Compositional Recurrence Analysis Revisited [*]

Zachary Kincaid
Princeton Univ., USA
zkincaid@cs.princeton.edu

Jason Breck
Ashkan Forouhi Boroujeni
Univ. of Wisconsin, USA
{jbreck,ashkanfb}@cs.wisc.edu

Thomas Reps
Univ. of Wisconsin and
GrammaTech, Inc., USA
reps@cs.wisc.edu

Abstract

Compositional recurrence analysis (CRA) is a static-analysis method based on a combination of symbolic analysis and abstract interpretation. This paper addresses the problem of creating a context-sensitive interprocedural version of CRA that handles recursive procedures. The problem is non-trivial because there is an "impedance mismatch" between CRA, which relies on analysis techniques based on *regular languages* (i.e., Tarjan's path-expression method), and the *context-free-language underpinnings* of context-sensitive analysis.

We show how to address this impedance mismatch by augmenting the CRA abstract domain with additional operations. We call the resulting algorithm Interprocedural CRA (ICRA). Our experiments with ICRA show that it has broad overall strength compared with several state-of-the-art software model checkers.

CCS Concepts **•Theory of computation → Program analysis**; **•Software and its engineering → Automated static analysis**

Keywords Invariant generation, Resource bounds

1. Introduction

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

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

Compositional recurrence analysis (CRA) [14] is a static-analysis method based on a combination of symbolic analysis and abstract interpretation. It performs abstract interpretation using an abstract domain of transition formulas, and thus works with quite precise models of a program's actions. It conservatively explores all possible behaviors of a loop by overapproximating the transitive closure of its body: the formula for the loop body is converted into a system of linear recurrences, which are then solved to create a transition formula for the loop that involves polynomial constraints.

Abstraction refinement is a widely used technique in static analysis that tailors an abstraction to a property of interest. Abstraction refinement enables analyzers to prove complicated properties, but can also cause them to fail to terminate. An interesting feature of CRA is that it can verify complicated properties *without* using abstraction refinement. CRA is guaranteed to terminate, and in practice it is generally faster than tools based on abstraction refinement. Moreover, it can *generate*—rather than just verify—invariants, including resource bounds (see §5), and non-linear invariants (see Ex. 4.12).

CRA analyzes a procedure's behavior using a two-step process, following Tarjan's path-expression method [40] for solving single-procedure dataflow-analysis problems. First, for each node n in the procedure's control flow graph (CFG), it creates a regular expression R_n that recognizes all paths from the procedure's entry to n. Second, it evaluates the regular expressions within the CRA domain by interpreting each regular expression operator as a corresponding operator on transition formulas: $+$ is disjunction, $\cdot$ is relational composition, and * over-approximates the reflexive transitive closure of a transition formula by generating and solving an appropriate system of recurrence equations.

[*] Supported, in part, by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies. T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

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

- At the intraprocedural level, CRA makes use of Tarjan's path-expression method to compose the meanings of subparts via the interpretations of $+$, $\cdot$, and * [14].
- At the interprocedural level, each procedure is analyzed independently of its calling context to produce a summary that is used to interpret calls to the procedure.

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

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

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

To create such a system, we must deal with the "impedance mismatch" between CRA's reliance on Tarjan's path-expression method, which handles *regular languages*, and the *context-free-language underpinnings* of context-sensitive interprocedural analysis [5, 15, 33, 34, 38]. This issue is challenging because at the technical level, the regular-language viewpoint is baked into CRA: CRA's recurrence-solving step is coupled with interpreting Kleene-star (*).

The inspiration for our work was the recently developed framework for Newtonian Program Analysis via Tensor Product (NPA-TP) [35]. At first blush, it appeared that NPA-TP overcomes the impedance mismatch because it provides a way to harness Tarjan's path-expression method for interprocedural analysis: NPA-TP has the surprising property of converting an interprocedural-analysis problem—i.e., a *context-free* path problem—into a sequence of *regular-language* path problems.

For NPA-TP to be applicable, the abstract domain must support a few non-standard operations (i.e., a so-called tensor-product operation and a detensor operation). While the CRA domain can be extended with these non-standard operations (§4.2), it fails to satisfy some properties on which NPA-TP relies: (i) CRA has *infinite ascending chains*, and thus NPA-TP[CRA] would not be guaranteed to terminate; (ii) CRA does not have an *effective equivalence procedure*, and thus it is not possible, in general, to ascertain whether an NPA-TP[CRA] analyzer has reached a fixpoint.

We address these issues by developing a new framework, which we call NPA-TP-GJ (NPA-TP with Gauss-Jordan elimination, §4.3). Whereas Kleene iteration, NPA [13], and NPA-TP do not converge at all when working with an abstract domain, such as the CRA domain, that has neither effective equivalence nor the ascending-chain condition, the NPA-TP-GJ algorithm (Alg. 4.20) is able to both detect and enforce convergence.

Contributions. Our work makes three main contributions:

- We extend CRA to create ICRA, a context-sensitive interprocedural version that handles loops and recursion—including non-linear recursion—in a *uniform* way (§4.3).
- We present the NPA-TP-GJ framework, which represents the interface and properties required for our approach. NPA-TP-GJ is an interprocedural-analysis method that can be used when the abstract domain has infinite ascending chains and does not support effective equivalence (§4.3).
- We present the results of experiments with an implementation of ICRA (§5). The experiments show that ICRA has broad overall strength, compared with several state-of-the-art software model checkers.

Outline. §2 summarizes CRA. §3 states the problem that the paper addresses. §4 presents the technical details of our solution. §5 describes the implementation of Algorithm NPA-TP-GJ and presents experimental results. §6 discusses related work. Proofs can be found in [27, App. A].

2. Background

2.1 Algebraic Program Analysis

Algebraic program analysis is an alternative to the classic iterative style of program analysis; it takes an algebraic (rather than order-theoretic) approach to approximating repetitive behavior. We illustrate the difference between the algebraic and iterative styles of program analysis with the example shown in Fig. 1. The program's CFG can also be represented as the following recursive system of equations:

$$S : \left\{ \begin{array}{lcl} X_0 & = & \epsilon + \left(X_2 \cdot \texttt{i := i - 1}\right) \\ X_1 & = & X_0 \cdot \texttt{i > 0} \\ X_2 & = & X_1 \cdot \left(\texttt{x := x + i} + \texttt{y := y - i}\right) \\ X_3 & = & X_0 \cdot \texttt{i <= 0} \end{array} \right\}$$

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

The problem that we are interested in is the classic problem from dataflow analysis of overapproximating the interpretation of all paths that start at the beginning of a program and end at a given program point [26, 38]. We formalize this problem as follows. Suppose that we are given a space of

```
while (i > 0)
  if (*) x := x + i
  else y := y - i
  i := i - 1
```

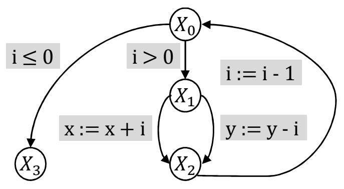

Figure 1: Example program and its CFG.

path properties D and a function $\llbracket\cdot\rrbracket : \Sigma \to D$ that maps each program instruction to a path property representing it (where $\Sigma \overset{\text{def}}{=} \{\texttt{i > 0}, \texttt{i := i - 1}, \ldots\}$ denotes the set of all program instructions). We suppose that D is equipped with a sequencing operator $\otimes$ (with unit $\underline{1}$) and a choice operator $\oplus$ (with unit $\underline{0}$). We define an approximation order $\gtrsim$ on D by defining $a \gtrsim b$ iff $a \oplus b = a$, and lift $\llbracket\cdot\rrbracket$ to paths by defining $\llbracket p_1 \ldots p_n \rrbracket \overset{\text{def}}{=} \llbracket p_1 \rrbracket \otimes \cdots \otimes \llbracket p_n \rrbracket$. Our goal is

$$\boxed{\text{For each variable } X_i\text{, compute some path property } \mathbf{D}(X_i) \in D\text{, such that for each path } p \in \mathit{Paths}(X_i)\text{, } p \text{ satisfies the property } \mathbf{D}(X_i) \text{ (i.e., } \mathbf{D}(X_i) \gtrsim \llbracket p \rrbracket\text{).}}$$

In the terminology of abstract interpretation [10], we wish to compute a function $\mathbf{D}$ that abstracts the *path semantics* of the program within some abstract domain D.

DEFINITION 2.1. *In the* ***path semantics****, the domain of path properties is the set of all path languages: $D \overset{\text{def}}{=} 2^{\Sigma^*}$. The operations are: $L_1 \otimes L_2 \overset{\text{def}}{=} \{p_1 p_2 : p_1 \in L_1 \wedge p_2 \in L_2\}$; $L_1 \oplus L_2 \overset{\text{def}}{=} L_1 \cup L_2$; $\underline{0} \overset{\text{def}}{=} \emptyset$; and $\underline{1} \overset{\text{def}}{=} \{\epsilon\}$. The semantic function $\llbracket a \rrbracket \overset{\text{def}}{=} \{a\}$ maps each instruction a to $\{a\}$.*

The classical way to establish that a function $\mathbf{D}$ is a solution to a system of equations S (in the sense that $\mathbf{D}$ abstracts the path semantics) is to show that $\mathbf{D}$ is a solution to a related system of *in*equations $\llbracket S \rrbracket$ over the abstract domain D, where $\llbracket S \rrbracket$ is obtained by re-interpreting the regular operators appearing in S with the corresponding operators in D. For the S that corresponds to Fig. 1, we have

$$\llbracket S \rrbracket : \left\{ \begin{array}{l} X_0 \gtrsim \underline{1} \oplus (X_2 \otimes \llbracket \texttt{i := i - 1} \rrbracket) \\ X_1 \gtrsim X_0 \otimes \llbracket \texttt{i > 0} \rrbracket \\ X_2 \gtrsim X_1 \otimes \left(\llbracket \texttt{x := x + i} \rrbracket \oplus \llbracket \texttt{y := y - i} \rrbracket \right) \\ X_3 \gtrsim X_0 \otimes \llbracket \texttt{i <= 0} \rrbracket \end{array} \right\}$$

We call a solution to $\llbracket S \rrbracket$ a ***post-fixpoint solution*** to S. It is easy to see that if $\mathbf{D}$ is a post-fixpoint solution to $\llbracket S \rrbracket$, then $\mathbf{D}$ overapproximates the path semantics.

In an *iterative* program analysis, a post-fixpoint solution is computed as the limit of a sequence of approximations. Define the ***Kleene iteration sequence*** $\langle \mathbf{D}^n : \{X_1, X_2, X_3, X_4\} \to D \rangle_{n \in \mathbb{N}}$ as follows:

$$\begin{aligned} \mathbf{D}^0(X_i) &= \underline{0} \\ \mathbf{D}^{n+1}(X_0) &= \underline{1} \oplus (\mathbf{D}^n(X_2) \otimes \llbracket \texttt{i := i - 1} \rrbracket) \\ \mathbf{D}^{n+1}(X_1) &= \mathbf{D}^n(X_0) \otimes \llbracket \texttt{i > 0} \rrbracket \\ \mathbf{D}^{n+1}(X_2) &= \mathbf{D}^n(X_1) \otimes (\llbracket \texttt{x := x + i} \rrbracket \oplus \llbracket \texttt{y := y - i} \rrbracket) \\ \mathbf{D}^{n+1}(X_3) &= \mathbf{D}^n(X_0) \otimes \llbracket \texttt{i <= 0} \rrbracket \end{aligned}$$

Define $\mathbf{D}(X_i) \overset{\text{def}}{=} \bigoplus_n \mathbf{D}^n(X_i)$ to be the limit of this sequence. When D has no infinite ascending chains, the sequence $\langle \mathbf{D}^n \rangle_{n \in \mathbb{N}}$ eventually stabilizes, and we may compute $\mathbf{D}$ effectively. If D fails to satisfy the ascending-chain condition, then we use a widening operator to ensure convergence [11]. Observe that if we take D to be the *path semantics*, then $\mathbf{D}(X_i)$ coincides with $\mathit{Paths}(X_i)$.

