difference $x_1 - x_2$. Let $Z = \{z_1, z_2\}$ and define the VAS $\mathcal{V}$ by:

$$\mathcal{V}(a) \triangleq z_1' = z_1 + 2 \wedge z_2' = z_2 \qquad \mathcal{V}(b) \triangleq z_1' = z_1 \wedge z_2' = z_2 - 2$$

The substitution $\sigma(z_1) = x_1 + x_2$, $\sigma(z_2) = x_1 - x_2$ is a linear simulation from $\#$ to $\mathcal{V}$, and $\mathcal{V}$ is the VAS reflection of $\#$: any VAS abstraction of $\#$ can only track linear combinations of $x_1 + x_2$ and $x_1 - x_2$ because this is the total space of linear terms over X. Then, there is a simulation τ from $\mathcal{V}$ to any such abstraction. In particular, the substitution $\tau(y) = z_1$ witnesses the arrow from $\mathcal{V}$ to $\mathcal{U}$ in the diagram below.



In the category $\mathbf{TS}(\Sigma)$, the objects are labeled transition systems over Σ defined by transition assignments, and the arrows are linear simulations between those systems. Our input program $\#$ is an object in this category. $\mathbf{VAS}(\Sigma)$ is a subcategory therein, in which the objects are VAS (and the arrows are, again, linear simulations).

The VAS reflection of $\#$ is the VAS $\mathcal{V}$ and the linear simulation σ from $\#$ to $\mathcal{V}$. This object has the property that for any other VAS $\mathcal{U}$ with a simulation κ from $\#$ to $\mathcal{U}$, there exists some simulation τ from $\mathcal{V}$ to $\mathcal{U}$. We saw in Subsection 3.1 that this property implies

that the summary computed from the reflection is at least as precise as the summary computed from any other VAS abstraction.

We can compute VAS reflections via a *divide-and-conquer* approach. First, we show that we can compute VAS reflections of individual transition formulas. We then show that we can combine two reflections over disjoint alphabets into a single reflection over the union of their alphabets. Together, these form an approach for computing the VAS reflection of any system $\not\!f$ in $\mathbf{TF}(\Sigma)$: compute a reflection of $\not\!f$ restricted to s for every $s \in \Sigma$, then repeatedly combine these individual reflections to form a global one.

3.3.1 Per-Letter VAS Reflections

This section details an approach to computing a VAS reflection $\mathcal{V} : \Sigma \to \mathbf{TF}(Y)$ and simulation $\sigma : Y \to \mathit{LinTerm}(X)$ of a transition assignment $\not\!f : \Sigma \to \mathbf{TF}(X)$ in the special case that Σ consists of a single character s. In other words, we show that any transition formula $F \in \mathbf{TF}(X)$ has a best abstraction as a VAS transition formula.

Every variable of any VAS abstraction of $\not\!f$ represents a linear term over X. We can define the set of pairs $\langle t, o\rangle$ over X for which $\not\!f(s)$ increments term t by o as:

$$\text{VASTerms}(\not\!f(s)) \triangleq \{\langle t, o\rangle : \not\!f(s) \models t' = t + o\}$$

This set is a vector space, a subspace of $\mathit{LinTerm}(X) \times \mathbb{Q}$: it is closed under addition and scalar multiplication. There exists a finite basis for this set. Informally, by constructing a VAS abstraction which encodes this finite basis, we are able to represent the whole set VASTerms($\not\!f(s)$)and guarantee that this abstraction is a reflection.

We can compute a basis for VASTerms($\not\!f(s)$) via Algorithm 1, which is a special case of the symbolic abstraction algorithm of Reps et al. [55]. The condition that transition formulas

are satisfiable, stated in their definition in Chapter 2, guarantees the existence of a model m satisfying $\mathit{tf}(s)$ on line 1.

Input: Transition formula $\mathit{tf}(s)$, variables X
Output: Basis of VASTerms($\mathit{tf}(s)$)
1 $m \leftarrow$ model of $\mathit{tf}(s)$
2 $\mathit{ans} \leftarrow \{\langle x, m(x') - m(x)\rangle : x \in X\}$
3 $\psi \leftarrow \mathit{tf}(s)$
4 **while** ψ *is satisfiable* **do**
5 $\quad m \leftarrow$ model of ψ
6 $\quad B \leftarrow \{\langle x, m(x') - m(x)\rangle : x \in X\}$
7 $\quad \mathit{ans} \leftarrow$ basis of $\mathrm{span}(\mathit{ans}) \cap \mathrm{span}(B)$
8 $\quad \psi \leftarrow \psi \wedge \neg\Big(\bigwedge_{\langle t,o\rangle \in \mathit{ans}} t' = t + o\Big)$
9 **return** *ans*

Algorithm 1: Compute Basis of VASTerms($\mathit{tf}(s)$)

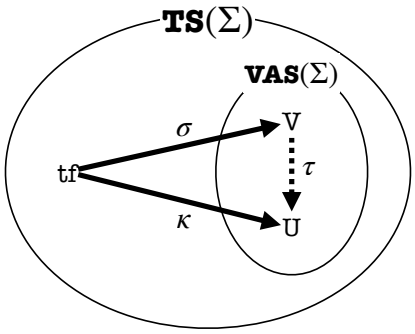
With a basis $\langle t_1, o_1\rangle, \ldots, \langle t_n, o_n\rangle$ for VASTerms($\mathit{tf}(s)$) in hand, we may define the VAS-reflection of tf to be the following:

$$\mathcal{V}(s) \triangleq \bigwedge_{i=1}^{n} y_i' = y_i + o_i$$

$$\sigma(y_i) = t_i$$

Lemma 2. Let $\mathit{tf} : \Sigma \to \mathbf{TF}(X)$ be a transition assignment over the singleton alphabet ($|\Sigma| = 1$). $\langle \sigma, \mathcal{V}\rangle$ is a VAS reflection of tf.

Proof. Let $\mathcal{U} : \Sigma \to \mathbf{TF}(Z)$ and simulation $\kappa : Z \to \mathit{LinTerm}(X)$ from tf to $\mathcal{U}$ be a different VAS abstraction. We will show that there exists a simulation τ from $\mathcal{V}$ to $\mathcal{U}$ such that $\sigma \circ \tau = \kappa$.



Consider variable $z \in Z$ of $\mathcal{U}$. We must have that $\langle \kappa(z), \mathit{Offset}(\mathcal{U}, s, z) \rangle$ is in VASTerms($\mathit{tf}(s)$): if $\mathit{tf}(s)$ holds, then $\mathcal{U}(s)[\bar{\kappa}]$ holds due to the linear simulation κ, and a conjunct of this formula is $\bar{\kappa}(z') = \bar{\kappa}(z) + \mathit{Offset}(\mathcal{U}, s, z)$.

Then, since $\langle t_1, o_1 \rangle, \ldots, \langle t_n, o_n \rangle$ is a basis for VASTerms($\mathit{tf}(s)$), for each $z \in Z$ there must exist coefficients $\alpha_1, \ldots \alpha_n$ such that $\langle \kappa(z), \mathit{Offset}(\mathcal{U}, s, z) \rangle = \sum_{i=1}^{n} \alpha_i \langle t_i, o_i \rangle$, and in particular such that $\mathit{Offset}(\mathcal{U}, s, z) = \sum_i \alpha_i o_i$.

Let $\tau : Z \to \mathit{LinTerm}(Y)$ be the substitution such that $\tau(z) = \sum_{i=1}^{n} \alpha_i y_i$ for each $z \in Z$. One may observe that $\sigma \circ \tau = \kappa$.

It remains to show that $\mathcal{V}(s) \models \mathcal{U}(s)[\tau]$ for all $s \in \Sigma$. This holds because $\mathcal{V}(s)$ asserts $y_i' = y_i + o_i$ for each i, so for each $z \in Z$:

$$\bar{\tau}(z') = \sum_i \alpha_i y_i' = \sum_i \alpha_i (y_i + o_i) = \bar{\tau}(z) + \mathit{Offset}(\mathcal{U}, s, z)$$

where the last equality uses $\mathit{Offset}(\mathcal{U}, s, z) = \sum_i \alpha_i o_i$.

Therefore, τ is a simulation from $\mathcal{V}$ to $\mathcal{U}$, and so $\langle \sigma, \mathcal{V} \rangle$ is a VAS reflection of tf. □

3.3.2 Combining VAS Reflections of Disjoint Alphabets

This subsection shows how, given VAS reflections of $\mathit{tf}|_{\Sigma_1}$ and $\mathit{tf}|_{\Sigma_2}$ for a partition $\langle \Sigma_1, \Sigma_2 \rangle$ of Σ, we may combine them into a reflection of tf. This procedure, in combination with

the previous subsection's technique for computing the VAS reflection of $\not{f}|_{\{s\}}$, can be used to compute the VAS reflection of any transition assignment: compute a reflection for each character, and combine them all together.

The key challenge in this combination problem is that the two reflections can track different linear terms of the original transition assignment, and therefore operate over distinct abstract state spaces. Computing a reflection of $\not{f}$ requires finding a maximal common abstract state space that both reflections simulate to. Consider the following example.

Example 3.3.2. Consider the following transition assignment $\not{f}$ over alphabet $\Sigma = \{a, b\}$:

$$\not{f}(a) \triangleq x_1' = x_1 + 1 \wedge x_2' = x_2 + 2 \qquad \not{f}(b) \triangleq x_1' + x_2' = x_1 + x_2 + 5$$

Consider the partition $\Sigma_1 = \{a\}$, $\Sigma_2 = \{b\}$ of Σ. The VAS-reflections $\langle \sigma_1, \mathcal{V}_1 \rangle$ and $\langle \sigma_2, \mathcal{V}_2 \rangle$ of $\not{f}|_{\Sigma_1}$ and $\not{f}|_{\Sigma_2}$ respectively are:

$$\mathcal{V}_1(a) \triangleq y_1' = y_1 + 1 \wedge y_2' = y_2 + 2 \qquad \mathcal{V}_2(b) \triangleq z_1' = z_1 + 5$$

$$\sigma_1(y_1) = x_1 \quad \sigma_1(y_2) = x_2 \quad \sigma_2(z_1) = x_1 + x_2$$

The reflection of $\not{f}|_{\Sigma_1}$ tracks both x_1 and x_2, but the reflection of $\not{f}|_{\Sigma_2}$ only tracks $x_1 + x_2$. The reflection of $\not{f}$ cannot track x_1 or x_2 because no VAS abstraction of $\not{f}|_{\Sigma_2}$ can track these terms, evidenced by the fact that there is no way to form these terms as a linear combination of the terms tracked by the reflection $\langle \sigma_2, \mathcal{V}_2 \rangle$. However, the reflection of $\not{f}$ can track $x_1 + x_2$ because this term can be expressed as a linear combination of the variables of the reflections of both $\not{f}|_{\Sigma_1}$ and $\not{f}|_{\Sigma_2}$.

Then, informally, our approach for computing the reflection of $\not\#$ from reflection of partitions of its alphabet is to find a basis for the greatest space of possibly trackable terms that is common between the two reflections, and to compute a VAS that tracks a basis for this space. In this example, that common space is $\text{span}(\{x_1 + x_2\})$, and so the VAS-reflection $\langle \sigma, \mathcal{V} \rangle$ of $\not\#$ is:

$$\mathcal{V}(a) \triangleq w' = w + 3 \qquad \mathcal{V}(b) \triangleq w' = w + 5$$

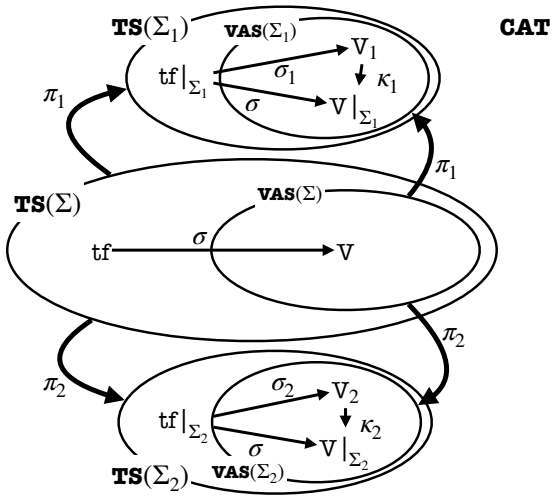
$$\sigma(w) = x_1 + x_2$$

We have that σ is a simulation from $\not\#$ to $\mathcal{V}$, and therefore that σ is a simulation from $\not\#|_{\Sigma_1}$ to $\mathcal{V}|_{\Sigma_1}$. Since $\langle \sigma_1, \mathcal{V}_1 \rangle$ is a VAS reflection of $\not\#|_{\Sigma_1}$ and σ is a simulation from $\not\#|_{\Sigma_1}$ to $\mathcal{V}|_{\Sigma_1}$, there exists a simulation κ_1 from $\mathcal{V}_1$ to $\mathcal{V}|_{\Sigma_1}$ such that $\sigma_1 \circ \kappa_1 = \sigma$. Symmetrically, there exists a simulation κ_2 from $\mathcal{V}_2$ to $\mathcal{V}|_{\Sigma_2}$ such that $\sigma_2 \circ \kappa_2 = \sigma$.

$$\kappa_1(w) = y_1 + y_2 \qquad \kappa_2(w) = z_1$$

Displaying the reflective property of $\mathcal{V}$ in a category-theoretic commuting diagram is complex because the transition systems inhabit different categories. We cannot put $\mathcal{V}_1$ and $\mathcal{V}_2$ in the same category because they operate over different alphabets, and therefore simulations would be ill-defined.

Instead, our diagram contains distinct categories for each alphabet considered in this example: $\Sigma_1 = \{a\}$, $\Sigma_2 = \{b\}$, and $\Sigma = \{a, b\}$.



The π_1 functor describes how objects in $\mathbf{TS}(\Sigma)$ (and in $\mathbf{VAS}(\Sigma)$) can be restricted down to objects in $\mathbf{TS}(\Sigma_1)$ (resp. $\mathbf{VAS}(\Sigma_1)$). Formally, we have $\pi_1(\not{tf}) = \not{tf}|_{\Sigma_1}$ and $\pi_1(\sigma) = \sigma$. The other functor π_2 is defined analogously.

The key takeaway from this diagram is that our combination procedure is actually taking place across 3 distinct categories. This section describes a technique for combining VAS reflections, and the next section (Section 3.4) extracts from this technique a generalized recipe for combining reflections. Informally, this recipe requires (1) a common base category where the above 3 categories can be combined and (2) a mechanism for transferring the resulting constructions back into the original categories.

We continue by describing a technique for combining VAS reflections over partitions $\langle \Sigma_1, \Sigma_2 \rangle$ into a global reflection over Σ.

Let $\langle \sigma_1 : Y \to \mathit{LinTerm}(X), \mathcal{V}_1 : \Sigma_1 \to \mathbf{TF}(Y) \rangle$ be a VAS reflection of $\not{tf}|_{\Sigma_1} : \Sigma_1 \to \mathbf{TF}(X)$ and let $\langle \sigma_2 : Z \to \mathit{LinTerm}(X), \mathcal{V}_2 : \Sigma_2 \to \mathbf{TF}(Z) \rangle$ be a VAS reflection of $\not{tf}|_{\Sigma_2} : \Sigma_2 \to \mathbf{TF}(X)$. Our goal is to compute the VAS reflection $\langle \sigma, \mathcal{V} \rangle$ of $\not{tf} : \Sigma \to \mathbf{TF}(X)$.

Our first step is to compute σ. We can express the space of terms that a VAS reflection of $\not\!\!f$ could possibly track as:

$$\mathrm{span}(\{\sigma_1(y) : y \in Y\}) \cap \mathrm{span}(\{\sigma_2(z) : z \in Z\})$$

Let $t_1 \dots t_n$ be a basis for this space. Let W be a fresh set of n variables. Let $\sigma : W \to \mathit{LinTerm}(X)$ be the substitution such that $\sigma(w_i) = t_i$.

Since all t_i were in $\mathrm{span}(\{\sigma_1(y) : y \in Y\})$, there must be a substitution $\kappa_1 : W \to \mathit{LinTerm}(Y)$ such that $\sigma_1(\kappa_1(w_i)) = t_i = \sigma(w_i)$. Symmetric reasoning argues for the existence of a substitution $\kappa_2 : W \to \mathit{LinTerm}(Z)$ such that $\sigma_2 \circ \kappa_2 = \sigma$.

We may now compute $\mathcal{V}$ such that σ is a simulation from $\not\!\!f$ to $\mathcal{V}$. Below, $\kappa_1(w)(y)$ refers to coefficient of y in the term $\kappa_1(w) \in \mathit{LinTerm}(Y)$. We define $\mathcal{V} : \Sigma \to \mathbf{TF}(W)$ by:

$$\text{For } s \in \Sigma_1, \quad \mathcal{V}(s) \triangleq \bigwedge_{w \in W} w' = w + \sum_{y \in Y} \kappa_1(w)(y)\, \mathit{Offset}(\mathcal{V}_1, s, y)$$

$$\text{For } s \in \Sigma_2, \quad \mathcal{V}(s) \triangleq \bigwedge_{w \in W} w' = w + \sum_{z \in Z} \kappa_2(w)(z)\, \mathit{Offset}(\mathcal{V}_2, s, z)$$

The substitution κ_1 is a simulation from $\mathcal{V}_1$ to $\mathcal{V}|_{\Sigma_1}$: for each $s \in \Sigma_1$, $\mathcal{V}_1(s)$ asserts $y' = y + \mathit{Offset}(\mathcal{V}_1, s, y)$ for each $y \in Y$. Then, we may conclude by the linearity of κ_1, for all $w \in W$:

$$\bar{\kappa}_1(w') = \bar{\kappa}_1(w) + \sum_{y \in Y} \kappa_1(w)(y)\, \mathit{Offset}(\mathcal{V}_1, s, y)$$

which is precisely a conjunct of $\mathcal{V}(s)[\bar{\kappa_1}]$. Similarly, κ_2 is a simulation from $\mathcal{V}_2$ to $\mathcal{V}|_{\Sigma_2}$. Since $\sigma = \sigma_1 \circ \kappa_1 = \sigma_2 \circ \kappa_2$, the simulation condition $\not\!\!f(s) \models \mathcal{V}(s)[\bar{\sigma}]$ follows immediately from the simulation conditions of $\mathcal{V}_1$ and $\mathcal{V}_2$.

Example 3.3.3. Returning to Example 3.3.2, we now show how the above technique allows us to compute $\langle \sigma, \mathcal{V} \rangle$ from $\langle \sigma_1, \mathcal{V}_1 \rangle$ and $\langle \sigma_2, \mathcal{V}_2 \rangle$.

First, we observe that:

$$\begin{aligned} \mathrm{span}(\{\sigma_1(y_1), \sigma_2(y_2)\}) \cap \mathrm{span}(\{\sigma_2(y_3)\}) &= \mathrm{span}(\{x_1, x_2\}) \cap \mathrm{span}(\{x_1 + x_2\} \\ &= \mathrm{span}(\{x_1 + x_2\}) \end{aligned}$$

A basis for this set is the term $x_1 + x_2$. Define a set of variables $W = \{w\}$. We may define $\sigma : W \to \mathit{LinTerm}(X)$ by $\sigma(w) = x_1 + x_2$.

There exist κ_1 and κ_2 such that $\sigma_1 \circ \kappa_1 = \sigma = \sigma_2 \circ \kappa_2$:

$$\kappa_1(w) = y_1 + y_2 \qquad \kappa_2(w) = z_1$$

We may now compute $\mathcal{V}$ such that σ is a simulation from $\not\!\!f$ to $\mathcal{V}$.

$$\begin{aligned} \mathcal{V}(a) &\triangleq w' = w + \kappa_1(w)(y_1)\,\mathit{Offset}(\mathcal{V}_1, s, y_1) + \kappa_1(w)(y_2)\,\mathit{Offset}(\mathcal{V}_1, s, y_2) \\ &\triangleq w' = w + 1 + 2 \\ &\triangleq w' = w + 3 \\ \mathcal{V}(b) &\triangleq w' = w + \kappa_2(w)(z)\,\mathit{Offset}(\mathcal{V}_2, s, z) \\ &\triangleq w' = w + 5 \end{aligned}$$

Lemma 3. Let $\not\!\!f : \Sigma \to \mathbf{TF}(X)$ be a transition assignment over a finite alphabet Σ. Let $\langle \Sigma_1, \Sigma_2 \rangle$ be a partition of Σ. Let $\langle \sigma_1, \mathcal{V}_1 \rangle$ be a VAS reflection of $\not\!\!f|_{\Sigma_1}$ and let $\langle \sigma_2, \mathcal{V}_2 \rangle$ be a VAS reflection of $\not\!\!f|_{\Sigma_2}$. $\langle \sigma, \mathcal{V} \rangle$ is a VAS reflection of $\not\!\!f$.

Proof. Let $\mathcal{V}' : \Sigma \to \mathbf{TF}(V)$ and simulation σ' from $\not{f}$ to $\mathcal{V}'$ be a different VAS abstraction. We will show that there exists a simulation τ from $\mathcal{V}$ to $\mathcal{V}'$ such that $\sigma \circ \tau = \sigma'$.

It is the case that $\mathcal{V}'|_{\Sigma_1}$ and σ' are an abstraction of $\not{f}|_{\Sigma_1}$. Because $\langle f, \mathcal{V}_1 \rangle$ is a reflection of $\not{f}|_{\Sigma_1}$, there exists a simulation κ_1' from $\mathcal{V}_1$ to $\mathcal{V}'|_{\Sigma_1}$ such that $\sigma_1 \circ \kappa_1' = \sigma'$. Symmetrically, there exists a simulation κ_2' from $\mathcal{V}_2$ to $\mathcal{V}'|_{\Sigma_2}$ such that $\sigma_2 \circ \kappa_2' = \sigma'$.

Because κ_1' and κ_2' exist, we know that $\sigma'(v) \in \text{span}(\{\sigma_1(y) : y \in Y\}) \cap \text{span}(\{\sigma_2(z) : z \in Z\})$ for all $v \in V$. Since $\sigma(w_1), \ldots, \sigma(w_n)$ is a basis for this space, there exists a unique $\tau : V \to \mathit{LinTerm}(W)$ such that $\sigma \circ \tau = \sigma'$. Since $\sigma = \sigma_1 \circ \kappa_1$, uniqueness of τ implies $\kappa_1 \circ \tau = \kappa_1'$ and $\kappa_2 \circ \tau = \kappa_2'$.

It remains to show $\mathcal{V}(s) \models \mathcal{V}'(s)[\tau]$ for all $s \in \Sigma$; we handle $s \in \Sigma_1$, with Σ_2 symmetric. Since κ_1' is a simulation from $\mathcal{V}_1$ to $\mathcal{V}'$, we have $\mathcal{V}_1(s) \models \mathcal{V}'(s)[\kappa_1']$, i.e., $\bar{\kappa_1'}(v') = \bar{\kappa_1'}(v) + \mathit{Offset}(\mathcal{V}', s, v)$ under $\mathcal{V}_1(s)$ for each $v \in V$. Since $\mathcal{V}_1(s)$ asserts $y' = y + \mathit{Offset}(\mathcal{V}_1, s, y)$ for each $y \in Y$, we conclude:

$$\mathit{Offset}(\mathcal{V}', s, v) = \sum_{y \in Y} \kappa_1'(v)(y) \cdot \mathit{Offset}(\mathcal{V}_1, s, y) = \sum_{y \in Y} \kappa_1(\tau(v))(y) \cdot \mathit{Offset}(\mathcal{V}_1, s, y)$$

where the second equality uses $\kappa_1' = \kappa_1 \circ \tau$. Since $\mathcal{V}(s)$ asserts $w' = w + \sum_{y \in Y} \kappa_1(w)(y) \cdot \mathit{Offset}(\mathcal{V}_1, s, y)$ for each $w \in W$, we have under $\mathcal{V}(s)$:

$$\tau(v)' = \tau(v) + \sum_{y \in Y} \kappa_1(\tau(v))(y) \cdot \mathit{Offset}(\mathcal{V}_1, s, y) = \tau(v) + \mathit{Offset}(\mathcal{V}', s, v)$$

Thus $\mathcal{V}(s) \models \mathcal{V}'(s)[\bar{\tau}]$ for all $s \in \Sigma$ and so τ is a simulation from $\mathcal{V}$ to $\mathcal{V}'$, and $\langle \sigma, \mathcal{V} \rangle$ is a reflection of $\not{f}$. □

The following theorem should be no surprise:

Theorem 4. For any transition assignment $\not{f} : \Sigma \to \mathbf{TF}(X)$ over a finite alphabet Σ, we can compute the VAS reflection $\langle \sigma, \mathcal{V} \rangle$ of $\not{f}$ in polynomial time.

Proof. We can use Lemma 2 to compute VAS reflections $\langle \sigma_s, \mathcal{V}_s \rangle$ of $\#|_{\{s\}}$ for all $s \in \Sigma$. We may then repeatedly use Lemma 3 to combine these reflections into a single reflection $\langle \sigma, \mathcal{V} \rangle$ of $\#$. □

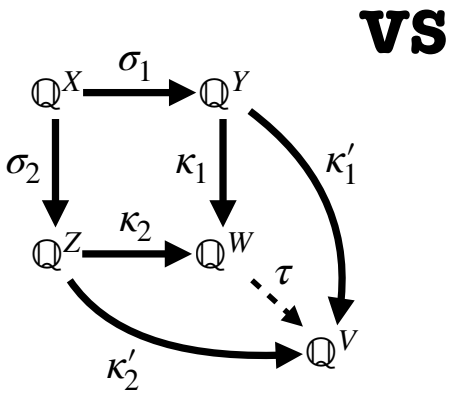
3.4 A Category-Theoretic Generalization of Computing Reflections

The previous two sections described a divide-and-conquer approach to computing VAS reflections of transition assignments. Our technique computed the reflection of the transition system restricted to each character, then combined these reflections to form a reflection of the full system. This subsection aims to generalize the steps of this abstraction process to a recipe that can be applied to the other abstract domains presented in this thesis. The necessary ingredients are (1) a technique for computing a reflection of a singleton transition formula system and (2) a technique for computing a reflection of a transition formula system over Σ from reflections of that system restricted to partitions $\langle \Sigma_1, \Sigma_2 \rangle$ of Σ.

The process of computing the best reflection of a singleton transition system is, in some sense, already generalized under the symbolic abstraction framework of [55]. We were able to identify that the space of terms that any VAS abstraction could possibly track had a finite generator representation (in particular, the space was a vector space and had a basis). Reps et al. [55] gave us algorithm to computing this generator representation, and we were able to compute the reflection directly from the generators. The guarantee that the reflection simulated any other abstraction (and therefore was best) was a direct consequence of the fact that these generators generated the space of possibly trackable terms.

The process of combining reflections requires a more thorough investigation. Given two VAS-reflections $\langle \sigma_1, \mathcal{V}_1 \rangle$ of $\not\parallel_{\Sigma_1}$ and $\langle \sigma_2, \mathcal{V}_2 \rangle$ of $\not\parallel_{\Sigma_2}$, we constructed the VAS reflection $\langle \sigma, \mathcal{V} \rangle$ in two steps.

Our first step was to compute σ, and for that we did not use any information from $\mathcal{V}_1$ or $\mathcal{V}_2$. Instead, we used linear algebra to complete the following commuting diagram in the most general way:



We defined $\sigma = \kappa_1 \circ \sigma_1 = \kappa_2 \circ \sigma_2$. This was a pushout, as described in Chapter 2, in the category of vector spaces **VS**.

In our second step, we computed a VAS $\mathcal{V}$ such that σ was a simulation from $\not\parallel$ to $\mathcal{V}$. To do so, we used κ_1 (and κ_2) to compute $\mathcal{V}|_{\Sigma_1}$ (resp. $\mathcal{V}|_{\Sigma_2}$). And so, informally speaking, we required that we could compute objects in $\mathbf{VAS}(\Sigma_1)$ (resp. $\mathbf{VAS}(\Sigma_2)$) which were simulated via arbitrary arrows from **VS**.

What follows is a category-theoretic formalization of the above intuition. We are able to distinguish the category in which the pushout is computed from the categories which the transition systems inhabit. We are also able to formalize the property that we can interpret the results of the pushout in our original categories. To formalize the relations between the

categories of transition systems over different alphabets and the underlying category of their state space, we use *concrete categories* [3].

Let **Base** be a category. A concrete category over **Base** is another category **C** along with a faithful functor ϕ from **C** to **Base**. The faithfulness of this functor implies that arrows in **C** map to distinct arrows in **Base**.

To make this concrete, consider the following two categories:

1. the category $\mathbf{TS}(\Sigma)$ of transition systems over a finite alphabet Σ defined by transition assignments in which the arrows are linear simulations

2. the category **VS** of finite-dimensional vector spaces over the rationals in which the arrows are linear functions

The category $\mathbf{TS}(\Sigma)$ is a concrete category over **VS**, witnessed by the faithful functor ϕ which maps each transition system $T = \langle \mathbb{Q}^X, \rightarrow_T \rangle$ in $\mathbf{TS}(\Sigma)$ to the underlying state space $\mathbb{Q}^X$, and which maps each arrow in $\mathbf{TS}(\Sigma)$ to the same linear function in **VS**. Every subcategory of $\mathbf{TS}(\Sigma)$, including the category $\mathbf{VAS}(\Sigma)$ of VAS, is similarly a concrete category over **VS**.

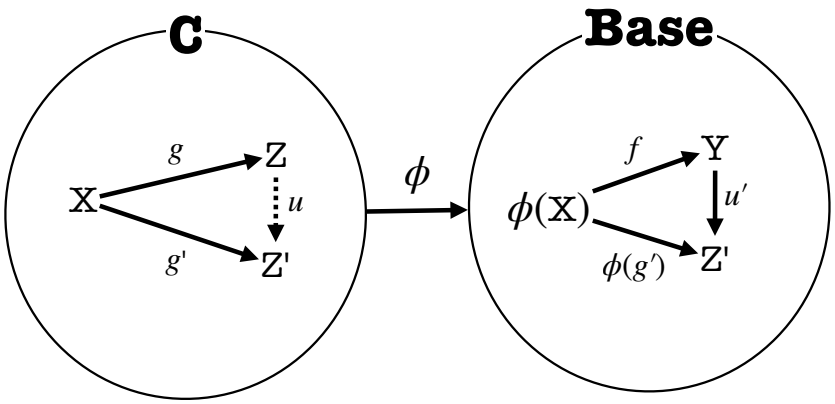
We may now formalize the first ingredient necessary for our combination procedure. To compute an $\mathbf{Abs}(\Sigma)$-reflection of $\not\Vdash$ from an $\mathbf{Abs}(\Sigma_1)$-reflection of $\not\Vdash|_{\Sigma_1}$ and an $\mathbf{Abs}(\Sigma_2)$-reflection of $\not\Vdash|_{\Sigma_2}$, we require that $\mathbf{Abs}(\Sigma)$, $\mathbf{Abs}(\Sigma_1)$, and $\mathbf{Abs}(\Sigma_2)$ are concrete categories over a common base category **Base** which admits pushouts. In our VAS example, we had that $\mathbf{VAS}(\Sigma)$, $\mathbf{VAS}(\Sigma_1)$, and $\mathbf{VAS}(\Sigma_2)$ were concrete categories over **VS**, and we used the pushout of **VS** to compute the simulation of our combined reflection.

The other necessary ingredient, defined formally below, is that we can lift arrows in the base category to arrows in the concrete category. In the VAS combination step, this amounted to taking the pushout maps κ_1 and κ_2 and constructing the combined VAS $\mathcal{V}$ from $\mathcal{V}_1$ and $\mathcal{V}_2$ such that κ_1 is a simulation from $\mathcal{V}_1$ to $\mathcal{V}|_{\Sigma_1}$ and κ_2 is a simulation from $\mathcal{V}_2$ to $\mathcal{V}|_{\Sigma_2}$. We

will find in subsequent chapters that this lifting is not possible over **VS** for more complex abstract domains, and that we therefore must enrich our base category.

