Compositional Recurrence Analysis Revisited *

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

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

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

Abstract

Compositional recurrence analysis (CRA) is a static-analysis method based on an interesting 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 address this issue by showing that we can make use of a recently developed framework—Newtonian Program Analysis via Tensor Product (NPA-TP)—that reconciles this impedance mismatch when the abstract domain supports a few special operations. Our approach introduces new problems that are not addressed by NPA-TP; however, we are able to resolve those problems. We call the resulting algorithm Interprocedural CRA (ICRA).

Our experimental study of ICRA shows that it has broad overall strength. The study showed that ICRA is both faster and handles more assertions than two state-of-the-art software model checkers. It also performs well when applied to the problem of establishing bounds on resource usage, such as memory used or execution time.

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*, in which the program is executed over an abstract domain that (deliberately) leaves out certain details of concrete execution states so that the analyzer can explore a program's behavior for all possible inputs and all states that the program can reach (as well as some unreachable states);
- *symbolic analysis*, which uses logical formulas to create precise models of a program's actions, but is usually forced to forgo an exploration that accounts for all of the reachable states.

Compositional recurrence analysis (CRA) [13] is a static-analysis method based on an interesting 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 procedure's actions. It carries out a bottom-up exploration that accounts for all of a procedure's reachable states by performing a non-trivial abstraction step at each loop: the formula for the loop body is converted into a system of linear recurrences, which are then solved to create a summary transformer that involves polynomial constraints. So that the analyzer can continue its bottom-up analysis—proceeding upward to analyze the loop's context—the summary transformer is converted into a transition formula.

The CRA domain meets the conditions required to apply Tarjan's path-expression method [33] for solving single-procedure dataflow-analysis problems. For each node n, a regular expression R_n is created that summarizes all paths from entry to n. In CRA, the regular expressions are then evaluated using an interpretation of the regular operators $+$, $\cdot$, and * as disjunction, sequential composition, and the generation and solving of an appropriate system of recurrence equations, respectively.

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

- At the intraprocedural level, CRA makes use of Tarjan's path-expression method to compose the meanings of sub-parts via the interpretations of $+$, $\cdot$, and * [13].
- 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 recoded 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. To solve this problem, 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, 14, 27, 28, 31]. This issue is challenging because at the technical level, the regular-

* 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.

language viewpoint is baked into CRA: CRA's recurrence-solving step is coupled with the interpretation of Kleene-star (*).

The inspiration for our work was to use the recently developed framework for Newtonian Program Analysis via Tensor Product (NPA-TP) [29] to address the aforementioned impedance mismatch. 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 does support these non-standard operations (see §4.3), it fails to meet some "standard" properties. CRA unfortunately has infinite ascending chains, and thus NPA-TP[CRA] would not be guaranteed to terminate. Moreover, CRA does not have an effective entailment procedure, and thus it is not possible, in general, to ascertain whether an NPA-TP[CRA] analyzer has reached a fixed point.

We address these issues by developing an algorithm that adopts, but adapts in significant ways, ideas used in the NPA-TP framework. (See §4.4.) The resulting algorithm, which we call NPA-TP-GJ (Alg. 4.17) extends the ideas of NPA-TP to make it even more generally applicable: Kleene iteration, NPA, and NPA-TP do not converge at all for the class of domains considered by NPA-TP-GJ (and in particular, the CRA domain). The problem that NPA-TP-GJ addresses is how to both detect and enforce convergence when working with an abstract domain that has neither decidable equivalence nor the ascending-chain condition.

NPA-TP-GJ has broad overall strength. Our experiments showed that NPA-TP-GJ instantiated for Interprocedural CRA (ICRA) is both faster and handles more assertions than two state-of-the-art software model checkers. (See §5.1.)

ICRA also provides a new, systematic approach to the problem of establishing bounds on resource usage, such as memory used or execution time. Recent work by Hoffmann et al. [6, 20] on establishing resource-usage bounds is able to provide polynomial upper bounds on the resource usage of programs with several arguments. It can even succeed when the bound requires an amortized-cost analysis. Hoffmann et al. use ad hoc techniques to analyze procedures (and recursive procedures), whereas ICRA is based on the classical idea of creating behavior summaries for procedures, including recursive procedures. In the case of resource-bound analysis, the summary for a procedure P characterizes the change in available resources caused by invoking P (and all procedures transitively invoked by P). Moreover, by approaching resource-bound analysis as a problem of finding invariant polynomial inequalities, ICRA is able to obtain *lower bounds* on the resources used, as well as *upper bounds*, in a uniform way. (See §5.2.)