Algebraic program analysis is an alternative to this iterative style. Rather than assuming the ascending-chain condition or a binary widening operator, we assume that the space of program properties is equipped with a unary iteration operator $*$. We compute $\mathbf{D}$ using a two-step process. The first step is to apply Tarjan's path-expression algorithm [40] to compute, for each variable X_i in the system, a regular expression that recognizes the path language $\mathit{Paths}(X_i)$:

$$\hat{S} : \left\{ \begin{array}{l} X_0 = \mathit{body}^* \\ X_1 = \mathit{body}^* \cdot \texttt{i > 0} \\ X_2 = \mathit{body}^* \cdot \texttt{i > 0} \cdot (\texttt{x := x+i} + \texttt{y := y-i}) \\ X_3 = \mathit{body}^* \cdot \texttt{i <= 0} \end{array} \right\}$$

where $\mathit{body} \overset{\text{def}}{=} \texttt{i > 0} \cdot \left(\begin{array}{l} \texttt{x := x+i} \\ + \texttt{y := y-i} \end{array} \right) \cdot \texttt{i := i-1}$. The second step is to evaluate each path expression within D, using the algebraic operators $\otimes$, $\oplus$, and $*$ to interpret the regular expression operators $\cdot$, $+$, and $*$, respectively:

$$\begin{aligned} \mathbf{D}(X_0) &\overset{\text{def}}{=} \llbracket \mathit{body} \rrbracket^* \\ \mathbf{D}(X_1) &\overset{\text{def}}{=} \llbracket \mathit{body} \rrbracket^* \otimes \llbracket \texttt{i > 0} \rrbracket \\ \mathbf{D}(X_2) &\overset{\text{def}}{=} \llbracket \mathit{body} \rrbracket^* \otimes \llbracket \texttt{i > 0} \rrbracket \otimes \left(\begin{array}{l} \llbracket \texttt{x := x + i} \rrbracket \\ \oplus \llbracket \texttt{y := y - i} \rrbracket \end{array} \right) \\ \mathbf{D}(X_3) &\overset{\text{def}}{=} \llbracket \mathit{body} \rrbracket^* \otimes \llbracket \texttt{i <= 0} \rrbracket \end{aligned}$$

where $\llbracket \mathit{body} \rrbracket \overset{\text{def}}{=} \llbracket \texttt{i > 0} \rrbracket \otimes \left(\begin{array}{l} \llbracket \texttt{x := x+i} \rrbracket \\ \oplus \llbracket \texttt{y := y-i} \rrbracket \end{array} \right) \otimes \llbracket \texttt{i := i-1} \rrbracket$. Again, if we take D to be the path semantics (where $L^* \overset{\text{def}}{=} \bigcup_n L^n$, $L^0 = \underline{1}$, and $L^{n+1} = L^n \otimes L$), then $\mathbf{D}(X_i)$ coincides with $\mathit{Paths}(X_i)$.

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

The method developed in this paper combines elements of both the iterative and algebraic styles of program analysis. First, the equations defining the paths through a program are transformed (§4.3.1). For *linear* recursive systems, the transformation results in a non-recursive system that can be solved merely by interpreting the right-hand side of each equation within a suitable algebra (as in the algebraic method). For general systems, the transformation results in a recursive system of a restricted form, which can be solved using an iterative method (§4.3.2). Post-fixpoint solutions of the transformed system are generally *not* post-fixpoint solutions of the original system; however, they do overapproximate the path semantics of the original system. The correctness of the approach is captured by the following principle:

OBSERVATION 2.1. **[Path-Preservation Principle]**. *A post-fixpoint solution of a transformed (but equivalent) system of equations overapproximates every path of the original equation system.* □

We make use of Obs. 2.1 in the soundness argument for the program-analysis algorithm presented in §4.

2.2 Compositional Recurrence Analysis

Compositional Recurrence Analysis (CRA) [14] is a program analysis that follows the algebraic style. The space of path properties supported by CRA (i.e., the carrier of the semantic algebra of CRA) is the set of *transition formulas* over (not necessarily linear) integer arithmetic. Letting $\mathbf{x}$ denote a finite set of program variables, a transition formula $\varphi(\mathbf{x}, \mathbf{x}')$ is a formula over the variables $\mathbf{x}$ plus a set of primed copies $\mathbf{x}'$, representing the values of the variables before and after executing a path, respectively. The sequencing operation is relational composition and choice is disjunction:[1]

$$\varphi \otimes \psi \overset{\text{def}}{=} \exists \mathbf{x}''.\varphi(\mathbf{x}, \mathbf{x}'') \wedge \psi(\mathbf{x}'', \mathbf{x}') \qquad \varphi \oplus \psi \overset{\text{def}}{=} \varphi \vee \psi$$

The heart of CRA is its iteration operator. Given as input a transition formula φ_{body} that summarizes the body of a loop, the iteration operator computes a formula φ_{body}^* that summarizes any number of iterations. The iteration operator of CRA uses an SMT solver to extract a system of recurrences entailed by φ_{body}, and then returns the closed form of the system as the abstraction of the loop.

We illustrate how CRA analyzes loops using the example system $\hat{S}$ from §2.1. (See [14] for algorithmic details.) To compute the value of $\mathbf{D}(X_0)$ (that is, a transition formula that approximates every path starting and ending at the beginning of the loop), we evaluate within the CRA algebra the right-hand side of the equation for X_0 in $\hat{S}$

$$\left(\texttt{i > 0} \cdot \left(\texttt{x := x+i} + \texttt{y := y-i}\right) \cdot \texttt{i := i-1}\right)^* .$$

The evaluation proceeds by first evaluating the expression inside the Kleene-star operator, which yields the following transition formula (representing one iteration of the body of the loop):

$$\varphi_{body} : \mathtt{i} > 0 \wedge \begin{pmatrix} (\mathtt{x}' = \mathtt{x} + \mathtt{i} \wedge \mathtt{y}' = \mathtt{y}) \\ \vee\ (\mathtt{x}' = \mathtt{x} \wedge \mathtt{y}' = \mathtt{y} - \mathtt{i}) \end{pmatrix} \wedge \mathtt{i}' = \mathtt{i} - 1$$

It then applies the iteration operator to φ_{body} to obtain a value for X_0. The iteration operator extracts the following recurrence (in)equations from φ_{body}, and computes their closed forms using symbolic summation:

Recurrence	Closed form
$\mathtt{i}' = \mathtt{i} - 1$	$\mathtt{i}^{(k)} = \mathtt{i}^{(0)} - k$
$\mathtt{x}' \geq \mathtt{x}$	$\mathtt{x}^{(k)} \geq \mathtt{x}^{(0)}$
$\mathtt{y}' \leq \mathtt{y}$	$\mathtt{y}^{(k)} \leq \mathtt{y}^{(0)}$
$\mathtt{y}' \geq \mathtt{y} - \mathtt{i}$	$\mathtt{y}^{(k)} \geq \mathtt{y}^{(0)} - k(k-1)/2 - k\mathtt{i}^{(0)}$
$\mathtt{x}' \leq \mathtt{x} + \mathtt{i}$	$\mathtt{x}^{(k)} \leq \mathtt{x}^{(0)} + k(k-1)/2 + k\mathtt{i}^{(0)}$
$\mathtt{x}' - \mathtt{y}' = \mathtt{x} - \mathtt{y} + \mathtt{i}$	$\mathtt{x}^{(k)} - \mathtt{y}^{(k)} = \mathtt{x}^{(0)} - \mathtt{y}^{(0)} + k(k-1)/2 + k\mathtt{i}^{(0)}$

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

$$\begin{aligned} \varphi_{body}^* : \exists k. k \geq 0 \wedge \mathtt{i}' = \mathtt{i} - k \wedge \mathtt{x}' \geq \mathtt{x} \wedge \mathtt{y}' \leq \mathtt{y} \\ \wedge\ \mathtt{y}' \geq \mathtt{y} - k(k-1)/2 - k\mathtt{i} \\ \wedge\ \mathtt{x}' \leq \mathtt{x} + k(k-1)/2 + k\mathtt{i} \\ \wedge\ \mathtt{x}' - \mathtt{y}' = \mathtt{x} - \mathtt{y} + k(k-1)/2 + k\mathtt{i} \end{aligned}$$

3. Problem Statement

The problem addressed in the paper is the following:

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

Recursion presents an obstacle because the set of paths through a program that uses recursion is not regular: it is context-free [33, 38], which places recursive behavior beyond the scope of what can be analyzed using CRA's iteration operator.

Recent work on Newtonian Program Analysis via Tensor Product (NPA-TP) [35] has provided a crucial piece of the puzzle. NPA-TP uses tensor products (§4.2) to extend the algebraic approach to program analysis to *linear* recursive systems of equations. To solve non-linear recursive equations, NPA-TP uses Newton iteration, an approach pioneered by Esparza et al. [13] that generalizes Newton's method for numerical analysis. NPA-TP is a hybrid iterative/algebraic approach in which a solution to a system of equations is computed as the limit of a sequence (like in iterative program analysis), but where each iterate is the solution to a linearized model of the equations (which is solved algebraically). However, neither Newton iteration nor Kleene iteration is compatible with CRA:

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

[1] In the implementation of sequential composition (and that of the detensor operation defined in §4.2), fresh Skolem constants are introduced for existentially quantified variables. We do not perform quantifier elimination because the formulas are in non-linear integer arithmetic, which does not admit quantifier elimination.

- The problem of determining whether two CRA transition formulas are equivalent is undecidable, so there is no way to tell if Newton/Kleene iteration has converged.

Our algorithm for ICRA, presented in §4.3, is based on a new analysis framework, called NPA-TP-GJ. NPA-TP-GJ adopts the idea from NPA-TP of working with a refactored equation system obtained by rearranging expressions using tensor product (§4.3.1). However, other aspects of NPA-TP-GJ are significantly different. For instance, the key step of NPA-TP-GJ has the flavor of Gauss-Jordan elimination: it repeatedly carries out (i) a symbolic variation of NPA-TP's linearization step (but only applied to a single equation), and (ii) substitution of a closed-form expression for the linearized symbol into the other equations. (See Alg. 4.14.)

4. Technical Details

4.1 Recursive Equation Systems

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

DEFINITION 4.1. *Let Σ be an alphabet and let $\mathcal{X} = \{X_1, \ldots, X_n\}$ be a finite set of variables. A* ***system of equations S with regular right-hand sides over Σ and $\mathcal{X}$*** *consists of one equation for each variable, $X_1 = R_1$, ..., $X_n = R_n$, where each R_i is a regular expression over the alphabet $\mathcal{X} \cup \Sigma$. More compactly, we write $S : \{X_i = R_i\}_{i=1}^n$.*

Quasi-weight domains are algebraic structures with operations to interpret the operators of regular expressions, the domain of CRA being the motivating example for our interest. The axioms of quasi-weight domains ensure that its operations overapproximate the operations of the path algebra described in Defn. 2.1.