Definition 7. Let **C** be a concrete category over **Base** with functor ϕ. We say C is **cofibered**[1] over **Base** if for every object X of **C**, every object Y of **Base**, and every arrow $f \in \mathbf{Base}(\phi(X), Y)$, there exists[2] an object Z of **C** and an arrow $g \in \mathbf{C}(X, Z)$ with $\phi(Z) = Y$ and $\phi(g) = f$, such that for any object Z' of **C** and arrow $g' \in \mathbf{C}(X, Z')$ for which there exists $u' \in \mathbf{Base}(Y, \phi(Z'))$ with $u' \circ f = \phi(g')$, there exists an arrow $u \in \mathbf{C}(Z, Z')$ with $\phi(u) = u'$ and $u \circ g = g'$.

This property represented in the following commutative diagram, in which we have that $f = \phi(g) = \phi(g')$ and $Y = \phi(Z) = \phi(Z')$:



The previous section used the fact that $\mathbf{VAS}(\Sigma_1)$ is cofibered over **VS** in order to compute $\mathcal{V}|_{\Sigma_1}$ from $\mathcal{V}_1$ and κ_1. As we explore more complex abstract domains in subsequent chapters, we will find that these domains are not cofibered over **VS**; there will exist linear functions

[1]The name arises from fibered categories, which is the same condition but with the arrow f reversed: given $f \in \mathbf{Base}(Y, \phi(X))$ pointing into the state space of X, a cartesian lift exists. See [58].

[2]The standard definition of a fibered/cofibered category additionally requires that Z and g are unique, but this is not required for our purposes. For the remainder of this thesis, we adopt the weakened definition of cofibered categories.

f which are not valid simulations between transition systems in our abstract domain. To recover the cofibered property, we will need to enrich the structure of our base category while retaining computable pushouts.

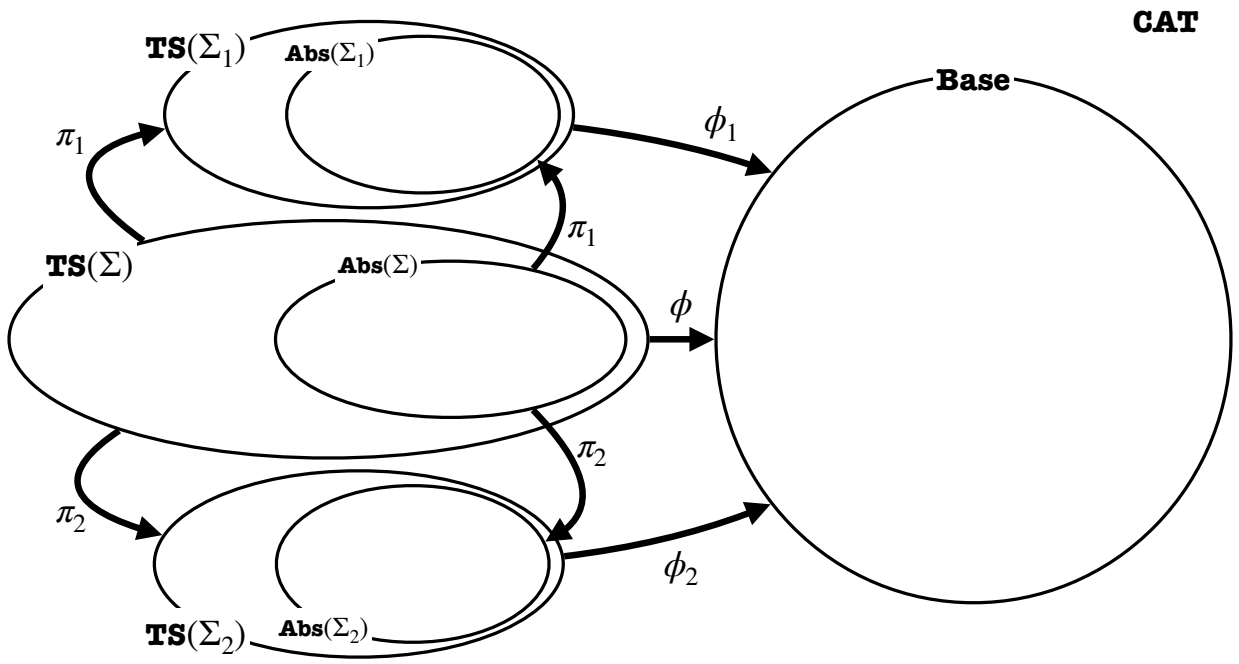
We formalize our category theoretic recipe for combining reflections via the following theorem.

Theorem 5. Let $\mathbf{Abs}(\Sigma)$ be a family of subcategories of $\mathbf{TS}(\Sigma)$. If there exists a category **Base** such that for all finite alphabets Σ we have:

1. $\mathbf{TS}(\Sigma)$ is a concrete category over **Base**
2. **Base** admits pushouts
3. $\mathbf{Abs}(\Sigma)$ is cofibered over **Base**

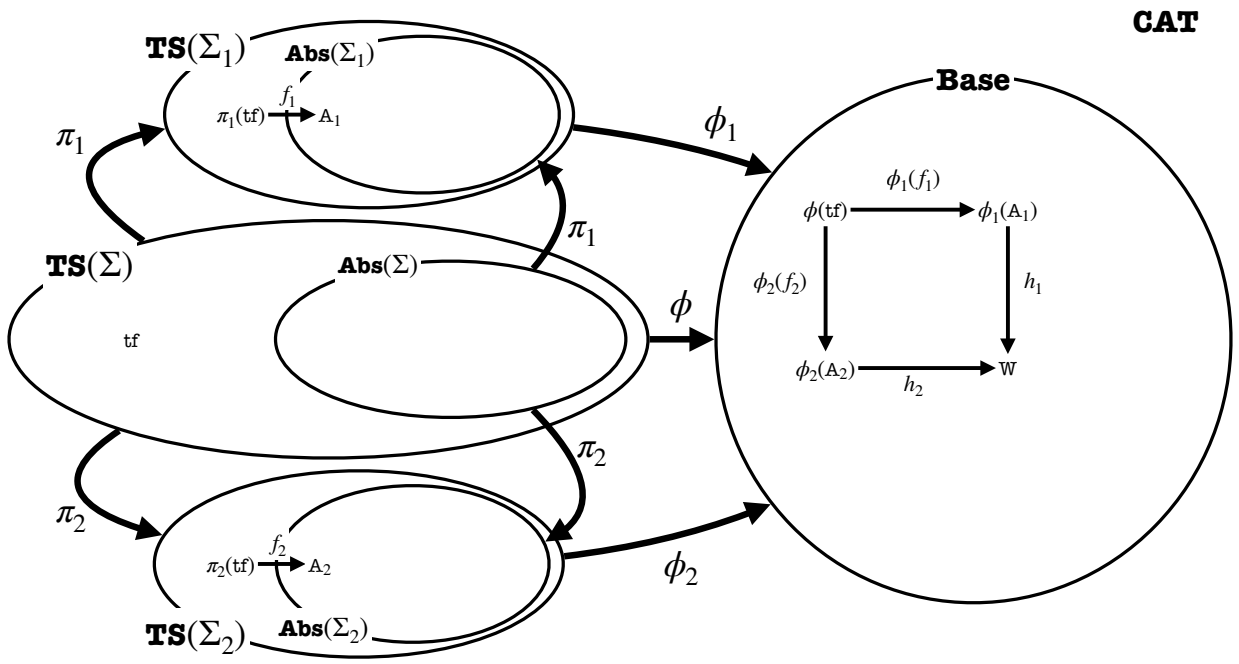
then for all partitions $\langle \Sigma_1, \Sigma_2 \rangle$ of a finite alphabet Σ if $\mathbf{Abs}(\Sigma_1)$ and $\mathbf{Abs}(\Sigma_2)$ are reflective subcategories of $\mathbf{TS}(\Sigma_1)$ and $\mathbf{TS}(\Sigma_2)$ respectively, then $\mathbf{Abs}(\Sigma)$ is a reflective subcategory of $\mathbf{TS}(\Sigma)$.

Proof. Our proof operates in the following diagram. The functors ϕ, ϕ_1 and ϕ_2 are the faithful functors evidencing that $\mathbf{TS}(\Sigma_1)$, $\mathbf{TS}(\Sigma_2)$, and $\mathbf{TS}(\Sigma)$ are concrete categories over **Base** (by assumption 1). The functor π_1 sends every transition system T in $\mathbf{TS}(\Sigma)$ (and $\mathbf{Abs}(\Sigma)$) to $\pi_1(T) = T|_{\Sigma_1}$ in $\mathbf{TS}(\Sigma)$ (resp. $\mathbf{Abs}(\Sigma)$) and every linear simulation f to $\pi(f) = f$. The functor π_2 is defined symmetrically with Σ_2 in place of Σ_1.



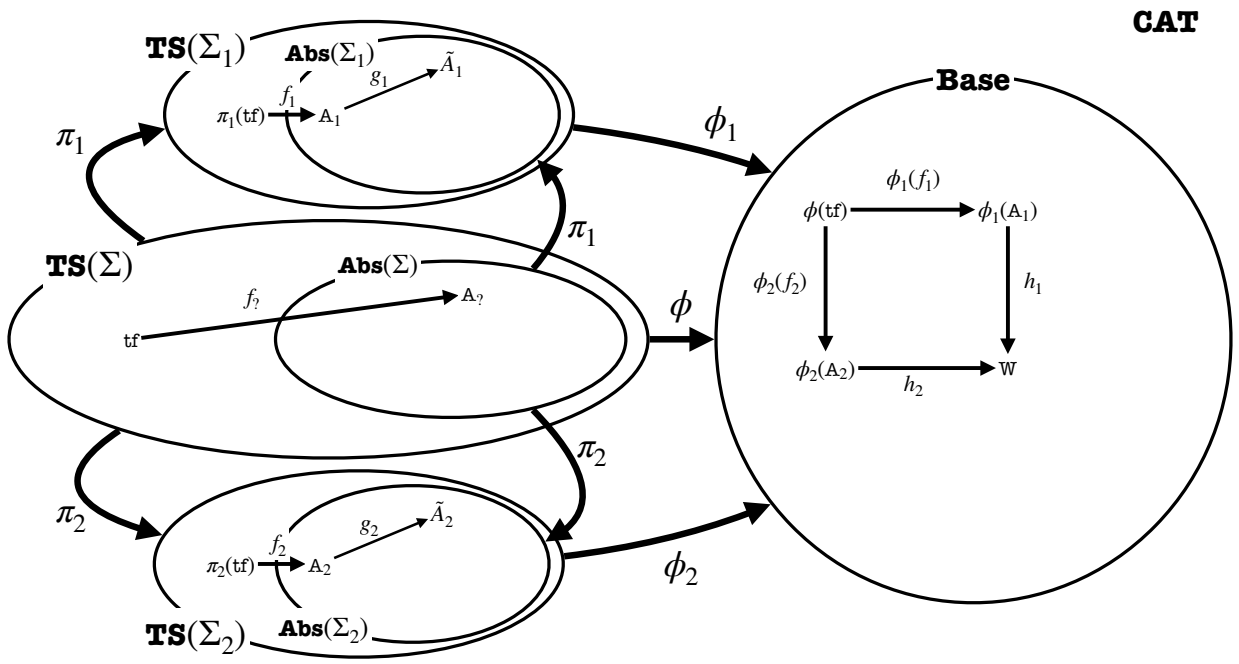
Let $\not{f}$ be any object of $\mathbf{TS}(\Sigma)$. Since $\mathbf{Abs}(\Sigma_1)$ and $\mathbf{Abs}(\Sigma_2)$ are reflective subcategories, let $\langle f_1, A_1\rangle$ and $\langle f_2, A_2\rangle$ be the $\mathbf{Abs}(\Sigma_1)$- and $\mathbf{Abs}(\Sigma_2)$-reflections of $\pi_1(\not{f}) = \not{f}|_{\Sigma_1}$ and $\pi_2(\not{f}) = \not{f}|_{\Sigma_2}$, respectively. We will construct the $\mathbf{Abs}(\Sigma)$-reflection $\langle f_?, A_?\rangle$ of $\not{f}$ and verify universality.

The arrows $\phi_1(f_1) : \phi(\not{f}) \to \phi_1(A_1)$ and $\phi_2(f_2) : \phi(\not{f}) \to \phi_2(A_2)$ are arrows in $\mathbf{Base}$. By condition (2), let (W, h_1, h_2) be a pushout of $\phi_1(f_1)$ and $\phi_2(f_2)$ in $\mathbf{Base}$, so that $h_1 \circ \phi_1(f_1) = h_2 \circ \phi_2(f_2)$.

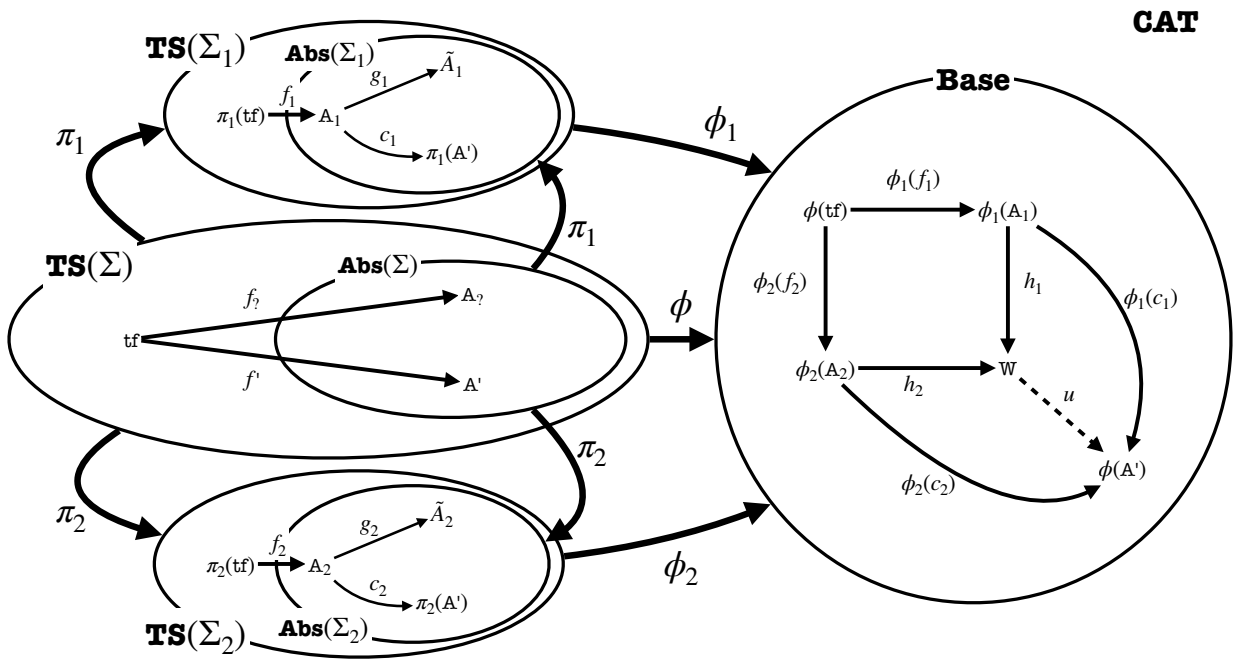


By condition (3), $\mathbf{Abs}(\Sigma_1)$ is cofibered over **Base**. Applying this to A_1, W, and $h_1 \in \mathbf{Base}(\phi_1(A_1), W)$, there exists an object $\tilde{A}_1$ of $\mathbf{Abs}(\Sigma_1)$ and an arrow $g_1 \in \mathbf{Abs}(\Sigma_1)(A_1, \tilde{A}_1)$ with $\phi_1(\tilde{A}_1) = W$ and $\phi_1(g_1) = h_1$. Symmetrically, $\mathbf{Abs}(\Sigma_2)$ is cofibered over **Base**, yielding $\tilde{A}_2$ in $\mathbf{Abs}(\Sigma_2)$ and $g_2 \in \mathbf{Abs}(\Sigma_2)(A_2, \tilde{A}_2)$ with $\phi_2(\tilde{A}_2) = W$ and $\phi_2(g_2) = h_2$.

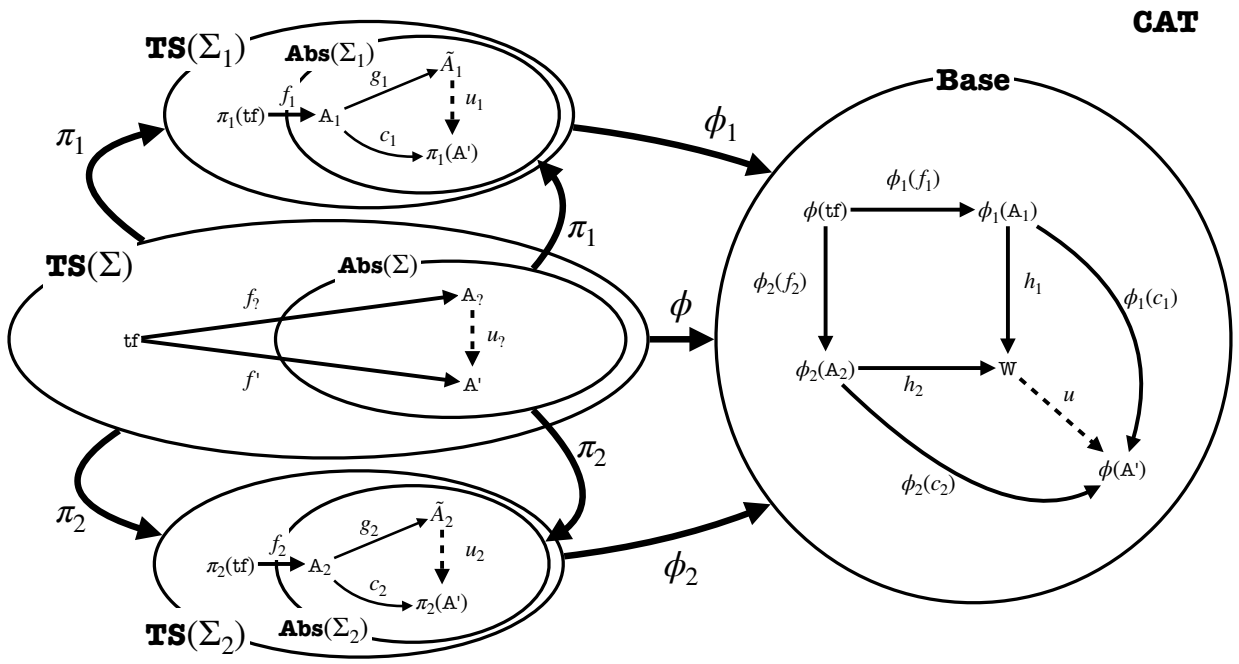
Since a transition system over $\Sigma = \Sigma_1 \cup \Sigma_2$ is precisely a pair of transition systems over Σ_1 and Σ_2 sharing a common state space, $\mathbf{TS}(\Sigma)$ can be identified with the *fibered product* $\mathbf{TS}(\Sigma_1) \times_{Base} \mathbf{TS}(\Sigma_2)$, the category in which the objects are pairs $\langle T_1, T_2 \rangle \in obj(\mathbf{TS}(\Sigma_1) \times obj(\mathbf{TS}(\Sigma_2)$ such that $\phi(T_1) = \phi(T_2)$ and in which the arrows are pairs $\langle h_1, h_2 \rangle$ with $\phi_1(h_1) = \phi_2(h_2)$. In this case, we may define $A_?$ to be the pair $\left\langle \tilde{A}_1, \tilde{A}_2 \right\rangle$ and $f_?$ to be the pair $\langle g_1 \circ f_1, g_2 \circ f_2 \rangle$. We define our $\mathbf{TS}(\Sigma)$-reflection of $\not\!\!f$ to be $\langle f_?, A_? \rangle$.



Universality. Let $\langle f', A' \rangle$ be any $\mathbf{Abs}(\Sigma)$-abstraction of $\not{tf}$. Since $\langle f', \pi_1(A') \rangle$ is an abstraction of $\pi_1(\not{tf})$ and $\langle f_1, A_1 \rangle$ is its reflection, there exists a simulation $c_1 \in \mathbf{Abs}(\Sigma_1)(A_1, \pi_1(A'))$ satisfying $c_1 \circ f_1 = f'$. Symmetrically, there exists $c_2 \in \mathbf{Abs}(\Sigma_2)(A_2, \pi_2(A'))$ satisfying $c_2 \circ f_2 = f'$. Because $\phi(c_1) \circ \phi(f_1) = \phi(f') = \phi(c_2) \circ \phi(f_2)$, the universal property of the pushout (W, a, b) yields a unique arrow $u : W \to \phi(A')$ in **Base** such that $u \circ h_1 = \phi(c_1)$ and $u \circ h_2 = \phi(c_2)$.



Since $\mathbf{Abs}(\Sigma_1)$ is cofibered over **Base**, the existence of the arrow u from W to $\phi(A')$ implies the existence of a simulation u_1 from $\tilde{A}_1$ to $\pi_1(A')$ such that $\phi_1(u_1) = u$. Symmetric reasoning says that there is a simulation u_2 from $\tilde{A}_2$ to $\pi_2(A')$ such that $\phi_2(u_2) = u$. We may define the simulation $u_?$ from $A_?$ to A' as $u_? = \langle u_1, u_2 \rangle$.



Therefore, $\langle f_?, A_? \rangle$ is a $\mathbf{TS}(\Sigma)$-reflection of tf, and so $\mathbf{Abs}(\Sigma)$ is a reflective subcategory.

□

This theorem will be a powerful tool used to compute reflections in the remainder of this thesis. By defining appropriate categories and showing that they meet the above properties, we immediately have a technique for computing reflections.

Chapter 4

Vector Addition Systems with Resets

This chapter introduces Vector Addition Systems with Resets (VASR), the central abstract domain of this thesis upon which subsequent chapters generalize.

Section 4.1 introduces VASRs formally and briefly recounts the historical usage of the model. Section 4.2 develops a polynomial-time procedure to compute a transition formula precisely encoding the context-free reachability relation of any VASR. Section 4.3 describes a divide-and-conquer approach to computing the VASR reflection of a program. These two sections constitute a technique for program analysis that is sound, monotone, and local. We implemented this technique as an analyzer for C programs and evaluated its capabilities in Section 4.4. Our evaluation shows that this technique is not competitive with the state of the art, largely due to our techniques' struggles to model conditional branching. Nevertheless, the theoretical foundations developed here underpin the more expressive domains of the following chapters, which overcome this limitation and advance the state of the art.

4.1 Definition and Examples

Vector Addition Systems with Resets (VASR) are an extension to Vector Addition Systems which allows operations to optionally reset counters before adding in an increment. We again adopt the rational-valued extension introduced in [27].

Definition 8. A **VASR transition** over a set of variables Y is a transition formula in $\mathbf{TF}(Y)$ of the form

$$\bigwedge_{y \in Y} y' = r_y y + a_y,$$

where $r_y \in \{0, 1\}$ and $a_y \in \mathbb{Q}$ for every $y \in Y$. When $r_y = 1$ we say the update is additive with offset a_y; when $r_y = 0$ we say the variable y is reset to the constant a_y.

Definition 9. A **Rational Vector Addition System with Resets (VASR)** over a set of variables Y and a finite alphabet Σ is a transition formula mapping $\mathcal{V} : \Sigma \to \mathbf{TF}(Y)$ in which $\mathcal{V}(s)$ is a VASR transition for every $s \in \Sigma$.

For a VASR $\mathcal{V}$ over Y and Σ, we write:

- $\mathit{Resets}(\mathcal{V}, y) \subseteq \Sigma$ for the set of symbols $s \in \Sigma$ such that $r_y = 0$ in $\mathcal{V}(s)$ (the symbols that reset y)
- $\mathit{Offset}(\mathcal{V}, s, y)$ for the rational a_y in $\mathcal{V}(s)$ (the constant offset added to y by symbol s).

Example 4.1.1. This example explores how VASRs can be used to abstract programs. Recall the grammar and transition assignment representing the `save_tree` program from Example 2.1.1:

$$G = \left\langle \begin{array}{c} \{P\}, \{a, b\}, \\ \{P \Rightarrow aPP, P \Rightarrow ab\}, P \end{array} \right\rangle$$

$$\not{f}(a) \triangleq \begin{pmatrix} \texttt{buf}' = \texttt{buf} + 1 \\ \wedge\ \texttt{mem\_ops}' = \texttt{mem\_ops} \end{pmatrix}$$

$$\not{f}(b) \triangleq \begin{pmatrix} \texttt{buf}' = 0 \\ \wedge\ \texttt{mem\_ops}' = \texttt{mem\_ops} + \texttt{buf} + 1 \end{pmatrix}$$

We can construct a VASR $\mathcal{V}$ over $Y = \{y_1, y_2\}$ that linearly simulates this program, where y_1 tracks `buf` and y_2 tracks `mem_ops` + `buf`. Concretely, the linear simulation f and the VASR $\mathcal{V}$ are:

$$\sigma(y_1) = \texttt{buf} \qquad \mathcal{V}(a) \triangleq y_1' = y_1 + 1 \wedge y_2' = y_2 + 1$$

$$\sigma(y_2) = \texttt{mem\_ops} + \texttt{buf} \qquad \mathcal{V}(b) \triangleq y_1' = 0 \wedge y_2' = y_2 + 1$$

Notably, the VASR tracks the compound term `mem_ops` + `buf` as a single abstract variable, capturing a nontrivial relationship between program variables. By precisely computing the context-free reachability relation of this VASR, we derive invariants about this term across all executions of `save_tree`.

4.2 Context-Free Reachability of VASRs

This section shows that, given a VASR $\mathcal{V}$ over variables Y and alphabet Σ and a context-free grammar G over the same alphabet Σ, we can compute in polynomial time a transition formula $\mathit{Reach}(\mathcal{V}, G) \in \mathbf{TF}(Y)$ that precisely characterizes the reachability relation of $\mathcal{V}$ over $\mathcal{L}(G)$:

$$[\rho, \rho'] \models \mathit{Reach}(\mathcal{V}, G) \iff \rho \xrightarrow{\mathcal{L}(G)}_{\mathcal{V}} \rho'.$$

The following example motivates the key definitions and the approach we take to computing $\mathit{Reach}(\mathcal{V}, G)$.

Example 4.2.1. Consider the VASR $\mathcal{V}$ of Example 4.1.1:

$$\mathcal{V}(a) \triangleq y_1' = y_1 + 1 \wedge y_2' = y_2 + 1$$
$$\mathcal{V}(b) \triangleq y_1' = 0 \wedge y_2' = y_2 + 1$$

As a starter problem, consider computing a transition formula F such that $\langle \rho, \rho' \rangle \models F$ if and only if $\rho \xrightarrow{ababa}_{\mathcal{V}} \rho'$.

For the second variable of the VASR, y_2, the composition of $\mathcal{V}(a)$ and $\mathcal{V}(b)$ along *ababa* can be computed from the character count of a and b within the trajectory, as all VASR transitions increment y_2 and therefore commute. Since there are 3 occurrences of a and 2 occurrences of b, $y_2' = y_2 + 3(1) + 2(1)$.

The transitions along *ababa* do not commute with respect to y_1 due to the reset incurred by $\mathcal{V}(b)$, so we cannot compute their composition from the Parikh image (the character count abstraction defined in Chapter 2) of *ababa*. For example, *aaabb* has the same Parikh image as *ababa* but the composition of operations along *aaabb* resets y_1 to 0 (the final b resets to 0) and composition of operations along *ababa* resets y_1 to 1 (the final b resets to 0 and the following a increments to 1).

Haase and Halfon [27] observed that it is sufficient to identify the final reset of y_2 from left to right and the Parikh image of the sub-word after it; the final reset nullifies the effects of the transitions before it and all transitions after increment the variable and therefore commute.

To formalize this idea, observe that any word $w \in \{a, b, c\}^*$ can be decomposed as $w = w_1 w_2 w_3$ where $w_3 \in \{a, c\}^*$, w_2 is either ϵ or b, and w_1 is in $\{a, c\}^*$ if w_2 is ϵ and is in $\{a, b, c\}^*$ otherwise. Intuitively, w_2 identifies the final reset of y_2, or occurrence of b, from left to right. The transition relation $\xrightarrow{w}_{\mathcal{V}}$ is uniquely determined by the Parikh images of w_1, w_2 and w_3.

As motivated in Example 4.2.1, our goal is to compute a variation of the Parikh image of the language of G which identifies the final time each variable is reset from left to right. Our approach takes inspiration from the generalized Parikh images of [27], but is distinct—see Chapter 8 for a detailed comparison.

Our approach computes abstract trajectories, an abstraction of context free languages that identifies an arbitrary number of symbols in each word and captures the Parikh images of the subwords in between. Any particular trajectory has many abstract trajectories that abstract it, and at least one such abstract trajectory identifies the final reset of each variable. We formally define abstract trajectories in Section 4.2.1, as well as a formalization of the *well-formedness* property that an abstract trajectory identifies the final resets of each variable of a VASR.

We decompose our approach for computing $\mathit{Reach}(\mathcal{V}, G)$ into two components; this division of tasks is our key insight to computing the context-free reachability relation of VASR. In Section 4.2.2, we compute a formula defining the abstract trajectories of a context-free language and conjoin additional formulas ensuring that the computed abstract trajectories are well-formed with respect to our VASR. We then compute a formula describing the transition over VASR variables associated with a well-formed abstract trajectory in Section 4.2.3.

4.2.1 Definitions

Definition 10. A d-marked **abstract trajectory** is a function $n : (\Sigma \times [2d+1]) \to \mathbb{N}$ such that for all even i we have $\sum_{s \in \Sigma} n(s, i) \leq 1$.

We interpret a d-marked abstract trajectory n as an abstraction of a word $w \in \Sigma^*$ that identifies up to d characters in w (at even-indexed positions) and records the Parikh images of the sub-words between them (at odd-indexed positions). For a trajectory $w \in \Sigma^*$ and d-marked abstract trajectory n, write $w \Vdash n$ if there exists a decomposition $w = w_1 \dots w_{2d+1}$ such that $n(s, i) = \pi(w_i)(s)$ for all i and s. For any even index i, the constraint $\sum_s n(s, i) \leq 1$ means there is at most one symbol with a positive count at that position, so the even-indexed "words" $w_2, w_4, \dots, w_{2d}$ are all either empty or single characters. The odd-indexed words $w_1, w_3, \dots, w_{2d+1}$ are arbitrary sub-words, recorded only by their Parikh images.

We want to compute the abstract trajectories of a context free language which identify the *final reset* of each variable $y \in Y$ of the VASR from left to right. An abstract trajectory with this property is called well-formed.

Definition 11. A $|Y|$-marked abstract trajectory n is **well-formed** with respect to a VASR $\mathcal{V}$ over variables Y if for every $y \in Y$ and every odd index $i \in [2|Y| + 1]$:

$$\begin{pmatrix} \text{there exists } s \in \mathit{Reset}(\mathcal{V}, y) \\ \text{with } n(s, i) > 0 \end{pmatrix} \implies \begin{pmatrix} \text{there exists even } j > i,\ s' \in \mathit{Reset}(\mathcal{V}, y) \\ \text{with } n(s', j) > 0 \end{pmatrix}$$

Informally, whenever a reset of y appears anywhere in an odd-indexed sub-word, a later reset must appear at a subsequent even-indexed position. This ensures that every identified character at an even position is a reset of a variable that occurs after all resets to that variable within odd-indexed sub-words, making it a "final reset" of that variable. Note that a well-formed abstract trajectory is not required to identify the final reset of a variable if that variable is not reset within a word.

Example 4.2.2. Consider the VASR $\mathcal{V}$ of Example 4.2.1:

$$\mathcal{V}(a) \triangleq y_1' = y_1 + 1 \wedge y_2' = y_2 + 1$$
$$\mathcal{V}(b) \triangleq y_1' = 0 \wedge y_2' = y_2 + 1$$

Since this VASR has 2 variables, we are interested in 2-marked abstract trajectories.

Consider the word $w = ababa$. One abstract trajectory n of w arises from the decomposition $w_1 = \epsilon$, $w_2 = \epsilon$, $w_3 = aba$, $w_4 = b$, and $w_5 = a$ (note that $w = w_1w_2w_3w_4w_5$):

$$n(a,3) = 2 \quad n(b,3) = 1 \quad n(b,4) = 1 \quad n(a,5) = 1$$

$$n(\cdot,\cdot) = 0 \text{ for all other inputs}$$

This trajectory is well-formed. The variable y_1 is never reset, so the condition is vacuously satisfied. The variable y_2 is reset at subword 3 ($n(b,3) > 0$), but there is a greater even index at which it is reset ($n(b,4) > 0$).