Contributions. Overall, our work makes three main contributions:

- We extend CRA to create a context-sensitive interprocedural version that handles loops and recursion—including non-linear recursion—in a *uniform* way (§4.3 and §4.4).
- We define an alternative to NPA-TP that can be used when the abstract domain of interest has infinite ascending chains and does not support effective entailment (§4.4). The method retains several ideas from NPA-TP, including that of using tensor products to rearrange the terms appearing in a system of (in)equations so that certain operations can be postponed.
- We report on the results of an experimental study (§5). The study showed that ICRA is both faster and handles more assertions than two state-of-the-art software model checkers. ICRA also performs well when applied to the problem of establishing resource-usage bounds.

Related work is discussed in §6. §7 concludes.

2. Background

2.1 Compositional Recurrence Analysis (CRA)

Tarjan's path-expression method [33] is an alternative to the classic iterative style of program analysis. In the iterative style, a program analysis is described by a lattice of program properties and an abstract transformer over the lattice. An abstract meaning of a program is computed by iterating the abstract transformer until a fixed point is reached (possibly using a widening operator to ensure convergence) [9]. In the algebraic style (following Tarjan), a program analysis is described by a *semantic algebra*: an algebraic structure whose elements represent path properties, and which is equipped with operators for composing properties via sequencing ($\otimes$), choice ($\oplus$), and Kleene-star (*). Computing an abstract meaning of a program can be seen as a two-step process. The first step computes a *path expression* for the program, which is a regular expression that recognizes the set of paths through the program's control-flow graph. The second step evaluates the path expression over the semantic algebra (i.e., using the algebra's operations to interpret the regular-expression operations) to arrive at a property that holds for all paths recognized by the path expression.

CRA follows this algebraic style [13]. The space of program 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 $\phi(\mathbf{x}, \mathbf{x}')$ 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. The sequencing operation is relational composition and choice is disjunction.[1]

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

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 works by using an SMT solver to extract a system of recurrences entailed by ϕ_{body}, and then using the closed form of the system as the abstraction of the loop.

We illustrate the high-level idea of how CRA analyzes loops using the example below. (See [13] for algorithmic details.)

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

CRA computes a transition formula for the above loop by applying the iteration operator to a transition formula representing its body:

$$\phi_{body} : \mathtt{i} > 0 \wedge \mathtt{i}' = \mathtt{i} - 1 \wedge \big((\mathtt{x}' = \mathtt{x} + \mathtt{i} \wedge \mathtt{y}' = \mathtt{y}) \vee (\mathtt{x}' = \mathtt{x} \wedge \mathtt{y}' = \mathtt{y} - \mathtt{i})\big) .$$

The iteration operator extracts the following recurrence (in)equations from ϕ_{body}, and computes their closed forms using symbolic summation:

[1] In the implementation of sequential composition (and that of the detensor-transpose operation defined in §4.3), 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.

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

(where $\mathtt{i}^{(k)}$ denotes the value of $\mathtt{i}$ on the k^{th} iteration of the loop). Finally, to compute a transition formula representing any number of iterations of ϕ_{body}, the iteration operator introduces an existentially quantified non-negative iteration variable k, and conjoins the closed form of every recurrence:

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

CRA is *compositional* in the sense that it computes the abstract meaning of a program by breaking it into parts, computing a meaning for each part, and then composing the meanings (via $\otimes$, $\oplus$, and *). This property makes it a natural fit for interprocedural analysis, because a procedure can be analyzed independently of its context to produce a summary that can be used to interpret its calls. *Recursive* procedures, however, present an obstacle. The set of paths through a program that uses recursion is not regular, but rather context-free [27, 31], which places recursive behavior beyond the scope of what can be analyzed using CRA's iteration operator. Recently, however, Reps et al. [29] showed that when a semantic algebra also provides a *tensor-product* operation, tensor-products can be used to convert linear context-free systems of (in)equations into *regular* systems of (in)equations, opening the door to applying CRA's loop analysis to recursive procedures.

2.2 Newtonian Program Analysis via Tensor Product (NPA-TP)

Esparza et al. [11, 12] generalized Newton's method—the classical numerical-analysis algorithm for finding roots of real-valued functions—to a method, called *Newtonian Program Analysis* (NPA), for finding fixed-points of systems of inequalities over semirings. NPA provides a new way to solve interprocedural dataflow-analysis problems. As in its real-valued counterpart, each iteration of NPA solves a simpler "linearized" problem.