DEFINITION 4.2. *A* ***quasi-weight domain*** *$\mathcal{D} = \langle D, \equiv, \oplus, \otimes, *, \underline{0}, \underline{1}\rangle$ is a set D equipped with an equivalence relation $\equiv$, a binary* ***combine*** *operator $\oplus$, a binary* ***extend*** *operator $\otimes$, a unary* ***closure*** *operator $*$, and distinguished elements $\underline{0}$ and $\underline{1}$, and that satisfies the following axioms:*

1. *The equivalence relation $\equiv$ is a congruence with respect to $\oplus$ and $\otimes$ ($a_1 \equiv a_2$ and $b_1 \equiv b_2$ implies $a_1 \oplus b_1 \equiv a_2 \oplus b_2$ and $a_1 \otimes b_1 \equiv a_2 \otimes b_2$). Note that $\equiv$ is not necessarily a congruence with respect to $*$.*
2. *$\langle D, \oplus, \otimes, \underline{0}, \underline{1}\rangle$ is an idempotent semiring "up to equivalence," meaning that for all $a, b, c \in D$ we have*
   (a) (Associativity) $(a \otimes b) \otimes c \equiv a \otimes (b \otimes c)$ and $(a \oplus b) \oplus c \equiv a \oplus (b \oplus c)$
   (b) (Unit) $a \otimes \underline{1} \equiv \underline{1} \otimes a \equiv a$ and $a \oplus \underline{0} \equiv \underline{0} \oplus a \equiv a$
   (c) (Commutativity) $a \oplus b \equiv b \oplus a$
   (d) (Distributivity) $a \otimes (b \oplus c) \equiv (a \otimes b) \oplus (a \otimes c)$ and $(b \oplus c) \otimes a \equiv (b \otimes a) \oplus (c \otimes a)$
   (e) (Idempotence) $a \oplus a \equiv a$
3. *The closure operator overapproximates reflexive transitive closure, in the following sense:*
   (a) (Reflexivity) $\underline{1} \lesssim a^*$
   (b) (Transitivity) $a^* \otimes a \lesssim a^*$ and $a \otimes a^* \lesssim a^*$
   where $\lesssim$ is the ***natural preorder*** *on $\mathcal{D}$: for any a, b in D, $a \lesssim b \iff a \oplus b \equiv b$.*

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

EXAMPLE 4.3. *The* ***quasi-weight domain of paths*** *$\langle 2^{\Sigma^*}, \equiv, \oplus, \otimes, *, \underline{0}, \underline{1}\rangle$ is defined using $\oplus$, $\otimes$, $\underline{0}$, and $\underline{1}$ from Defn. 2.1. The equivalence relation $\equiv$ is equality on languages, and the iteration operation $L^* \overset{def}{=} \bigcup_{i=0}^{\infty} L^i$ is Kleene closure.*

EXAMPLE 4.4. *Let $\mathbf{x}$ denote a finite set of program variables. In the* ***quasi-weight domain of CRA*** *over the variables $\mathbf{x}$, the carrier D is the set of all transition formulas $\varphi(\mathbf{x}, \mathbf{x}')$ over the variables $\mathbf{x}$ and primed copies $\mathbf{x}'$. The equivalence relation $\equiv$ is logical equivalence; the sequencing operator $\varphi \otimes \psi \overset{def}{=} \exists \mathbf{x}''.\varphi(\mathbf{x}, \mathbf{x}'') \wedge \psi(\mathbf{x}'', \mathbf{x}')$ is sequential composition; the choice operation $\varphi \oplus \psi \overset{def}{=} \varphi \vee \psi$ is disjunction; the iteration operation φ^* is the defined by extracting recurrences from φ and computing their closed forms as described in §2.2; $\underline{1} \overset{def}{=} (\mathbf{x} = \mathbf{x}')$ is the identity transition; and $\underline{0} \overset{def}{=}$ false is the empty transition.*

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

DEFINITION 4.5. *Let $\mathcal{D} = \langle D, \equiv, \oplus, \otimes, *, \underline{0}, \underline{1}\rangle$ be a quasi-weight domain, Σ be an alphabet, $\mathcal{X} = \{X_1, ..., X_n\}$ be a set of variables, $[\![\cdot]\!] : \Sigma \to D$ be an alphabet interpretation, and $\sigma : \mathcal{X} \to D$ be a variable interpretation. We use $[\![\cdot]\!]_\sigma$ to denote the function that maps any regular expression over Σ and $\mathcal{X}$ to an element of D by using $[\![\cdot]\!]$ to interpret Σ, σ to interpret $\mathcal{X}$, and by using the operations of $\mathcal{D}$ in place of the regular-expression operators 0, 1, $+$, $\cdot$, $*$. If R contains no variables, we may omit the subscript σ.*

We use $[\![\cdot]\!]^P$ to denote the obvious alphabet interpretation for the quasi-weight domain of paths (as described in §2.1): for any regular expression R, $[\![R]\!]^P$ is the set of paths recognized by R. For any system of equations with *regular right-hand sides* $S : \{X_i = R_i\}$ *over* Σ *and* $\mathcal{X} = \{X_1, ..., X_n\}$*, we define* $\mathit{Paths}_S : \mathcal{X} \to 2^{\Sigma^*}$ *to be the least function (in pointwise set-inclusion order) such that for each* i*,* $\mathit{Paths}_S(X_i) = [\![R_i]\!]^P_{\mathit{Paths}_S}$.

We can now formalize the problem of interest.

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

In what follows, we will describe a method for solving systems of equations with regular right-hand sides. Our presentation of the method proceeds in two steps: first, we review *tensor products*, which can be used to solve *linear* recursive systems [35]; second, we present a hybrid iterative/algebraic technique for solving general recursive systems based on tensor products and iteration domains.

4.2 Tensor Products

Intuitively, quasi-weight domains are well-suited for solving *intra*procedural program analysis problems, such as the one described in §2.1. Intraprocedural program analysis problems correspond to *left-linear* systems of equations (in which each right-hand-side is of the form $\bigoplus_i X_i \cdot R_i$ where R_i does not contain variables). Left-linear systems correspond to regular languages, and the operators of quasi-weight domains correspond to those of regular expressions. For *inter*procedural-analysis problems, however, these operations are not sufficient. Just as one cannot describe the set of paths of a recursive procedure using a regular expression, one cannot use the operators of a quasi weight domain to describe the operation of recursive procedures.

Recently, Reps et al. showed that algebraic program analysis can be extended to a *linear* system of equations by using a *tensored* domain [35]. As a warm-up exercise for §4.3, where we present our method for solving an arbitrary system of equations, this section describes a slight variation of the Reps et al. method. Essentially all of the machinery introduced in this section carries over to §4.3, but the setting in which it is used here is somewhat less complicated.

Following Obs. 2.1, we proceed in two steps. First, we define a way to transform a linear equation system into a different, but equivalent, form (Eqn. (1)). The transformation involves the tensor-product domain of paths (Defn. 4.6), and converts a linear equation system with regular right-hand sides into a *non-recursive* equation system in which the right-hand sides are regular expressions extended with additional operators from the tensor-product domain. Second, we describe the properties that a tensor-product domain must satisfy to be used to interpret an extended regular expression, and then describe the ICRA tensor-product domain.

DEFINITION 4.6. *Let* Σ *be an alphabet. Define a quasi-weight domain* $\mathcal{D}_\mathcal{T} = \langle D_\mathcal{T}, \equiv_\mathcal{T}, \oplus_\mathcal{T}, \otimes_\mathcal{T}, *_\mathcal{T}, \underline{0}_\mathcal{T}, \underline{1}_\mathcal{T}\rangle$ *of* ***tensored paths*** *as follows: a tensored path* $\langle \underline{p}, \overline{p}\rangle$ *is a pair consisting of one "backward" path* $\underline{p}$ *and one "forward" path* $\overline{p}$*; the carrier* $D_\mathcal{T} \overset{\text{def}}{=} 2^{\Sigma^* \times \Sigma^*}$ *is the powerset of tensored paths. The* $\otimes_\mathcal{T}$ *operator is coordinate-wise concatenation, with concatenation reversed for backward paths:*

$$T_1 \otimes_\mathcal{T} T_2 \overset{\text{def}}{=} \{\langle \underline{p}_2\underline{p}_1, \overline{p}_1\overline{p}_2\rangle : \langle \underline{p}_1, \overline{p}_1\rangle \in T_1, \langle \underline{p}_2, \overline{p}_2\rangle \in T_2\}$$

Similarly to the quasi-weight domain of untensored paths, $\equiv_\mathcal{T}$ *is equality,* $\oplus_\mathcal{T}$ *is union, and* $T^{*_\mathcal{T}} \overset{\text{def}}{=} \bigcup_i T^i$ *where* $T^0 \overset{\text{def}}{=} \{\langle \epsilon, \epsilon\rangle\}$ *and* $T^{i+1} = T^i \otimes_\mathcal{T} T$*. The unit of* $\oplus_\mathcal{T}$ *is* $\underline{0}_\mathcal{T} \overset{\text{def}}{=} \emptyset$*, and the unit of* $\otimes_\mathcal{T}$ *is* $\underline{1}_\mathcal{T} \overset{\text{def}}{=} \{\langle \epsilon, \epsilon\rangle\}$.

The ***tensor-product domain of paths*** $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_\mathcal{T}, \odot, \ltimes\rangle$ *consists of the domain of paths* $\mathcal{D}$*, the domain of tensored paths* $\mathcal{D}_\mathcal{T}$*, a tensor operation* $\odot : \mathcal{D} \times \mathcal{D} \to \mathcal{D}_\mathcal{T}$ *and a detensor-product operation* $\ltimes : \mathcal{D} \times \mathcal{D}_\mathcal{T} \to \mathcal{D}$*, defined as follows:*

$$L_1 \odot L_2 \overset{\text{def}}{=} \{\langle p_1, p_2\rangle : p_1 \in L_1, p_2 \in L_2\}$$
$$L \ltimes T \overset{\text{def}}{=} \{\underline{p}p\overline{p} : p \in L, \langle \underline{p}, \overline{p}\rangle \in T\}$$

DEFINITION 4.7. *Let* Σ *be an alphabet and let* $\mathcal{X} = \{X_1, \ldots, X_n\}$ *be a finite set of variables. A* ***(tensored) extended regular expression*** *over* Σ *and* $\mathcal{X}$ *is an expression generated by the following grammar:*

$$\begin{aligned} E, F \in \mathit{Ext}(\Sigma, \mathcal{X}) &::= a \in \Sigma \mid X_i \in \mathcal{X} \mid 0 \mid 1 \\ &\quad\mid E + F \mid E \cdot F \mid E^* \mid E \ltimes E_\mathcal{T} \\ E_\mathcal{T}, F_\mathcal{T} \in \mathit{Ext}_\mathcal{T}(\Sigma, \mathcal{X}) &::= (E \odot F) \mid 0_\mathcal{T} \mid 1_\mathcal{T} \\ &\quad\mid E_\mathcal{T} +_\mathcal{T} F_\mathcal{T} \mid E_\mathcal{T} \cdot_\mathcal{T} F_\mathcal{T} \mid E_\mathcal{T}^{*_\mathcal{T}} \end{aligned}$$

For any function $\sigma : \mathcal{X} \to 2^{\Sigma^*}$ *and any extended regular expression* E*, we use* $[\![E]\!]^P_\sigma$ *to denote the set of paths in the language of* E*.* $[\![E]\!]^P_\sigma$ *is defined as the interpretation of* E *using the operations of* $\mathscr{T}$ *from Defn. 4.6 in place of the operators* 0*,* 1*,* $+$*,* $\cdot$*,* $*$*,* $0_\mathcal{T}$*,* $1_\mathcal{T}$*,* $+_\mathcal{T}$*,* $\cdot_\mathcal{T}$*,* $*_\mathcal{T}$*,* $\odot$*, and* $\ltimes$.

For extended regular expressions E *and* F*, we write* $E \simeq F$ *to denote that for every function* $\sigma : \mathcal{X} \to 2^{\Sigma^*}$*, we have* $[\![E]\!]^P_\sigma = [\![F]\!]^P_\sigma$.

As a concrete example, $\{a^i b^i : i \geq 0\}$ is the classic example of a non-regular language. It is not the language of any regular expression, but it is the language of the *extended* regular expression $1 \ltimes (a \odot b)^{*_\mathcal{T}}$.