A second abstract trajectory n' of w arises from the decomposition $w_1 = \epsilon$, $w_2 = \epsilon$, $w_3 = a$, $w_4 = b$, and $w_5 = aba$:

$$n'(a,3) = 1, \quad n'(b,4) = 1, \quad n'(a,5) = 2, \quad n'(b,5) = 1$$

$$n'(\cdot,\cdot) = 0 \text{ for all other inputs}$$

This trajectory is not well-formed. The variable y_2 is reset at subword 5 ($n'(b,5) > 0$) but there is no greater even index which resets the variable.

Intuitively, n is sufficient to compute the composition of VASR operations along *ababa*, as all words that are represented by it increment y_1 by 5 and reset y_2 to 1. However, n' is insufficient because it could also represent *abaab*, which would reset y_2 to 2.

4.2.2 Abstract Trajectories of Context-Free Languages

We start by constructing a formula $AT(d, G)$ that characterizes exactly the set of d-marked abstract trajectories realizable by words in $\mathcal{L}(G)$. We then refine this formula to only identify well-formed abstract trajectories relative to a VASR $\mathcal{V}$.

Formally, we aim to compute, given a context-free grammar $G = \langle N, \Sigma, R, s_0 \rangle$ and a VASR $\mathcal{V}$ over Y, a formula $AT(|Y|, G)$ that represents the set of $|Y|$-marked abstract trajectories n such that $w \Vdash n$ for some trajectory w in $\mathcal{L}(G)$. This formula has free variables $c_{s,i}$ for $s \in \Sigma$ and $i \in [2|Y| + 1]$. It meets the condition that $AT(|Y|, G)[c_{s,i} \mapsto n(s, i)]$ holds if and only if there exists some $w \in \mathcal{L}(G)$ such that $w \Vdash n$.

Our strategy is to define $AT(|Y|, G)$ as the Parikh image formula of an expanded grammar derived from G by replacing each character with tagged copies that record which sub-word it occupies. We may identify the correspondence between abstract trajectories and Parikh images as follows. Consider the following regular language $\mathcal{O}$, in which each $\Sigma_i \triangleq \{\langle s, i \rangle : s \in \Sigma\}$ is a copy of Σ in which all characters are "tagged" with numbers. Observe that the Parikh image of $\mathcal{O}$ is equal to the set of all abstract trajectories over Σ

$$\mathcal{O} \triangleq \Sigma_1^*(\Sigma_2 + \epsilon)\Sigma_3^* \dots \Sigma_{2|Y|-1}^*(\Sigma_{2|Y|} + \epsilon)\Sigma_{2|Y|+1}^*$$

Let $h : (\Sigma \times [2|Y| + 1])^* \to \Sigma^*$ be the homomorphism that maps $\langle a, i \rangle \mapsto a$ for all i. Consider the language $h^{-1}(\mathcal{L}(G)) \cap \mathcal{O}$; since context-free languages are closed under inverse homomorphism and intersection with regular languages, this language is context-free. This

language represents the possible decompositions of words w in the language of G into numbered decompositions $w_1 \dots w_{2|Y|+1}$. One can then observe that $n \in \pi(h^{-1}(\mathcal{L}(G)) \cap \mathcal{O})$ if and only if there exists some trajectory $w \in \mathcal{L}(G)$ such that $w \Vdash n$.

This section assumes G is in Chomsky Normal Form. We proceed by constructing a grammar $\mathcal{I}(G, |Y|) \triangleq \langle N_{[]}, \Sigma \times [2|Y|+1], R_{[]}, s_{[1,2|Y|+1]} \rangle$ that recognizes the language $h^{-1}(\mathcal{L}(G)) \cap \mathcal{O}$ as follows; the formula $AT(|Y|, G)$ is defined to be its Parikh image formula $Parikh(I(G, |Y|))$.

- The non-terminal symbols are defined to be

$$N_{[]} \triangleq \left\{ n_{[2i+1:2j+1]} : n \in N, 0 \le i \le j \le |Y| \right\}$$

  The intention of the grammar design is that the set of words derivable from $n_{[i:j]}$ is $\mathcal{L}_{I(G,Y)}(n_{[i:j]}) = h^{-1}(\mathcal{L}_G(n)) \cap (\Sigma_i^*(\Sigma_{i+1} + \epsilon) \dots (\Sigma_{j-1} + \epsilon)\Sigma_j^*)$.

- The productions are defined to be

$$\begin{aligned} R_{[]} \triangleq\ & \left\{ A_{[2i+1:2j+1]} \Rightarrow B_{[2i+1:2k+1]} C_{[2k+1:2j+1]} : A \Rightarrow BC \in R, 0 \le i \le k \le j \le |Y| \right\} \\ & \cup \left\{ A_{[2i+1:2j+1]} \Rightarrow \langle a, k \rangle : A \Rightarrow a \in R, 2i+1 \le k \le 2j+1, 0 \le i \le j \le |Y| \right\} \\ & \cup \left\{ s_{[1:2d+1]} \Rightarrow \epsilon : s \Rightarrow \epsilon \in R \right\} \end{aligned}$$

The design of the production rules maintains the invariant throughout the derivation of any word that for any even k, there is at most one $n_{[i:j]}$ capable of producing a terminal symbol in Σ_k. This ensures that the output of the grammar is in $\mathcal{O}$; all derived words are additionally in $h^{-1}(\mathcal{L}_G(n))$ because all production rules are structurally identical to those of G.

Theorem 6. For any grammar $G = (N, \Sigma, R, s_0)$ (in Chomsky Normal Form), we have

$$\mathcal{L}(\mathcal{I}(G, Y)) = h^{-1}(\mathcal{L}(G)) \cap \mathcal{O}$$

Moreover, observe that $\mathcal{I}(G, Y)$ has $O(|Y||\Sigma|)$ terminals, $O(|Y|^2|N|)$ nonterminals, $O(|Y|^3|R|)$ production rules, and can be constructed in polynomial time.

We proceed by defining following formula $WF(\mathcal{V})$ to ensure the abstract trajectory is well-formed. We first define a helper formula: $FR(\mathcal{V}, y, j)$ is true if and only if the even index j carries the "final reset" of variable y (some reset symbol of y appears at position j, and no reset symbol of y appears at any later even position).

$$FR(\mathcal{V}, y, j) \triangleq \left(\bigvee_{s \in Resets(\mathcal{V}, y)} c_{s,j} > 0 \right) \wedge \left(\bigwedge_{\substack{s \in Resets(\mathcal{V}, y) \\ j < k \leq 2|Y|+1}} c_{s,k} = 0 \right)$$

$$WF(\mathcal{V}) \triangleq \bigwedge_{y \in Y} \left(\bigvee_{j=1}^{|Y|} FR(\mathcal{V}, y, 2j) \vee \bigwedge_{\substack{s \in Reset(\mathcal{V}, i) \\ k \in [2|Y|+1]}} c_{s,k} = 0 \right)$$

Theorem 7. Let G be a context-free grammar and $\mathcal{V}$ be a VASR over Y. Define $AT(|Y|, G) \triangleq Parikh(\mathcal{I}(G, Y))$. Then for all $|Y|$-marked abstract trajectories n that are well-formed with respect to $\mathcal{V}$, we have:

$$(AT(|Y|, G) \wedge WF(\mathcal{V}))\,[c_{s,i} \mapsto n(s, i)] \iff w \Vdash n \text{ for some } w \in \mathcal{L}(G)$$

4.2.3 Transitions of Abstract Trajectories

We now define a formula $Transition(\mathcal{V})$ that, given a well-formed abstract trajectory n such that $w \Vdash n$, computes the state transformation corresponding to executing w in $\mathcal{V}$. The free variables of $Transition(\mathcal{V})$ are the pre- and post-state variables Y and Y', along with integer variables $c_{s,k}$ for all $s \in \Sigma$ and $k \in [2|Y| + 1]$; each variable $c_{s,k}$ of the latter represents the count $n(s, k)$ of a $|Y|$-marked abstract trajectory.

We first define a helper term: $\mathit{After}(y, j)$ is an integer-valued term that is the total offset accumulated by y from position j onwards.

$$\mathit{After}(y, j) \triangleq \sum_{\substack{s \in \Sigma \\ k \geq j}} \mathit{Offset}(\mathcal{V}, s, y) \cdot c_{s,k}$$

The $\mathit{Transition}(\mathcal{V})$ formula handles two cases for each variable y: either y is reset at some even position (in which case y' equals the offset accumulated after that final reset), or y is never reset (in which case y' equals y plus the total offset across all positions). Recall that $\mathit{FR}(\mathcal{V}, y, j)$ is true if and only if the even index j carries the final reset of variable y, as defined in Section 4.2.2.

$$\mathit{Transition}(\mathcal{V}) \triangleq \bigwedge_{y \in Y} \left(\begin{array}{c} \bigvee_{j=1}^{|Y|} \Big(\mathit{FR}(\mathcal{V}, y, 2j) \ \wedge \ y' = \mathit{After}(y, 2j) \Big) \\ \vee \ \left(\left(\bigwedge_{\substack{k \in [2|Y|+1] \\ s \in \mathit{Resets}(\mathcal{V}, y)}} c_{s,k} = 0 \right) \ \wedge \ y' = y + \mathit{After}(y, 1) \right) \end{array} \right)$$

The correctness of this formula is stated by the following theorem.

Theorem 8. Let $\mathcal{V}$ be a VASR over variables Y, let $w \in \Sigma^*$ be a trajectory, and let n be a $|Y|$-marked abstract trajectory that is well-formed with respect to $\mathcal{V}$ and satisfies $w \Vdash n$. Then for all states $\rho, \rho' \in \mathbb{Q}^Y$:

$$[\rho, \rho'] \models \mathit{Transition}(\mathcal{V})[c_{s,i} \mapsto n(s, i)] \iff \rho \xrightarrow{w}_{\mathcal{V}} \rho'.$$

Proof sketch. VASR transitions update each variable independently, so it suffices to check each $y \in Y$ separately. If y is never reset along w, all offsets accumulate additively and $y' = y + \mathit{After}(y, 1)$. Otherwise, well-formedness guarantees a unique final-reset index $2j$: by definition of a reset, the final reset nullifies all preceding contributions to y, and since no

reset appears after position $2j$, the remaining transitions contribute purely additively. Thus $y' = \mathit{After}(y, 2j)$. Both cases are precisely the disjuncts of $\mathit{Transition}(\mathcal{V})$. □

We may conclude this section by defining $\mathit{Reach}(\mathcal{V}, G)$.

With $C \triangleq \{c_{s,i} : s \in \Sigma, i \in [2|Y| + 1]\}$, define:

$$\mathit{Reach}(\mathcal{V}, G) \triangleq \exists C. \mathit{AT}(\mathcal{V}, G) \wedge \mathit{WF}(\mathcal{V}) \wedge \mathit{Transition}(\mathcal{V})$$

Theorem 9. There is a polynomial-time procedure which, given a VASR $\mathcal{V}$ over alphabet Σ and a grammar G over the same alphabet, computes a formula $\mathit{Reach}(\mathcal{V}, G)$ such that:

$$[\rho, \rho'] \models \mathit{Reach}(\mathcal{V}, G) \iff \rho \xrightarrow{\mathcal{L}(G)}_{\mathcal{V}} \rho'$$

Proof sketch. The formula $\mathit{Reach}(\mathcal{V}, G)$ is the conjunction of three components whose correctness is established individually: $\mathit{AT}(|Y|, G) \wedge \mathit{WF}(\mathcal{V})$ identifies all well-formed abstract trajectories of $\mathcal{L}(G)$ (Theorem 7) and $\mathit{Transition}(\mathcal{V})$ computes the correct state transformation for any well-formed abstract trajectory (Theorem 8). All three are polynomial-size, so the construction runs in polynomial time. □

4.3 Best VASR Abstractions

Following the recipe introduced in Section 3.1, the next ingredient in developing a monotone program analysis based on VASR is to compute the VASR reflection of any transition system in our program model. We follow the structure used in Chapter 3 to compute VAS reflections: we define a procedure to compute the VASR reflection of a singleton transition assignment, then describe a procedure for combining reflections over disjoint alphabets.

4.3.1 Per-Letter VASR Reflections

This section shows how to compute the VASR reflection $\langle \sigma, \mathcal{V} \rangle$ of a transition assignment $\mathit{ff} : \Sigma \to \mathbf{TF}(X)$ in the special case that Σ consists of a single character s. In other words, we show that any transition formula $F \in \mathbf{TF}(X)$ has a best abstraction as a VASR transition formula.

The VASR transitions that can abstract $\mathit{ff}(s)$ are constrained to track terms over X that are either reset or additively offset in all models of F. The key observation is that the set of such terms are each vector spaces, and their bases determine a canonical best abstraction.

For a transition formula $F \in \mathbf{TF}(X)$, define:

$$\begin{aligned} \mathit{Res}(F) &\triangleq \{\langle t, a\rangle \in \mathit{LinTerm}(X) \times \mathbb{Q} : F \models t' = a\} \\ \mathit{Add}(F) &\triangleq \{\langle t, b\rangle \in \mathit{LinTerm}(X) \times \mathbb{Q} : F \models t' = t + b\} \end{aligned}$$

$\mathit{Res}(F)$ is the set of all terms that F always resets to a constant; $\mathit{Add}(F)$ is the set of all terms that F increments by a fixed offset. Both sets are vector spaces (closed under addition and scalar multiplication). We can compute bases for each via the symbolic abstraction framework of [55]; specifically, we may use Algorithm 1 directly to compute a basis for $\mathit{Add}(F)$ and replace $m(x') - m(x)$ with $m(x')$ on lines 2 and 6 to compute a basis for $\mathit{Res}(F)$.

Let $\{\langle t_1, a_1\rangle, \ldots, \langle t_n, a_n\rangle\}$ and $\left\{\left\langle \hat{t}_1, b_1\right\rangle, \ldots, \left\langle \hat{t}_m, b_m\right\rangle\right\}$ be bases of $\mathit{Res}(F)$ and $\mathit{Add}(F)$ respectively. Then, the VASR reflection $\langle \sigma, \mathcal{V} \rangle$ of $\mathit{ff}|_{\{s\}}$ can be defined as the following.

$$\mathcal{V}(s) \triangleq \left(\bigwedge_{i=1}^{n} y_i' = a_i\right) \wedge \left(\bigwedge_{i=1}^{m} z_i' = z_i + b_i\right)$$
$$\sigma(y_i) = t_i \qquad \sigma(z_i) = \hat{t}_i$$

Lemma 4. $\langle \sigma, \mathcal{V} \rangle$ is a VASR reflection of $\mathit{ff}|_{\{s\}}$.

Proof. Let $\langle \sigma', \mathcal{V}' \rangle$ be a different VASR abstraction of $\not\!\!f$ over variables V. We will show that there exists a simulation τ from $\mathcal{V}$ to $\mathcal{V}'$.

Consider a particular $v \in V$. If $\mathcal{V}'(s)$ resets v (if $s \in \mathit{Resets}(\mathcal{V}', v)$) then we must have that $\langle \sigma'(v), \mathit{Offset}(\mathcal{V}', s, v) \rangle \in \mathit{Res}(\not\!\!f(s))$:

$$\begin{aligned} [\rho, \rho'] \models \not\!\!f(s) &\implies [\rho, \rho'] \models \mathcal{V}'(s)[\bar{\sigma'}] \\ &\implies \sigma'(v) = \mathit{Offset}(\mathcal{V}', s, v) \end{aligned}$$

Then, since $\{\langle t_1, a_1 \rangle \ldots \langle t_n, a_n \rangle\}$ is a basis for this space, there must exist coefficients $\alpha_1, \ldots, \alpha_n$ such that $\langle \sigma'(v), \mathit{Offset}(\mathcal{V}', s, v) \rangle = \sum_{i=1}^{n} \alpha_i \langle t_i, a_i \rangle$, and in particular such that $\mathit{Offset}(\mathcal{V}', s, v) = \sum_{i=1}^{n} \alpha_i a_i$.

By symmetric reasoning, if $\mathcal{V}'(s)$ increments v then we must have that $\langle \sigma'(v), \mathit{Offset}(\mathcal{V}', s, v) \rangle \in \mathit{Add}(\not\!\!f(s))$. Since $\left\{\left\langle \hat{t}_1, b_1 \right\rangle, \ldots, \left\langle \hat{t}_m, b_m \right\rangle\right\}$ is a basis for this space, there exists coefficients $\beta_1, \ldots, \beta_m$ such that $\langle \sigma'(v), \mathit{Offset}(\mathcal{V}', s, v) \rangle = \sum_{j=1}^{m} \beta_j \left\langle \hat{t}_j, b_j \right\rangle$, and in particular such that $\mathit{Offset}(\mathcal{V}', s, v) = \sum_{j=1}^{m} \beta_j b_j$.

Then, let $\tau : V \to \mathit{LinTerm}(Y) \cup \mathit{LinTerm}(Z)$ be the substitution such that $\tau(v) = \sum_{i=1}^{n} \alpha_i y_i$ for all $v \in V$ such that $\not\!\!f(s)$ resets v, and $\tau(v) = \sum_{j=1}^{m} \beta_j z_j$ for all $v \in V$ such that $\not\!\!f(s)$ increments v. One can observe that $\sigma \circ \tau = \sigma'$.

It remains to show that $\mathcal{V}(s) \models \mathcal{V}'(s)[\bar{\tau}]$. This holds because $\mathcal{V}(s)$ asserts $y'_i = a_i$, and so for each $v \in V$ such that $\mathcal{V}'(s)$ resets v we have:

$$\begin{aligned} \bar{\tau}(v') &= \sum_i \alpha_i y'_i \\ &= \sum_i \alpha_i a_i \\ &= \mathit{Offset}(\mathcal{V}', s, v) \end{aligned}$$

Similarly, $V(s)$ asserts $z'_j = z_j + b_j$ and so for each $v \in V$ such that $\mathcal{V}'(s)$ increments v we have:

$$\begin{aligned}\bar{\tau}(v') &= \sum_j \beta_j z'_j \\ &= \sum_j \beta_j (z_j + b_j) \\ &= \bar{\tau}(v) + \mathit{Offset}(\mathcal{V}', s, v)\end{aligned}$$

Therefore, $\langle \sigma, \mathcal{V} \rangle$ is a VASR reflection of $\not{f}|_{\{s\}}$. □

Example 4.3.1. Recall the following transition formula from Example 4.1.1:

$$\not{f}(b) \triangleq \begin{pmatrix} \texttt{buf}' = 0 \\ \wedge\ \texttt{mem\_ops}' = \texttt{mem\_ops} + \texttt{buf} + 1 \end{pmatrix}$$

The bases of $\mathit{Res}(\not{f}(b))$ and $\mathit{Add}(\not{f}(b))$ are respectively:

$$\{\langle \texttt{buf},\, 0 \rangle\} \text{ and } \{\langle \texttt{mem\_ops} + \texttt{buf}, 1 \rangle\}$$

. Thus the reflection $\langle \sigma, \mathcal{V} \rangle$of $\not{f}|_{\{b\}}$ is:

$$\sigma(y_1) = \texttt{buf} \qquad \sigma(z_1) = \texttt{mem\_ops} + \texttt{buf}$$

$$\mathcal{V}(b) \triangleq y'_1 = 0 \wedge z'_1 = z_1 + 1$$

4.3.2 Combining VASR Reflections over Disjoint Alphabets

Given VASR reflections of $\not{\mathit{f}}|_{\Sigma_1}$ and $\not{\mathit{f}}|_{\Sigma_2}$ for a partition Σ_1, Σ_2 of Σ, we now show how to combine them into a reflection of the whole $\not{\mathit{f}}$. Our technique is an adaptation of [62, Algorithm 2] to the setting of labeled transition systems; for a detailed comparison, see Chapter 8. Our technique is a direct instantiation of the category theoretic recipe of Chapter 3; we define appropriate categories and use Theorem 5 to combine VASR reflections.

The following example describes the computation of a VASR reflection and motivates our approach.

Example 4.3.2. Recall the transition formula mapping $\not{\mathit{f}}$ of Example 4.1.1:

$$\not{\mathit{f}}(a) \triangleq \begin{pmatrix} \mathtt{buf}' = \mathtt{buf} + 1 \\ \wedge\ \mathtt{mem\_ops}' = \mathtt{mem\_ops} \end{pmatrix} \qquad \not{\mathit{f}}(b) \triangleq \begin{pmatrix} \mathtt{buf}' = 0 \\ \wedge\ \mathtt{mem\_ops}' = \mathtt{mem\_ops} + \mathtt{buf} \end{pmatrix}$$

Consider the partition $\Sigma_1 = \{a\}$, $\Sigma_2 = \{b\}$ of $\Sigma = \{a, b\}$. Consider the following VASR reflections $\langle \sigma_1, \mathcal{V}_1 \rangle$ of $\not{\mathit{f}}|_{\Sigma_1}$ and $\langle \sigma_2, \mathcal{V}_2 \rangle$ of $\not{\mathit{f}}|_{\Sigma_2}$:

$$\begin{matrix} \sigma_1(x_1) = \mathtt{buf} \\ \sigma_1(x_2) = \mathtt{mem\_ops} \end{matrix} \qquad \mathcal{V}_1(a) \triangleq x_1' = x_1 + 1 \wedge x_2' = x_2$$

$$\begin{matrix} \sigma_2(y_1) = \mathtt{buf} \\ \sigma_2(y_2) = \mathtt{mem\_ops} + \mathtt{buf} \end{matrix} \qquad \mathcal{V}_2(b) \triangleq y_1' = 0 \wedge y_2' = y_2 + 1$$

The VASR reflection of $\not{\mathit{f}}$ can clearly track the term $\mathtt{buf}$, as this term is tracked by both of the individual reflections above. However, it cannot track the term $\mathtt{mem\_ops}$ because although the term can be expressed as a linear combination of the terms that the reflection of $\not{\mathit{f}}|_{\Sigma_2}$ tracks, the combination of a reset term and an additive term does

not yield a well-formed VASR transition. It can track $\texttt{mem\_ops} + \texttt{buf}$, as this term is tracked by the reflection of $\not\#|_{\Sigma_2}$ and can be expressed as a linear combination of terms tracked by the reflection of $\not\#|_{\Sigma_1}$ that are all additive.

The VASR reflection $\langle \sigma, \mathcal{V} \rangle$ of $\not\#$ is

$$\sigma(z_1) = \texttt{buf} \qquad \mathcal{V}(a) \triangleq z_1' = z_1 + 1 \wedge z_2' = z_2 + 1$$

$$\sigma(z_2) = \texttt{mem\_ops} + \texttt{buf} \qquad \mathcal{V}(b) \triangleq z_1' = 0 \wedge z_2' = z_2 + 1$$

At a high level, the key difference between VAS and VASR regarding combining reflections is that simulations between VASRs require additional structure - they cannot track terms which combine variables that are reset and variables that are incremented.

We aim to combine VASR reflections via Theorem 5, which requires that we define a base category **Base** such that:

1. $\mathbf{TS}(\Sigma)$ is a concrete category over **Base**
2. **Base** admits pushouts
3. $\mathbf{VASR}(\Sigma)$ is cofibered over **Base**

In Chapter 3, we computed VAS reflections with this recipe using the category of vector spaces **VS**. The following lemma proves that **VS** is inadequate for computing VASR reflections because $\mathbf{VASR}(\Sigma)$ is not cofibered over **VS**, intuitively because some linear functions within **VS** can combine VASR dimensions that are incremented and reset.

After noting this problem, we instrument **VS** with more structure, creating a new category **Sep**. The objects in this category are *separated spaces*, linear spaces equipped with a canonical decomposition, and the arrows are *coherent linear maps*, linear which preserve these decompositions. We can use separated spaces to decompose the state space into subspaces which are either reset or incremented; the coherence property then coincides with

the desired property to not combine dimensions that are reset and incremented. We show that this category fulfills the conditions of Theorem 5, and that we may therefore use this theorem in order to combine VASR reflections into a global one.

Observation 1. $\mathbf{VASR}(\Sigma)$ is not cofibered over $\mathbf{VS}$.

Proof. Consider the VASR $\mathcal{V}(a) \triangleq x_1' = 0 \wedge x_2' = x_2 + 1$ and the linear substitution $\sigma(y) = x_1 + x_2$. We have that σ is an arrow in $\mathbf{VS}$, but there does not exist any VASR $\mathcal{V}'$ such that σ is a simulation from $\mathcal{V}$ to $\mathcal{V}'$. The essential problem is that σ mixes variables that are reset and incremented, and so the resulting transition does not reset or increment its variable, making it not a VASR.

Suppose for the sake of contradiction that a VASR $\mathcal{V}'$ existed such that σ was a simulation from $\mathcal{V}$ to $\mathcal{V}'$. Then, $\mathcal{V}'(a) \triangleq y' = r * y + o$ for some $r \in \{0, 1\}$ and $o \in \mathbb{Q}$. The simulation implies:

$$x_1' = 0 \wedge x_2' = x_2 + 1 \models x_1' + x_2' = r * (x_1 + x_2) + o$$

Substituting x_1' and x_2' based on the left hand side yields $x_2 + 1 = r * (x_1 + x_2) + o$, which must hold for all possible x_1, x_2. Setting $x_1 = x_2 = 0$ yields $o = 1$ and then setting $x_1 = x_2 = 1$ yields $1 = r * 2$, which is impossible if $r \in \{0, 1\}$.

Therefore σ is not a simulation to any VASR, and so $\mathbf{VASR}(\Sigma)$ is not cofibered over $\mathbf{VS}$. □

Intuitively, the problem with $\mathbf{VS}$ is that its arrows carry no information about which dimensions are reset and which are incremented, so there is no way to enforce that a linear map respects this structure. The category $\mathbf{Sep}$ fixes this by equipping each space with an explicit decomposition and enforcing that arrows respect this structure. The following example illustrates the idea before we give the formal definitions.

Example 4.3.3. Consider the following two VASRs:

$$\mathcal{V}(a) \triangleq x_1' = x_1 + 1 \wedge x_2' = x_2 + 3 \wedge x_3' = 5 \wedge x_4' = 2$$

$$\mathcal{V}'(a) \triangleq y_1' = y_1 + 4 \wedge y_2' = 7$$

The substitution $\sigma(y_1) = x_1 + x_2$, $\sigma(y_2) = x_3 + x_4$ is a simulation from $\mathcal{V}$ to $\mathcal{V}'$. This simulation is valid because it respects the natural decomposition of the state space of $\mathcal{V}$ into additive dimensions $\{x_1, x_2\}$ and reset dimensions $\{x_3, x_4\}$.

We can formalize why this simulation is valid by associating $\mathcal{V}$ with the spaces $\{span(\{x_1, x_2\}), span(\{x_3, x_4\})\}$ and $\mathcal{V}'$ with the spaces $\{span(\{y_1\}), span(\{y_2\})\}$. There exists a witness function w_σ recognizing the coherence of σ with respect to these spaces by mapping $span(\{y_1\})$ to $span(\{x_1, x_2\})$ and $span(\{y_2\})$ to $span(\{x_3, x_4\})$. We have that for all spaces C' of $\mathcal{V}'$ and all terms $t \in C'$, we have $\sigma(t) \in w_\sigma(C')$.

We formalize this idea by defining **Sep**, a category which does fulfill the necessary conditions of Theorem 5 for VASR.

Definition 12. A **separated space** $S = \langle V, D \rangle$ consists of a rational vector space V and a finite set $D = \{C_1, \ldots, C_k\}$ whose direct sum is $V = \bigoplus_i C_i$. For each $C \in D$, we use $\text{proj}_C : V \to C$ for the projection onto C.

Definition 13. A coherent linear map from $S = \langle V, D \rangle$ to $S' = \langle V', D' \rangle$ is a pair $\langle f, w \rangle$ where $f : V \to V'$ is a linear map and $w : D' \to D$ is a witness function such that for all $C \in D$ and $C' \in D'$:

$$C \neq w(C') \implies \text{proj}_{C'}(f(v)) = 0 \quad \text{for all } v \in C$$

Composition between coherent linear maps is defined as $\langle f, w\rangle \circ \langle f', w'\rangle = \langle f \circ f', w' \circ w\rangle$.

Intuitively, a coherent linear map f respects the direct-sum structure of its domain: each component $\text{proj}_{C'}(f(v))$ of the output depends only on the single component $\text{proj}_{w_f(C')}(v)$ of the input designated by the witness function.

Definition 14. Sep refers to the category in which the objects are separated spaces and the arrows are coherent linear maps.

We now go about proving the conditions necessary to applying Theorem 5:

1. $\mathbf{TS}(\Sigma)$ is a concrete category over **Sep**
2. **Sep** admits pushouts
3. $\mathbf{VASR}(\Sigma)$ is cofibered over **Sep**

We begin by establishing that the additional structure in **Sep** restores the cofibered property in Lemma 6; for this to be well-defined, we first establish that $\mathbf{VASR}(\Sigma)$ is a concrete category over **Sep**.

Lemma 5. $\mathbf{VASR}(\Sigma)$ is a concrete category over **Sep**.

Proof. We define a faithful functor ϕ mapping objects and arrows of $\mathbf{VASR}(\Sigma)$ to **Sep**.

We say that a VASR transition $F \triangleq \bigwedge_{y \in Y} y' = r_y \cdot y + a_y$ resets a variable y if $r_y = 0$ and that it adds to a variable y if $r_y = 1$. A **coherence class** of $\mathcal{V}$ is a linear subspace of $\mathbb{Q}^Y$ of the form $\bigcap_{s \in \Sigma} \mathbb{Q}^{\{Y_R\}}$ where for each s, we have that Y_R is either the set of variables that is incremented or reset by $\mathcal{V}(s)$. Any two coherence classes only intersect at the zero state, and the direct sum of all coherence classes is $\mathbb{Q}^Y$. Given a VASR $\mathcal{V}$ over Y, we define $\phi(\mathcal{V}) = \left\langle \mathbb{Q}^Y, D \right\rangle$ where D is the set of all coherence classes of $\mathcal{V}$.

Let f be a linear simulation from $\mathcal{V} : \Sigma \to \mathbf{TF}(Y)$ to $\mathcal{V}' : \Sigma \to \mathbf{TF}(Z)$ and let D and D' be their coherence classes respectively. We define $\phi(f) = \langle f, w_f \rangle$ where $w_f : D' \to D$ is constructed as follows.

We first claim that for every $C' \in D'$, there is at most one $C \in D$ such that $\mathrm{proj}_{C'} \circ f \circ \mathrm{proj}_C$ is non-zero. Suppose for the sake of obtaining a contradiction that two distinct classes $C_1, C_2 \in D$ both have this property, witnessed by $\rho_1 \in C_1$ and $\rho_2 \in C_2$ with $\mathrm{proj}_{C'}(f(\rho_1))$ and $\mathrm{proj}_{C'}(f(\rho_2))$ non-zero. Since $C_1 \neq C_2$, there exists $s \in \Sigma$ such that $\mathcal{V}(s)$ increments ρ_1 and resets ρ_2 (or vice versa). Since f is a simulation, $\mathcal{V}'(s)$ must increment $f(\rho_1)$ and reset $f(\rho_2)$, and therefore must increment $\mathrm{proj}_{C'}(f(\rho_1))$ and reset $\mathrm{proj}_{C'}(f(\rho_2))$. But both belong to the same coherence class C', so $\mathcal{V}'(s)$ must either increment or reset the entire class — a contradiction, since it cannot do both.

We may therefore define $w_f(C')$ to be the unique $C \in D$ with $\mathrm{proj}_{C'} \circ f \circ \mathrm{proj}_C \neq 0$, or an arbitrary element of D if no such C exists. By construction, $\langle f, w_f \rangle$ satisfies the coherence condition, so $\phi(f)$ is a valid coherent linear map. □

Lemma 6. $\mathbf{VASR}(\Sigma)$ is cofibered over $\mathbf{Sep}$.

Proof. Let ϕ be the functor from Lemma 5. Consider a VASR $\mathcal{V}$ over variables Y and a coherent linear map $\langle f, w \rangle$ from $\phi(\mathcal{V}) = \langle \mathbb{Q}^Y, D \rangle$ to a separated space $S = \langle \mathbb{Q}^Z, D' \rangle$.