Using extended regular expressions, we can transform a linear system of equations
$S : \{X_i = r_i + s_{i,1} \cdot X_{j_1} \cdot t_{i,1} + \ldots + s_{i,m_i} \cdot X_{j_{m_i}} \cdot t_{i,m_i}\}_{i=1}^n$
over an alphabet Σ and set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ as follows. Let $\mathcal{Z} = \{Z_1, \ldots, Z_n\}$ be a set of variables (one for each X_i), and define the following *left-linear* system:

$$S_\mathcal{T} : \left\{\begin{matrix} Z_i = (1 \odot r_i) +_\mathcal{T} Z_{j_1} \cdot_\mathcal{T} (s_{i,1} \odot t_{i,1}) +_\mathcal{T} \ldots \\ \qquad +_\mathcal{T} Z_{j_{m_i}} \cdot_\mathcal{T} (s_{i,m_i} \odot t_{i,m_i}) \end{matrix}\right\}_{i=1}^n \quad (1)$$

Using Tarjan's path-expression algorithm, we can transform $S_\mathcal{T}$ into an equivalent *non-recursive* system of equations $\hat{S}_{\mathcal{T}} : \{Z_i = E_i\}_{i=1}^n$ in which the right-hand sides are extended regular expressions. Finally, define system $\hat{S}$ to be $\{X_i = 1 \ltimes E_i\}_{i=1}^n$. We have the following path-preservation property:

PROPOSITION 4.8. *[35] For each i, we have*

$$Paths_{\hat{S}}(X_i) = Paths_S(X_i)\ .$$

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

DEFINITION 4.9. *A* ***tensor-product domain*** $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_{\mathcal{T}}, \odot, \ltimes \rangle$ *consists of two quasi-weight domains, $\mathcal{D}$ and $\mathcal{D}_{\mathcal{T}}$, along with the following two operations:*

- *A* ***tensor-product*** *operator, denoted by $\odot : D \times D \to D_{\mathcal{T}}$, such that for all $a, b, c, a_1, b_1, a_2, b_2 \in D$,*

$$\begin{aligned} \underline{0} \odot a \equiv_{\mathcal{T}} a \odot \underline{0} &\equiv_{\mathcal{T}} \underline{0}_{\mathcal{T}} \\ a \odot (b \oplus c) &\equiv_{\mathcal{T}} (a \odot b) \oplus_{\mathcal{T}} (a \odot c) \\ (b \oplus c) \odot a &\equiv_{\mathcal{T}} (b \odot a) \oplus_{\mathcal{T}} (c \odot a) \\ (a_1 \odot b_1) \otimes_{\mathcal{T}} (a_2 \odot b_2) &\equiv_{\mathcal{T}} (a_2 \otimes a_1) \odot (b_1 \otimes b_2) \end{aligned}$$

  and for all $a_1 \equiv a_2$ and $b_1 \equiv b_2$, we have $a_1 \odot b_1 \equiv_{\mathcal{T}} a_2 \odot b_2$.
- *A* ***detensor-product*** *operation $\ltimes : D \times D_{\mathcal{T}} \to D$, such that for all $a, b \in D$ and all $p, q \in D_{\mathcal{T}}$ we have*

$$\begin{aligned} a \ltimes (p \oplus_{\mathcal{T}} q) &\equiv (a \ltimes p) \oplus (a \ltimes q) \\ (a \oplus b) \ltimes p &\equiv (a \ltimes p) \oplus (b \ltimes p) \\ a \ltimes (p \otimes_{\mathcal{T}} (b \odot c)) &\equiv b \otimes (a \ltimes p) \otimes c \\ a \ltimes (p \otimes_{\mathcal{T}} q) &\equiv (a \ltimes p) \ltimes q \\ a \ltimes \underline{1}_{\mathcal{T}} &\equiv a \end{aligned}$$

  and for all $a \equiv b$ and $p \equiv_{\mathcal{T}} q$, we have $a \ltimes p \equiv b \ltimes q$.

One can check that the tensor-product domain of paths (Defn. 4.6) satisfies the conditions of Defn. 4.9. A second example is the ***tensor-product domain of CRA***, denoted by *CRA*$_{\mathcal{T}}$.

EXAMPLE 4.10. *Recall from Ex. 4.4 that CRA weights are formulas over some specified set of variables $\mathbf{x}$ and primed copies $\mathbf{x}'$. The weights in CRA$_{\mathcal{T}}$ are formulas over* four *sets of variables $\underline{\mathbf{x}}, \underline{\mathbf{x}}', \overline{\mathbf{x}}$, and $\overline{\mathbf{x}}'$. It is instructive to think of such a formula as a set of pairs of transitions $((\underline{s}, \underline{s}'), (\overline{s}, \overline{s}'))$. Under this interpretation, the semantic definitions of the $\otimes_{\mathcal{T}}$, $\odot$, and $\ltimes$ operators are as follows:*

$$\begin{aligned} T_1 \otimes_{\mathcal{T}} T_2 &= \left\{ ((\underline{s}, \underline{s}'), (\overline{s}, \overline{s}')) : \exists \underline{s}'', \overline{s}''. \begin{matrix} ((\underline{s}'', \underline{s}'), (\overline{s}, \overline{s}'')) \in T_1, \\ ((\underline{s}, \underline{s}''), (\overline{s}'', \overline{s}')) \in T_2 \end{matrix} \right\} \\ R_1 \odot R_2 &= \{((s_1, s_1'), (s_2, s_2')) : (s_1, s_1') \in R_1, (s_2, s_2') \in R_2\} \\ R \ltimes T &= \{(\underline{s}, \overline{s}') : \exists \underline{s}', \overline{s}. (\underline{s}', \overline{s}) \in R \wedge ((\underline{s}, \underline{s}'), (\overline{s}, \overline{s}')) \in T\} \end{aligned}$$

The operations of CRA$_{\mathcal{T}}$ can be carried out syntactically using variable renaming and existential quantification:

$$\begin{aligned} \varphi_{\mathcal{T}} \otimes_{\mathcal{T}} \psi_{\mathcal{T}} &\overset{\text{def}}{=} \exists \underline{\mathbf{x}}'', \overline{\mathbf{x}}''. \begin{pmatrix} \varphi_{\mathcal{T}}[\underline{\mathbf{x}} \mapsto \underline{\mathbf{x}}'', \overline{\mathbf{x}}' \mapsto \overline{\mathbf{x}}''] \\ \wedge\ \psi[\underline{\mathbf{x}}' \mapsto \underline{\mathbf{x}}'', \overline{\mathbf{x}} \mapsto \overline{\mathbf{x}}''] \end{pmatrix} \\ \varphi_{\mathcal{T}} \oplus_{\mathcal{T}} \psi_{\mathcal{T}} &\overset{\text{def}}{=} \varphi_{\mathcal{T}} \vee \psi_{\mathcal{T}} \\ 0_{\mathcal{T}} &\overset{\text{def}}{=} \mathit{false} \\ 1_{\mathcal{T}} &\overset{\text{def}}{=} \underline{\mathbf{x}} = \underline{\mathbf{x}}' \wedge \overline{\mathbf{x}} = \overline{\mathbf{x}}' \\ \varphi \odot \psi &\overset{\text{def}}{=} (\varphi[\mathbf{x} \mapsto \underline{\mathbf{x}}, \mathbf{x}' \mapsto \underline{\mathbf{x}}']) \wedge (\psi[\mathbf{x} \mapsto \overline{\mathbf{x}}, \mathbf{x}' \mapsto \overline{\mathbf{x}}']) \\ \varphi \ltimes \psi_{\mathcal{T}} &\overset{\text{def}}{=} \exists \underline{\mathbf{x}}, \overline{\mathbf{x}}. \begin{pmatrix} \varphi[\mathbf{x} \mapsto \underline{\mathbf{x}}', \mathbf{x}' \mapsto \overline{\mathbf{x}}] \\ \wedge\ \psi_{\mathcal{T}}[\underline{\mathbf{x}} \mapsto \mathbf{x}, \overline{\mathbf{x}}' \mapsto \mathbf{x}'] \end{pmatrix} \end{aligned}$$

The iteration operator $*_{\mathcal{T}}$ operates by finding closed forms for recurrences just as described in §2.2, except over a vocabulary of four sets of variables.

Tensor-product domains can be used to evaluate extended regular expressions in the same way that quasi-weight domains can be used to evaluate regular expressions (Defn. 4.5). Let $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_{\mathcal{T}}, \odot, \ltimes \rangle$ be a tensor-product domain, $[\![\cdot]\!] : \Sigma \to \mathcal{D}$ be an interpretation of an alphabet Σ, and $\sigma : \mathcal{X} \to \mathcal{D}$ be an interpretation of a set of variables $\mathcal{X}$. We use $[\![\cdot]\!]_\sigma$ to denote the function that evaluates an extended regular expression within $\mathscr{T}$ by interpreting the extended-regular-expression operators 0, 1, $+$, $\cdot$, $*$, $0_{\mathcal{T}}$, $1_{\mathcal{T}}$, $+_{\mathcal{T}}$, $\cdot_{\mathcal{T}}$, $*_{\mathcal{T}}$, $\odot$, and $\ltimes$ using their counterparts in $\mathscr{T}$, using $[\![\cdot]\!]$ to interpret symbols in Σ, and σ to interpret variables. We omit the σ subscript for extended regular expressions without variables. The crucial property of tensor-product domains is that evaluation of extended regular expressions within a tensor-product domain overapproximates the path semantics:

PROPOSITION 4.11. *Let Σ be an alphabet, let $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_{\mathcal{T}}, \odot, \ltimes \rangle$ be a tensor-product domain, and let $[\![\cdot]\!] : \Sigma \to \mathcal{D}$ be an interpretation for the alphabet. For any regular expression E over Σ and any path $p \in [\![E]\!]^P$, we have $[\![E]\!] \gtrsim [\![p]\!]$.*

Together, Props. 4.8 and 4.11 give a complete recipe for applying Obs. 2.1 to solve a linear equation system S over a tensor-product domain:

- Transform S to eliminate recursion—while preserving paths—using Eqn. (1) and Tarjan's method.
- Evaluate the right-hand sides of the resulting system within the tensor-product domain.

EXAMPLE 4.12. *We demonstrate the use of the CRA tensor-product domain with a recursive multiplication routine.*

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

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

$$S : \left\{ X = \begin{matrix} \texttt{y = 0} \\ + \ (\texttt{y != 0} \cdot \texttt{y := y - 1} \cdot X \cdot \texttt{m := m + x}) \end{matrix} \right\}$$

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

$$\hat{S} : \left\{ X = \ltimes \begin{array}{l} \mathtt{y\ =\ 0} \\ ((\mathtt{y\ !=\ 0} \cdot \mathtt{y\ :=\ y\ -\ 1}) \odot (\mathtt{m\ :=\ m\ +\ x}))^{*\mathcal{T}} \end{array} \right\}$$

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

$$\underline{\mathtt{y}} \neq 0 \land \underline{\mathtt{y}}' = \underline{\mathtt{y}} - 1 \land \overline{\mathtt{m}}' = \overline{\mathtt{m}} + \overline{\mathtt{x}} \land \underline{\mathtt{m}}', \underline{\mathtt{x}}', \overline{\mathtt{y}}', \overline{\mathtt{x}}' = \underline{\mathtt{m}}, \underline{\mathtt{x}}, \overline{\mathtt{y}}, \overline{\mathtt{x}}$$

From this formula, we extract the recurrences shown to the right.