For each $s \in \Sigma$, let $\mathbf{o}_s \triangleq (\mathit{Offset}(\mathcal{V}, s, y))_{y \in Y} \in \mathbb{Q}^Y$ be the offset vector of $\mathcal{V}(s)$. Define the VASR $\mathit{image}(\mathcal{V}, f)$ over Z by:

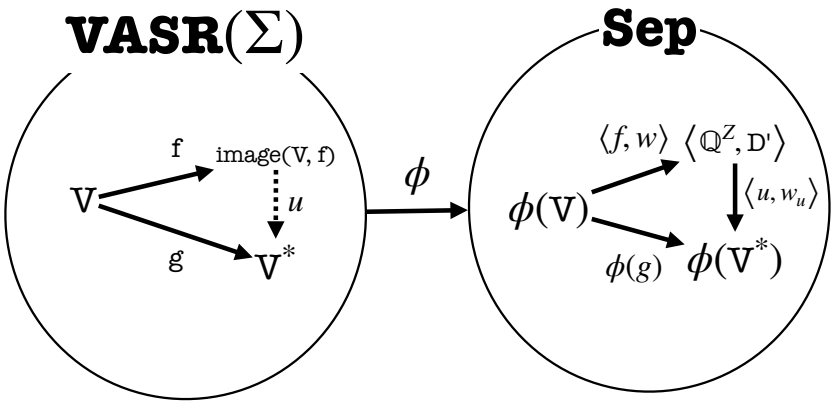
$$\mathit{image}(\mathcal{V}, f)(s) \triangleq \bigwedge_{z \in Z} z' = r_z^s \cdot z + f(\mathbf{o}_s)(z),$$

where $r_z^s = 0$ if $\mathcal{V}(s)$ resets $w(C'_z)$, and $r_z^s = 1$ if $\mathcal{V}(s)$ increments $w(C'_z)$, where $C'_z \in D'$ is the unique class containing $\mathbb{Q}^{\{z\}}$. This is well-defined since $w(C'_z)$ is a coherence class of $\mathcal{V}$.

To show that f is a simulation from $\mathcal{V}$ to $\mathit{image}(\mathcal{V}, f)$, consider any $\rho, \rho' \in \mathbb{Q}^Y$ such that $[\rho, \rho'] \models \mathcal{V}(s)$. Consider some $z \in Z$. If $\mathcal{V}(s)$ resets $w(C'_z)$, then $\rho'(y) = \mathit{Offset}(\mathcal{V}, s, y)$ for y

with $\mathbb{Q}^{\{y\}} \in w(C'_z)$, and so $f(\rho')(z) = f(\mathbf{o}_s)(z)$ by linearity of f. The case where $\mathcal{V}(s)$ adds to $w(C'_z)$ is analogous. Since this holds for all $z \in Z$, we have that $[f(\rho), f(\rho')] \models \mathit{image}(\mathcal{V}, f)$.

It remains to show the universal property of this object. Let $\mathcal{V}^*$ be any other VASR over variables V and g be a linear simulation from $\mathcal{V}$ to $\mathcal{V}^*$ such that there exists a coherent linear map $\langle u, w_u \rangle$ such that $\langle u, w_u \rangle \circ \langle f, w \rangle = \phi(g)$. We must find a simulation u from $\mathit{image}(\mathcal{V}, f)$ to $\mathcal{V}^*$ such that $\phi(u) = \langle u, w_u \rangle$.



We show that u is a simulation from $\mathit{image}(\mathcal{V}, f)$ to $\mathcal{V}^*$. Denote $\phi(g)$ as $\langle g, w_g \rangle$. Since g is a simulation from $\mathcal{V}$ to $\mathcal{V}^*$ and $\langle 0, \mathbf{o}_s \rangle \models \mathcal{V}(s)$, we have that $\langle 0, g(\mathbf{o}_s) \rangle \models \mathcal{V}^*(s)$. We therefore have that for all $v \in V$ we have $\mathit{Offset}(\mathcal{V}^*, s, v) = g(\mathbf{o}_s)(v) = u(f(\mathbf{o}_s))$.

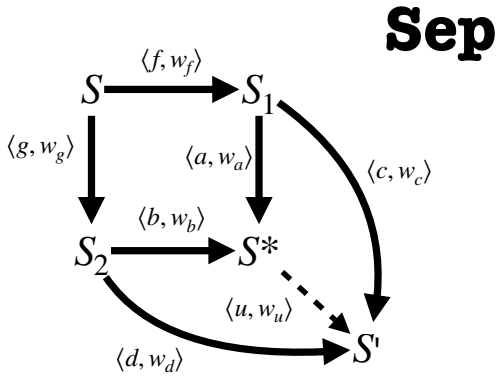
Fix $s \in \Sigma$. Consider any $\rho, \rho' \in \mathbb{Q}^Z$ such that $[\rho, \rho'] \models \mathit{image}(\mathcal{V}, f)(s)$. Fix $v \in V$ and let C^*_v be the unique coherence class of $\mathcal{V}^*$ containing $\mathbb{Q}^{\{v\}}$. Let $\mathbf{r} = (r^s_z)_{z \in Z} \in \{0, 1\}^Z$ be the reset vector of $\mathit{image}(\mathcal{V}, f)(s)$. Since $[\rho, \rho'] \models \mathit{image}(\mathcal{V}, f)(s)$, we have that $\rho' = \mathbf{r} * \rho + f(\mathbf{o}_s)$ where $*$ denotes element-wise multiplication. By the linearity of u, we have that:

$$
\begin{aligned}
u(\rho')(v) &= u(\mathbf{r} * \rho)(v) + u(f(\mathbf{o}_s))(v) \\
&= \mathrm{proj}_{C^*_v} \circ u \circ \mathrm{proj}_{w_u(C^*_v)}(\mathbf{r} * \rho)(v) + u(f(\mathbf{o}_s))(v) \\
&= r * u(\rho)(v) + g(\mathbf{o}_s)(v) \qquad \text{where } r = r^s_z \text{ for any } z \in w_u(C^*_v)
\end{aligned}
$$

By the definition of r_z^s, we have that the value r is determined by whether $\mathcal{V}(s)$ resets or adds to $w(w_u(C_v^*)) = w_g(C_v^*)$. Since g is a simulation from $\mathcal{V}$ to $\mathcal{V}^*$, this implies that r is the coefficient of $\mathcal{V}^*(s)$ on s. Combined with $g(\mathbf{o}_s)(v) = \mathit{Offset}(\mathcal{V}', s, v)$, this means that the above is precisely one of the conjuncts of $\mathcal{V}^*(s)$. Since this holds for all $v \in V$, we may conclude that $[u(\rho), u(\rho')] \models \mathcal{V}^*(s)$. □

Lemma 7. Sep admits pushouts.

Proof. Let $\langle f, w_f \rangle : S \to S_1$ and $\langle g, w_g \rangle : S \to S_2$ be coherent linear maps, where $S = \langle V, D \rangle$, $S_1 = \langle V_1, D_1 \rangle$, and $S_2 = \langle V_2, D_2 \rangle$. Consider the following pushout diagram.



Denote $S^* = \langle V^*, D^* \rangle$. For the above diagram to commute, witness functions $w_a : D^* \to D_1$ and $w_b : D^* \to D_2$ must satisfy $w_f \circ w_a = w_g \circ w_b$, so each $C^* \in D^*$ must be assigned a pair $(w_a(C^*), w_b(C^*))$ mapping to the same component of D. We therefore define:

$$\texttt{Matching} \triangleq \{\, (C_1, C_2) \in D_1 \times D_2 : w_f(C_1) = w_g(C_2) \,\}$$

We construct one component of S^* for each pair therein. For each $p = \langle C_1, C_2 \rangle \in$ `Matching`, let

$$\langle C_p,\ a_p,\ b_p\rangle \ =\ \mathit{pushout}_{\mathbf{VS}}(\mathrm{proj}_{C_1}\circ f,\ \mathrm{proj}_{C_2}\circ g).$$

Fix an enumeration $\texttt{Matching} = \{p_1, \ldots, p_n\}$ and define

$$V^* \triangleq C_{p_1} \times \cdots \times C_{p_n}, \qquad C_i^* \triangleq \{\,(0,\ldots,v,\ldots,0) : v \in C_{p_i}\,\}, \qquad D^* \triangleq \{C_1^*, \ldots, C_n^*\},$$

so that $V^* = \bigoplus_i C_i^*$ by construction. Define $a : V_1 \to V^*$ and $b : V_2 \to V^*$ componentwise by $a = \langle a_{p_1}, \ldots, a_{p_n}\rangle$ and $b = \langle b_{p_1}, \ldots, b_{p_n}\rangle$, and set $w_a(C_i^*) = C_1$, $w_b(C_i^*) = C_2$ for $p_i = \langle C_1, C_2\rangle$. The commutativity conditions $a \circ f = b \circ g$ follows componentwise from each $\mathit{pushout}_{\mathbf{VS}}$, and $w_f \circ w_a = w_g \circ w_b$ holds by definition of `Matching`.

We now show universality of $\langle S^*, \langle a, w_a\rangle, \langle b, w_b\rangle\rangle$. Suppose $\langle c, w_c\rangle : S_1 \to S'$ and $\langle d, w_d\rangle : S_2 \to S'$ also make the diagram commute. Denote $S' = \langle V', D'\rangle$. For each $C' \in D'$, commutativity gives $w_f(w_c(C')) = w_g(w_d(C'))$, so $p_{C'} \triangleq \langle w_c(C'),\ w_d(C')\rangle \in \texttt{Matching}$; let $i(C')$ be its index and define $w_u(C') \triangleq C^*_{i(C')}$. By universality of $\mathit{pushout}_{\mathbf{VS}}$, for each $C' \in D'$ there exists a unique $u_{C'} : C_{p_{C'}} \to V'$ with $u_{C'} \circ a_{p_{C'}} = \mathrm{proj}_{C'} \circ c$ and $u_{C'} \circ b_{p_{C'}} = \mathrm{proj}_{C'} \circ d$. Setting $u \triangleq (u_{C'})_{C' \in D'}$ gives a coherent linear map $\langle u, w_u\rangle : S^* \to S'$ making the diagram commute. □

Lemma 8. $\mathbf{TS}(\Sigma)$ is a concrete category over **Sep**.

Proof. Via Lemma 5, we already know that the subcategory $\mathbf{VASR}(\Sigma)$ is concrete over Σ - we extend the definition of ϕ to operate over all objects and arrows in $\mathbf{TS}(\Sigma)$. For all objects $T = \left\langle \mathbb{Q}^X, \to_T \right\rangle$ in $\mathbf{TS}(\Sigma)$ but not in $\mathbf{VASR}(\Sigma)$, we define $\phi(T) = \left\langle \mathbb{Q}^X, \left\{\mathbb{Q}^X\right\}\right\rangle$. Arrows which are not fully in $\mathbf{VASR}(\Sigma)$ may be extended by trivial witness functions and achieve coherence. □

With these lemmas in hand, we can use Theorem 5 to combine VASR reflections. Together with Lemma 4, we have a divide-and-conquer approach to computing the VASR reflection

of any transition assignment: compute the VASR reflection of $\#$ restricted to each character, then repeatedly combine the reflections.

4.4 Evaluation

We implemented the summarization procedure described above in a tool called **LiP**.[1] LiP uses a monotone variant of Compositional Recurrence Analysis (CRA) [21] as its intra-procedural back-end: it generates procedure summaries using our technique and passes them to CRA, which uses them as invariants when verifying safety properties of recursive programs.

Baselines and comparisons. We compared our technique to:

- **CRA**, a monotone instantiation of the abstraction-interpretation technique Compositional Recurrence Analysis.
- **Goblint** [64], the (tied) first-place non-monotone abstract interpreter in SV-COMP 2023 [6].
- **Korn** [19] and **UAutomizer** [29], the first- and second-place finishers in the ReachSafety-Recursive category of SV-COMP 2023, representing the state of the art in software model checking.

We note an important qualitative distinction between the two categories of tools. Abstract interpreters (LiP, CRA, Goblint) are terminating invariant-generation algorithms; software model checkers (Korn, UAutomizer) are semi-algorithms capable of both verifying and refuting safety properties. Our evaluation compares them only on safe verification tasks, which lies at the intersection of their capabilities.

[1] Source code available at `https://github.com/nikhilpim/duet`.

Benchmarks. Procedure summarization is relevant only for recursive programs; on non-recursive inputs our tool reduces to CRA. We therefore restrict evaluation to recursive, numerical verification tasks.

We compiled a benchmark suite **Rec-Supreme** (60 tasks) consisting of variants of 17 distinct recursive functions covering a diverse range of algorithmic patterns including tree traversal, sorting, modular arithmetic, and lexer simulation. We supplement this with the safe tasks from the SV-COMP **Recursive** (17 tasks) and **Recursive-Simple** (35 tasks) benchmark sets. We note that the SV-COMP sets have limited diversity: together they contain 18 variants of the Fibonacci function, 12 variants of the identity function, and 8 variants of recursive addition, and many tasks can be verified by simple loop unrolling. Rec-Supreme was designed to compensate for these weaknesses. Experiments ran on an Ubuntu 20.04 virtual machine with 8 GB RAM and a 2.3 GHz Intel i7 CPU, with a 10-minute time limit per task.

Table 4.1: Number of tasks solved (✓), returning unknown (?), and timed out (TO) by each tool on each benchmark set. Tools above the midrule are monotone.

	Rec-Supreme (60)			**Recursive (17)**			**Rec-Simple (35)**		
	✓	?	TO	✓	?	TO	✓	?	TO
LiP (*VASR*)	14	46	0	2	15	0	13	22	0
CRA	16	44	0	4	13	0	14	21	0
Korn	18	1	39	13	1	3	35	0	0
UAutomizer	23	0	37	11	0	6	23	0	12
Goblint	7	53	0	3	14	0	20	15	0

Results. LiP performs roughly comparably to CRA across all benchmarks, and both are outperformed by the model checkers Korn and UAutomizer. Since CRA makes the same monotonicity and locality guarantees as our technique, this comparison is the most mean-

ingful: LiP instantiated with VASR offers no practical advantage over existing monotone program analyzers.

The root cause is a mismatch between VASR and conditional branching. A conditional branch produces a transition formula such as $x \leq 0$, which constrains the pre-state rather than specifying a fixed update. No VASR transition can represent such a formula, since the offset a_y must be a constant independent of the current state. As a result, the best VASR abstraction of a conditional branch discards the guard entirely; this loss of precision typically prevents properties from being verified. This is not a limitation of our algorithm but of the abstract domain itself.

The theoretical contributions of this chapter — polynomial-time CFL-reachability and a divide-and-conquer procedure for computing best VASR abstractions — remain the conceptual foundation for the next chapter. There, we introduce a non-deterministic extension of our VASR-based technique which outperforms existing abstract-interpreters and is even competitive with model checkers on some benchmarks.

Chapter 5

Lossy Vector Addition Systems with Resets

The preceding chapter described an inter-procedural program analysis technique based on vector addition systems with resets (VASR). It introduced several advancements to the theory of vector addition systems, including a logical encoding of the context-free reachability relation of a VASR and the development of a theory of VASR reflections.

This chapter builds upon these foundations to produce a program analysis technique that is effective in practice. We introduce an extension of our technique to Lossy VASRs, a relaxation of the VASR model that replaces equalities with inequalities. Every VASR can be precisely simulated by a Lossy VASR, and so the best Lossy VASR abstraction always retains more information about the semantics of the original program than the best VASR abstraction. This increase in precision is critical to developing an effective program analyzer for real-world benchmarks, in which conditionals are frequently used in order to define control flow.

The organization of this chapter mimics that of Chapter 4. First we formally define Lossy VASRs and describe how to logically encode their context-free reachability relation

in Section 5.1. Next, we show how to compute Lossy VASR reflections of LIRA transition assignments in Section 5.2. Together, these ingredients form a program analysis technique that is sound, monotone, and local. Section 5.3 reports an experimental evaluation of the resulting tool, LiP, comparing against state-of-the-art abstract interpreters and software model checkers on a suite of recursive numerical programs. Our evaluation shows that LiP is a step forward for abstract-interpretation based program analyzers and in some cases is even competitive with model-checkers.

5.1 Definitions and Reachability Relation

This subsection discusses an extension of our summarization procedure to Lossy VASRs, highlighting the extensibility of the theory built in Chapter 4. "Lossy" means that we replace the equalities in our VASR transitions with inequalities.

Definition 15. A **Lossy VASR transition** over a set of variables Y is a transition formula in $\mathbf{TF}(Y)$ of the form

$$\bigwedge_{y \in Y} y' \leq r_y y + a_y,$$

where $r_y \in \{0, 1\}$ and $a_y \in \mathbb{Q}$ for every $y \in Y$.

Definition 16. A **Lossy VASR** over a set of variables Y and a finite alphabet Σ is a transition formula mapping $\mathcal{L} : \Sigma \to \mathbf{TF}(Y)$ in which $\mathcal{L}(s)$ is a Lossy VASR transition for every $s \in \Sigma$.

Replacing the equalities in the transitions of vector addition systems with inequalities is a standard relaxation typically used to make reachability problems easier; here, we show that Lossy VASRs are strictly more powerful than VASR abstractions, akin to how polyhedral abstractions are strictly more powerful than affine relation abstractions. We first illustrate this idea by example.

Example 5.1.1. Consider the following transition assignment:

$$\#(s) \triangleq x' = x + 1 \vee x' = x + 2$$

Disjunctions arise naturally in transition assignments modeling programs with conditional branching. It is clear that # is not a VASR. Moreover, there is no non-trivial VASR abstraction of #, as x is not updated according to a VASR transition.

However, the Lossy VASR reflection of this transition assignment, diagrammed below, is able to capture non-trivial facts about #:

$$\mathcal{L}(s) \triangleq y_1' \leq y_1 - 1 \wedge y_2' \leq y_2 + 2$$

$$\sigma(y_1) = -x \quad \sigma(y_2) = x$$

Intuitively, the Lossy VASR reflection of # models both a lower and upper bound on x. In this way, by relaxing the structure of VASR transitions to Lossy VASR transitions, we are able to capture information about bounded updates to terms in our transition assignment.

Lossy VASR transitions are a relaxation of VASR transitions to inequalities, but counter-intuitively Lossy VASR abstractions are strictly more powerful than VASR abstractions. This is because every VASR is precisely simulated by a Lossy VASR: for any VASR $\mathcal{V}$ over Y, we can construct an LVASR $\mathcal{L}$ over a set of variables $\{lo_y : y \in Y\} \cup \{hi_y : y \in Y\}$ and a *precise*

simulation σ from $\mathcal{V}$ to $\mathcal{L}$ such that for all $s \in \Sigma$ it is the case that $\mathcal{V}(s) \iff \mathcal{L}(s)$.

$$\mathcal{V}(s) \triangleq \bigwedge_{y \in Y} (y' = r_y\, y + a_y)$$

$$\sigma(lo_y) \triangleq y, \qquad \sigma(hi_y) \triangleq -y$$

$$\mathcal{L}(s) \triangleq \bigwedge_{y \in Y} (lo'_y \leq r_y\, lo_y + a_y) \;\wedge\; \bigwedge_{y \in Y} (hi'_y \leq r_y\, hi_y - a_y)$$

Then, a LVASR reflection of a program is guaranteed to capture all information that a VASR reflection does. Moreover, LVASR reflections can frequently lead to more precise procedure summaries. Conditionals in programs, particularly those involving inequalities, generally have trivial VASR reflections but may have useful LVASR reflections.

The context-free reachability relation of a Lossy VASR admits logical encoding, obtained by replacing the equalities in $\mathit{Transition}(\mathcal{V})$ from Section 4.2.3 with inequalities.

Theorem 10. There is a polynomial-time procedure which, given a Lossy VASR $\mathcal{L}$ over alphabet Σ and a grammar G over the same alphabet, computes a formula $\mathit{Reach}(\mathcal{L}, G)$ such that:

$$[\rho, \rho'] \models \mathit{Reach}(\mathcal{L}, G) \iff \rho \xrightarrow{\mathcal{L}(G)}_{\mathcal{L}} \rho'.$$

Proof. The proof is identical to that of Theorem 9, with equalities replaced by inequalities in $\mathit{Transition}(\mathcal{L})$. □

5.2 Best Lossy VASR Abstractions

Following the same divide-and-conquer strategy as Section 4.3, we compute Lossy VASR reflections by first computing a per-letter reflection for each character in Σ and then combining them.

The key difference between VASR and Lossy VASR reflections is that linear simulations between Lossy VASRs must be non-negative. VASR transitions are conjunctions of equalities, and equalities are stable under arbitrary linear combinations: if $t_1 = t_2$ then $\alpha t_1 = \alpha t_2$ for any α. LVASR transitions, however, are conjunctions of inequalities, and inequalities are only stable under non-negative scaling: $t_1 \leq t_2$ implies $\alpha t_1 \leq \alpha t_2$ only when $\alpha \geq 0$. The consequence of this difference is that **Sep**, the category used to combine VASR reflections, is not adequate because $\mathbf{LVASR}(\Sigma)$ is not cofibered over **Sep** because it contains negative maps. We develop a variant $\mathbf{Sep}^{\leq}$, a variant that only considers non-negative maps, which fulfills the conditions of Theorem 5 and can therefore be used to combine LVASR reflections.

5.2.1 Per-Letter Lossy VASR Reflections

This section shows how to compute the Lossy VASR reflection $\langle \sigma, \mathcal{L} \rangle$ of a transition assignment $\mathit{ff} : \Sigma \to \mathbf{TF}(X)$ in the special case that Σ is singleton. In other words, we show that every transition formula $\mathit{ff}(s) \in \mathbf{TF}(X)$ has a best abstraction as a Lossy VASR transition.

Definition 17. For a transition formula $F \in \mathbf{TF}(X)$, define:

$$\begin{aligned} \mathit{L\text{-}Res}(F) &\triangleq \{\langle t, a\rangle \in \mathit{LinTerm}(X) \times \mathbb{Q} : F \models t' \leq a\} \\ \mathit{L\text{-}Add}(F) &\triangleq \{\langle t, b\rangle \in \mathit{LinTerm}(X) \times \mathbb{Q} : F \models t' \leq t + b\} \end{aligned}$$

L-Res(F) and L-Add(F) are convex cones, in that they are closed under addition and *positive* scalar multiplication. In fact, they are polyhedral cones [38], which means that they additionally have a finite generator representation: we may compute $\langle t_1, a_1\rangle, \ldots, \langle t_n, a_n\rangle$ and $\langle \hat{t}_1, b_1\rangle, \ldots, \langle \hat{t}_m, b_m\rangle$ such that:

$$\mathit{L\text{-}Res}(F) = \left\{ \sum_{i=1}^{n} \alpha_i \langle t_i, a_i\rangle : \alpha_i \in \mathbb{Q}^{\geq 0} \right\}$$

$$L\text{-}Add(F) = \left\{ \sum_{i=1}^{m} \alpha_i \langle t_i, b_i \rangle : \alpha_i \in \mathbb{Q}^{\geq 0} \right\}$$

The generators of a polyhedral cone are analogous to the basis of a linear space, which was used in Section 4.3.1 to compute a VASR reflection of a singleton transition assignment. The key difference is that a basis generates a linear space via arbitrary linear combinations, whereas the generators of a polyhedral cone generate it via *non-negative* linear combinations. This shift is due to the shift from equalities to inequalities.

Then, we may define the LVASR reflection $\langle \sigma, \mathcal{L} \rangle$ of $\not\Vdash$ as:

$$\mathcal{L}(s) \triangleq \left(\bigwedge_{i=1}^{n} y_i' \leq a_i \right) \wedge \left(\bigwedge_{i=1}^{m} z_i' \leq z_i + b_i \right)$$
$$\sigma(y_i) = t_i \qquad \sigma(z_i) = \hat{t}_i$$

Lemma 9. $\langle \sigma, \mathcal{L} \rangle$ is a Lossy VASR reflection of $\not\Vdash|_{\{s\}}$.

Proof. Follows similar reasoning to Lemma 4 □

5.2.2 Combining Lossy VASR Reflections over Disjoint Alphabets

Following the pattern developed in Section 3.4, we may reduce the problem of merging two LVASR abstractions over disjoint alphabets to defining the relevant categories qualifying the assumptions of Theorem 5. We begin by observing that **Sep**, the category used to combine VASR reflections in Section 3.3, is inadequate for LVASR because LVASR are not cofibered over **Sep**.

Observation 2. LVASR(Σ) is not cofibered over **Sep**.

Proof. Consider the LVASR:

$$\mathcal{L}(s) \triangleq x \leq 0$$

We may associate $\mathcal{L}$ with the separated space $\left\langle \mathbb{Q}^{\{x\}}, \left\{ \mathbb{Q}^{\{x\}} \right\} \right\rangle$. Consider the coherent linear substitution $\langle \sigma, w_\sigma \rangle$ in which $\sigma(y) = -x$ and w_σ maps $\mathbb{Q}^{\{y\}}$ to $\mathbb{Q}^{\{x\}}$.

Suppose σ is a simulation to some $\mathcal{L}'(s) \triangleq y' \leq r \cdot y + a$; then we have that $x' \leq 0 \models -x' \leq r * (-x) + o$ for some $r \in \{0, 1\}$ and $o \in \mathbb{Q}$. However setting x' to M as $M \to -\infty$ yields a contradiction. Therefore, σ is not a simulation to any $\mathcal{L}$, and so $\mathbf{LVASR}(\Sigma)$ is not cofibered over $\mathbf{Sep}$. □

Intuitively, the additional constraint (besides coherence) that must be satisfied by a linear simulation σ from $\mathcal{L}$ to $\mathcal{L}'$ is that it is *non-negative*: for each variable z of $\mathcal{L}'$, we must have $\sigma(z) = a_1 y_1 + \ldots a_n y_n$ where each $a_i \geq 0$.

We thus consider the following variation of the category $\mathbf{Sep}$ from Section 4.3.

Definition 18. Let $V \subseteq \mathbb{Q}^X$ and $V' \subseteq \mathbb{Q}^Y$ be vector spaces over the rationals. Consider the partial order $\leq_V$ (and $\leq_{V'}$) defined by $\rho \leq_V \rho'$ iff $\rho(x) \leq \rho'(x)$ for all $x \in X$ ($\leq_{V'}$ defined analogously). A **positive map** f between V and V' is a linear map that is monotone with respect to these orders: $u \leq_V v$ implies that $f(u) \leq_{V'} f(v)$.

Definition 19. $\mathbf{Sep}^{\leq}$ is the category in which the objects are separated spaces over the rationals and the arrows are positive coherent linear maps.

We now go about proving the necessary conditions to apply Theorem 5:

1. $\mathbf{TS}(\Sigma)$ is a concrete category over $\mathbf{Sep}^{\leq}$
2. $\mathbf{Sep}^{\leq}$ admits pushouts
3. $\mathbf{LVASR}(\Sigma)$ is cofibered over $\mathbf{Sep}^{\leq}$

We begin by showing that $\mathbf{LVASR}(\Sigma)$ is cofibered over $\mathbf{Sep}^{\leq}$ (for which we require that $\mathbf{LVASR}(\Sigma)$ is a concrete category over $\mathbf{Sep}^{\leq}$). We continue by showing the other two conditions above.

Lemma 10. $\mathbf{LVASR}(\Sigma)$ is a concrete category over $\mathbf{Sep}^{\leq}$.

Proof. We define a faithful functor ϕ mapping objects and arrows of $\mathbf{LVASR}(\Sigma)$ to $\mathbf{Sep}^{\leq}$.

Given a LVASR $\mathcal{L}$ over variables Y, define $\phi(\mathcal{L}) = \langle \mathbb{Q}^Y, D, \leq \rangle$ where D is the coherence classes of $\mathcal{L}$ (as defined in Lemma 5).

Let f be a linear simulation from $\mathcal{L}$, defined over variables Y, to $\mathcal{L}'$, defined over variables Y'. The witness function w_f is constructed by an identical at-most-one argument as in Lemma 5: two distinct coherence classes of $\mathcal{L}$ cannot contribute non-trivially to the same coherence class of $\mathcal{L}'$, as this would require $\mathcal{L}'(s)$ to simultaneously increment and reset C' for some s. We define w_f to map each coherence class C' of $\mathcal{L}'$ to the unique coherence class C of $\mathcal{L}$ such that $\text{proj}_C \circ f \circ \text{proj}_{C'}$ is non-zero (and an arbitrary coherence class if no such C exists).

It remains to show that f is a positive map with respect to the component-wise orders on $\mathbb{Q}^Y$ and $\mathbb{Q}^{Y'}$. Suppose for the sake of obtaining a contradiction that f is not positive. Let σ be the substitution corresponding to f. Denote $\sigma(z) = \sum_y \alpha_y y$. We know by the fact that f is not positive that there must be some $\alpha_y < 0$. Fix this y.

Fix some $s \in \Sigma$, and consider some states ρ, ρ' such that $[\rho, \rho'] \models \mathcal{L}(s)$. There exists an infinite decreasing sequence $\rho' = \rho_1, \rho_2, \ldots$ such that $[\rho, \rho_i] \models \mathcal{L}(s)$, obtained by $\rho_{i+1}(y) = \rho_i(y) - 1$ and $\rho_{i+1}(y') = \rho_i(y)$ for all other $y' \neq y$ for all i.

Observe that $\rho_{i+1} \leq \rho_i$ for all i, and that $f(\rho_{i+1}) \geq f(\rho_i)$. Since f is a simulation, we have that there is an infinite increasing sequence $f(\rho_1), f(\rho_2), \ldots$ such that $[\rho, \rho_i] \models \mathcal{L}'(s)$ for all i. However, this is not possible because LVASR transitions are bounded above. Therefore, f is positive.

We may therefore define $\phi(f) = \langle f, w_f \rangle$, which is a positive coherent linear map by the above. □

Lemma 11. LVASR(Σ) is cofibered over $\mathbf{Sep}^{\leq}$.

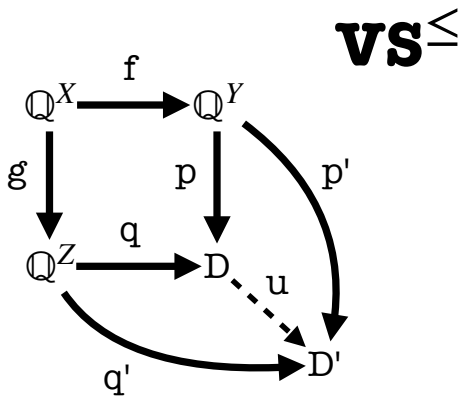
Proof. Similar to Lemma 6, in which the inequalities are stable due to the positivity of the maps. □

To define the pushout of **Sep**, we used the pushout of the category of vector spaces **VS** as a helper subroutine. Analogously, we define the pushout of $\mathbf{Sep}^{\leq}$ using the pushout of the subcategory category of vector spaces $\mathbf{VS}^{\leq}$ in which all arrows are positive maps as a helper routine.

Definition 20. The category $\mathbf{VS}^{\leq}$ has as objects rational vector spaces of the form $\mathbb{Q}^X$ and as arrows positive linear maps.

Lemma 12. $\mathbf{VS}^{\leq}$ admits pushouts.

Proof. Let $f : \mathbb{Q}^X \to \mathbb{Q}^Y$ and $g : \mathbb{Q}^X \to \mathbb{Q}^Z$ be positive linear maps.



Let $e_y \in \mathbb{Q}^Y$ denote the vector where $e_y(y) = 1$ and $e_y(y') = 0$ for all $y' \neq y$, and let e_z be defined similarly. Consider the set of pairs of positive functionals that are compatible

with respect to f and g:

$$\mathcal{F} \triangleq \left\{ \langle p, q\rangle \in (\mathbb{Q}^Y \to_{lin} \mathbb{Q}) \times (\mathbb{Q}^Z \to_{lin} \mathbb{Q}) \ : \ p \circ f = q \circ g, \bigwedge_y p(e_y) \geq 0 \ \wedge \ \bigwedge_z q(e_z) \geq 0 \right\}.$$

$\mathcal{F}$ is a polyhedral cone; let $\langle p_1, q_1\rangle, \ldots, \langle p_k, q_k\rangle$ be its generators. Define the pushout object $D \triangleq \mathbb{Q}^k$, and define $p : B \to D$ and $q : C \to D$ by:

$$p(v) \triangleq \langle p_1(v), \ldots, p_k(v)\rangle, \qquad q(w) \triangleq \langle q_1(w), \ldots, q_k(w)\rangle.$$

We have that $p \circ f = q \circ g$ holds componentwise since each $p_i \circ f = q_i \circ g$ by definition of $\mathcal{F}$. We additionally have that p and q are positive component-wise since each p_i and q_i are positive because they are positive over the standard basis vectors by the construction of $\mathcal{F}$.