Recurrence	Closed form
$\underline{\mathtt{y}}' = \underline{\mathtt{y}} - 1$	$\underline{\mathtt{y}}^{(k)} = \underline{\mathtt{y}}^{(0)} - k$
$\overline{\mathtt{m}}' = \overline{\mathtt{m}} + \overline{\mathtt{x}}$	$\overline{\mathtt{m}}^{(k)} = \overline{\mathtt{m}}^{(0)} + k\overline{\mathtt{x}}^{(0)}$

The output of the iteration operator is the following tensored formula:

$$\exists k. k \geq 0 \land \left(\begin{array}{l} \underline{\mathtt{y}}' = \underline{\mathtt{y}} - k \land \overline{\mathtt{m}}' = \overline{\mathtt{m}} + k\overline{\mathtt{x}} \\ \land\ \underline{\mathtt{m}}', \underline{\mathtt{x}}', \overline{\mathtt{y}}', \overline{\mathtt{x}}' = \underline{\mathtt{m}}, \underline{\mathtt{x}}, \overline{\mathtt{y}}, \overline{\mathtt{x}} \end{array} \right)$$

Finally, by evaluating the detensor-product operation, we get the following summary for `mul`*:*

$$\exists k. k \geq 0 \land \mathtt{y}' = 0 \land \mathtt{y}' = \mathtt{y} - k \land \mathtt{m}' = \mathtt{m} + k\mathtt{x} \land \mathtt{x}' = \mathtt{x}$$

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

4.3 Non-Linear Systems

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

Our method overcomes two fundamental challenges:

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

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

- When there are infinite ascending chains, Kleene iteration may not converge.
- When equivalence is undecidable, even if Kleene iteration does converge, there is no way to tell.

However, as we show in the remainder of this section, it is possible to carry out Kleene iteration on the "regularized" model, due to its special properties (see Alg. 4.14). Our solution is motivated by the following observation about the CRA domain, which can be used to enforce and detect convergence for that domain:

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

We motivate our approach using the CRA domain. The iteration operator of CRA can be thought of as a two-phase process (cf. §2.2): given a formula φ representing the body of a loop, we (1) compute a set of recurrences entailed by φ using an SMT solver, and (2) compute closed forms for the recurrences to use as an approximation of the reflexive transitive closure of φ. The loop-body formula φ is expressed in a rich assertion language, which includes disjunction, quantifiers, and non-linear terms. The *recurrences* computed by phase (1), on the other hand, are relatively simple: each recurrence (e.g., $\mathtt{x}' = \mathtt{x} + 1$, representing that `x` is incremented in the loop) is a linear constraint, and so a set of recurrences can be represented by a convex polyhedron. Thus, we can express the iteration operator of CRA as the composition of (i) an *abstraction* function that computes a polyhedron (representing recurrences) from a transition formula, and (ii) an *abstract-closure* function that computes a transition formula from a polyhedron by computing closed forms. A key feature of our approach is that the widening operator is defined on the *iteration domain*—polyhedra—rather than on the domain of CRA itself (i.e., transition formulas over non-linear arithmetic). Polyhedra are a well-studied abstract domain, for which equivalence is decidable and widening operators are readily available. Our implementation uses the NewPolka implementation in APRON [1].

EXAMPLE 4.13. *We use the Fibonacci function to show how our analysis technique exploits the two-phase structure of CRA's iteration operator. The code for* `fib` *is pictured to the right; it uses two global variables,* `p` *(representing the parameter) and* `r` *(representing the return value), and two local variables,* `n` *and* `t`*. (Our technique handles local variables using essentially the same technique presented in [35, §8], but we omit the details.) The paths through* `fib` *are captured by the following equation:*

```
fib():
  n := p
  if (n <= 1) r := 1
  else
    p := n-1
    fib()
    t := r; p := n-2
    fib()
    r := r+t
```

$$S : \left\{ X = \mathtt{n{:=}p} \cdot \left(\begin{array}{l} \mathtt{n\ <=\ 1} \cdot \mathtt{r{:=}1} \\ +\ \mathtt{n\ >\ 1} \cdot \mathtt{p{:=}n{-}1} \cdot X \cdot \mathit{right} \end{array} \right) \right\},$$

where right $\overset{\text{def}}{=}$ $\texttt{t:=r} \cdot \texttt{p:=n-2} \cdot X \cdot \texttt{r:=r+t}$. *The first step in solving S is to re-arrange the system to use the tensored iteration operator to model the* ***first*** *recursive call*

$$\hat{S} : \left\{ X = \begin{array}{l} (\texttt{n:=p} \cdot \texttt{n <= 1} \cdot \texttt{r:=1}) \\ \ltimes\ (\texttt{n:=p} \cdot \texttt{n > 1} \cdot \texttt{p:=n-1} \odot \textit{right})^{*\mathcal{T}} \end{array} \right\}$$

Unlike the case of linear recursion (e.g., Ex. 4.12), $\hat{S}$ is still recursive: X appears within the expression "right." However, $\hat{S}$ has the property that ***every variable appears below a star****. We say that an occurrence of a variable in a (tensored) extended regular expression is* ***guarded*** *if it appears below a star; a variable with an* ***un****guarded occurrence is called* ***free****. The significance of $\hat{S}$ having no free variables is that it allows us to detect and enforce convergence of a successive-approximation process via a "derived sequence" of polyhedra.*

We define two sequences by mutual recursion: $\langle \mathbf{D}^k \rangle_{k\in\mathbb{N}}$ is a sequence of transition formulas representing summaries for `fib`*, and $\langle \beta^k \rangle_{k\in\mathbb{N}}$ is a sequence of polyhedra representing transitions of the "loop body" (appearing below the $^{*\mathcal{T}}$):*

$$\textit{body} = \texttt{n:=p} \cdot \texttt{n > 1} \cdot \texttt{p:=n-1} \odot \texttt{t:=r} \cdot \texttt{p:=n-2} \cdot X \cdot \texttt{r:=r+t}\ .$$

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

$$\mathbf{D}^1(X) = \mathtt{p} \leq 1 \wedge \mathtt{r}' = 1 \wedge \mathtt{p}' = \mathtt{p}$$

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

$$\begin{aligned} \beta^2 \ &= \bot \mathbin{\triangledown} \begin{pmatrix} \underline{\mathtt{p}}' = \underline{\mathtt{p}} - 1 \wedge 1 < \underline{\mathtt{p}} \leq 3 \wedge \underline{\mathtt{r}}' = \underline{\mathtt{r}} \\ \wedge\ \overline{\mathtt{p}}' = \underline{\mathtt{p}} - 2 \wedge \overline{\mathtt{r}}' = \overline{\mathtt{r}} + 1 \end{pmatrix} \\ &= \begin{pmatrix} \underline{\mathtt{p}}' = \underline{\mathtt{p}} - 1 \wedge 1 < \underline{\mathtt{p}} \leq 3 \wedge \underline{\mathtt{r}}' = \underline{\mathtt{r}} \\ \wedge\ \overline{\mathtt{p}}' = \underline{\mathtt{p}} - 2 \wedge \overline{\mathtt{r}}' = \overline{\mathtt{r}} + 1 \end{pmatrix} \end{aligned}$$

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

$$\begin{aligned} \mathbf{D}^2(X) &= \begin{pmatrix} \mathtt{p}' = \mathtt{p} \leq 1 \wedge \mathtt{r}' = 1 \\ \vee\ \exists k. k \geq 1 \wedge \mathtt{r}' = k + 1 \wedge k = \mathtt{p} - 1 \wedge \mathtt{p} \leq 3 \end{pmatrix} \\ \beta^3 &= \underline{\mathtt{p}}' = \underline{\mathtt{p}} - 1 \wedge 1 < \underline{\mathtt{p}} \wedge \underline{\mathtt{r}}' = \underline{\mathtt{r}} \wedge \overline{\mathtt{r}}' \geq \overline{\mathtt{r}} + 1 \\ \mathbf{D}^3(X) &= \begin{pmatrix} \mathtt{p}' = \mathtt{p} \leq 1 \wedge \mathtt{r}' = 1 \\ \vee\ \exists k. k \geq 1 \wedge \mathtt{r}' \geq k + 1 \wedge k = \mathtt{p} - 1 \end{pmatrix} \end{aligned}$$

When computing β^4, we find that the sequence has converged: $\beta^4 = \beta^3$. Because every variable is guarded by a Kleene-star, the fact that the sequence of loop-body polyhedra has converged implies that the sequence of function summaries has also converged (Thm. 4.18), and $\mathbf{D}^3(X) \equiv \mathbf{D}^4(X)$ is an overapproximating summary for `fib`*.*

More abstractly, the equation system for `fib` *consists of a single recursive equation with just one variable:*

$$S : \{X = d + (a \cdot X \cdot b \cdot X \cdot c)\}\ . \qquad (2)$$

By treating the first occurrence of X on the right-hand side of Eqn. (2) as a variable and the second occurrence as a symbolic constant, we created an equivalent system of equations where every variable appears below a star

$$\hat{S} : \{X = (\underline{1} \odot d) \ltimes (a \odot (b \otimes X \otimes c))^{*\mathcal{T}}\}.$$

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

$$\beta^k \overset{\text{def}}{=} \beta^{k-1} \mathbin{\triangledown} \widetilde{\alpha}_{\textit{poly}}(a \odot (b \otimes \mathbf{D}^k(X) \otimes c)),$$

where $\widetilde{\alpha}_{\textit{poly}}$ is a heuristic that creates a polyhedron that overapproximates a transition formula [14, §IV & Alg. 2]. Convergence of the derived sequence—i.e., $\beta^k = \beta^{k-1}$ for some k—is ***enforced*** *via the widening operator of the polyhedral domain. When the derived sequence converges, so does the $\mathbf{D}$ sequence (in particular, $\mathbf{D}^{k+1}(X) \equiv \mathbf{D}^k(X)$).*

We now show how this process can be carried out in a more abstract setting, for an arbitrary equation system over any number of variables. Following the Path-Preservation Principle (Obs. 2.1), we proceed in two steps. First, in §4.3.1, we show how to transform a system of equations into an equivalent system of equations with no free variables. Second, in §4.3.2, we formalize *iteration domains* (of which polyhedra are one concrete instance), and show how to solve systems of equations without free variables using a tensor-product domain equipped with an iteration domain.

4.3.1 Eliminating free variables

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

(i) Rewrite E as $F + X \ltimes F_{\mathcal{T}}$, where X is not free in F.
(ii) Replace the equation $X = E$ with the equation $X = F \ltimes F_{\mathcal{T}}^*$ (in which X is not free by construction), and replace every free occurrence of X within the system of equations with $F \ltimes F_{\mathcal{T}}^*$.

$$\begin{aligned}
\mathit{factor}_X(a) &\stackrel{\text{def}}{=} a + X \ltimes 0_{\mathcal{T}} \\
\mathit{factor}_X(X_i) &\stackrel{\text{def}}{=} \begin{cases} 0 + X \ltimes 1_{\mathcal{T}} & \text{if } X = X_i \\ X + X \ltimes 0_{\mathcal{T}} & \text{otherwise} \end{cases} \\
\mathit{factor}_X(E + E') &\stackrel{\text{def}}{=} (F + F') + X \ltimes (F_{\mathcal{T}} +_{\mathcal{T}} F'_{\mathcal{T}}) \\
&\quad \text{where } \mathit{factor}_X(E) = F + X \ltimes F_{\mathcal{T}} \\
&\quad \text{and } \mathit{factor}_X(E') = F' + X \ltimes F'_{\mathcal{T}} \\
\mathit{factor}_X(E \cdot E') &\stackrel{\text{def}}{=} (F \cdot F') + \left(X \ltimes \left(\begin{array}{r} F_{\mathcal{T}}(\underline{1} \odot F') \\ +_{\mathcal{T}} F'_{\mathcal{T}}(E \odot \underline{1}) \end{array} \right) \right) \\
&\quad \text{where } \mathit{factor}_X(E) = F + X \ltimes F_{\mathcal{T}} \\
&\quad \text{and } \mathit{factor}_X(E') = F' + X \ltimes F'_{\mathcal{T}} \\
\mathit{factor}_X(E^*) &\stackrel{\text{def}}{=} E^* + X \ltimes 0_{\mathcal{T}} \\
\mathit{factor}_X(E \ltimes E_{\mathcal{T}}) &\stackrel{\text{def}}{=} (F \ltimes E_{\mathcal{T}}) + X \ltimes (F_{\mathcal{T}} \cdot_{\mathcal{T}} E_{\mathcal{T}}) \\
&\quad \text{where } \mathit{factor}_X(E) = F + X \ltimes F_{\mathcal{T}}
\end{aligned}$$

Figure 2: $\mathit{factor}_X(E)$.

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

ALGORITHM 4.14 (Free-Variable Elimination). *The input is a system of equations $S : \{X_i = R_i\}_{i=1}^n$ with regular right-hand sides over an alphabet Σ and a finite set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$. We transform S into an equivalent system $\hat{S}$ with no free variables by eliminating variables one at a time, in a style reminiscent of Gauss-Jordan elimination. We use $S_k : \{X_i = E_{i,k}\}_{i=1}^n$ to denote the system of equations after k elimination rounds (we take $S_0 \stackrel{\text{def}}{=} S$ to be the original system of equations, noting that each regular expression R_i from the original system of equations is also a normal extended regular expression $E_{i,0}$).*

```
1. Repeat for each $k = 1$ to $n$:
    (a) Let $factor_{X_k}(E_{k,k}) = F + (X_k \ltimes F_{\mathcal{T}})$
    (b) Define $S_{k+1} : \{X_i = E_{i,k+1}\}_{i=1}^n$ by:
        • $E_{k,k+1} \stackrel{\text{def}}{=} F \ltimes F_{\mathcal{T}}^{*\mathcal{T}}$
        • For $i \neq k$, $E_{i,k+1}$ is obtained from $E_{i,k}$ by replacing each free occurrence of $X_k$ with $F \ltimes F_{\mathcal{T}}^{*\mathcal{T}}$.
2. Return $\hat{S} \stackrel{\text{def}}{=} S_{n+1}$.
```

THEOREM 4.15. *Given a system of equations $S : \{X_i = R_i\}_{i=1}^n$ with regular right-hand sides over an alphabet Σ and a finite set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$, Alg. 4.14 computes a system of equations $\hat{S} : \{X_i = \hat{E}_i\}_{i=1}^n$ where the right-hand sides are normal extended regular expressions over Σ and $\mathcal{X}$ with no free variables, and which is equivalent to S in the sense that $\mathit{Paths}_S(X_i) = \mathit{Paths}_{\hat{S}}(X_i)$ for all X_i.*

Thm. 4.15 shows that the transformation performed by Alg. 4.14 results in an equivalent system of equations. Following Obs. 2.1, any post-fixpoint solution to the transformed system overapproximates every path of the original equation system. §4.3.2 shows how to compute such a post-fixpoint solution.

EXAMPLE 4.16. *We now illustrate Alg. 4.14 on the recursive-descent parser shown in Fig. 3. The initial equation system has the form*

$$\begin{aligned}
E &= a \cdot T \cdot (b \cdot E + (c + 0)) \\
T &= d \cdot ((e + f \cdot E \cdot (g + 0)) + 0) \cdot (h \cdot T + (i + 0)),
\end{aligned}$$

where 0 = `abort()`, a = `cost++`, b = `currentToken == '+'` $\cdot$ `getNextToken()`, *etc. The equation system simplifies to*

$$E = a \cdot T \cdot (b \cdot E + c) \qquad (3)$$

$$T = d \cdot (e + f \cdot E \cdot g) \cdot (h \cdot T + i). \qquad (4)$$

Let E be variable X_1 and T be X_2. Step 1a of Alg. 4.14 applies factor_E to the right-hand side of Eqn. (3), to produce a new equation for E. After simplification, we have the following extended regular expression for E:

$$E = (a \cdot T \cdot c) \ltimes ((a \cdot T \cdot b) \odot 1)^{*\mathcal{T}}, \qquad (5)$$

Eqn. (5) is a non-recursive version of Eqn. (3): Eqn. (5) says that an E consists of some number—say, n—of copies of $a \cdot T \cdot b$, followed by $a \cdot T \cdot c$, followed by n copies of 1.[2] *Step 1b of Alg. 4.14 substitutes the right-hand side of Eqn. (5) for the free occurrence of E in Eqn. (4), and thus round 1 of step 1 of Alg. 4.14 produces Eqn. (5) and*

$$T = d \cdot (e + f \cdot [(a \cdot T \cdot c) \ltimes ((a \cdot T \cdot b) \odot 1)^{*\mathcal{T}}] \cdot g) \cdot (h \cdot T + i).$$

All occurrences of E have been eliminated, but three free occurrences of T and two guarded occurrences of T remain. The second and final round of Alg. 4.14 eliminates the free occurrences of T, producing

$$\begin{aligned}
E &= a \cdot ((d \cdot e \cdot i) \ltimes (\mathit{left} + \mathit{right})^{*\mathcal{T}}) \cdot c \ltimes ((a \cdot T \cdot b) \odot 1)^{*\mathcal{T}} \\
T &= (d \cdot e \cdot i) \ltimes (\mathit{left} + \mathit{right})^{*\mathcal{T}}, \text{ where} \\
\mathit{left} &= (a \odot c) \cdot_{\mathcal{T}} ((a \cdot T \cdot b) \odot 1)^{*\mathcal{T}} \cdot_{\mathcal{T}} ((d \cdot f) \odot (g \cdot i)) \\
\mathit{right} &= d \cdot (e + f \cdot [(a \cdot T \cdot c) \ltimes ((a \cdot T \cdot b) \odot 1)^{*\mathcal{T}}] \cdot g) \cdot h \odot 1.
\end{aligned}$$

As desired, all free occurrences of all variables have been eliminated. When the method described in §4.3.2 is used to solve this system, it computes summaries of procedures E and T. Using the domain of CRA, the summary is strong enough to prove that after parsing completes, `cost` $\leq$ `tokenCount` $+ 1$*—that is, the cost of parsing is linear in the number of tokens.*

4.3.2 Solving an equation system with no free variables

We now show how to solve a system of equations over extended regular expressions with no free variables. The key

[2] A similar characterization of E via an ordinary regular expression can be obtained by rewriting Eqn. (3) as $E = (a \cdot T \cdot b) \cdot E + (a \cdot T \cdot c)$, which has the solution $(a \cdot T \cdot b)^*(a \cdot T \cdot c)$.

```
void getNextToken() { tokenCount++; ... }
void E() { // E ::= T ('+' E)*
   cost++; T();
   if (currentToken == '+') {
      getNextToken(); E();
   } else if (currentToken == EOF_TOKEN ||
              currentToken == ')') return;
   else abort();
}
void T() { // T ::= ( ATOM | '(' E ')' ) ('*' T)*
   cost++;
   if (currentToken == ATOM) getNextToken();
   else if (currentToken == '(') {
      getNextToken(); E();
      if (currentToken == ')') getNextToken();
      else abort();
   } else abort();
   if (currentToken == '*') {
      getNextToken(); T();
   } else if (currentToken == EOF_TOKEN ||
              currentToken == '+' ||
              currentToken == ')') return;
   else abort();
}
```

Figure 3: A recursive-descent parser.

idea is to equip the tensor-product domain with an *iteration domain*, defined below.

DEFINITION 4.17. *Let $\mathcal{D} = \langle D, \equiv, \oplus, \otimes, *, \underline{0}, \underline{1}\rangle$ be a quasi-weight domain. An* ***iteration domain for*** *$\mathcal{D}$, $\mathcal{D}^\sharp = \langle D^\sharp, \leq^\sharp, \triangledown, \alpha, cl\rangle$, is a partially ordered set $\langle D^\sharp, \leq^\sharp\rangle$ along with the following operations.*

An ***abstraction*** *operator $\alpha : D \to D^\sharp$ and an* ***abstract-closure*** *operator $cl : D^\sharp \to D$ such that the following properties hold:*

1. *For all $a \in D$, $cl(\alpha(a)) = a^*$*
2. *For all $a^\sharp, b^\sharp \in D^\sharp$ such that $a^\sharp \leq^\sharp b^\sharp$ we have $cl(a^\sharp) \lesssim cl(b^\sharp)$.*

A ***widening*** *operator $\triangledown : D^\sharp \times D^\sharp \to D^\sharp$ such that the following properties hold:*

1. *For all $a^\sharp, b^\sharp \in D^\sharp$, $a^\sharp \triangledown b^\sharp$ is an upper bound of $a^\sharp$ and $b^\sharp$ ($a^\sharp \leq^\sharp a^\sharp \triangledown b^\sharp$ and $b^\sharp \leq^\sharp a^\sharp \triangledown b^\sharp$)*
2. *For every infinite sequence $\langle a_r^\sharp : D^\sharp\rangle_{r\in\mathbb{N}}$, the ascending chain $\langle b_r^\sharp : D^\sharp\rangle_{r\in\mathbb{N}}$ defined as*
$$b_1^\sharp = a_1^\sharp \qquad b_{r+1}^\sharp = b_r^\sharp \triangledown a_{r+1}^\sharp$$
eventually stabilizes.

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

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

$$\begin{aligned}
\mathit{eval}^k(a) &= [\![a]\!]\\
\mathit{eval}^k(X_i) &= \mathbf{D}^{k-1}(X_i)\\
\mathit{eval}^k(E \oplus F) &= \mathit{eval}^k(E) \oplus \mathit{eval}^k(F)\\
\mathit{eval}^k(E \otimes F) &= \mathit{eval}^k(E) \otimes \mathit{eval}^k(F)\\
\mathit{eval}^k(E^*) &= cl\big(\mathit{body}^k(E)\big)\\
\mathit{eval}^k(E \ltimes E_\mathcal{T}) &= \mathit{eval}^k(E) \ltimes \mathit{eval}^k_\mathcal{T}(E_\mathcal{T})\\
\mathit{eval}^k_\mathcal{T}(E \odot F) &= \mathit{eval}^k_\mathcal{T}(E) \odot \mathit{eval}^k_\mathcal{T}(F)\\
\mathit{eval}^k_\mathcal{T}(E_\mathcal{T} \oplus_\mathcal{T} F_\mathcal{T}) &= \mathit{eval}^k_\mathcal{T}(E_\mathcal{T}) \oplus_\mathcal{T} \mathit{eval}^k_\mathcal{T}(F_\mathcal{T})\\
\mathit{eval}^k_\mathcal{T}(E_\mathcal{T} \otimes_\mathcal{T} F_\mathcal{T}) &= \mathit{eval}^k_\mathcal{T}(E_\mathcal{T}) \otimes_\mathcal{T} \mathit{eval}^k_\mathcal{T}(F_\mathcal{T})\\
\mathit{eval}^k_\mathcal{T}(E_\mathcal{T}^{*_\mathcal{T}}) &= cl_\mathcal{T}(\mathit{body}^k_\mathcal{T}(E_\mathcal{T}))
\end{aligned}$$

$$\begin{aligned}
\text{and}\quad \mathit{body}^k(E) &= \mathit{body}^{k-1}(E) \triangledown \alpha([\![E]\!]_{\mathbf{D}^{k-1}})\\
\mathit{body}^k_\mathcal{T}(E_\mathcal{T}) &= \mathit{body}^{k-1}_\mathcal{T}(E_\mathcal{T}) \triangledown_\mathcal{T} \alpha_\mathcal{T}([\![E_\mathcal{T}]\!]_{\mathbf{D}^{k-1}})\ .
\end{aligned}$$

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

$$\mathit{body}^k(E) = \mathit{body}^{k-1}(E),$$

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

$$\mathit{body}^k_\mathcal{T}(E_\mathcal{T}) = \mathit{body}^{k-1}_\mathcal{T}(E_\mathcal{T})\ .$$

THEOREM 4.18. *There exists a k such that S stabilizes at k, and $\mathbf{D}^k$ is a post-fixpoint solution to S.*