It remains to show universality of $\langle D, p, q\rangle$. Consider another vector space $\mathbb{Q}^V$ and positive linear maps $p' : \mathbb{Q}^Y \to \mathbb{Q}^V$, $q' : \mathbb{Q}^Z \to \mathbb{Q}^V$ such that the diagram commutes. For each $v \in V$, let $p'_v : \mathbb{Q}^Y \to \mathbb{Q}$ be the functional $p'_v(b) \triangleq p'(b)(v)$, and define q'_v similarly. Observe that for each $v \in V$ the pair $\langle p'_v, q'_v\rangle$ lies in $\mathcal{F}$, and so can be written as a nonnegative combination $\langle p'_v, q'_v\rangle = \sum_{i=1}^k \lambda_{v,i} \langle p_i, q_i\rangle$ of the generators. The coefficients then define the map witnessing universality: set $u(d)(v) \triangleq \sum_{i=1}^k \lambda_{v,i}\, d(i)$, which is positive and satisfies $u \circ p = p'$ and $u \circ q = q'$

□

Lemma 13. $\mathbf{Sep}^{\leq}$ admits pushouts.

Proof. Similar to Lemma 7, with $\mathbf{VS}^{\leq}$ in place of $\mathbf{VS}$. □

With these lemmas in hand, we can use Theorem 5 to combine VASR reflections. Together with Lemma 4, we have a divide-and-conquer approach to computing the VASR reflection of any transition assignment: compute the VASR reflection of $\#$ restricted to each character, then repeatedly combine the reflections.

5.3 Evaluation

We implemented LVASR based summarization inside of LiP and evaluated it in the same environment as Chapter 4.

Table 5.1: Number of tasks solved (✓), returning unknown (?), and timed out (TO) by each tool on each benchmark set. Tools above the midrule are monotone.

	Rec-Supreme (60)			**Recursive (17)**			**Rec-Simple (35)**		
	✓	?	TO	✓	?	TO	✓	?	TO
LiP (*LVASR*)	26	34	0	3	13	1	20	15	0
LiP (*VASR*)	14	46	0	2	15	0	13	22	0
CRA	16	44	0	4	13	0	14	21	0
LiP $\wedge$ CRA	29	31	0	5	11	1	20	15	0
Korn	18	1	39	13	1	3	35	0	0
UAutomizer	23	0	37	11	0	6	23	0	12
Goblint	7	53	0	3	14	0	20	15	0

Precision among abstract interpreters. On Rec-Supreme, LiP with LVASR reflections solves 26 tasks, compared to 16 for CRA and 7 for Goblint. Since LiP and CRA are both monotone, their summaries can be conjoined to form LiP $\wedge$ CRA, which solves 29 tasks; this highlights a benefit of designing monotone program analyses, which is that techniques are naturally compatible. We can conjoin two monotone summarization techniques to produce a joint technique that is also monotone and guaranteed to be stronger than either of its subparts.

Comparison with model checkers. On Rec-Supreme, LiP (LVASR) solves 26 tasks versus 23 for UAutomizer and 18 for Korn, demonstrating that our monotone summarization approach is competitive with state-of-the-art verification tools on diverse recursive programs. On the SV-COMP sets the model checkers hold an advantage, particularly on tasks that reduce to bounded unrolling (Fibonacci, identity, recursive addition), for which fully general

summarization provides little benefit. Korn times out on 39 Rec-Supreme tasks and UAutomizer on 37, while LiP never times out—a reflection of the guarantee that our summarization procedure always terminates. It is important to note the qualitative differences between abstract interpreters and model checkers; LiP could serve as a pre-pass to generate initial candidate summaries for model checkers, which could then refine them via counterexample-guided abstraction.

Contribution of LVASR refinement. Comparing the VASR and LVASR variants of LiP directly isolates the contribution of the Lossy extension. The programs most affected are those in which branches constrain the recursive argument with an inequality: VASR reflections cannot represent the inequality branch, so the resulting summary is too imprecise to verify the post-condition; LVASR reflections capture it exactly.

Chapter 6

Semi-Linear Vector Addition Systems with Resets

Our program analysis recipe requires classes of abstractions which are both tractable and expressive. Chapters 4 and 5 described a progression of techniques that increased in expressiveness while retaining tractability; Lossy VASRs exceeded the theoretical and practical value of VASRs for program analysis by capturing inequalities, rather than equalities, over terms that were optionally reset then incremented. This chapter takes this progression to its limit. Semi-Linear VASRs (SVASRs) equip each transition with a semi-linear set of possible offsets. The SVASR reflection of any LIA-definable transition assignment has a step-wise transition relation in exact correspondence with the original. In this sense, SVASRs represent a completion of the abstraction approaches presented in previous chapters. Translating this theoretical result into a practical tool, however, remains an open challenge.

The chapter is organized around three contributions. Section 6.2 shows that L-reachability of a SVASR reduces to L'-reachability of an ordinary VASR, where the blowup from L to L' is linear in the size of the generator representations of the offset semi-linear sets. Since Section 4.2 showed that we can compute the context-free reachability

relation of any VASR, this gives a procedure for computing the context-free reachability relation of SVASRs. Section 6.3 shows that every transition assignment has a SVASR reflection. The reflection, however, has a state space of dimension exponential in the size of the alphabet, making it impractical to construct explicitly. Section 6.4 resolves this by exploiting the symmetric structure of the reflection to compute, in polynomial time, a formula that is semantically equivalent to the one that would result from explicitly constructing the reflection and computing its context-free reachability relation — effectively using an exponentially large object without ever building it. This removes a key barrier to the use of SVASRs in scalable inter-procedural program analysis. We conclude by discussing remaining issues standing in the way of an implementation in Section 6.5.

6.1 Definitions

A semi-linear set [46] over $\mathbb{Z}^n$ is the finite union of linear sets in $\mathbb{Z}^n$. A linear set S in $\mathbb{Z}^n$ is generated by a *base point* b in $\mathbb{Z}^n$ and a set of *periods* $p_1, \ldots, p_n$ in $\mathbb{Z}^n$:

$$S \triangleq \{b + \lambda_1 p_1 + \cdots + \lambda_n p_n : \lambda_1, \ldots, \lambda_n \in \mathbb{N}\}$$

The generator representation of a semi-linear set is not unique. For convenience, we fix one such representation for each semi-linear set and refer to it as the **basis** of that set; nothing in the following sections depends on the particular choice (Equality of semi-linear sets is decidable, so in our algorithms, one can always test whether a semi-linear set is represented by an existing set of generators). A basis $\mathbb{Z}^n$ is an element of $(\mathbb{Z}^n \times (\mathbb{Z}^n)^*)^*$. Each element of this representation is the generator representation of a linear set; that is, a base pointer followed by a sequence of periods. Let $B(S)$ refer to the basis of semilinear set S and let $S(B)$ be the semilinear set defined by basis B.

In this section, we restrict our attention from transition formulas expressed in LIRA to transition formulas expressed in LIA. A classical result is that LIA-definable sets coincide with semi-linear sets [26]. Then, for every transition formula $F \in \mathbf{TF}(X)$, there exists a semilinear set S such that $S = \{m : m \models F\}$. For any such formula F, let $B(F)$ be the basis of the underlying semi-linear set of models. This observation motivates the definition of semi-linear VASR: we instrument VASRs with semi-linear sets to make them expressive enough to capture the semi-linearity of our program model.

A labeled semi-linear integer vector addition system with resets (**SVASR**) over variables X and alphabet Σ is a labeled transition system $\mathcal{SV} = \left\langle \mathbb{Z}^X, \rightarrow_{\mathcal{SV}} \right\rangle$. For each symbol $s \in \Sigma$ there is an offset semi-linear set $S_s \subseteq \mathbb{Z}^X$ and a reset vector $r_s \in \{0,1\}^X$ such that $\rho \xrightarrow{s}_{\mathcal{SV}} \rho'$ if and only if $\bigwedge_{x \in X} \rho'(x) = r_s(x)\rho(x) + v(x)$ for some $v \in S_s$. Let $RV(\mathcal{SV}, s)$ and $OS(\mathcal{SV}, s)$ denote r_s and the basis of S_s respectively. Note that $\rightarrow_{\mathcal{SV}}$ is uniquely determined by $RV(\mathcal{SV}, s)$ and $OS(\mathcal{SV}, s)$ for all s.

6.2 SVASR Reachability Relations in Polynomial Time

We show that context-free reachability of SVASR can be reduced to context-free reachability of VASR. Our reduction creates VASR transitions representing the generator representations of the semi-linear sets of the SVASR transitions and encodes the structure of these sets as a regular language over these transitions. A SVASR transition resets some part of the state then adds a vector from a semi-linear set; our key insight is that this is equivalent to a VASR transition applying the same reset and adding one of the base points of the semi-linear set followed by some number of VASR transitions each of which perform no reset and add one of the associated periods. The following example illustrates the key idea.

Example 6.2.1. Consider the SVASR defined by:

$$\mathcal{SV}(s) \triangleq x_1' = x_1 + o_1 \wedge x_2' = o_2 \quad \text{where } \langle o_1, o_2 \rangle \in \{\langle 1,2\rangle + \lambda \langle 3,4\rangle : \lambda \in \mathbb{N}\}$$

We may define a VASR $\mathcal{V}$ and regular language $R \subseteq \{s_r, s_p\}^*$ for this SVASR such that the transition relation $\xrightarrow{s}_{\mathcal{SV}}$ is equivalent to the regular reachability relation $\xrightarrow{R}_{\mathcal{V}}$.

$$\mathcal{V}(s_r) \triangleq x_1' = x_1 + 1 \wedge x_2' = 2 \qquad \mathcal{V}(s_p) \triangleq x_1' = x_1 + 3 \wedge x_2' = x_2 + 4$$

$$R \triangleq s_r(s_p)^*$$

A single application of s in the SVASR with offset $\langle 1,2\rangle + \lambda \langle 3,4\rangle$ is then simulated by one s_r step in the VASR (resetting x_2 and adding the base point) followed by λ copies of s_p (each adding the period without resetting). The star in R ranges over the choice of λ.

Consider a SVASR $\mathcal{SV} = \left\langle \mathbb{Z}^X, \rightarrow_{\mathcal{SV}} \right\rangle$ over alphabet Σ and a language $L \subseteq \Sigma^*$. In what follows, $\lambda x.1$ denotes the constant-1 vector in $\{0,1\}^X$. This is the appropriate reset component for the period-handling transitions in the construction below, playing the role of s_p in the above example. Define a new alphabet $\Sigma_{\mathcal{SV}} \subseteq \{0,1\}^X \times \mathbb{Z}^X$ to be the least set such that:

- For all $s \in \Sigma$, for all $\langle b, P\rangle \in OS(\mathcal{SV}, s)$, we have $\langle RV(\mathcal{SV}, s), b\rangle \in \Sigma_{\mathcal{SV}}$
- For all $s \in \Sigma$, for all $\langle b, p_1 \ldots p_n\rangle \in OS(\mathcal{SV}, s)$, we have $\langle \lambda x.1, p_i\rangle \in \Sigma_{\mathcal{SV}}$ for all $i \in [1,n]$

Define a VASR $\mathcal{V}_{\mathcal{SV}} \triangleq \left\langle \mathbb{Z}^X, \rightarrow_{\mathcal{V}_{\mathcal{SV}}} \right\rangle$ over variables X and alphabet $\Sigma_{\mathcal{SV}}$ where for all $\langle r, v\rangle \in \Sigma_{\mathcal{SV}}$ we have:

$$\mathcal{V}_{\mathcal{SV}}(s) = \bigwedge_{x \in X} x' = r * x + v$$

Finally, for each $s \in S$, define the following regular language $R_s \subseteq \Sigma^*_{\mathcal{SV}}$:

$$R_s \triangleq \bigcup_{\langle b,p_1,\ldots,p_n\rangle \in OS(\mathcal{SV},s)} \langle RV(\mathcal{SV},s), b\rangle \langle \lambda x.1, p_1\rangle^* \ldots \langle \lambda x.1, p_n\rangle^*$$

Lemma 14. Consider an SVASR $\mathcal{SV}$ over alphabet Σ. For all $s \in \Sigma$, we have:

$$\left(\rho \xrightarrow{s}_{\mathcal{SV}} \rho'\right) \iff \left(\rho \xrightarrow{R_s}_{\mathcal{V}_{\mathcal{SV}}} \rho'\right)$$

Proof. ($\implies$) Let X denote the variables of $\mathcal{SV}$. Consider any states ρ, ρ' such that $\rho \xrightarrow{s}_{\mathcal{SV}} \rho'$. We have that $\bigwedge_{x\in X} \rho'(x) = RV(\mathcal{SV},s)(x)\rho(x) + v(x)$ for some $v \in OS(\mathcal{SV},s)$. Then, there must be some $\langle b, p_1 \ldots p_n\rangle \in OS(\mathcal{SV},s)$ such that $v = b + \sum_{i=1}^n \lambda_i p_i$ for some $\lambda_1, \ldots, \lambda_n \in \mathbb{N}$. Then, observe that the word

$$w \triangleq \langle RV(\mathcal{SV},s), b\rangle \langle \lambda x.1, p_1\rangle^{\lambda_1} \ldots \langle \lambda x.1, p_n\rangle^{\lambda_n}$$

belongs to R_s and that $\rho \xrightarrow{w}_{\mathcal{V}(\mathcal{SV})} \rho'$, and so $\rho \xrightarrow{R_s}_{\mathcal{V}_{\mathcal{SV}}} \rho'$.

($\impliedby$) Consider any states ρ, ρ' such that $\rho \xrightarrow{R_s}_{\mathcal{V}_{\mathcal{SV}}} \rho'$. Then there must be some $w \in R_s$ such that $\rho \xrightarrow{w}_{\mathcal{V}_{\mathcal{SV}}} \rho'$. By the definition of R_s, for some $\langle b, p_1 \ldots p_n\rangle \in OS(\mathcal{SV},s)$ we have that

$$w = \langle RV(\mathcal{SV},s), b\rangle \langle \lambda x.1, p_1\rangle^{\lambda_1} \ldots \langle \lambda x.1, p_n\rangle^{\lambda_n}$$

for some $\lambda_1, \ldots, \lambda_n \in \mathbb{N}$. By the definition of $\mathcal{V}(\mathcal{SV})$, this implies that $\bigwedge_{x\in X} \rho'(x) = RV(\mathcal{SV},s)(x)\rho(x) + (b + \sum_{p\in P} \lambda_p p)$. Then, since $b + \sum_{i=1}^n \lambda_i p_i \in OS(\mathcal{SV},s)$, we have that $\rho \xrightarrow{s}_{\mathcal{SV}} \rho'$. □

The above lemma corresponds the step-wise transition relation of any SVASR with the regular reachability relation of a VASR. We may thereby reduce the task of computing the L-

reachability relation of any SVASR to the task of computing the reachability of a VASR over a language derived from L by replacing every character with its associated regular language. Formally, let $R(L)$ be the language replacing all characters in Σ with their corresponding regular languages: $R(L) \triangleq \{w_1 \dots w_n : \exists s_1 \dots s_n \in L, w_i \in R_{s_i}\}$.

Theorem 11. For language $L \subseteq \Sigma^*$ and semi-linear VASR $\mathcal{SV}$ we have

$$\left(\rho \xrightarrow{L}_{\mathcal{SV}} \rho'\right) \iff \left(\rho \xrightarrow{R(L)}_{\mathcal{V}_{\mathcal{SV}}} \rho'\right)$$

Proof. Follows from Lemma 14. □

Therefore, L-reachability of $\mathcal{SV}$ reduces to $R(L)$-reachability of $\mathcal{V}$. Observe that if L is a context-free language, then $R(L)$ is also a context-free language. The grammar generating $R(L)$ extends the grammar generating L by replacing each terminal $s \in \Sigma$ with the start nonterminal of a grammar generating R_s; this adds a number of productions linearly proportional to the total size of the bases $OS(\mathcal{SV}, s)$ across all $s \in \Sigma$.

6.3 Best SVASR Abstractions of LIA-Definable Transition Systems

We now shift focus to the other ingredient of our recipe for computing monotone program analyses: a technique to compute the SVASR abstraction of an LIA transition assignment representing our program of interest.

We show a direct construction of the SVASR reflection. The reflection is exponentially sized with respect to the original transition system, making it impractical for direct usage. We show that we can exploit symmetries in the reflection to compute its induced summary in polynomial time in Section 6.4.

6.3.1 Best SVASR Abstractions

We define the SVASR reflection of a transition assignment $\mathit{tf} : \Sigma \to \mathrm{TF}(X)$ via a direct construction. Although the divide-and-conquer strategy of Chapter 3 applies here with **Sep** (from Chapter 4) as the underlying category, we bypass it and present the reflection directly. This is possible because the variables of the SVASR reflection are determined entirely by X and Σ, and is independent of the actual formulas assigned by tf.

The SVASR reflection $\langle \sigma, \mathcal{SV} \rangle$ of a transition assignment tf over variables X and alphabet Σ can be defined as follows. The state space of $\mathcal{SV}$ is $\mathbb{Z}^{X \times \{0,1\}^\Sigma}$. Intuitively, for each variable x and character $s \in \Sigma$, we must make a choice of whether to treat s as a reset of x. We introduce $2^{|\Sigma|}$ copies of each variable x to encode all possible choices. Thus, a "variable" of the SVASR reflection is a pair $\langle x, C \rangle \in X \times \{0,1\}^\Sigma$ where x is variable of tf, and $C(s)$ indicates whether s is to be treated as an increment ($C(s) = 1$) or reset ($C(s) = 0$). The simulation $\sigma : X \times \{0,1\}^\Sigma \to \mathit{LinTerm}(X)$ is defined as:

$$\sigma(\langle x, C \rangle) = x$$

The relation $\to_{\mathcal{SV}}$ is defined in terms of $RV(\mathcal{SV}, s)$ and $OS(\mathcal{SV}, s)$ as:

$$RV(\mathcal{SV}, s) = \lambda \langle x, C \rangle . C(s)$$

$$OS(\mathcal{SV}, s) = \left\{ \lambda \langle x, C \rangle . \text{ if } C(s) \text{ then } \rho'(x) - \rho(x) \text{ else } \rho'(x) : \rho \xrightarrow{s}_{\mathit{tf}} \rho' \right\}$$

A basis for $OS(\mathcal{SV}, s)$ can be computed from a basis for $\xrightarrow{s}_T$ as follows. For any $t = \langle \rho, \rho' \rangle \in \mathbb{Z}^X \times \mathbb{Z}^X$, define

$$\hat{t} \triangleq \lambda \langle x, C \rangle . \text{ if } C(s) \text{ then } \rho'(x) - \rho(x) \text{ else } \rho'(x) \ .$$

Then we define

$$OS(\mathcal{SV}, s) \triangleq \left\{ \left\langle \hat{b}; \hat{p}_1, \ldots, \hat{p}_n \right\rangle : \langle b, p_1 \ldots p_n \rangle \in B(\not{tf}(s)) \right\} .$$

Theorem 12. The pair $\langle f, \mathcal{SV} \rangle$ is a SVASR reflection of T.

Proof. First, we show that f is a simulation from $\not{tf}$ to $\mathcal{SV}$. Consider any states ρ, ρ' such that $\rho \xrightarrow{s}_{\not{tf}} \rho'$. By the definition of $OS(\mathcal{SV}, s)$, we have that the vector $v = (\lambda \langle x, C \rangle \,.\, \text{if } C(s) \text{ then } \rho'(x) - \rho(x) \text{ else } \rho'(x))$ is in $OS(\mathcal{SV}, s)$. Then, since $RV(\mathcal{SV}, s)(x, C) = C(s)$ for all $\langle x, C \rangle$, we may immediately conclude:

$$\bigwedge_{\langle x,C\rangle \in X \times \{0,1\}^\Sigma} f(\rho')(\langle x, C \rangle) = RV(\mathcal{SV}, s)(\langle x, C \rangle) f(\rho)(\langle x, C \rangle) + v(\langle x, C \rangle)$$

because $f(\rho')(\langle x, C \rangle) = \rho'(x)$ and $f(\rho)(\langle x, C \rangle) = \rho(x)$; then, each conjunct holds by substituting the definitions of $RV(\mathcal{SV}, s)(\langle x, C \rangle)$ and $v(\langle x, C \rangle)$.

Consider another SVASR abstraction $\langle f', \mathcal{SV}' \rangle$ over variables Y. We will show that the following function $f^* : \mathbb{Z}^{X \times \{0,1\}^\Sigma} \to \mathbb{Z}^Y$ is a simulation from $\mathcal{SV}$ to $\mathcal{SV}'$.

$$f^*(\sigma) = \lambda y. f'(\lambda x. \sigma(\langle x, C_y \rangle))(y) \text{ with } C_y = \lambda s. RV(\mathcal{SV}', s)(y)$$

A piece of intuition for this definition is that $\langle x, C_y \rangle$ is the variable of $\mathcal{SV}$ that abstracts the variable x of T and that experiences the same resets per-character as y in $\mathcal{SV}'$.

First, observe that $f^* \circ f = f'$:

$$\begin{aligned} f^*(f(\rho)) &= \lambda y. f'(\lambda x. f(\rho)(x, C_y))(y) \\ &= \lambda y. f'(\lambda x. \rho(x))(y) = f'(\rho) \end{aligned}$$

To show f^* is a simulation from $\mathcal{SV}$ to $\mathcal{SV}'$, consider states $\sigma, \sigma' \in \mathbb{Z}^{X \times \{0,1\}^\Sigma}$ such that $\sigma \xrightarrow{s}_{\mathcal{SV}} \sigma'$. By the definition of $OS(\mathcal{SV}, s)$, there must be ρ, ρ' such that $\rho \xrightarrow{s}_{\not\sharp} \rho'$ and:

$$\bigwedge_{\langle x,C\rangle \in X \times \{0,1\}^\Sigma, C(s)=1} \sigma'(\langle x, C\rangle) = 1\sigma(\langle x, C\rangle) + (\rho'(x) - \rho(x))$$
$$\wedge \bigwedge_{\langle x,C\rangle \in X \times \{0,1\}^\Sigma, C(s)=0} \sigma'(\langle x, C\rangle) = 0\sigma(\langle x, C\rangle) + (\rho'(x))$$

Thus, for all $\langle x, C\rangle \in X \times \{0,1\}^\Sigma$, if $C(s) = 0$ then $\sigma'(\langle x, C\rangle) = \rho'(x)$ and if $C(s) = 1$ then $\sigma(\langle x, C\rangle) - \sigma'(\langle x, C\rangle) = \rho'(x) - \rho(x)$.

Since $\rho \xrightarrow{s}_T \rho'$, we have that $f'(\rho) \xrightarrow{s}_{\mathcal{SV}'} f'(\rho')$. Then, there is some $v \in OS(\mathcal{SV}', s)$ such that:

$$\bigwedge_{y \in Y} f'(\rho')(y) = RV(\mathcal{SV}', s)(y) f'(\rho)(y) + v(y) \tag{6.1}$$

Note that $RV(\mathcal{SV}', s)(y) = C_y(s)$ for all s. Then, for all variables $y \in Y$ of SVASR $\mathcal{SV}'$, if $RV(\mathcal{SV}', s)(y) = 0$ then $\sigma'(x, C_y) = \rho'(x)$ by previous reasoning. In such cases:

$$f^*(\sigma')(y) = f'(\lambda x.\sigma'(x, C_y))(y) = f'(\lambda x.\rho'(x))(y) = f'(\rho')(y)$$

Substituting $f'(\rho')(y)$ with $f^*(\sigma')(y)$ in Equation (1) and using $RV(\mathcal{SV}', s)(y) = 0$, we have that $f^*(\sigma')(y) = RV(\mathcal{SV}', s)(y) f^*(\sigma)(y) + v(y)$.

In the other case, if $RV(\mathcal{SV}, s)(y) = 1$ then $C_y(s) = 1$ and so $\sigma'(x, C_y) - \sigma(x, C_y) = \rho'(x) - \rho(x)$. Then, using the linearity of f', we have:

$$\begin{aligned} f^*(\sigma')(y) - f^*(\sigma)(y) &= f'(\lambda x.\sigma'(x, C_y))(y) - f'(\lambda x.\sigma(x, C_y))(y) \\ &= f'(\lambda x.\sigma'(x, C_y) - \sigma(x, C_y))(y) \\ &= f'(\lambda x.\rho'(x) - \rho(x))(y) = f'(\rho')(y) - f(\rho)(y) \end{aligned}$$

Then, substituting $f'(\rho')(y) - f'(\rho)(y)$ with $f^*(\sigma')(y) - f^*(\sigma)(y)$ in (1), we have that $f^*(\sigma')(y) = RV(\mathcal{SV}', s)(y)f^*(\sigma)(y) + v(y)$.

Therefore, by cases, we have that:

$$\bigwedge_{y \in Y} f^*(\sigma')(y) = RV(\mathcal{SV}', s)(y)f^*(\sigma')(y) + v(y)$$

We can conclude that $f^*(\sigma) \xrightarrow{s}_{\mathcal{SV}'} f^*(\sigma')$, that f^* is a simulation from $\mathcal{SV}$ to $\mathcal{SV}'$, and that $\langle f, \mathcal{SV} \rangle$ is a reflection of T. □

6.4 Over-Approximate Semi-Linear Transition System Reachability in Polynomial Time

Following the recipe introduced in Chapter 3, given a transition assignment $\not\Vdash$, we can compute a formula F such that $[\rho, \rho'] \models F$ if and only if $f(\rho) \xrightarrow{\mathcal{L}(G)}_{\mathcal{SV}} f(\rho')$ where $\langle f, \mathcal{SV} \rangle$ is the SVASR-reflection of $\not\Vdash$. This formula is an over-approximation of the context-free reachability relation of $\not\Vdash$, but requires exponential space w.r.t. $\not\Vdash$ because the dimension of $\mathcal{SV}$ is exponential in the size of the alphabet. We show here that we can compute a formula equivalent to F in **polynomial time** w.r.t. $\not\Vdash$.

The key is to never explicitly compute the SVASR-reflection $\langle f, \mathcal{SV} \rangle$. Given a semi-linear transition system T, let $M_{\not\Vdash} : \Sigma \to (\mathbb{Z}^{X \times \{0,1\}} \times (\mathbb{Z}^{X \times \{0,1\}})^*)^*$ be the function mapping each $s \in \Sigma$ to a basis of the following semilinear set:

$$M_{\not\Vdash}(s) \triangleq \left\{ \lambda \langle x, r \rangle \,.\, \text{if } r = 1 \text{ then } \rho'(x) - \rho(x) \text{ else } \rho'(x) : \rho \xrightarrow{s}_T \rho' \right\}$$

This set is semi-linear: it is LIA-definable in terms of $\not\Vdash(s)$. A basis of this set can be computed from a basis of $\not\Vdash(s)$ in polynomial time, as in Section 6.3.1. Given a transition

assignment $\#$ over variables X, this subsection computes $G \in TF(X)$ such that $[\rho, \rho'] \models G$ if and only if $f(\rho) \xrightarrow{L}_{\mathcal{SV}} f(\rho')$ in polynomial time w.r.t. $M_{\#}$.

Our approach to computing G is based on a subtle reframing of the construction used to compute the context-free reachability relation of VASRs in Section 4.2. There, we computed a formula representing the abstract trajectories of $\mathcal{L}(G)$ and further constrained the free variables of that formula to identify the final resets of each dimension of the VASR. Directly considering every variable of the SVASR reflection involves identifying exponentially many final resets. However, the final reset of every SVASR variable will occur at the final occurrence of some character $s \in \Sigma$, and at the final occurrence of every character, at least one SVASR variable for each variable $x \in X$ has its final reset; namely $\langle x, C \rangle$ in which $C(s) = 0$ and $C(s') = 1$ for all $s' \neq s$. Although there are 2^Σ dimensions of the SVASR reflection associated with each variable, at most $|\Sigma|$ of these dimensions have different values.

Our approach then is to mark the final occurrence of each *symbol* and to conjoin a formula per final occurrence representing the transition of all variables which experience their final reset there. This approach ultimately exploits the symmetry about each variable of the SVASR reflection.

We first use the same formula $AT(|\Sigma|, G)$ defining the $|\Sigma|$-marked abstract trajectories of $\mathcal{L}(G)$ from Theorem 7. We then define a new formula $WF(\Sigma)$ ensuring that our symbolic $|\Sigma|$-marked abstract trajectory marks the final occurrence of every $s \in \Sigma$:

$$WF(\Sigma) = \bigwedge_{s \in \Sigma} \left(\bigwedge_{i=1}^{2|\Sigma|+1} \left(c_{s,i} > 0 \implies \bigvee\nolimits_{\text{even } k \geq i} c_{s,k} > 0 \right) \wedge \left(\sum_{j=1}^{|\Sigma|} c_{s,2j} \leq 1 \right) \right)$$

Then, we define the formula $\mathit{Transition}(M_T, \Sigma)$ which describes the transition corresponding to the symbolic abstract trajectory value. This formula uses the following sets of variables

to symbolically pick the SVASR translation vector corresponding to each occurrence of $c_{s,i}$:

$$D \triangleq \{d_{s,b,i} : s \in \Sigma, \langle b, P\rangle \in M_T(s), i \in [1, 2|\Sigma| + 1]\}$$
$$E \triangleq \{e_{s,b,p,i} : s \in \Sigma, \langle b, P\rangle \in M_T(s), p \in P, i \in [1, 2|\Sigma| + 1]\}$$

The following formula $\mathit{Corr}(\Sigma, M_T)$ corresponds variables $c_{s,i}$ to the relevant variables $d_{s,b,i}$ and $e_{s,b,p,i}$. The variable $c_{s,i}$ captures how many times s appears in subword i - for each such appearance, we must pick a single base vector and any number of periods.

$$\mathit{Corr}(\Sigma, M_T) \triangleq \bigwedge_{s\in\Sigma} \bigwedge_{i=1}^{2|\Sigma|+1} \left(\sum_{\langle b,P\rangle \in M_T(s)} d_{s,b,i} = c_{s,i} \right) \wedge \bigwedge_{\langle b,P\rangle \in M_T(s)} \bigwedge_{p\in P} (e_{s,b,p,i} > 0 \implies d_{s,b,i} > 0)$$

We define $\mathit{ResetAt}(i, x, M_T, \Sigma)$ to compute the value that x would be reset to by the ith even subword and $\mathit{AddsAfter}(i, x, M_T, \Sigma)$ to compute the value of the increments to x after the ith subword.

$$\mathit{ResetAt}(i, x, M_T, \Sigma) = \sum_{s\in\Sigma} \sum_{\langle b,P\rangle \in M_T(s)} \left(d_{s,b,i} b(\langle x, 0\rangle) + \sum_{p\in P} e_{s,b,p,i} p(\langle x, 0\rangle) \right)$$

$$\mathit{AddsAfter}(i, x, M_T, \Sigma) = \sum_{j=i+1}^{2|\Sigma|+1} \sum_{s\in\Sigma} \sum_{\langle b,P\rangle \in M_T(s)} \begin{pmatrix} d_{s,b,j} b(\langle x, 1\rangle) + \\ \sum_{p\in P} e_{s,b,p,j} p(\langle x, 1\rangle) \end{pmatrix}$$

And subsequently define:

$$\mathit{Transition}(X, M_T, \Sigma) \triangleq \bigwedge_{x \in X} \left(\begin{array}{c} x' = x + \mathit{AddsAfter}(0, x, M_T, \Sigma) \wedge \\ \bigwedge_{k=1}^{|\Sigma|} \left(\begin{array}{c} \sum_{s \in \Sigma} c_{s,2k} > 0 \implies \\ x' = \mathit{ResetAt}(2k, x, M_T, \Sigma) + \\ \mathit{AddsAfter}(2k, x, M_T, \Sigma) \end{array} \right) \end{array} \right)$$

Finally, we conjoin these formulae to produce our summary.

$$\mathit{Summary}(X, M_T, G, \Sigma) = \begin{array}{c} \exists \{c_{s,i} \geq 0 : s \in \Sigma, i \in [1, 2|\Sigma| + 1]\} \\ \exists \{d_{s,b,i} \geq 0 : d_{s,b,i} \in D\} \exists \{e_{s,b,p,i} \geq 0 : e_{s,b,p,i} \in E\} \\ \left(\begin{array}{c} \mathit{Transition}(X, M_T, \Sigma) \wedge \mathit{AT}(L, |\Sigma|) \\ \wedge \mathit{WF}(\Sigma) \wedge \mathit{Corr}(\Sigma, M_T) \end{array} \right) \end{array}$$

Theorem 13. Consider a transition assignment $\#$ and a context-free language $L(G) \subseteq \Sigma^*$. Let $\langle f, \mathcal{SV} \rangle$ be the SVASR-reflection of T defined in Section 6.3. Then,

$$[\rho, \rho'] \models \mathit{Summary}(X, M_T, G, \Sigma) \iff f(\rho) \xrightarrow{\mathcal{L}(G)}_{\mathcal{SV}} f(\rho')$$

Proof. Firstly, note that there is a one-to-one correspondence between the elements of the set generated by $M_{\#}(s)$ and the set generated by $\mathit{OS}(\mathcal{SV}, s)$, as both are defined by ρ, ρ' such that $\rho \xrightarrow{s}_T \rho'$. For each $s \in \Sigma$, define $\psi_s : \mathbb{Z}^{X \times \{0,1\}} \to \mathbb{Z}^{X \times \{0,1\}^\Sigma}$ by $\psi_s(v)(x, C) \triangleq v(x, C(s))$ to translate between the elements of the set generated by $M_{\#}(s)$ and that of $\mathit{OS}(\mathcal{SV}, s)$. Let ψ_s^{-1} be a left inverse.

($\implies$) Consider ρ, ρ' such that $[\rho, \rho'] \models \mathit{Summary}(X, M_T, L, \Sigma)$. By its definition, there exists a valuation $A : \{c_{s,i} : s \in \Sigma, i \in [1, 2|\Sigma| + 1]\} \cup D \cup E \to \mathbb{N}$ such that $(\mathit{Transition}(X, M_T, \Sigma) \wedge \mathit{AT}(L, |\Sigma|) \wedge \mathit{WF}(\Sigma) \wedge \mathit{Corr}(\Sigma, M_T))$ holds when each $c_{s,i}$ is re-

placed with $A(c_{s,i})$, each $d_{s,b,i}$ is replaced with $A(d_{s,b,i})$, and each $e_{s,b,p,i}$ is replaced with $A(e_{s,b,p,i})$.

Let $n : (\Sigma \times [1, 2|\Sigma| + 1]) \to \mathbb{N}$ be the function mapping each $\langle s, i\rangle$ to $A(c_{s,i})$. Since this valuation satisfies $AT(L, |\Sigma|)$, we have that n is a $|\Sigma|$-marked abstract trajectory over Σ such that there exists a word $w = s_1 \dots s_{|w|} \in L$ such that $w \Vdash n$. We will show that $f(\rho) \xrightarrow{w}_{\mathcal{SV}} f(\rho')$, therefore showing $f(\rho) \xrightarrow{L}_{\mathcal{SV}} f(\rho')$.

Consider any $\langle x, C\rangle \in X \times \{0, 1\}^{\Sigma}$ such that $RV(\mathcal{SV}, s_i)(\langle x, C\rangle) = 1$ for all s_i in w. Observe that the first conjunct of $Transition(X, M_T, \Sigma)$ ensures that $\rho'(x) = \rho(x) + \sum_{i=1}^{|w|} v_i(\langle x, 1\rangle)$ where each $v_i \in S(M_T(s_i))$. Since it is the case that $v_i(x, 1) = \psi_{s_i}(v_i)(\langle x, C\rangle)$ for all i in $[|w|]$ since $C(s_i) = 1$, we have that $f(\rho')(\langle x, C\rangle) = f(\rho)(\langle x, C\rangle) + \sum_{i=1}^{|w|} \psi_{s_i}(v_i)(\langle x, C\rangle)$.

Consider any $\langle x, C\rangle \in X \times \{0, 1\}^{\Sigma}$ such that $RV(\mathcal{SV}, s_i)(\langle x, C\rangle) = 0$ for some s_i in w. Let j be the highest index such that $RV(\mathcal{SV}, s_j)(\langle x, C\rangle) = 0$. $WF(\Sigma)$ ensures that $A(c_{s_j,2k}) = 1$ for some k. The corresponding conjunct of $Transition(X, M_T, \Sigma)$ ensures that $\rho'(x) = v_j(\langle x, 0\rangle) + \sum_{i=j+1}^{|w|} v_i(\langle x, 1\rangle)$ where each $v_i \in S(M_T(s_i))$. Since it is the case that $v_j(\langle x, 0\rangle) = \psi_{s_j}(v_j)(\langle x, C\rangle)$ since $C(s_j) = 0$ and $v_i(\langle x, 1\rangle) = \psi_{s_i}(v_i)(\langle x, C\rangle)$ for all $i \in [j + 1, |w|]$ since $C(s_i) = 1$, we have that $f(\rho')(\langle x, C\rangle) = \psi_{s_j}(v_j)(\langle x, C\rangle) + \sum_{i=j+1}^{|w|} \psi_{s_i}(v_i)(\langle x, C\rangle)$.

Observe that for all $i \in [1, |w|]$, we have $\psi_{s_i}(v_i) \in OS(\mathcal{SV}, s_i)$. Then, by the above casework over all $\langle x, C\rangle$, we have that $f(\rho) \xrightarrow{L}_{\mathcal{SV}} f(\rho')$.

($\Longleftarrow$) Consider ρ, ρ' such that $f(\rho) \xrightarrow{L}_{\mathcal{SV}} f(\rho')$. There exists $w = s_1 \dots s_n \in L$ such that $f(\rho) \xrightarrow{w}_{\mathcal{SV}} f(\rho')$. Let d be the number of unique characters in w and let $i_1 \dots i_d$ be the indexes of the final occurrence of each letter; that is, character s_{i_j} does not appear in subword $s_{i_j+1} \dots s_n$. For all $j \in [1, d]$, let word w_{2j} be the character s_{i_j} and let word w_{2j-1} be the subword $s_{i_{j-1}+1} \dots s_{i_j-1}$; let w_{2d+1} through $w_{2|\Sigma|+1}$ be empty. Let $n : (\Sigma \times [1, 2|\Sigma| + 1]) \to \mathbb{N}$ be the function such that $n(s, i)$ is the number of occurrences of s in w_i. Observe that n is a $|\Sigma|$-marked abstract trajectory and $w \Vdash n$.

By assumption, we have $f(\rho) = \sigma_1 \xrightarrow{s_1}_{\mathcal{SV}} \ldots \xrightarrow{s_n} \sigma_{n+1} = f(\rho')$. For all $i \in [1, n]$, let $o_i \in \mathit{OS}(\mathcal{SV}, s_i)$ be the offset vector used in the transition $\sigma_i \xrightarrow{s_i} \sigma_{i+1}$. There exists some $\langle b, P\rangle \in M_T(s_i)$ such that $\psi(o_i) = b + \sum_{p\in P} \lambda_p p$. Fix such a representation for each o_i. Let $\phi : (D \cup E) \to \mathbb{N}$ be the function mapping each $d_{s,b,i}$ to the number of times b occurs in the representations of the o_j corresponding to all s_j in w_i and mapping each $e_{s,b,p,i}$ to the sum of λ_p in the representations of the o_j corresponding to all s_j in w_i.

Finally, we can observe that $\mathit{Summary}(X, M_T, L, \Sigma)$ holds when all $c_{s,i}$ are set to $n(s, i)$, all $d \in D$ are set to $\phi(d)$, and all $e \in E$ are set to $\phi(e)$. The subformulas $\mathit{AT}(L, |\Sigma|) \wedge \mathit{WF}(\Sigma)$ and $\mathit{Corr}(\Sigma, M_T)$ hold by construction of n and ϕ respectively. The first conjunct of $\mathit{Transition}(X, M_T, \Sigma)$ holds because the transition $f(\rho) \xrightarrow{w}_{\mathcal{SV}} f(\rho')$ implies that $f(\rho')(x, \lambda s.1) = f(\rho)(x, \lambda s.1) + \sum_{i=1}^{n} o_i(x, \lambda s.1)$ or equivalently $\rho'(x) = \rho(x) + \sum_{i=1}^{n} \psi(o_i)(x, 1)$. For the remainder of the conjuncts, with $k \in [1, d]$ the transition implies that $f(\rho')(x, \lambda s.s = s_{i_k}) = o_{i_k}(x, \lambda s.s = s_{i_k}) + \sum_{j=i_k+1}^{n} o_j(x, \lambda s.s = s_{i_k})$ or equivalently $\rho'(x) = \psi(o_{i_k})(x, 0) + \sum_{j=i_k+1}^{n} \psi(o_j)(x, 1)$. Therefore, $\mathit{Transition}(X, M_T, \Sigma)$ holds, and we therefore have that $\mathit{Summary}(X, M_T, L, \Sigma)$ holds. □

6.5 Discussion

The central result of this chapter is that we can use the SVASR-reflection of any transition assignment to over-approximate its context-free reachability relation. This is perhaps surprising: the SVASR reflection constructed in Section 6.3 has a state space exponential in the size of the alphabet. The key insight of Section 6.4 is that this exponential structure is highly symmetric and this symmetry can be exploited to compute the implied reachability formula without ever materializing the reflection itself.

Following the recipe introduced in Chapter 3, the resulting program analysis is sound, monotone, and local. However, a significant obstacle stands between these theoretical results

and a practical implementation. The polynomial time bound is with respect to the sizes of the bases of the semi-linear sets $M_{\sharp}(s)$. These bases can be arbitrarily large. For example, consider the following transition formula for an arbitrary integer $n > 0$:

$$0 < x \wedge x \leq n$$

The semi-linear set of models of this formula is:

$$\bigcup_{i=1}^{n} \{i\}$$

and therefore has a basis representation of size n. This means that even a single variable formula has no bound on the size of the basis of its models. Finding a way to handle the space overhead incurred by semi-linear sets in formulas is an active area of research [42].

Chapter 7

Almost-Commuting Transition Systems

In this section, we aim to investigate the formal foundations making our reachability constructions possible for VASR, LVASR, and SVASR. A central tool in these constructions is Parikh's Theorem, which allows us to compute the set of possible character counts of words in a context-free language. This character count is the *universal commutative abstraction* of a language of paths: if every operation in our program commuted, we could use the count of every operation to infer the net effect of all operations, as order would not matter.

VASR, LVASR, and SVASR do not have commutative operations, making their successful use of Parikh's Theorem surprising. Their reachability constructions work by computing the Parikh image of an unfolded copy of the grammar of a context-free language, which decomposes each word in the language into a finite number of phases. Each phase corresponds to a subsequence in which the transitions do commute, enabling Parikh-based reasoning to proceed piecewise.

This discussion raises the following question:

Is there a generalization of these works that extends the power of Parikh's Theorem to reason about systems that are only partially commutative?

In this chapter, we answer this question affirmatively by identifying a new algebraic structure that combines a finite-state control component with a commutative-state component. This class is neither finite nor commutative, but remains amenable to Parikh-based analysis. Our key technical result is that the image of a context-free language under a homomorphism into an *almost-commuting monoid* is a semi-linear set and definable in polynomial time. This result expands the applicability of Parikh's Theorem, and may be of independent interest.

The motivating application for this result is in solving CFL-reachability problems for transition systems represented by almost-commuting monoids, which we call *almost-commuting transition systems*. We identify several concrete examples of transition systems that non-trivially inhabit this algebraic structure, including a generalization of the abstract domain seen thus far in this thesis. Our technical result yields an automatic decision procedure for solving CFL-reachability queries concerning all of our examples, and more generally all almost-commuting transition systems meeting a *queryability* condition.

Our work can be viewed as a potential "backend" for automated program analysis. A client would have two obligations: (i) identifying a suitable family of almost-commuting transition systems S in which to model programs, and (ii) developing an abstraction procedure to translate a given program into an S system that over-approximates the behavior of the program. Our work then allows the computation of a summary formula describing the possible executions of the abstraction over a context-free set of control paths.

7.1 Overview: Generalizing VASR

Our approach to defining the reachability relation of vector addition systems with resets was based on computing the set of abstract trajectories of a context-free language of paths which

captured enough information to infer the net effect of the related VASR transitions. These abstract trajectories were an extension of the Parikh image, or character count, in which a finite set of characters are identified along with the sub-words in between. We used the finite set of characters to mark the *final reset* of each dimension of the VASR. The key fact that enabled the use of the Parikh image was the commutativity of operations within each subword between final resets.

The generalized approach presented in this chapter, when applied to VASR, directly computes these commutative subwords by expanding the underlying grammar. The advantage of this formulation of VASR reachability is that it can be readily generalized. For example, consider the following numeric programs:

```
def f(x):
    if (*):
        return 7
    else:
      return g(x + 1)
```

```
def g(x):
    if (*):
        return -x + 2
    else if (*):
        return f(-x)
    else:
        return f(x) + 1
```

Just as our formulation for VASR reachability decomposed runs into phases before and after the final reset of each variable, we may decompose runs through `f` into phases based on the algebraic structure of the operations. All operations are of the form $x' = r * x + o$ in which x and x' are the pre-state and post-state values respectively, $r \in \{-1, 0, 1\}$, and $o \in \mathbb{Z}$, and this form is preserved under composition. Intuitively, we decompose runs through these functions into phases based on the possible value of r while modifying the additive component o based on the r context. We may display this composition by unfolding each procedure above into multiple copies annotated by the r context.

```
def f_shell(x):
  if (*):
    return x + f_one()
  elif (*):
    return -x + f_neg()
  else:
    return f_zero()
```

```
def f_neg():
  return -1 + g_neg()

def f_zero():
  if (*):
    return 7
  else:
    return g_zero()
```

```
def g_neg():
  if (*):
    return 2
  elif (*):
    return f_one()
  else:
    return f_neg() + 1
```

```
def f_one():
  return 1 + g_one()
```

```
def g_one():
  if (*):
    return f_neg()
  else:
    return f_one() + 1
```

```
def g_zero():
  if (*):
    return f_zero()
  else:
    return f_zero() + 1
```

The `f` and `g` procedures have been unfolded into a shell procedure `f_shell`, which has the same input-output behavior as `f`, and six context-labeled procedures in which the contexts are `_one`, `_neg`, and `_zero`. The context tracks the effect of later operations on the current increment—i.e., how future manipulations of the x parameter (of the original program) affect the local increment. For example, because the body of `f_neg` presumes that calling `g_neg` will negate future increments, it negates the increment performed in procedure `f` (*decrementing* by 1 rather than *incrementing* by 1).

For example, consider the run of `f` with input $x = 3$ and returns -2 via the path:

`f(x)` → `g(x+1)` → `f(x+1)+1` → `g((x+1)+1)+1` → `(-((x+1)+1)+2)+1` → `-x+1` → `-2`

This run corresponds to the following run in `f_shell` with the same input-output behavior:

$$\texttt{f\_shell(x)} \to \texttt{-x+f\_neg()} \to \texttt{-x-1+g\_neg()} \to \texttt{-x-1+f\_neg()+1}$$

$$\to \texttt{-x-1-1+g\_neg()+1} \to \texttt{-x-1-1+2+1} \to \texttt{-x+1} \to \texttt{-2}$$

Each run of `f_shell` contains (i) exactly one initial operation, which sets the context and adds in the value of x relevant to that context, (ii) a sequence of operations performed in `f_one`, `f_neg`, `f_zero`, `g_one`, `g_neg`, and `g_zero`, and (iii) exactly one constant return from `f_zero` or `g_neg`. All arithmetic operations in (ii) are increments to the top-level return variable, which commute with each other.

The occurrence of the "-x" term in the second branch of `f_shell` is not an instance of a non-commuting operation in the sequence of operations performed during the run. One should think of "-x" as preceding the sequence, and setting the context of what is being computed. Then, our transformations on this program allow us to reinterpret our program as a purely commutative program with a finite control skeleton. Because `f_shell` is semantically equivalent to `f`, the transformation allows us to compute the input-output relation of `f` via the Parikh image of the runs through `f_shell`, as desired.

7.2 Technical Definitions

This section defines *almost-commuting monoids* (ACMs). The following sections will provide examples of such monoids and examine their properties.

Definition 21. A monoid $\langle M, \cdot, 1\rangle$ consists of a universe of elements M, a binary operator $(-)\cdot(-) : M \times M \to M$, and an identity element 1, where:

- (Associativity) For any $a, b, c \in M$ we have $(a \cdot b) \cdot c = a \cdot (b \cdot c)$

- (Identity) For any $a \in M$ we have $1 \cdot a = a \cdot 1 = a$

An almost-commuting monoid (ACM) is a monoid $\langle M, \cdot, 1 \rangle$ whose elements can be fully described as the image of a finite set $F \subseteq M$ under a right action $\triangleleft$ by a commutative monoid $\langle C \subseteq \mathbb{Z}^n, +, 0 \rangle$. The action $\triangleleft$ respects both the monoid structure of M (via a kind of "reassociativity" rule for $\cdot$ and $\triangleleft$), and the additive structure of C (via a kind of "mixed" associativity rule for $\triangleleft$ and $+$).

Definition 22 (Almost-Commuting Monoid)**.** A monoid $\langle M, \cdot, 1 \rangle$ is **almost-commuting** if there exists a finite subset F of M, a commutative integer monoid $\langle C \subseteq \mathbb{Z}^n, +, 0 \rangle$, and an action $\triangleleft : M \times C \to M$ such that:

1. $M = \{f \triangleleft c : f \in F, c \in C\}$
2. (Reassociativity) For all $m_1, m_2 \in M$ and all $c \in C$, we have $(m_1 \cdot m_2) \triangleleft c = m_1 \cdot (m_2 \triangleleft c)$
3. (Action) For all $m \in M$ and all $c_1, c_2 \in C$, we have $(m \triangleleft c_1) \triangleleft c_2 = m \triangleleft (c_1 + c_2)$ and $m \triangleleft 0 = m$
4. There is an algorithm that, given $m \in M$, computes $f \in F$ and $c \in C$ such that $m = f \triangleleft c$.

(Note the mnemonic: F and (possibly subscripted) f for "finite"; C and (possibly subscripted) c for "commutative.")

Example 7.2.1. Consider the monoid $\langle \{0, 1\} \times \mathbb{Z}, \cdot, \langle 1, 0 \rangle \rangle$ in which $\cdot$ is defined as:

$$\langle a, b \rangle \cdot \langle a', b' \rangle = \langle aa', a'b + b' \rangle .$$

This monoid is an almost-commuting monoid, with finite subset $F = \{\langle 0,0\rangle, \langle 1,0\rangle\}$, commutative monoid $\langle \mathbb{Z}, +, 0\rangle$, and action $\triangleleft$ defined by $\langle a, b\rangle \triangleleft c = \langle a, b + c\rangle$. The first three conditions can be straightforwardly checked; the algorithm for the fourth condition is component-wise projection.

Our goal is to compute the images of context-free languages under homomorphisms into almost-commuting monoids. Effective representation of such images hinges on identifying a class of sets that is both expressive and tractable. Semi-linear sets fit this role, and we represent them using formulas in LIA.

Definition 23. Let M be an ACM parametrized by finite subset F, commutative monoid $\langle C \subseteq \mathbb{Z}^n, +, 0\rangle$, and action $\triangleleft : M \times C \to M$. A *linear set* over M is a set of the form:

$$S = \{f \triangleleft (b + \lambda_1 p_1 + \cdots + \lambda_k p_k) \mid \lambda_i \in \mathbb{N}\}$$

for $f \in F$ and $b, p_1, \ldots, p_k \in C$. **Semi-linear sets** are finite unions of linear sets.

Definition 24. Let M be an ACM parametrized by finite subset $F = \{f_1, \ldots, f_{|F|}\}$, commutative monoid $\langle C \subseteq \mathbb{Z}^n, +, 0\rangle$, and action $\triangleleft : M \times C \to M$. We say that LIA formula $\phi(x_0, \ldots, x_n)$ *weakly defines* a subset $S \subseteq M$ if

$$S = \{f_i \triangleleft (c_1, \ldots, c_n) \mid \phi(i, c_1, \ldots, c_n) \text{ holds}\}.$$

LIA formula $\phi(x_0, \ldots, x_n)$ *strongly defines* subset $S \subseteq M$ if

$$\{\langle f_i, c_1, \ldots, c_n\rangle \mid \phi(i, c_1, \ldots, c_n) \text{ holds}\} = \{\langle f, c_1, \ldots, c_n\rangle \mid f \triangleleft (c_1, \ldots, c_n) \in S\}.$$

The distinction between a formula strongly defining a subset S versus weakly defining S is that strongly defining means that the formula recognizes *all* $\langle f, c\rangle$ such that $f \triangleleft c$ lies within

S, whereas weakly defining means that the formula recognizes *at least one* $\langle f, c\rangle$ such that $f \triangleleft c = m$ for each element m of S. As expected, a formula that strongly defines a subset also weakly defines it. Weakly definable sets coincide with semi-linear sets over almost-commuting monoids.

The above definitions allow us to use LIA formulas to define semi-linear subsets of almost-commuting monoids. Note that what the formulas define is not quite the same as the notion of definability in the traditional sense: our formulas do not identify the elements of a subset of an ACM, they identify a set of *representations* of these elements. We provide several examples of ACMs that model transition systems in §7.3, and in §7.4, we show that even though our formulas identify representations of sets of interest, they still allow CFL-reachability queries to be answered.

Our central technical result shows that we can compute a LIA formula that weakly defines the image of a context-free language under a monoid homomorphism into an ACM. That is, let $\mathcal{L}(G) \subseteq \Sigma^*$ be a context-free language over a finite alphabet Σ, let $v : \Sigma \to M$ be a valuation of that alphabet, and let $v^* : \Sigma^* \to M$ be v extended to a monoid homomorphism over ACM $\langle M, \cdot, 1\rangle$. Then, the set:

$$\{v^*(w) \mid w \in \mathcal{L}(G)\}$$

is semi-linear, and we can compute a LIA formula that weakly defines it in polynomial time.

This result generalizes Parikh's Theorem beyond commutative monoids, and may be of independent technical interest. Our motivation for this generalization is its application to answering context-free-language reachability queries involving transition systems that are represented by almost-commuting monoids.

Definition 25 (Almost-Commuting Transition System (ACTS))**.** A labeled transition system $\langle \mathbb{Z}^X, \to \rangle$ over finite alphabet Σ is *almost-commuting* if the monoid $\langle M, \cdot, \mathbb{I} \rangle$ is almost-commuting, in which:

- M is the closure of $\left\{ \xrightarrow{s} : s \in \Sigma \right\}$ under relational composition,
- $\cdot$ denotes relational composition, and
- $\mathbb{I} \subseteq \mathbb{Z}^X \times \mathbb{Z}^X$ is the identity relation.

Our main technical result in the context of transition systems means that for any context-free language $\mathcal{L}(G)$ and any almost-commuting transition system T, we can compute a representation of all transition relations corresponding to $\mathcal{L}(G)$ paths through T. That is, we can weakly define the set:

$$\left\{ \xrightarrow{s_1}_T \circ \cdots \circ \xrightarrow{s_n}_T : s_1 \dots s_n \in \mathcal{L}(G) \right\}$$

However, this result is not a sufficient representation of the $\mathcal{L}(G)$-reachability relation for T because we are unable to test for membership. That is, given two states of the transition system, we do not have a way to test if any of our transition relations contain the states. To bridge this gap, we introduce the following definition:

Definition 26. Let T be a labeled transition system over finite alphabet Σ and let M be the closure of $\left\{ \xrightarrow{s}_T : s \in \Sigma \right\}$ under relational composition. T is a **queryable ACTS** if T is an ACTS and there is an algorithm that computes, for any states $\rho, \rho' \in S$, a LIA formula strongly defining the set

$$\{ \to \in M \mid \rho \to \rho' \}$$

With this condition in hand, given two states ρ, ρ' of the transition system, we may conjoin the formula that strongly defines the set of transitions between the states with the

formula that weakly defines the set of $\mathcal{L}(G)$ paths through the transition system to produce a formula that weakly defines the set:

$$\left\{ \xrightarrow{s_1}_T \circ \cdots \circ \xrightarrow{s_n}_T \mid \rho \xrightarrow{s_1}_T \circ \cdots \circ \xrightarrow{s_n}_T \rho', s_1 \ldots s_n \in \mathcal{L}(G) \right\}$$

CFL-reachability between ρ and ρ' then reduces to checking whether this set is empty; that is, satisfiability of this formula.

In the following section, we present several concrete examples of almost-commuting transition systems; we also show that these transition systems are queryable. In §7.4, we formally prove that we have a decision procedure for CFL-reachability problems over queryable ACTS.

7.3 Examples of ACTS

This section provides examples of almost-commuting transition systems. Theorem 17 in §7.4 implies that we immediately have a decision procedure for context-free reachability problems involving all of these systems. §7.5 shows that ACTS are closed under several operations, enabling us to generalize all of these examples to arbitrary dimension and allowing us to build complex ACTS out of simple components. The three families studied in earlier chapters—VASR, LVASR, and SVASR—are each subsumed by or reducible to our first example, called finite monoid affine vector addition systems.

7.3.1 Finite Monoid Affine Vector Addition Systems

Finite Monoid Affine Vector Addition Systems (FMAVAS) are a generalization of integer VAS that were introduced in [8], drawing on prior work on computing closures of affine relations [9, 22]. Their work showed that the regular-language reachability relation of FMAVAS is computable via a reduction to the regular-language reachability relation of vector addition

systems. We show here that FMAVAS are queryable almost-commuting transition systems, and thereby show that we can solve CFL-reachability problems over them.

Definition 27. A *finite monoid affine vector addition system* over a finite alphabet Σ is a labeled transition system $\langle \mathbb{Z}^n, \{\rightarrow_s |\ s \in \Sigma\}\rangle$ equipped with a finite monoid $\langle L, \circ, \mathbb{I}\rangle$ of linear functions $\mathbb{Z}^n \rightarrow \mathbb{Z}^n$, such that for each label $s \in \Sigma$, the transition relation $\rightarrow_s$ satisfies:

$$x \rightarrow_s x' \iff x' = f(x) + o \quad \text{for some } f \in L,\ o \in \mathbb{Z}^n$$

The program in §7.1 is an FMAVAS. The following is an example of a 2-dimensional FMAVAS.

Example 7.3.1. Let the alphabet Σ consist of a single character a . Let $\langle \mathbb{Z}^2, \{\rightarrow_a\}\rangle$ be a FMAVAS in which:

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightarrow_a \begin{pmatrix} x' \\ y' \end{pmatrix} \Leftrightarrow \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 3 \\ 5 \end{pmatrix}$$

This FMAVAS is equipped with finite monoid $\langle L, \circ, \mathrm{id}\rangle$ in which:

$$L = \left\{ \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \mathrm{id} \right\}$$

One can observe that VASRs are a special case of FMAVAS in which the finite monoid $\langle L, \circ, \mathbb{I}\rangle$ is of the form:

$$L = \{\lambda\rho.r * \rho : r \in \{0,1\}^n\}$$

The following theorems, which establish that every FMAVAS is an almost-commuting transition system and that they are queryable, establish the same properties for VASR. The following theorems are also in principle adaptable to prove that every Lossy FMAVAS (defined by replacing the $=$ in the above definition with $\leq$ and requiring a positivity constraint over the matrices in L) is an almost-commuting transition system and that they are queryable, and therefore that these properties also hold for Lossy VASR. Semi-Linear VASRs are not obviously almost-commuting transition systems, but we can use the machinery developed in this chapter to analyze Semi-Linear VASRs using their correspondence to VASRs that was described in Chapter 6.

Finally, we observe that every FMAVAS is an almost-commuting transition system, and that they are queryable.

Lemma 15. Every finite-monoid affine vector addition system is an almost-commuting transition system.

Proof. Let $\langle \mathbb{Z}^n, \{\rightarrow_s | \ s \in \Sigma\}\rangle$ be an FMAVAS over finite alphabet Σ, equipped with a finite monoid $\langle L, \circ, \mathbb{I}\rangle$ of linear functions. Let M denote the closure of $\{\rightarrow_s | \ s \in \Sigma\}$ under relational composition $\cdot$, and let $\mathbb{I}$ be the identity relation on $\mathbb{Z}^n$. We will show that $\langle M, \cdot, \mathbb{I}\rangle$ is an almost-commuting monoid.

For each $f \in L$, define $R_f = \{(x, x') \mid x' = f(x)\}$. Let $F = \{R_f \mid f \in L\}$ and let $C = \mathbb{Z}^n$. Define the action $\triangleleft : M \times C \rightarrow M$ by:

$$m \triangleleft c = \{(x, x' + c) \mid (x, x') \in m\}.$$

To verify Condition (1) of Definition 22, observe that every generator $\rightarrow_s$ of M is of the form:

$$\{(x, x') \mid x' = f(x) + o\} = R_f \triangleleft o$$

for some $f \in L, o \in C$. Additionally, we have the following closure property: for all $f, f' \in L$ and $o, o' \in C$:

$$(R_f \triangleleft o) \cdot (R_{f'} \triangleleft o') = R_{f' \circ f} \triangleleft (f'(o) + o'),$$

Because $\{R_f \triangleleft o : R_f \in F, o \in \mathbb{Z}\}$ contains the generators of M and is closed under composition, it is equal to M. Thus, every element of M is of the form $R_f \triangleleft o$ for some $R_f \in F$ and $o \in \mathbb{Z}^n$.

Conditions (2) and (3) follow directly from the definition of $\triangleleft$. Condition (4) is straightforward: the affine transformation $x' = f(x) + o$ decomposes into $R_f \in F$ and $o \in C$. □

Lemma 16. Finite-monoid affine vector addition systems are queryable.

Proof. Let $\langle \mathbb{Z}^n, \{\rightarrow_s |\ s \in \Sigma\}\rangle$ be an FMAVAS over finite alphabet Σ, equipped with a finite monoid $\left\langle L = \left\{f_1 \dots f_{|L|}\right\}, \circ, \mathbb{I}\right\rangle$ of linear functions. Let M denote the closure of $\{\rightarrow_s |\ s \in \Sigma\}$ under relational composition.

Consider any states $\rho, \rho' \in \mathbb{Z}^n$. We can define the set $\{\rightarrow \in M \mid \rho \rightarrow \rho'\}$ with the following formula:

$$\bigvee_{i=1}^{|L|} x_0 = i \wedge \rho' = f_i(\rho) + (x_1, \dots, x_n),$$

where x_0 encodes the choice of $f_i \in L$ and $(x_1, \dots, x_n)$ encodes the offset vector in $\mathbb{Z}^n$. This formula thus strongly defines (Definition 24) the required subset of M. □

7.3.2 Natural Offset Max-Plus Linear Systems

Max-Plus Linear Systems arise from the use of the max-plus algebra, where "addition" is replaced by taking maximums and "multiplication" by addition. Certain discrete-event systems, including production lines, railway networks, and communication protocols, can be modeled within this algebra [17]. Max-Plus Linear Systems are transition systems in which the transitions correspond to the analogue of affine updates in the max-plus algebra. We

show here that the subclass of Max-Plus Linear Systems in which the offset coefficients are non-negative are queryable almost-commuting transition systems.

Definition 28. A *natural offset max-plus linear system* over a finite alphabet Σ is a labeled transition system $\langle \mathbb{Z}, \{\rightarrow_s |\ s \in \Sigma\}\rangle$ equipped with functions $a : \Sigma \to \mathbb{Z}$ and $b : \Sigma \to \mathbb{N}$ such that for each label $s \in \Sigma$, the transition relation $\rightarrow_s$ satisfies:

$$x \rightarrow_s x' \iff x' = \max(x, a(s)) + b(s)$$

We now observe that these transition systems are almost-commuting and queryable.

Theorem 14. Every Natural Offset Max-Plus Linear System is an almost-commuting transition system.