EXAMPLE 4.19. *Ex. 4.13 is a high-level overview of how our approach analyzes the Fibonacci function. Here we illustrate one step of this process in greater detail: computing the third iterate $\mathbf{D}^3$ of the system $\hat{S}$. Write the system $\hat{S}$ as*

$$\hat{S} : \{X = \mathit{base} \ltimes (\mathit{rec}^{*_\mathcal{T}})\}\ ,\ \textit{where}$$

$$\begin{aligned}
\mathit{base} &\overset{\mathit{def}}{=} \texttt{n:=p} \cdot \texttt{n <= 1} \cdot \texttt{r:=1}\\
\mathit{rec} &\overset{\mathit{def}}{=} \begin{pmatrix} (\texttt{n:=p} \cdot \texttt{n > 1} \cdot \texttt{p:=n-1})\\ \odot\ (\texttt{t:=r} \cdot \texttt{p:=n-2} \cdot X \cdot \texttt{r:=r+t}) \end{pmatrix}\ .
\end{aligned}$$

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

$$\begin{aligned}
\mathbf{D}^3(X) &= \mathit{eval}^3\,(\mathit{base} \ltimes (\mathit{rec}^{*_\mathcal{T}}))\\
&= \mathit{eval}^3(\mathit{base}) \ltimes \mathit{eval}^3_\mathcal{T}(\mathit{rec}^{*_\mathcal{T}})\\
&= \mathit{eval}^3(\mathit{base}) \ltimes cl_\mathcal{T}(\mathit{body}^3_\mathcal{T}(\mathit{rec})).
\end{aligned}$$

The expression base contains no variables, so it can be reinterpreted within the CRA quasi-weight domain:

$$\begin{aligned}
\mathit{eval}^3(\mathit{base}) &= [\![\texttt{n:=p}]\!] \otimes [\![\texttt{n <= 1}]\!] \otimes [\![\texttt{r:=1}]\!]\\
&= \big(\texttt{n}' = \texttt{p} = \texttt{p}' \wedge \texttt{n}' \leq 1 \wedge \texttt{r}' = 1 \wedge \texttt{t}' = \texttt{t}\big)
\end{aligned}$$

The term $\mathit{body}^3_\mathcal{T}(\mathit{rec})$ refers to the third-iteration loop-body polyhedron (called β^3 in Ex. 4.13). Following the definition *of $body_{\mathcal{T}}$ above, we have*

$$body^3_{\mathcal{T}}(rec) = body^2_{\mathcal{T}}(rec) \nabla_{\mathcal{T}} \widetilde{\alpha}_{poly,\mathcal{T}}([\![rec]\!]_{\mathbf{D}^2}) .$$

where $body^2_{\mathcal{T}}(rec)$ is the second-iteration loop-body polyhedron (β^2 in Ex. 4.13) and $[\![rec]\!]_{\mathbf{D}^2}$ is the value obtained by evaluating rec in the tensor-product domain of CRA, using $\mathbf{D}^2(X)$ to interpret the variable X:

$$\begin{aligned}
[\![rec]\!]_{\mathbf{D}^2} &= \begin{pmatrix} ([\![\texttt{n:=p}]\!] \otimes [\![\texttt{n > 1}]\!] \otimes [\![\texttt{p:=n-1}]\!]) \\ \odot\ ([\![\texttt{t:=r}]\!] \otimes [\![\texttt{p:=n-2}]\!] \otimes \mathbf{D}^2(X) \otimes [\![\texttt{r:=r+t}]\!]) \end{pmatrix} \\
&\quad (\mathtt{n}' = \mathtt{p} \wedge \mathtt{n}' > 1 \wedge \mathtt{p}' = \mathtt{n}' - 1 \wedge \mathtt{r}' = \mathtt{r} \wedge \mathtt{t}' = \mathtt{t}) \\
&= \odot \begin{pmatrix} (\mathtt{t}' = \mathtt{r} \wedge \mathtt{n}' = \mathtt{n}) \\ \wedge \begin{bmatrix} (\mathtt{p}' = \mathtt{n} - 2 \leq 1 \wedge \mathtt{r}' = 1 + \mathtt{t}') \\ \vee (\exists k . k \geq 1 \wedge k = \mathtt{n}-3 \wedge \mathtt{r}' = k+1+\mathtt{t}' \wedge \mathtt{n} \leq 5) \end{bmatrix} \end{pmatrix} \\
&= \begin{pmatrix} (\underline{\mathtt{n}}' = \underline{\mathtt{p}} \wedge \underline{\mathtt{n}}' > 1 \wedge \underline{\mathtt{p}}' = \underline{\mathtt{n}}' - 1 \wedge \underline{\mathtt{r}}' = \underline{\mathtt{r}} \wedge \underline{\mathtt{t}}' = \underline{\mathtt{t}}) \\ \wedge (\overline{\mathtt{t}}' = \overline{\mathtt{r}} \wedge \overline{\mathtt{n}}' = \overline{\mathtt{n}}) \\ \wedge \begin{bmatrix} (\overline{\mathtt{p}}' = \overline{\mathtt{n}} - 2 \leq 1 \wedge \overline{\mathtt{r}}' = 1 + \overline{\mathtt{t}}') \\ \vee (\exists k . k \geq 1 \wedge k = \overline{\mathtt{n}}-3 \wedge \overline{\mathtt{r}}' = k+1+\overline{\mathtt{t}}' \wedge \overline{\mathtt{n}} \leq 5) \end{bmatrix} \end{pmatrix}
\end{aligned}$$

We then project out the local variables n *and* t *and apply the abstraction operator to obtain the following polyhedron:*

$$\widetilde{\alpha}_{poly,\mathcal{T}}([\![rec]\!]_{\mathbf{D}^2}) = (\underline{\mathtt{p}}' = \underline{\mathtt{p}} - 1 \wedge 1 < \underline{\mathtt{p}} \leq 5 \wedge \underline{\mathtt{r}}' = \underline{\mathtt{r}} \wedge \overline{\mathtt{r}}' \geq \overline{\mathtt{r}} + 1).$$

Returning to the evaluation of $body^3_{\mathcal{T}}(rec)$, we apply polyhedra widening to the previous iterate β^2 and $\widetilde{\alpha}_{poly,\mathcal{T}}([\![rec]\!]_{\mathbf{D}^2})$:

$$\begin{aligned}
body^3_{\mathcal{T}}(rec) &= \beta^2 \nabla_{\mathcal{T}} \widetilde{\alpha}_{poly,\mathcal{T}}([\![rec]\!]_{\mathbf{D}^2}) \\
&= (\underline{\mathtt{p}}' = \underline{\mathtt{p}} - 1 \wedge 1 < \underline{\mathtt{p}} \wedge \underline{\mathtt{r}}' = \underline{\mathtt{r}} \wedge \overline{\mathtt{r}}' \geq \overline{\mathtt{r}} + 1)
\end{aligned}$$

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

$$\begin{aligned}
\mathbf{D}^3(X) &= eval^3(base) \ltimes cl_{\mathcal{T}}(body^3_{\mathcal{T}}(rec)) \\
&= \begin{pmatrix} \mathtt{p}' = \mathtt{p} \leq 1 \wedge \mathtt{r}' = 1 \\ \vee\ \exists k . k \geq 1 \wedge \mathtt{r}' \geq k + 1 \wedge k = \mathtt{p} - 1 \end{pmatrix} .
\end{aligned}$$

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

ALGORITHM 4.20 (NPA-TP-GJ). *The input is a system of equations $S : \{X_i = R_i\}_{i=1}^n$ with regular right-hand sides over an alphabet Σ and a finite set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$, a tensor-product domain $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_{\mathcal{T}}, \odot, \ltimes \rangle$, and an interpretation $[\![\cdot]\!] : \Sigma \to \mathcal{D}$. The output is a mapping $\mathbf{D} : \mathcal{X} \to \mathcal{D}$ such that for all i and all $p \in Paths_S(X_i)$, $\mathbf{D}(X_i) \gtrsim [\![p]\!]$.*

```
1. Use Alg. 4.14 to transform $S$ into a system of equations
   $\hat{S} : \{X_i = \hat{E}_i\}_{i=1}^n$, in which each $\hat{E}_i$ is an extended reg-
   ular expression over $\Sigma$ and $\mathcal{X}$ that has no free variables.
2. $k \leftarrow 0$; $\mathbf{D}^0 \leftarrow \lambda x.\underline{0}$
3. Repeat
     (a) $k \leftarrow k + 1$
     (b) $\mathbf{D}^k(X_i) \leftarrow eval^k(\hat{E}_i)$
   until $\hat{S}$ stabilizes
4. Return $\mathbf{D}^k$
```

In the degenerate case when S is linear, left-linear, or right-linear, the call on Alg. 4.14 in step (1) returns a solution to S (or, more precisely, a system of equations $\hat{S}$ in which the right-hand-sides do not contain *any* variables). In this case, the evaluation loop stabilizes on the first pass.

5. Implementation and Experiments

NPA-TP-GJ is implemented on top of the WALi [25] system for weighted pushdown systems. In particular, it uses the implementation of Tarjan's method from the FWPDS solver [29] of WALi to create the initial equation system with regular right-hand sides from a specification of the problem to be solved as a weighted pushdown system. We instantiated NPA-TP-GJ for ICRA by augmenting the implementation of CRA [14] (which, in turn, makes use of APRON [1] and Z3 [12]) with $\odot$ and $\ltimes$ to create a tensor-product domain, and by implementing a subclass of WALi's tensor-product domain interface that makes appropriate calls to operations of the CRA abstract domain.

The implementation supports programs with multiple procedures, including recursion and mutual recursion. It supports local variables via the method explained in [35, §8]. The merge function used when returning from a procedure call is based on [35, Eqn. (54)].

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

1. How fast is ICRA, compared to other tools?
2. How many assertions can ICRA prove, compared to other tools?
3. How often is ICRA able to prove bounds on a program's resource usage, compared with C4B [7]?

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

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

Assertion Checking. To assess ICRA's assertion-checking capabilities, we ran it on (i) the benchmarks in the *Loops* and *Recursive* subcategories of the *Integers and Control Flow* category from SV-COMP16 [39]; (ii) the suite of programs from C4B [8, App. A], annotated with upper-bound resource assertions of the bounds reported by C4B;[3] and (iii) three groups of new recursive programs: two groups are modified versions of the SV-COMP16 subcategories *loop-lit* and *loop-new* in which all loops were converted into recursive procedures; the third group consists of a few newly-written recursive programs, plus a few recursive programs from the test suite of the CRA static-analysis tool. We compared ICRA's results against four state-of-the-art software model checkers: CPAchecker [3] and Ultimate Automizer [22], based on predicate abstraction; LPI [24], based on policy iteration;

[3] C4B obtains bounds for 34 of the 35 programs. The assertion in the 35th program is the bound from SPEED [20].