Proof. Let $\langle \mathbb{Z}, \{\rightarrow_s |\ s \in \Sigma\}\rangle$ be a natural offset max-plus linear system equipped with functions $a : \Sigma \to \mathbb{Z}$ and $b : \Sigma \to \mathbb{N}$. Let M denote the closure of $\{\rightarrow_s |\ s \in \Sigma\}$ under relational composition $\cdot$ and let $\mathbb{I}$ be the identity relation on $\mathbb{Z}$. We will show that $\langle M, \cdot, \mathbb{I}\rangle$ is an almost-commuting monoid.

Let I be the closed interval $[\min_s a(s), \max_s a(s)]$ and define for all $v \in I$:

$$R_v = \{(x, x') \mid x' = \max(x, v)\}$$

We define:

$$F = \{R_v \mid v \in I\} \quad C = \mathbb{N} \quad m \triangleleft c = \{(x, x' + c) \mid (x, x') \in m\}$$

Observe that for every generator $\rightarrow_s$ of M, we have that:

$$\rightarrow_s = R_{a(s)} \triangleleft b(s)$$

We now show closure under composition. For all $R_v, R_{v'} \in F$ and $c, c' \in \mathbb{N}$:

$$\begin{aligned}
R_v \triangleleft c \cdot R_{v'} \triangleleft c' &= \{\langle x, x'' \rangle \mid x' = \max(x, v) + c, x'' = \max(x', v') + c'\} \\
&= \{\langle x, x'' \rangle \mid x'' = \max(\max(x, v) + c, v') + c'\} \\
&= \{\langle x, x' \rangle \mid x' = \max(\max(x, v) + c, v') + c'\} \\
&= \{\langle x, x' \rangle \mid x' = \max(\max(x + c, v + c), v') + c'\} \\
&= \{\langle x, x' \rangle \mid x' = \max(x + c, \max(v + c, v')) + c'\} \\
&= \{\langle x, x' \rangle \mid x' = \max(x, \max(v, v' - c)) + c + c'\} \\
&= R_{\max(v, v' - c)} \triangleleft (c + c')
\end{aligned}$$

Because $v, v' \in I$ and $c \in \mathbb{N}$, $\max(v, v' - c)$ also lies in I. Hence, we have that compositions remain in the form $R_v \triangleleft c$, and thus by structural induction over M we have that every element therein can be represented in that form (Condition (1) of Definition 22). Conditions (2) and (3) follow straightforwardly from the definition of $\triangleleft$. Condition (4) is straightforward: transformer $x' = max(x, a) + b$ is associated with $R_a \in F$ and $b \in \mathbb{N}$. □

Lemma 17. Natural Offset Max-Plus Linear Systems are queryable.

Proof. Let $\langle \mathbb{Z}, \{\rightarrow_s | \ s \in \Sigma\} \rangle$ be a natural offset max-plus linear system equipped with functions $a : \Sigma \to \mathbb{Z}$ and $b : \Sigma \to \mathbb{N}$. Let M denote the closure of $\{\rightarrow_s | \ s \in \Sigma\}$ under relational composition. Let $l = \min_s a(s)$ and $r = \max_s a(s)$. Let the finite subset of M be denoted $F = \{R_l, \ldots, R_r\}$, where $R_v = \{(x, x') \mid x' = \max(x, v)\}$ for all $v \in [l, r]$.

Consider any states $\rho, \rho' \in \mathbb{Z}$. We can define the set $\{\rightarrow \in M \mid \rho \rightarrow \rho'\}$ with the following formula:

$$\bigvee_{i=1}^{|F|} x_0 = i \wedge \rho' = \max(\rho, l + i - 1) + x_1$$

where x_0 encodes the choice of the threshold in the max, and x_1 encodes the offset. Note that max can be straightforwardly encoded into LIA arithmetic. □

7.3.3 Parity-Guarded Systems

Parity-Guarded Systems are transition systems whose update rule depends on the parity of the current state. They are a simple example of how guarded updates can be accommodated within our framework. We show here that Parity-Guarded Systems are *dual* ACTS.

Definition 29. A *parity-guarded system* over a finite alphabet Σ is a labeled transition system $\langle \mathbb{Z}, \{\rightarrow_s | \ s \in \Sigma\}\rangle$ equipped with functions $a, b : \Sigma \rightarrow \mathbb{Z}$ such that for each label $s \in \Sigma$, the transition relation $\rightarrow_s$ satisfies:

$$x \rightarrow_s x' \Leftrightarrow x' = \text{if } x \text{ is even then } x + a(s) \text{ else } x + b(s)$$

Parity-Guarded Systems follow a related structure to ACTS.

Definition 30. A *dual* almost-commuting monoid is a monoid $\langle M, \cdot, 1\rangle$ such that there exists a finite subset F of M, a commutative monoid $\langle C, +, 0\rangle$, and a function $\triangleleft : M \times C \rightarrow M$ such that:

1. Conditions (1), (3), and (4) of Definition 22 are met
2. (Dual Reassociativity) For all $m_1, m_2 \in M$ and all $c \in C$, we have $(m_1 \cdot m_2) \triangleleft c = (m_1 \triangleleft c) \cdot m_2$

Definition 31. A *dual* almost-commuting transition system is defined analogously to an ACTS with "dual ACM" in place of "ACM".

Theorem 15. Every Parity-Guarded System is a dual almost-commuting transition system.

Proof. Let $\langle \mathbb{Z}, \{\rightarrow_s |\ s \in \Sigma\}\rangle$ be a parity-guarded system equipped with functions $a, b : \Sigma \to \mathbb{Z}$. Let M denote the closure of $\{\rightarrow_s |\ s \in \Sigma\}$ under relational composition $\cdot$ and let $\mathbb{I}$ be the identity relation on $\mathbb{Z}$. We will show that $\langle M, \cdot, \mathbb{I}\rangle$ is a dual ACM.

We define:

$$F = \left\{ \begin{array}{c} f_{00} = \{\langle x, x'\rangle \mid x' = \text{if } x \text{ is even then } x \text{ else } x\}, \\ f_{10} = \{\langle x, x'\rangle \mid x' = \text{if } x \text{ is even then } x+1 \text{ else } x\}, \\ f_{01} = \{\langle x, x'\rangle \mid x' = \text{if } x \text{ is even then } x \text{ else } x+1\}, \\ f_{11} = \{\langle x, x'\rangle \mid x' = \text{if } x \text{ is even then } x+1 \text{ else } x+1\} \end{array} \right\}$$

$$C = \langle \mathbb{Z} \times \mathbb{Z}, +, \langle 0, 0\rangle\rangle \text{ where } \langle a, b\rangle + \langle c, d\rangle = \langle a + c, b + d\rangle$$

$$m \triangleleft \langle c_e, c_o\rangle = \{\langle x, x''\rangle \mid \langle x, x'\rangle \in m, x'' = \text{if } x \text{ is even then } x' + 2c_e \text{ else } x' + 2c_o\}$$

Observe that for every generator $\rightarrow_s$ of M we have:

$$\rightarrow_s = \{\langle x, x'\rangle \mid x' = \text{if } x \text{ is even then } x + (a(s) \bmod 2) \text{ else } x + (b(s) \bmod 2)\} \triangleleft \left\langle \left\lfloor \frac{a(s)}{2} \right\rfloor, \left\lfloor \frac{b(s)}{2} \right\rfloor \right\rangle$$

We investigate composition $f \triangleleft \langle c_e, c_o\rangle \cdot f' \triangleleft \langle c'_e, c'_o\rangle$ by case analysis over f and f'. In the following, if x or y appears in the subscript of an f, the equation holds regardless of whether

the x or y is replaced with a 1 or a 0.

$$f_{00} \triangleleft \langle c_e, c_o \rangle \cdot f_{xy} \triangleleft \langle c'_e, c'_o \rangle = f_{xy} \triangleleft \langle c_e + c'_e, c_o + c'_o \rangle$$

$$f_{01} \triangleleft \langle c_e, c_o \rangle \cdot f_{0x} \triangleleft \langle c'_e, c'_o \rangle = f_{01} \triangleleft \langle c_e + c'_e, c_o + c'_e \rangle$$

$$f_{01} \triangleleft \langle c_e, c_o \rangle \cdot f_{1x} \triangleleft \langle c'_e, c'_o \rangle = f_{10} \triangleleft \langle c_e + c'_e, c_o + c'_e + 1 \rangle$$

$$f_{10} \triangleleft \langle c_e, c_o \rangle \cdot f_{x0} \triangleleft \langle c'_e, c'_o \rangle = f_{10} \triangleleft \langle c_e + c'_o, c_o + c'_o \rangle$$

$$f_{10} \triangleleft \langle c_e, c_o \rangle \cdot f_{x1} \triangleleft \langle c'_e, c'_o \rangle = f_{01} \triangleleft \langle c_e + c'_o + 1, c_o + c'_o \rangle$$

$$f_{11} \triangleleft \langle c_e, c_o \rangle \cdot f_{00} \triangleleft \langle c'_e, c'_o \rangle = f_{11} \triangleleft \langle c_e + c'_o, c_o + c'_e \rangle$$

$$f_{11} \triangleleft \langle c_e, c_o \rangle \cdot f_{01} \triangleleft \langle c'_e, c'_o \rangle = f_{01} \triangleleft \langle c_e + c'_o + 1, c_o + c'_e \rangle$$

$$f_{11} \triangleleft \langle c_e, c_o \rangle \cdot f_{10} \triangleleft \langle c'_e, c'_o \rangle = f_{10} \triangleleft \langle c_e + c'_o, c_o + c'_e + 1 \rangle$$

$$f_{11} \triangleleft \langle c_e, c_o \rangle \cdot f_{11} \triangleleft \langle c'_e, c'_o \rangle = f_{00} \triangleleft \langle c_e + c'_o + 1, c_o + c'_e + 1 \rangle$$

Therefore, we have that composition retains the structure $f \triangleleft c$ and that therefore by structural induction all elements within M are representable in that form (Condition (1) of Definition 22). Condition (3) of Definition 22 follows from the definition of $\triangleleft$, and Condition (4) is straightforward. Finally, note from the case analysis above that $(m_1 \cdot m_2) \triangleleft c = (m_1 \triangleleft c) \cdot m_2$. Thus, we have that parity-guarded systems are dual ACTS. □

Lemma 18. Parity-Guarded Systems are queryable.

Proof. Let $\langle \mathbb{Z}, \{\rightarrow_s | \ s \in \Sigma\} \rangle$ be a parity-guarded system. Let M denote the closure of $\{\rightarrow_s | \ s \in \Sigma\}$ under relational composition $\cdot$ and let $\mathbb{I}$ be the identity relation on $\mathbb{Z}$. Denote the finite subset $F = \{f_0 = f_{00}, f_1 = f_{01}, f_2 = f_{10}, f_3 = f_{11}\}$. In the following formula, the x_0 variable is a selector variable over these elements: when $x_0 = i$, the formula is selecting f_i.

Consider any states $\rho, \rho' \in \mathbb{Z}$. We can define the set $\{\rightarrow \in M \mid \rho \rightarrow \rho'\}$ with the following formula, in which x_1 and x_2 refer to the even-case and odd-case increments of the commutative component respectively:

$$\begin{aligned}
\exists y.\rho = 2y \wedge (((x_0 = 0 \vee x_0 = 1) \wedge \rho' = \rho + 2x_1) \\
\vee ((x_0 = 2 \vee x_0 = 3) \wedge \rho' = \rho + 1 + 2x_1)) \vee \\
\exists y.\rho = 2y + 1 \wedge (((x_0 = 0 \vee x_0 = 2) \wedge \rho' = \rho + 2x_2) \\
\vee ((x_0 = 1 \vee x_0 = 3) \wedge \rho' = \rho + 1 + 2x_2))
\end{aligned}$$

in which x_0 selects a base relation in F, and (x_1, x_2) encode the even- and odd-case offsets of the commutative component. □

7.4 Context-Free Reachability of ACTS

This section presents the main result of this chapter: that we can efficiently define the images of context-free languages under homomorphisms into an almost-commuting monoid. In the context of transition systems, this result means that we can answer context-free-language reachabilty queries about queryable almost-commuting transition systems. Our results are stated in the following theorems:

Theorem 16. Let $\langle M, \cdot, 1\rangle$ be an almost-commuting monoid, let Σ be a finite alphabet, and let $\mathcal{L}(G) \subseteq \Sigma^*$ be a context-free language. Let $v^* : \Sigma^* \rightarrow M$ be a homomorphism. Then, the set

$$\{v^*(w) \mid w \in \mathcal{L}(G)\}$$

is semi-linear and can be weakly defined in time $O(|G||F|^3)$ where $F \subseteq M$ is the finite subset of M.

Theorem 17. Let T be a queryable ACTS defined over a finite alphabet Σ, and let $\mathcal{L}(G) \subseteq \Sigma^*$ be a context-free language. Then there is a decision procedure to query the $\mathcal{L}(G)$-reachability relation of T for membership.

We present our construction for Theorem 16 diagrammatically in §7.4.1, and rigorously in §7.4.2.

7.4.1 Theorem 16 in Attribute-Grammar Diagrams

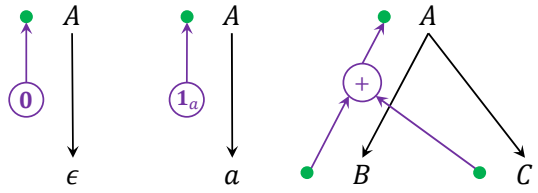
In this section, we give a proof of Theorem 16, using attribute-grammar diagrams to portray graphically the problem-transformation steps that establish the theorem. Readers who prefer a proof without such visual aids should skip to §7.4.2.

We begin by formulating the computation of the Parikh image of a context-free language using attribute grammars. Parikh's Theorem establishes that images of context-free languages computed via such attribute grammars are semi-linear and efficiently definable. We then formulate the computation of the image of a context-free language in an almost-commuting monoid using attribute grammars. We apply a sequence of equivalence-preserving rewrites to these attribute grammars to produce one that mirrors that of the Parikh image. We thereby show that we can compute the image of a context-free language in an almost-commuting monoid via the Parikh image of this grammar.

Without loss of generality, we can assume that the context-free grammar G is in Chomsky Normal Form. For each $a \in \Sigma$, let $\mathbf{1}_a : \Sigma \to \mathbb{N}$ be the function that maps a to 1 and all other characters $a' \neq a$ to 0. Let $\mathbf{0} : \Sigma \to \mathbb{N}$ denote the function $\lambda\sigma.0$. We can define an addition operation for such functions as follows:

$$+ : (\Sigma \to \mathbb{N}) \times (\Sigma \to \mathbb{N}) \to (\Sigma \to \mathbb{N}) \qquad g_1 + g_2 = \lambda\sigma.g_1(\sigma) + g_2(\sigma).$$

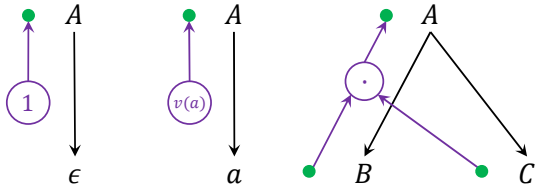
The following diagram depicts the form of an attribute grammar that computes the Parikh image of each word in $\mathcal{L}(G)$.



Each nonterminal propagates the Parikh image of the sub-word that it generates to its parent non-terminal. Nonterminals that derive the empty string ϵ pass up the $\mathbf{0}$ function; nonterminals that derive a single character a pass up $\mathbf{1}_a$; all other nonterminals pass up the sum of the attributes computed by their children. Parikh's Theorem implies that the set of attribute values that are computed via the above strategy is semi-linear and definable in polynomial time.

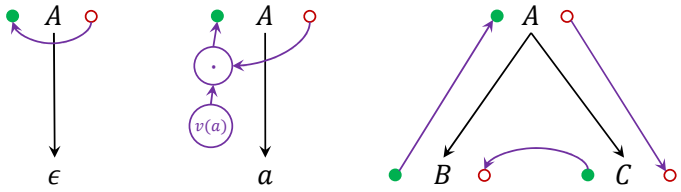
By Condition (4) of Definition 22, there exists an algorithm that, given any element $m \in M$, computes a decomposition $m = f \triangleleft c$. Let $\mathcal{A} : M \to F \times C$ be the function computing that decomposition.

Consider the following bottom-up strategy for computing $v^*(w)$, for all $w \in \mathcal{L}(G)$:



To characterize the values computed for this language via Parikh's Theorem, we apply a series of equivalence-preserving transformations to the underlying grammar.

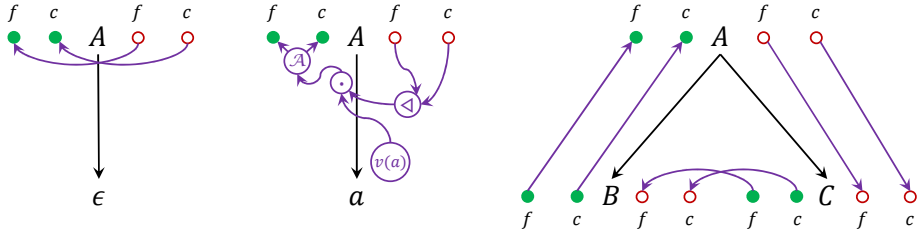
First, we change the order of evaluation to use to right-to-left threading:



(7.1)

(The initial attribute of the ROOT nonterminal would be assigned 1.) This transformation maintains equivalence by associativity of the monoid operation "$\cdot$".

We now take advantage of the properties of the almost-commuting monoid M. First, because all inherited and synthesized attribute values are elements of M, by Condition (4) of Definition 22, we know that each attribute value $m \in M$ can be decomposed into $(f, c) = \mathcal{A}(m)$, where (i) $f \in F \subseteq M$, (ii) F is a finite subset of M, (iii) $c \in$ commutative monoid C, and (iv) $m = f \triangleleft c$.



(The f and c initial attributes of the ROOT nonterminal would be assigned 1 and 0, respectively.)

We now perform four rewrites, shown below, of the attribute-definition functions in the production $A \to a$. (Rewritten elements are shown as dashed blue lines.)

- To obtain (i), we use Condition (2) of Definition 22: $v(a) \cdot (f \triangleleft c) = (v(a) \cdot f) \triangleleft c$.
- To obtain (ii), we use the identity

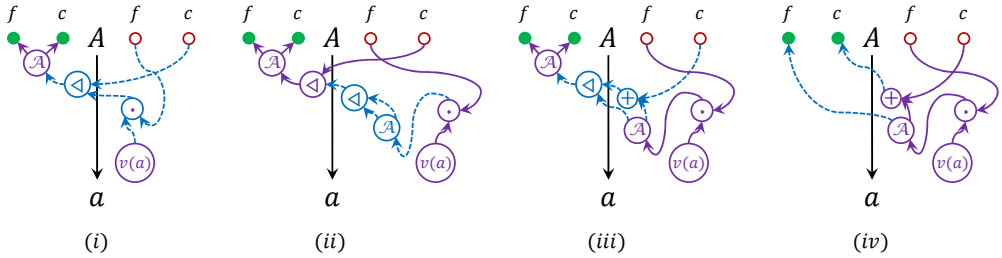
$$\text{for all } m, m = \text{let } \langle f_1, c_1 \rangle = \mathcal{A}(m) \text{ in } f_1 \triangleleft c_1.$$

- To obtain (iii), we use Condition (3) of Definition 22 in the form $(f_1 \triangleleft c_1) \triangleleft c = f_1 \triangleleft (c_1 + c)$.

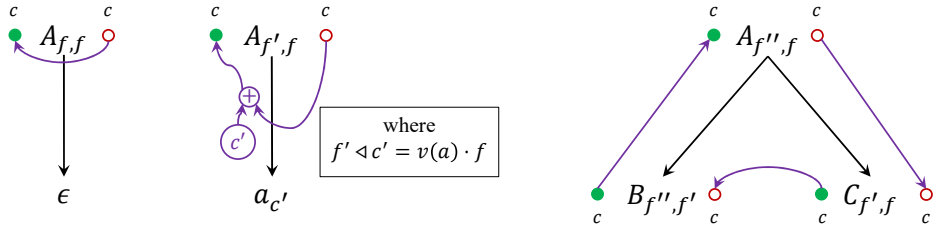
- To obtain (iv), we observe that because

$$\text{for all } f_1 \text{ and } c_1, (f_1 \triangleleft c_1) = \text{ let } \langle f, c\rangle = \mathcal{A}(f_1 \triangleleft c_1) \text{ in } (f \triangleleft c)$$

we can pass through $\langle f_1, c_1\rangle$ in place of $\mathcal{A}(f_1 \triangleleft c_1)$.



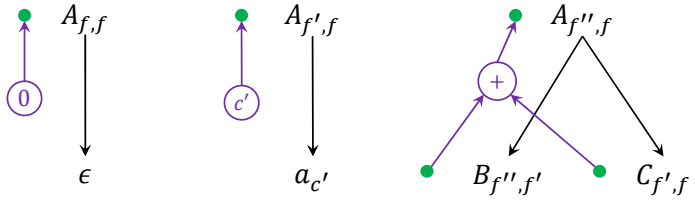
The next grammar transformation involves labeling the nonterminals with values of inherited and synthesized f attributes. Because F is finite, this transformation incurs only a constant-factor blow-up in the number of productions.



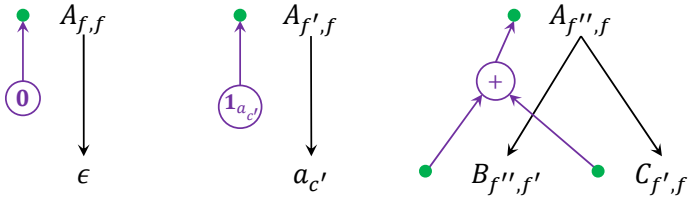
(7.2)

This step also expands the alphabet Σ, but again the expansion is only by at most a constant factor. In particular, every production of the form $A \to a$ becomes $A_{f,f'} \to a_{c'}$, where (i) $f \in F$ is a possible value of the inherited attribute f of the original left-hand-side nonterminal A, (ii) $f' \triangleleft c' = v(a) \cdot f$, and (iii) $a_{c'}$ is a new terminal symbol. Note that $v(a) \in M$ is a fixed known value associated with alphabet symbol a. Condition (4) of Definition 22 ensures that $v(a) \cdot f$ can be decomposed into $f' \triangleleft c' = v(a) \cdot f$, and thus, for a given terminal symbol a and value $f \in F$, $c' \in C$ is also a fixed known value.

The attribute grammar above accumulates a value in C via right-to-left threading. Because the operator $+$ of the commutative monoid C is associative, we can rewrite the attribute grammar to use a bottom-up accumulation pattern:



Finally, we transform the grammar to the form for computing the Parikh image of a word:



Because $+ : C \times C \to C$ is commutative, we can obtain a value in C of the Parikh image of a given word w. For instance, if $a_{c'}$ has a count of 3 in the Parikh image of w, then the contribution to the value of w in C is $c' + c' + c'$. The value of w is the sum of the contributions of each terminal symbol in w.

We can relate the values of Parikh images obtained in this way to the values of the words in the original grammar as follows. Consider the Parikh images of languages of the form $\mathcal{L}(\mathrm{ROOT}_{f,1})$. For a given word $w \in \mathcal{L}(\mathrm{ROOT}_{f,1})$, let $c_w \in C$ be the value of w computed from the Parikh image of w as discussed above. Let $\widehat{w}$ be the word in the original alphabet Σ obtained by dropping all c labels from the terminal-symbols in w. Then the $v^*(\widehat{w}) = f \triangleleft c_w$.

7.4.2 Formal Proofs

This subsection presents formal proofs of Theorems 16 and 17.

Theorem 16. Let $\langle M, \cdot, 1\rangle$ be an almost-commuting monoid, let Σ be a finite alphabet, and let $\mathcal{L}(G) \subseteq \Sigma^*$ be a context-free language. Let $v^* : \Sigma^* \to M$ be a homomorphism. Then, the set

$$\{v^*(w) \mid w \in \mathcal{L}(G)\}$$

is semi-linear and can be weakly defined in time $O(|G||F|^3)$ where $F \subseteq M$ is the finite subset of M.

Proof. Let finite subset $F \subseteq M$, commutative monoid $\langle C, +, 0\rangle$, and action $\triangleleft : M \times C \to M$ be the objects witnessing that $\langle M, \cdot, 1\rangle$ is almost-commuting.

By Condition (4) of Definition 22, there exists an algorithm that, given any element $m \in M$, computes a decomposition $m = f \triangleleft c$. Let $\mathcal{A} : M \to F \times C$ be the function computing that decomposition.

Our strategy is to construct a grammar G' such that $\{v^*(w) \mid w \in \mathcal{L}(G)\}$ can be calculated from the Parikh image of $\mathcal{L}(G')$. This strategy is enabled by the observation that the homomorphism $v^* : \Sigma^* \to M$ factors as $v^*(w) = \sigma(\phi(w, 1))$, where $\sigma : F \times C^* \to M$ is defined by

$$\sigma(f, c_1 \ldots c_n) \triangleq f \triangleleft (c_1 + \cdots + c_n),$$

Consider the function $\phi : \Sigma^* \times F \to F \times C^*$, in which the aim of $\phi(w, f)$ is to calculate a representation of $v^*(w) \cdot f$ by a "right fold" (analogous to the "right-to-left" evaluation strategy depicted in the set of diagrams labeled (7.1) in §7.4.1). Formally, ϕ is defined by

$$\begin{aligned}
\phi(\epsilon, f) &\triangleq \langle f, \epsilon\rangle \\
\phi(wa, f) &\triangleq \langle f'', tc\rangle \text{ where } \langle f'', t\rangle = \phi(w, f') \text{ and } \langle f', c\rangle = \mathcal{A}(v(a) \cdot f)
\end{aligned}$$

We show that for all $f \in F$, and all $w \in \Sigma^*$, we have $v^*(w) \cdot f = \sigma(\phi(w, f))$ by induction on w. For the base case we have

$$v^*(\epsilon) \cdot f = 1 \cdot f = f = \sigma(f, \epsilon) = \sigma(\phi(\epsilon, f)) \ .$$

For the induction step, we have

$$\begin{aligned}
v^*(wa) \cdot f &= v^*(w) \cdot v(a) \cdot f && \text{Homomorphism} \\
&= v^*(w) \cdot (f' \triangleleft c) && \text{where } \langle f', c\rangle = \mathcal{A}(v(a) \cdot f) \\
&= (v^*(w) \cdot f') \triangleleft c && \text{Definition 22(2) (Reassociativity)} \\
&= (\sigma(\phi(w, f'))) \triangleleft c && \text{Induction hypothesis} \\
&= (\sigma(f'', t_1 \dots t_n)) \triangleleft c && \text{where } \langle f'', t_1 \dots t_n\rangle = \phi(w, f') \\
&= (f'' \triangleleft (t_1 + \cdots + t_n)) \triangleleft c && \text{Definition of } \sigma \\
&= f'' \triangleleft (t_1 + \cdots + t_n + c) && \text{Definition 22(3) (Action)} \\
&= \sigma(f'', t_1 \dots t_n c) && \text{Definition of } \sigma \\
&= \sigma(\phi(wa, f)) && \text{Definition of } \phi
\end{aligned}$$

Our next task is to define a grammar G' that recognizes $\{\phi(w, 1) \mid w \in \mathcal{L}(G)\}$. Although formally $\phi(w, 1)$ belongs to $F \times C^*$, it may be thought of as a word over the alphabet $F \cup \hat{C}$, where

$$\hat{C} \triangleq \{c \in C \mid \exists s \in \Sigma, f \in F, \mathcal{A}(v(s) \cdot f) = \langle f', c\rangle\}$$

is a finite subset of C (the fact that $\sigma(w, f)$ belongs to $(F \times \hat{C})^*$ can be seen by inspection of the definition of σ).

Given a context-free grammar $G = \langle N, \Sigma, R, S \rangle$ in Chomsky Normal Form, we construct the grammar G' as follows:

$$
\begin{aligned}
G' &= \left\langle N', F \cup \hat{C}, R', S^* \right\rangle \\
N' &= \{A_{f',f} : A \in N, f, f' \in F\} \cup \{S^*\} \\
R' &= \{A_{f'',f} \to B_{f'',f'}\ C_{f',f} : A \to B\ C \in R, f, f', f'' \in F\} \\
&\quad \cup \left\{ A_{f',f} \to c : A \to a \in R, f, f' \in F, c \in \hat{C}, f' \triangleleft c = v(a) \cdot f \right\} \\
&\quad \cup \{A_{f,f} \to \epsilon : A \to \epsilon, f \in F\} \\
&\quad \cup \{S^* \to f\ S_{f,1} : f \in F\}
\end{aligned}
$$

The definition of the new production rules R' mirrors the set of diagrams labeled (7.2) in §7.4.1. In the last line, the first f on the right-hand side of $S^* \to f\ S_{f,1}$ is a "terminal-symbol tag" indicating that the F value for each word derived from $S_{f,1}$ is f. (The diagrams in (7.2) do not depict the productions of the form $S^* \to f\ S_{f,1}$.)

Then we have the following property:

$$
\{v^*(w) \mid w \in \mathcal{L}(G)\} = \left\{ f \triangleleft \left(\sum_{c \in \hat{C}} \phi(c) c \right) \;\middle|\; \phi \in \mathit{Parikh}(\mathcal{L}(G')), \phi(f) = 1 \right\} \tag{7.3}
$$

Here, $\phi(f) = 1$ and $f \triangleleft (\ldots)$ serve to (i) "read" the terminal-symbol tag f, and (ii) incorporate f into the value computed for a word, respectively.

We may thereby use the formula $\psi_{G'}$ representing the Parikh image of G' defined in Chapter 2 to define the following formula defining our subset of interest. The free variables of $\psi_{G'}$ correspond to the terminals of the grammar G', which we write as $Y = \{y_i \mid 1 \leq i \leq |F|\}$ and $Z = \left\{ z_c \mid c \in \hat{C} \right\}$. Then, the formula that weakly defines (Definition 24) $\{v^*(w) \mid w \in \mathcal{L}(G)\}$

is:

$$\exists Y, Z. \left(\psi_{G'}(Y, Z) \wedge \left(\bigwedge_{i=1}^{|F|} y_i = 1 \Leftrightarrow x_0 = i \right) \wedge (x_1, \ldots, x_n) = \sum_{c \in \hat{C}} z_c \cdot c \right).$$

□

Theorem 17. Let T be a queryable ACTS defined over a finite alphabet Σ, and let $\mathcal{L}(G) \subseteq \Sigma^*$ be a context-free language. Then there is a decision procedure to query the $\mathcal{L}(G)$-reachability relation of T for membership.

Proof. Given any states ρ, ρ', we can compute a formula that strongly defines the set $\{\rightarrow \mid \rho \rightarrow \rho'\}$ because T is queryable, conjoin this formula with that weakly defining $\{\rightarrow \mid \rightarrow = \rightarrow_{s_1} \cdot \ldots \cdot \rightarrow_{s_n}, s_1 \ldots s_n \in \mathcal{L}(G)\}$ via Theorem 16, and test for satisfiability. Because satisfiability of LIA formulas is decidable, this procedure constitutes a decision procedure for checking membership of $\langle \rho, \rho' \rangle$ in the $\mathcal{L}(G)$-reachability relation of T. □

We have that similar theorems hold for dual ACM and dual ACTS.

Theorem 18. Let $\langle M, \cdot, 1 \rangle$ be a dual almost-commuting monoid, let Σ be a finite alphabet, and let $\mathcal{L}(G) \subseteq \Sigma^*$ be a context-free language. Let $v : \Sigma \rightarrow M$ be a valuation function and let $v^* : \Sigma^* \rightarrow M$ be that valuation function extended to monoid homomorphism. Then, the set:

$$\{v^*(w) \mid w \in \mathcal{L}(G)\}$$

is semi-linear and definable in polynomial time.

Proof. This theorem is proven analogously to Theorem 16, except with the relevant subset of the commutative monoid $\hat{C}$ and the production rules R' of the blown up grammar G'

defined as:

$$\begin{aligned}
\hat{C} &= \{c \in C \mid s \in \Sigma, f, f' \in F, f' \triangleleft c = f \cdot v(s)\} \\
R' &= \{A_{f,f''} \to B_{f,f'}\ C_{f',f''} \mid A \to B\ C \in R, f, f', f'' \in F\} \\
&\quad \cup \left\{A_{f,f'} \to c \mid A \to a \in R, f, f' \in F, c \in \hat{C}, f' \triangleleft c = f \cdot v(a)\right\} \\
&\quad \cup \{A_{f,f} \to \epsilon \mid A \to \epsilon, f \in F\} \\
&\quad \cup \{S^* \to f\ S_{1,f} \mid f \in F\}
\end{aligned}$$

Diagrammatically, this construction follows the same reasoning as that shown in §7.4.1 with "left-to-right" threading in place of "right-to-left" threading.