Benchmark	Total	ICRA		UAut.		CPA		LPI		SEA	
Suite	#A	Time	#A	Time	#A	Time	#A	Time	#A	Time	#A
recursive	18/7	**40.7**	7	1952.1	8	1817.8	10	62.0	0	1334.0	**14**
rec.-simple	36/38	**168.7**	21	6979.3	28	2760.4	32	179.5	3	743.8	**36**
Rec. (tot.)	54/45	**209.4**	28	8931.4	36	4578.1	42	241.5	3	2077.8	**50**
loop-accel.	19/16	**20.8**	13	6696.5	7	4565.7	13	4227.7	13	2713.1	**15**
loop-invgen	18/1	**53.1**	**16**	1876.2	7	4909.6	2	1282.3	15	506.0	**16**
loop-lit	15/1	316.5	12	2722.9	5	2720.6	7	444.9	**13**	**305.2**	**13**
loops	34/32	**209.7**	22	3984.1	19	4380.1	**28**	3356.8	26	1821.5	27
loop-new	8/0	304.8	**7**	2147.9	1	1866.1	3	929.6	4	**302.8**	6
Loops (tot.)	94/50	**904.8**	70	17427.6	39	18442.2	53	10241.3	71	5648.6	**77**
C4B	35/0	**30.3**	**30**	6156.6	1	7817.8	2	6726.7	0	1867.6	29
misc	10/4	76.7	**10**	492.2	8	334.4	7	332.2	1	**5.3**	**10**
rec-loop-lit	15/1	312.7	9	2755.5	3	51.0	6	**40.4**	0	922.6	**12**
rec-loop-new	8/0	**6.2**	**5**	1546.9	2	25.6	2	19.6	0	905.7	4
Misc.-Rec.	33/5	395.6	24	4794.6	13	410.9	15	**392.2**	1	1833.7	**26**

Table 1: Column 2 shows the breakdown of programs into ones with valid and invalid assertions. "X/Y" means that X is the number of programs containing valid assertions and Y is the number containing invalid assertions. The total number of programs is X + Y. The two columns for each tool show the running time (in seconds) and the number of assertions proved. In each row, the fastest time and the greatest number of assertions proved are shown in bold font.

and SeaHorn [21], a Horn-clause solver based on IC3. We ran the versions of these tools that were submitted to SV-COMP16.

Tab. 1 shows that the answers to questions 1 and 2 are
(1) Overall ICRA was faster, sometimes dramatically so. The main reason is that ICRA had many fewer timeouts.
(2) ICRA correctly handled more valid assertions than all the other tools except for SeaHorn.

The five tools have different success/failure-versus-timeout profiles, and there are examples on which one tool succeeds quickly but the other tools time out or fail. For instance, ICRA took only 0.73 seconds to prove the assertion in rec-gauss_sum_true-unreach-call (shown below), and was the only tool to do so. This benchmark, from *rec-loop-new*, is a recursive equivalent of an iterative SV-COMP16 benchmark.

```
int n, sum, i;
void rec() {
  if (i <= n) { sum = sum + i; i++; rec(); }
}
int main() {
  n = __VERIFIER_nondet_int();
  __VERIFIER_assume(1 <= n && n <= 1000);
  sum = 0; i = 1; rec();
  __VERIFIER_assert(2*sum == n*(n+1));
  return 0;
}
```

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

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

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

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

The procedure in Fig. 4(a) illustrates the application of ICRA to resource-bound analysis. Statements of the form `tick(`k`)` represent manipulations of a resource such as memory or time, by consuming some of the resource (if k is positive) or freeing some of the resource (if k is negative).

ICRA can compute terms that upper-bound and lower-bound both the *final* resource usage and the *high-water mark* of resource usage.[4] When computing bounds, it is useful to treat positive and negative values of a program variable separately, so ICRA searches for bounds that include terms of the form $max(0, x)$ and $max(0, y-x)$ (as in [8]), and we use the notation $|[0, x]|$ and $|[x, y]|$, respectively, for such terms. For the function `perform`, ICRA computes the following four bounds, each of which is achieved by following paths of a particular form:

Bound	Path (line numbers)
final $\geq 4|[0, n]| + 4$	$1 \rightarrow 2 \rightarrow 4 \rightarrow$ `perform(n-1)` $\rightarrow 5$
final $\leq 6|[0, n]| + 6$	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow$ `perform(n-1)` $\rightarrow 5$
hwm $\geq 7|[0, n]| + 7$	$1 \rightarrow 2 \rightarrow 4 \rightarrow$ `perform(n-1)` $\rightarrow 5$
hwm $\leq 9|[0, n]| + 9$	$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow$ `perform(n-1)` $\rightarrow 5$

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

[4] The result of using ICRA for resource-bound analysis is only sound under the assumption that, for all inputs, the program in question terminates.

```
void perform(int n) {
  tick(7);
  if(*)
    tick(2);
  if(n>0) perform(n-1);
  tick(-3);
}
```

(a)

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

(b)

Figure 4: (a) Recursive procedure to show upper and lower resource bounds; (b) speed_popl10_simple_single_2.c from C4B.

On the C4B test suite, ICRA is able to generate upper bounds on resource usage for 28 of the 35 programs, and nontrivial lower bounds on resource usage for 30 of the programs. Whenever ICRA generates an upper bound, it is the same as C4B's bound (modulo four cases where it was unclear where to materialize C4B's implicit `tick` call, which caused ICRA's bound to be one unit different than C4B's bound). In seven cases, ICRA's bound is more precise: it generates both C4B's bound, plus one or more additional incomparable constraints. For example, for the procedure shown in Fig. 4(b), ICRA generates the following bounds:

$$\mathit{final} \leq |[0,n]| + |[n,m]| \qquad \mathit{final} \leq |[0,m]| + |[m,n]|$$
$$\mathit{final} \leq |[0,n]| + |[0,m]|$$

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

6. Related Work

The NPA-TP-GJ algorithm and its instantiation for ICRA rely heavily on prior work on CRA [14], NPA [13], and NPA-TP [35]. In particular, the NPA-TP-GJ algorithm adopts NPA-TP's tensor-product operation so that various terms in a non-linear equation can be rewritten into the "symbolically" left-linear form $X = F \oplus X \ltimes F_{\mathcal{T}}$, and then further rewritten as $X = F \ltimes F_{\mathcal{T}}{}^{*_{\mathcal{T}}}$.

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

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

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

ICRA relies on the fact that the logic fragment that CRA employs to solve recurrences supports effective equivalence. That property—along with the widening performed during the *body* subroutine of *eval*, and the stabilization result given in Thm. 4.18—ensures that ICRA will always terminate.

The elimination step in NPA-TP-GJ is related to Gauss-Jordan elimination, in that variables are eliminated in some order. Unlike in Gauss-Jordan elimination, variables are only partially eliminated; so-called "free variables" are eliminated, whereas variables that appear under * or $^{*_{\mathcal{T}}}$ are not eliminated. Past work on Gauss-Jordan elimination for semirings or regular expressions includes [40], [17, §4.5], [36, §4], [9, §7], and [2, §4].

The power of CRA is due to the ability of the iteration operator to summarize loops. There are a number of related techniques for loop summarization [4, 19, 28] and abstract loop-acceleration [18, 23, 32]. (The latter computes the *post-image* of a loop, rather than a summary.) Loop summarization in CRA is based on computing the closed forms for recurrences satisfied by loop bodies. ICRA harnesses the power of CRA's loop summarization for computing summaries of recursive procedures by computing abstract transitive closures of tensored transition formulas; i.e., in ICRA, the input and output of the iteration operator are formulas that each have four distinct vocabularies.

Ganty et al. [16] have created a Newton-based analysis tool for recursive programs over the integers. Like ICRA, it repeatedly applies a more basic analyzer to a transformed version of the original program. However, there are several differences in capabilities and approach.

- Their tool computes *underapproximations* of procedures; ICRA computes *overapproximations*.
- When used with FLATA [6], their tool is limited to finding summaries that are linear inequalities with at most two variables. ICRA can compute polynomial summaries.
- Their tool performs a transformation that restructures the linearized, possibly recursive, multi-procedure program P_k obtained for Newton-round k into a non-recursive program P'_k. The transformation introduces occurrences of `havoc` and `assume` so that, during the analysis of P'_k, the various parts of P_k are analyzed in an order different from the order in which they would be executed in P_k. In contrast, ICRA is based on algebraic program analysis, and relies on the properties of its tensor-product domain to tease out the properties of the original program from the restructured equation system that it creates.

The relationship of ICRA to Ganty et al. is similar to the relationship of Lal et al. [31] to Lal and Reps [30]. The subjects in the two pairs of papers are different—interprocedural analysis of sequential programs and analysis of concurrent programs, respectively. However, in each pair, the first paper performs algebraic program analysis using a tensor-product domain, whereas the second uses a code transformation.

Some additional related work is discussed in [27, §6].

Acknowledgments

We thank the anonymous reviewers and shepherd A. Bouajjani for their comments on the submission.

References

[1] APRON. APRON numerical abstract domain library. URL http://apron.cri.ensmp.fr/library/.

[2] R. Backhouse and B. Carré. Regular algebra applied to path-finding problems. *J. Inst. Maths. Applics.*, 15, 1975.

[3] D. Beyer and M. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, 2011.

[4] S. Biallas, J. Brauer, A. King, and S. Kowalewski. Loop leaping with closures. In *SAS*, 2012.

[5] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR*, 1997.

[6] M. Bozga, R. Iosif, F. Konečný, and T. Vojnar. Tool demonstration of the FLATA counter automata toolset. In *Workshop on Invariant Generation*, 2012.

[7] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. In *PLDI*, 2015.

[8] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds (extended version). YALEU/DCS/TR-1505, Yale Univ., New Haven, CT, Apr. 2015.

[9] B. Carré. An algebra for network routing problems. *J. Inst. Maths. Applics.*, 7, 1971.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, 1978.

[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[13] J. Esparza, S. Kiefer, and M. Luttenberger. Newtonian program analysis. *J. ACM*, 57(6), 2010.

[14] A. Farzan and Z. Kincaid. Compositional recurrence analysis. In *FMCAD*, 2015.

[15] A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *ENTCS*, 9, 1997.

[16] P. Ganty, R. Iosif, and F. Konečný. Underapproximation of procedure summaries for integer programs. *Softw. Tools for Tech. Transfer*, 2016. Corrected version available as arXiv:1210.4289v3 (10.1007/s10009-016-0420-7).

[17] M. Gondran and M. Minoux. *Graphs, Dioids and Semirings: New Models and Algorithms*. Springer-Verlag, 2010.

[18] L. Gonnord and P. Schrammel. Abstract acceleration in linear relation analysis. *SCP*, 93, 2014.

[19] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, 2010.

[20] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, 2009.

[21] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. Navas. The SeaHorn verification framework. In *CAV*, 2015.

[22] M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. Ultimate Automizer with SMTInterpol (competition contribution). In *TACAS*, 2013.

[23] B. Jeannet, P. Schrammel, and S. Sankaranarayanan. Abstract acceleration of general linear loops. In *POPL*, 2014.

[24] E. Karpenkov, D. Monniaux, and P. Wendler. Program analysis with local policy iteration. In *VMCAI*, 2016.

[25] N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. URL http://www.cs.wisc.edu/wpis/wpds/download.php.

[26] G. Kildall. A unified approach to global program optimization. In *POPL*, 1973.

[27] Z. Kincaid, J. Breck, A. Forouhi Boroujeni, and T. Reps. Compositional recurrence analysis revisited. TR-1840, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, Oct. 2016. Revised, Apr. 2017.

[28] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. Wintersteiger. Loop summarization using abstract transformers. In *ATVA*, 2008.

[29] A. Lal and T. Reps. Improving pushdown system model checking. In *CAV*, 2006.

[30] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35(1):73–97, 2009.

[31] A. Lal, T. Touili, N. Kidd, and T. Reps. Interprocedural analysis of concurrent programs under a context bound. In *TACAS*, 2008.

[32] J. Leroux and G. Sutre. Accelerated data-flow analysis. In *SAS*, 2007.

[33] T. Reps. Program analysis via graph reachability. *IST*, 40, 1998.

[34] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP*, 58, 2005.

[35] T. Reps, E. Turetsky, and P. Prabhu. Newtonian program analysis via tensor product. In *POPL*, 2016.

[36] G. Rote. Path problems in graphs. In *Computational Graph Theory (Computing Supplementum 7)*. Springer-Verlag, 1990.

[37] B. Ryder and M. Paul. Elimination algorithms for data flow analysis. *ACM Comput. Surv.*, 18(3):277–316, 1986.

[38] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[39] SVCOMP16. 5th Int. competition on software verification (SV-COMP16), 2016. URL https://sv-comp.sosy-lab.org/2016/.

[40] R. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.