□

Theorem 19. Let T be a queryable dual ACTS defined over finite alphabet Σ and let $\mathcal{L}(G) \subseteq \Sigma^*$ be a context-free language. Then, there is a decision procedure to query the $\mathcal{L}(G)$-reachability relation of T for membership.

Proof. Similar to the proof of Theorem 17. □

7.5 Closure Properties of ACTS

In this section, we demonstrate that almost-commuting transition systems (ACTS) enjoy robust closure properties. These properties ensure that complex systems constructed from smaller, analyzable components remain within the ACTS class, preserving tractability of CFL-reachability problems. Specifically, we (i) show that ACTS are closed under direct product and (ii) generalize the framework to encompass a recursive class of systems we call generalized ACTS. We also provide a concrete example of a transition system that lies in this broader class.

7.5.1 Direct Product

We begin with the result that the direct product of two almost-commuting monoids is itself an almost-commuting monoid.

Theorem 20. Let $\langle M, \cdot, 1\rangle$ and $\langle M', \cdot', 1'\rangle$ be almost-commuting monoids. Then, their direct product $\langle M \times M', \otimes, \langle 1, 1'\rangle\rangle$ is an almost-commuting monoid, in which:

$$\langle m_1, m_1'\rangle \otimes \langle m_2, m_2'\rangle = \langle m_1 \cdot m_2, m_1' \cdot' m_2'\rangle$$

Proof. By the definition of ACMs, there exist finite subsets $F \subseteq M$ and $F' \subseteq M'$, commutative monoids $\langle C, +, 0\rangle$ and $\langle C', +', 0'\rangle$, and actions $\triangleleft : M \times C \to M$ and $\triangleleft' : M' \times C' \to M'$ fulfilling the conditions of Definition 22 for $\langle M, \cdot, 1\rangle$ and $\langle M', \cdot', 1'\rangle$, respectively.

Then, finite subset $F \times F' \subseteq M \times M'$, commutative monoid $\langle C \times C', \oplus, \langle 0, 0'\rangle\rangle$ in which $\langle c_1, c_1'\rangle \oplus \langle c_2, c_2'\rangle = \langle c_1 + c_2, c_1' +' c_2'\rangle$, and action $\blacktriangleleft$: $(M \times M') \times (C \times C') \to (M \times M')$ defined as $\langle m, m'\rangle \blacktriangleleft \langle c, c'\rangle = \langle m \triangleleft c, m' \triangleleft' c'\rangle$ fulfill the conditions of Definition 22. Therefore, $\langle M \times M', \otimes, \langle 1, 1'\rangle\rangle$ is an almost-commuting monoid. □

This result implies that the ACTS framework is compositional: multiple ACTS components can be combined in parallel while remaining analyzable. The following example shows this property concretely.

Example 7.5.1. Let $T_1 = \left\langle \mathbb{Z}, \left\{ \xrightarrow{1}_s : s \in \Sigma \right\} \right\rangle$ and $T_2 = \left\langle \mathbb{Z}, \left\{ \xrightarrow{2}_s : s \in \Sigma \right\} \right\rangle$ be Natural Offset Max-Plus Linear Systems (from §7.3.2) over finite alphabet Σ. By Theorem 14, we have that both of these transition systems are ACTS.

Let $T_3 = \left\langle \mathbb{Z}^2, \left\{ \xrightarrow{3}_s : s \in \Sigma \right\} \right\rangle$ be a transition system in which for all $\rho_1, \rho_2, \rho_1', \rho_2' \in \mathbb{Z}$ and all $s \in \Sigma$ we have:

$$\langle \rho_1, \rho_2 \rangle \xrightarrow{3}_s \langle \rho_1', \rho_2' \rangle \Leftrightarrow \left(\rho_1 \xrightarrow{1}_s \rho_1' \text{ and } \rho_2 \xrightarrow{2}_s \rho_2' \right)$$

Theorem 20 implies that T_3 too is an ACTS, because its underlying monoid of transition relations is the direct product of that of T_1 and T_2. This theorem thus shows that any ACTS can be generalized to an arbitrary dimensional state and that multiple ACTS can be joined in parallel while retaining the same algebraic structure.

7.5.2 Generalized ACTS

Next, we show that our construction can be generalized to a hierarchy of algebraic structures beyond almost-commuting monoids. This generalization allows the "commutative component" of an ACM to itself be a (generalized) ACM.

Definition 32. A commutative integer monoid is a rank-0 generalized almost-commuting monoid. A monoid $\langle M, \cdot, 1 \rangle$ is a rank-n generalized almost-commuting monoid if there exists a finite subset F of M, a rank-$(n-1)$ generalized almost-commuting monoid $\langle C, +, 0 \rangle$, and a function $\triangleleft : M \times C \to M$ that follow the conditions of Definition 22 or Definition 30.

Definition 33. A generalized almost-commuting transition system is defined analogously to an ACTS with "generalized ACM" in place of "ACM."

The context-free-language-reachability relation remains semi-linear and efficiently definable over generalized ACTS, just as it does with basic ACTS.

Theorem 21. Let $\langle M, \cdot, 1 \rangle$ be a generalized almost-commuting monoid, let Σ be a finite alphabet, and let $\mathcal{L}(G) \subseteq \Sigma^*$ be a context-free language. Let $v : \Sigma \to M$ be a valuation function and let $v^* : \Sigma^* \to M$ be that valuation function extended to a monoid homomorphism.

Then, the set

$$\{v^*(w) \mid w \in \mathcal{L}(G)\}$$

is semi-linear and weakly definable in polynomial time.

Proof. We can define our construction recursively in terms of the construction in the proof of Theorem 16.

If the underlying monoid $\langle C, +, 0\rangle$ is commutative, then $\langle M, \cdot, 1\rangle$ is in fact an almost-commuting monoid and we can directly apply the construction of Theorem 16.

If the underlying monoid $\langle C, +, 0\rangle$ is a generalized almost-commuting monoid, then this theorem states that we can compute a formula Ψ that weakly defines the $\mathcal{L}(G)$ reachability of C. We then apply the construction of Theorem 16 replacing Equation (7.3) with:

$$\{v^*(w) \mid w \in \mathcal{L}(G)\} = \left\{ f \triangleleft \left(\sum_{c \in \hat{C}} \phi(c)c \right) \mid \phi \models \Psi, \phi(f) = 1 \right\}.$$

□

Theorem 22. Let T be a queryable generalized ACTS defined over finite alphabet Σ and let $\mathcal{L}(G) \subseteq \Sigma^*$ be a context-free language. Then, there is a decision procedure to query the $\mathcal{L}(G)$-reachability relation of T for membership.

Proof. Similar to Theorem 17 using Theorem 21. □

Example 7.5.2. We now present a concrete example of a system that is not a basic ACTS but is a generalized ACTS. This example combines features from both FMAVAS and parity-guarded systems.

Definition 34. A *parity-guarded rotating system* over a finite alphabet Σ is a labeled transition system $\langle \mathbb{Z}, \{\rightarrow_s |\ s \in \Sigma\}\rangle$ equipped with functions $a, c : \Sigma \to \{-1, 0, 1\}$ and $b, d : \Sigma \to \mathbb{Z}$ such that for each label $s \in \Sigma$, the transition relation $\rightarrow_s$ satisfies:

$$x \rightarrow_s x' \Leftrightarrow x' = \text{if } x \text{ is even then } a(s)x + b(s) \text{ else } c(s)x + d(s)$$

Theorem 23. A parity-guarded rotating system is a generalized ACTS.

Proof. Let $\langle \mathbb{Z}, \{\rightarrow_s |\ s \in \Sigma\}\rangle$ be a parity-guarded rotating system. Let M denote the closure of $\{\rightarrow_s |\ s \in \Sigma\}$ under relational composition $\cdot$, and let $\mathbb{I}$ be the identity relation on $\mathbb{Z}$. We will show that $\langle M, \cdot, \mathbb{I}\rangle$ is a generalized ACM.

Consider the monoid $\langle M' = \{-1, 0, 1\} \times 2\mathbb{Z}, \otimes, \langle 1, 0\rangle\rangle$ in which:

$$\langle a, b\rangle \otimes \langle a', b'\rangle = \langle a * a', a' * b + b'\rangle$$

Let $F' \subseteq M' = \{\langle -1, 0\rangle, \langle 0, 0\rangle, \langle 1, 0\rangle\}$, let $C = \langle 2\mathbb{Z}, +, 0\rangle$, and define action $\triangleleft' : M' \times 2\mathbb{Z} \to M'$ as $\langle a, b\rangle \triangleleft' c = \langle a, b + c\rangle$. These definitions fulfill the conditions of Definition 22, making $\langle M', \otimes, \langle 1, 0\rangle\rangle$ an almost-commuting monoid.

By Theorem 20, we have that $\langle M' \times M', \oplus, \langle \langle 1, 0\rangle, \langle 1, 0\rangle\rangle\rangle$ is a generalized almost-commuting monoid, where we define $\langle m_1, m_2\rangle \oplus \langle m_1', m_2'\rangle = \langle m_1 \otimes m_1', m_2 \otimes m_2'\rangle$.

Now, define finite subset $F \subseteq M$ and action $\triangleleft : M \times (M' \times M') \to M$ to be:

$$F = \left\{ \begin{array}{c} f_{00} = \{\langle x, x'\rangle \mid x' = \text{if } x \text{ is even then } x \text{ else } x\}, \\ f_{10} = \{\langle x, x'\rangle \mid x' = \text{if } x \text{ is even then } x + 1 \text{ else } x\}, \\ f_{01} = \{\langle x, x'\rangle \mid x' = \text{if } x \text{ is even then } x \text{ else } x + 1\}, \\ f_{11} = \{\langle x, x'\rangle \mid x' = \text{if } x \text{ is even then } x + 1 \text{ else } x + 1\} \end{array} \right\}$$

$$m \triangleleft \langle \langle a, b \rangle \langle c, d \rangle \rangle = \{x, x'' \mid \langle x, x' \rangle \in m, x'' = \text{if } x \text{ is even then } a * x' + b \text{ else } c * x' + d\}$$

Every generator $\rightarrow_s$ can be expressed as $f \triangleleft c$ for some $f \in F$ and $c \in M' \times M'$. Composition $f \triangleleft \langle \langle a, b \rangle, \langle c, d \rangle \rangle \cdot f' \triangleleft \langle \langle a', b' \rangle, \langle c', d' \rangle \rangle$ follows the same reasoning as that in the proof of Theorem 15. Therefore, $\langle M, \cdot, \mathbb{I} \rangle$ is a generalized almost-commuting monoid.

□

7.6 Discussion

This chapter presented a framework for solving context-free-language reachability problems for almost-commuting transition systems. The techniques used to compute the reachability relations of VASR, LVASR, and SVASR can all be reduced to the generalized technique presented here. While this technique is more general, it is less efficient - using the generalized technique for ACTS would produce exponentially larger formulas for VASR and LVASR-based summarization and would produce doubly exponentially larger formulas for SVASR-based summarization.

This result opens a rich new frontier of candidates for monotone inter-procedural program analysis. Given a technique to compute a reflection of a program in a queryable class of almost-commuting transition systems, the framework presented here would immediately yield a monotone procedure summarization technique.

Chapter 8

Related Work

The three papers forming the contributions of this thesis share two broad technical themes: (i) the reachability theory of Vector Addition Systems and their extensions, and (ii) inter-procedural program analysis, including loop summarization, monotone invariant generation, and resource-bound analysis. This chapter surveys the prior work most relevant across all three papers.

8.1 Reachability of Vector Addition Systems and Extensions

Vector Addition Systems (VAS) are a classical model of computation which operate over a vector of counters. Karp and Miller [34] originally introduced them in the context of identifying models of computation whose properties could be effectively verified. They defined VAS to operate over natural numbers and established initial results regarding the reachability set of these systems. Today, we know that the reachability problem for natural-valued VAS is decidable [40] but non-elementary [39]. VAS are highly related to another modeling language: Petri nets [47]. A Petri net consists of a finite set of places, each containing some number of

tokens; transitions remove tokens from some places and add tokens to others. Equivalence between VAS and Petri nets (which associates each dimension of the state of the VAS with a place in the Petri net) has been well-understood since the 70s [28]. VAS and Petri nets have proven to be practical modeling languages for concurrency [5, 25] and distributed systems [53]. They have even found application in biology and business management [1].

The non-elementary lower bound on reachability motivated the study of integer-valued VAS by Haase and Halfon [27]. The L-reachability relation of integer-valued VAS can be straightforwardly encoded into a LIA formula via Parikh's Theorem [46] when L is Σ^*, a regular language, or a context-free language. Haase and Halfon went further by considering VAS with resets, and showed that the L-reachability relation for this class was computable when L is Σ^* and a regular language. They (with Chistikov) [12] further extended this result to Petri-net languages, which are strictly between regular and context-free languages. However, their approach is not obviously extensible to context-free languages.

Their result was based on an extension of Parikh's Theorem [46], and is based on representing a particular counting abstraction, the generalized Parikh image, of L using a LIA formula. The generalized Parikh image of a word consists of a permutation over characters in a monitored sub-alphabet describing the order of final occurrences with respect to left-to-right order and the Parikh images of the sub-words in between each final occurrence. Haase and Halfon extended Seidl et al.'s method [60] for computing Parikh images based on a correspondence between Parikh images and *connected flows* through an automaton recognizing the language. Haase and Halfon [27] modified this construction to compute generalized Parikh images by encoding multiple connected sub-flows representing consecutive sub-words and symbolically encoding permutations of final occurrences in LIA. However, it is not clear how to extend this construction to compute the generalized Parikh image of a context-free language from its representation as a pushdown automaton or a context-free grammar; it is difficult to represent the configurations of a pushdown machine between flows

in linear arithmetic and the generalization of Seidl et al. [60] to grammars breaks the desired correspondence between sub-flows and consecutive sub-words.

The reachability relation encodings in this thesis, and in particular our encoding of the context-free reachability relation of VASR, employed *abstract trajectories* as an alternative to generalized Parikh images. We show that the problem of computing abstract trajectories can essentially be reduced to computing ordinary Parikh images by using standard language-theoretic operations (inverse homomorphism and intersection with a regular language). Our methods could in principle be used to compute generalized Parikh images, but the need to encode a permutation of the alphabet Σ yields an exponential-space reduction. Abstract trajectories allow us to side-step this blowup by identifying d arbitrary positions in a word rather than a permutation of the alphabet.

The context-free reachability of vector addition systems over the naturals was investigated in [39, 41, 24]. Whether reachability is decidable for this model is an open problem. Reachability problems for vector addition systems with resets over the naturals are undecidable [18].

The relaxation of deterministic transition to nondeterministic lossy systems was originally introduced by Abdulla and Jonsson [2], in the context of messages being dropped in communication protocols. Lossy transition systems relax the semantics to allow the state to spontaneously decrease at any point. Reachability of lossy variants corresponds to *coverability* of the original class.

Coverability of VAS has been widely studied. Chistikov et al [12] showed that coverability of integer-valued VASRs is decidable over Σ^*, regular languages, and Petri-net languages. Coverability is decidable for natural-valued VASRs over Σ^* [18] but is undecidable over context-free languages [59]. Our results for Lossy VASR amount to showing coverability for integer-valued VASRs over context-free languages.

Our extension to Semi-Linear VASR is inspired by a line of work [51, 42] that has investigated extending linear integer arithmetic to include a star operator, effectively computing reachability for Semi-Linear Vector Addition Systems, but not considering resets.

Blondin et al [8] investigated the extension of integer VAS to affine transformations beyond resets. Our work on ACTS extends this technique in two ways: (1) we "algebraize" the method, expressing the essence of the techniques as the laws of almost-commuting monoids, and (2) we extend the applicability from a model with states to one with a pushdown stack (i.e., *context-free* reachability rather than *regular* reachability).

8.2 Inter-procedural Program Analysis

Computing summaries that approximate the dynamics of recursive procedures is a classical problem in program analysis [13, 61]. The dominant approach is based on *iterative approximation*, which uses the limit of a Kleene iteration sequence as a summary. For abstract domains that fail the ascending chain condition, the limit can be over-approximated using widening; however, this results in a non-monotone analysis.

The classical iterative method for program analysis is monotone for abstract domains satisfying the ascending chain condition, such as affine relation analysis [35, 45] and the Houdini algorithm [23]. A recent line of work has designed monotone program analyses [62, 65, 66, 16] using *algebraic program analysis* [36]. This work is intra-procedural, and falls back on an iterative strategy to summarize recursive procedures.

Our work is closely related to Silverman et al's [62] loop summarization technique. Given a transition formula F representing the body of a loop, their method computes an (unlabeled) VASR abstraction of F, and then uses the reachability relation of the VASR (computed via [27]) to over-approximate the reflexive transitive closure of F. [62] also extend their loop summarization approach to VASR with states (equivalently, VASR restricted to a regular

language of trajectories). Using the algebraic program analysis framework [36], loop summarization can be extended to summarize (non-recursive) procedures by computing a regular expression representing all paths through the procedure and then re-interpreting each regular expression operator with a corresponding operation on transition formulas (and in particular, interpreting the Kleene $*$ operator using the aforementioned loop summarization algorithm).

Our work differs in several respects. Foremost, our approach can be applied to recursive procedures. Algebraic program analysis relies on the inductive structure of regular expressions to enable a simple "bottom-up" summarization strategy; in the presence of recursion, path languages can be context-free and so do not possess such an inductive structure. Our approach overcomes this barrier by directly extending VASR-based summarization to recursive procedures rather than relying upon algebraic program analysis. Our approach necessitated two technical innovations: (1) we must compute a "global" VASR abstraction for the whole procedure (rather than an independent "local" abstraction of each loop as in [62]), and (2) we must compute context-free reachability relations for VASR (rather than regular reachability as in [27]). Secondly, we extend the approach to the more expressive abstract domain of lossy VASR. Finally, our VASR abstraction algorithm computes a *best* abstraction for LIRA formulas, whereas [62]'s algorithm is only best for LRA formulas, since it relies upon the fact that LRA is a convex theory.

One framework for computing procedure summaries is to model the (abstract) meaning of a program as values drawn from a semiring, with the multiplication operation corresponding to sequential composition of actions along two program paths, and the addition operation to joining information across parallel program paths. In this setting, the problem of interest is to compute the sum of the weights of all inter-procedurally-valid paths from one point to another—that is, to compute $\sum_{\pi \in \mathcal{L}(G)} w(\pi)$, where $\mathcal{L}(G)$ is a (context-free) language of paths of interest, and $w(\pi)$ is the weight of path π obtained by multiplying (in order) the semiring value on each edge of π. There is no general method for computing this value, but there are

algorithms for some particular cases: (1) semirings in which there are no infinite ascending chains [10, 56], and (2) semirings in which the sequencing operation is commutative [20]. Our work on ACTS offers a new method for solving this problem for semirings of semi-linear sets (see [10, §3.4.4]) over an almost-commuting monoid. Alternatively, one may view this as computing $\{w(\pi) : \pi \in \mathcal{L}(G)\}$—the *set* of all weights of all paths belonging to a context-free language $\mathcal{L}(G)$—rather than $\sum_{\pi \in \mathcal{L}(G)} w(\pi)$—the *sum* of such weights.

Chapter 9

Conclusion

This thesis extends the frontier of robust program analysis to the inter-procedural setting. We focused upon monotonicity and locality as robustness properties that allow tool users to anticipate how code changes will affect analysis results without needing to understand the tool's internal algorithms. By framing program analysis as procedure summarization, we immediately obtained locality – each procedure's summary depends only on the code within its connected component, so changes elsewhere cannot affect it. Monotonicity is harder to achieve, and is the central technical concern of this thesis.

We identified a recipe for monotone procedure summarization with two key components. First, for a given abstract domain, we computed a reflection of the input program: a best abstraction of the program within that domain, in the sense that it universally simulates every other abstraction of the same class. Second, we precisely computed the reachability relation of this reflection over the context-free language of inter-procedurally-valid paths through the program. Together, these steps yield a monotone summarization technique: if program A is a refinement of program B, then a reflection R_A of A simulates any reflection R_B of B, so the reachability relation of R_A simulates that of R_B, and therefore the procedure summary for A logically entails that of B.

We instantiated this recipe with four abstract domains related to vector addition systems, a sub-Turing complete model of computation that operates over a set of independent counters. Within abstractions, each counter represented a linear term over the variables of the program. This meant that even though our abstractions operated over counters with simple updates, those updates could model complex numerical relationships between program variables. Our evaluation confirmed that invariants derived from our abstractions are practically useful; our tool exceeded state-of-the-art abstract interpreters in verification capability.

In each domain, our reflections were derived from the generator representation of the space of linear terms that our counters could possibly track. This means the counters of any other abstraction are expressible in terms of the reflection's counters, which is precisely what guarantees that the reflection universally simulates all other abstractions in its class. Our techniques for computing reflections used a divide-and-conquer approach: we computed a reflection of each individual operation, then repeatedly combined these into a whole-program reflection. The central challenge in each domain was identifying the right structured space of trackable terms over which reflections could be combined. We provided a category-theoretic formalization of sufficient conditions for this structured space, or base category, enabling the computation of reflections.

Domain	**Transition semantics**	**Combination Space**	**Chapter**
VAS	$x' = x + a$	Linear Space	3
VASR	$x' = rx + a$	Separated Space	4
LVASR	$x' \leq rx + a$	Ordered Separated Space	5
SVASR	$\mathbf{x}' = r * \mathbf{x} + o, \quad o \in S$	Separated Space	6

Our techniques for computing context-free reachability relations leveraged Parikh's Theorem, a result from formal language theory that commutativity collapses the structure of context-free languages onto that of regular languages. Although the abstract domains presented in this thesis (with the exception of VAS) are not commutative, we were successfully

able to use the universal commutative abstraction of context-free languages provided by Parikh's Theorem to compute their context-free reachability relation. Our key insight was that, in each of these domains, runs could be decomposed into a finite number of phases within which operations commute. We formalized this idea by algebraically characterizing almost-commuting transition systems, whose underlying mathematical structure allows this decomposition.

In summary, this thesis developed new techniques for monotone inter-procedural program analysis. Our implementation using Lossy VASRs exceeded the state-of-the-art for abstract-interpretation-based program analyzers while also providing a monotonicity guarantee. From these techniques, we extracted generalizable frameworks for this problem, facilitating future development of such techniques. Specifically, this thesis presented a category-theoretic divide-and-conquer framework for computing reflections and a characterization of the class of almost-commuting transition systems, whose context-free reachability relations are computable via Parikh's Theorem.

9.1 Future Work

Implementing SVASR Analysis with Approximations Chapter 6 developed the theoretical foundations for a monotone program analysis technique using Semi-Linear VASRs as an abstract domain, but an implementation was not achieved. The key remaining obstacle is that computing a generator representation of a semi-linear set is prohibitively expensive. Levatich et al. [42] developed a decision procedure for reasoning about an extension of LIA with a star operator which represents semi-linear sets efficiently by computing over- and under-approximations of the source formula. Further investigation is required to determine if such an approach could be made compatible with our reachability technique for SVASR.

Abstraction Techniques to Almost-Commuting Transition Systems A natural direction for future work is to identify new classes of queryable almost-commuting transition systems and developing techniques for computing reflections of program within those classes. Such techniques would immediately yield monotone program analysis techniques, as the classes would inherit our Parikh-based reachability construction.

Monotone Analysis over More Expressive Logics This thesis began from a program model which expressed program semantics as LIRA formulas. Doing so excluded many important program behaviors—most notably, operations interacting with heap-allocated memory. Extending the framework to more expressive logics would substantially broaden the class of programs amenable to monotone inter-procedural analysis. Doing so is algorithmically challenging, as several of our techniques are dependent on specialized subroutines developed for LIRA. One path forward would be to compute best LIRA abstractions of programs expressed in more expressive logics.

Bibliography

[1] Wil Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8:21–66, 02 1998.

[2] P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 160–170, 1993.

[3] Jiri Adamek, Horst Herrlich, and George E. Strecker. *Abstract and Concrete Categories : The Joy of Cats*. Wiley, New York, 1990.

[4] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electronic Notes in Theoretical Computer Science*, 267(1):3–16, 2010. Proceeding of the Second International Workshop on Numerical and Symbolic Abstract Domains: NSAD 2010.

[5] Thomas Ball, Sagar Chaki, and Sriram K. Rajamani. Parameterized verification of multithreaded software libraries. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, pages 158–173. Springer-Verlag, 2001.

[6] Dirk Beyer. Competition on software verification and witness validation: Sv-comp 2023. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 495–522, Cham, 2023. Springer Nature Switzerland.

[7] Black Duck. What could cause findings to disappear and re-appear on Coverity Connect on an unchanged code base? Black Duck Community Knowledge Base, 2023. Black Duck Community Knowledge Base, accessed April 2026.

[8] Michael Blondin, Christoph Haase, Filip Mazowiecki, and Mikhail Raskin. Affine extensions of integer vector addition systems with states. *Logical Methods in Computer Science (LMCS)*, 17(3), 2021.

[9] Bernard Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, University of Liège, Belgium, 1998.

[10] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 15-17, 2003*, pages 62–73. ACM, 2003.

[11] Shai Caspin, Nikhil Pimpalkhare, and Amit Levy. From rust till run: Extending memory safety from rust to cryptographic assembly. In *Proceedings of the 13th Workshop on Programming Languages and Operating Systems*, PLOS '25, page 108–117, New York, NY, USA, 2025. Association for Computing Machinery.

[12] Dmitry Chistikov, Christoph Haase, and Simon Halfon. Context-free commutative grammars with integer counters and resets. *Theoretical Computer Science*, 735:147–161, 2018. Reachability Problems 2014: Special Issue.

[13] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.

[14] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, June 1996.

[15] Patrick Cousot. Abstracting induction by extrapolation and interpolation. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 19–42, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[16] John Cyphert and Zachary Kincaid. Solvable polynomial ideals: The ideal reflection for program analysis. *Proc. ACM Program. Lang.*, 8(POPL), January 2024.

[17] Bart De Schutter and Ton van den Boom. Max-plus algebra and max-plus linear discrete event systems: An introduction. In *2008 9th International Workshop on Discrete Event Systems*, pages 36–42, 2008.

[18] C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, pages 103–115, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[19] Gidon Ernst. A complete approach to loop verification with invariants and summaries. *CoRR*, abs/2010.05812, 2020.

[20] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *J. ACM*, 57(6):33:1–33:47, 2010.

[21] Azadeh Farzan and Zachary Kincaid. Compositional recurrence analysis. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, page 57–64, Austin, Texas, 2015. FMCAD Inc.

[22] Alain Finkel and Jérôme Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In Manindra Agrawal and Anil Seth, editors, *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science, 22nd Conference Kanpur, India, December 12-14, 2002, Proceedings*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

[23] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, page 500–517, Berlin, Heidelberg, 2001. Springer-Verlag.

[24] Moses Ganardi, Rupak Majumdar, and Georg Zetzsche. The complexity of bidirected reachability in valence systems. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, July 1992.

[26] Seymour Ginsburg and Edwin H. Spanier. Bounded algol-like languages. *Transactions of the American Mathematical Society*, 113(2):333–368, 1964.

[27] Christoph Haase and Simon Halfon. Integer vector addition systems with states. In Joël Ouaknine, Igor Potapov, and James Worrell, editors, *Reachability Problems*, pages 112–124, Cham, 2014. Springer International Publishing.

[28] M. Hack. Decidability questions for petri nets. Technical report, USA, 1976.

[29] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 36–52, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[30] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson/Addison Wesley, Boston, third edition, 2007.

[31] Ankur Jain and G Ann Campbell. Unexplained new findings shown in unchanged code. `https://community.sonarsource.com/t/unexplained-new-findings-shown-in-unchanged-code/109154`, 2024. SonarSource Community Forum, accessed April 2026.

[32] Jim Jastrzebski. 4 reasons scan results may differ over time. `https://www.veracode.com/blog/4-reasons-scan-results-may-differ-over-time/`, 2022. Veracode Blog, accessed April 2026.

[33] JetBrains. Qodana calculates 0% fresh coverage for annotation-only changes in PHP classes. `https://youtrack.jetbrains.com/issue/QD-13213`, 2024. JetBrains YouTrack Issue QD-13213, accessed April 2026.

[34] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[35] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6(2):133–151, jun 1976.

[36] Zachary Kincaid, Thomas Reps, and John Cyphert. Algebraic program analysis. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 46–83, Cham, 2021. Springer International Publishing.

[37] Zachary Kincaid and Shaowei Zhu. A categorical basis for robust program analysis, 2026.

[38] Nicolas Koh. Consequence-finding in theories of arithmetic, 2026.

[39] Ranko Lazic. The reachability problem for vector addition systems with a stack is not elementary. *CoRR*, abs/1310.1767, 2013.

[40] Jérôme Leroux. Vector addition system reachability problem: a short self-contained proof. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 307–316, New York, NY, USA, 2011. Association for Computing Machinery.

[41] Jérôme Leroux, Grégoire Sutre, and Patrick Totzke. On the coverability problem for pushdown vector addition systems in one dimension. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming*, pages 324–336, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[42] Maxwell Levatich, Nikolaj Bjørner, Ruzica Piskac, and Sharon Shoham. *Solving* LIA* *Using Approximations*, pages 360–378. Springer International Publishing, 01 2020.

[43] Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., USA, 1967.

[44] Anders Møller and Michael I. Schwartzbach. Static program analysis. In *Encyclopedia of Cryptography and Security*, 2011.

[45] Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 330–341, New York, NY, USA, 2004. Association for Computing Machinery.

[46] Rohit J. Parikh. On context-free languages. *J. ACM*, 13(4):570–581, oct 1966.

[47] C. A. Petri. Kommunikation mit automaten. 1962.

[48] Nikhil Pimpalkhare and Zachary Kincaid. Monotone procedure summarization via vector addition systems and inductive potentials. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024.

[49] Nikhil Pimpalkhare and Zachary Kincaid. Semi-linear vasr for over-approximate semi-linear transition system reachability. In Laura Kovács and Ana Sokolova, editors, *Reachability Problems*, pages 154–166, Cham, 2024. Springer Nature Switzerland.

[50] Nikhil Pimpalkhare, Zachary Kincaid, and Thomas Reps. Context-free-language reachability for almost-commuting transition systems. *Proc. ACM Program. Lang.*, 10(POPL), January 2026.

[51] Ruzica Piskac and Viktor Kuncak. Linear arithmetic with stars. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 268–280, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[52] Raluca Ada Popa and Four Flynn. Introducing CodeMender: An AI agent for code security. Blog Post / Announcement, 2025. Accessed: 2026-04-30.

[53] Krithi Ramamritham, Robert M Keller, and Robert M Keller. Distributed software system design representation using modified petri nets. *IEEE Transactions on Software Engineering*, SE-9:733–745, 1983.

[54] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery.

[55] Thomas Reps, Mooly Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 252–266, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[56] Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[57] H. Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[58] Emily Riehl. *Category Theory in Context*. Aurora: Dover Modern Math Originals. Dover Publications, 2017.

[59] Sylvain Schmitz and Georg Zetzsche. Coverability is undecidable in one-dimensional pushdown vector addition systems with resets. In Emmanuel Filiot, Raphaël Jungers, and Igor Potapov, editors, *Reachability Problems*, pages 193–201, Cham, 2019. Springer International Publishing.

[60] Helmut Seidl, Thomas Schwentick, Anca Muscholl, and Peter Habermehl. Counting in trees for free. In *International Colloquium on Automata, Languages and Programming*, 2004.

[61] M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY, 1978.

[62] Jake Silverman and Zachary Kincaid. Loop summarization with rational vector addition systems. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 97–115, Cham, 2019. Springer International Publishing.

[63] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. On the complexity of equational horn clauses. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20*, pages 337–352, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[64] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 391–402, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Shaowei Zhu and Zachary Kincaid. Reflections on termination of linear loops. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, pages 51–74, Cham, 2021. Springer International Publishing.

[66] Shaowei Zhu and Zachary Kincaid. Termination analysis without the tears. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1296–1311, New York, NY, USA, 2021. Association for Computing Machinery.