NPA solves systems of polynomial inequalities over semirings. Program-analysis problems give rise to such systems as follows: if P calls Q twice on some path, one has the inequality $X_P \gtrsim X_Q \otimes X_Q$, which is quadratic. If there are actions a, b, and c before and after the calls, the inequality would be

$$X_P \gtrsim a \otimes X_Q \otimes b \otimes X_Q \otimes c. \tag{1}$$

In contrast, an inequality is *linear*[2] if each summand on the right-hand side contains only a single occurrence of a variable. For instance, the following (recursive) linear inequality

$$X_P \gtrsim \begin{array}{l}(a \otimes X_Q \otimes b) \oplus (c \otimes X_Q \otimes d) \\ \oplus\ (e \otimes X_R \otimes f) \oplus (g \otimes X_P \otimes h).\end{array}$$

corresponds to P calling Q twice in different branches, R once in a third branch, and P once in yet a fourth branch.

NPA is "Newtonian" because—like Newton's method for numerical analysis—it performs a succession of rounds; on each round, it uses a linear model (a "differential"[3])—built from the system of inequalities, together with the current approximation to the answer—to find the next approximation to the answer.

Operationally, NPA performs a kind of "linear-language sampling" of the state space of a program: if procedure P has multiple call-sites along a given path, then the linearized program used during a given round of Newton's method allows the analyzer to sample the state space of P by taking the $\oplus$ of various linear-language paths through P. Along each such path through P, the abstract values for the call-sites encountered are held fixed, except for possibly one call-site on the path, which is explored by visiting (the linear model of) the called procedure. The abstract value for P and all other procedures are updated according to the results of this state-space exploration, and the algorithm proceeds to the next Newton round. For instance, for Eqn. (1), the linear model would be

$$Y_P \gtrsim (a \otimes \underline{\nu}_Q \otimes b \otimes Y_Q \otimes c) \oplus (a \otimes Y_Q \otimes b \otimes \underline{\nu}_Q \otimes c)$$

(where $\underline{\nu}_Q$ denotes the value for Y_Q obtained on the previous round), which has just the one variable Y_Q in each summand.

A program with recursion is modeled with a recursive inequality, e.g.,

$$X \gtrsim d \oplus a \otimes X \otimes b \otimes X \otimes c. \tag{2}$$

The associated linear model is the recursive linear inequality

$$Y \gtrsim d \oplus (a \otimes \underline{\nu} \otimes b \otimes Y \otimes c) \oplus (a \otimes Y \otimes b \otimes \underline{\nu} \otimes c). \tag{3}$$

There is an important difference between the dataflow-analysis and numerical-analysis contexts: when Newton's method is used on numerical-analysis problems, multiplicative commutativity is relied on to rearrange expressions of the form "$e \otimes Y \otimes f$" into "$e \otimes f \otimes Y$." An inequality of the form "$Y \gtrsim g \oplus e \otimes f \otimes Y$" is similar to the right-linear grammar "$Y \rightarrow g \mid e\ f\ Y$," which corresponds to the regular language $L(Y) = (ef)^*g$. In contrast, when Newton's method is used for interprocedural dataflow analysis, the "multiplication" operation involves function composition, and hence is non-commutative: "$Y \gtrsim g \oplus e \otimes Y \otimes f$" cannot be rearranged into "$Y \gtrsim g \oplus e \otimes f \otimes Y$."

The inequality "$Y \gtrsim g \oplus e \otimes Y \otimes f$" is similar to the linear context-free grammar "$Y \rightarrow g \mid e\ Y\ f$." The latter describes the linear context-free language (LCFL) $L(Y) = \{e^i g f^i\}$, each word of which has "mirrored symmetry." Consequently, the linear models that arise in NPA, such as Eqn. (3), correspond to LCFLs with mirrored symmetry (albeit somewhat more complicated mirrored symmetry than $L(Y)$).

In a generalization of the work of Esparza et al. called *Newtonian Program Analysis via Tensor Product* (NPA-TP), Reps et al. [29] presented a way to convert each LCFL problem into a regular-language problem. Their construction allows them to apply Tarjan's path-expression method [33] as an *interprocedural* analysis engine on each Newton round.

Given that the language $a^i b^i$ is the canonical example of an LCFL that is not regular, the NPA-TP transformation sounds impossible. The secret is that one is not working with words: the

[2] In this paper, there is the potential for confusion between similar terms that are used for different things in CRA and NPA-TP. In particular, in the context of NPA-TP, "linear" and "polynomial" refer to the number of occurrences of variables on the right-hand sides of inequalities that capture a language of paths through the program. In the context of CRA, "linear" and "polynomial" refer to the properties of terms in the arithmetic transition formulas that encode individual statement actions, loop summaries, or procedure summaries. The intended meaning should be clear from context.

[3] In NPA, the notion of a "differential" is not based on freshman calculus (so no limits as $\Delta x \rightarrow 0$). Instead, NPA just adopts the rewriting rules for creating differentials (which are similar to the familiar rules for differentiation) to define a transformation on expressions that is used to create the linear model [11, 12, 29].

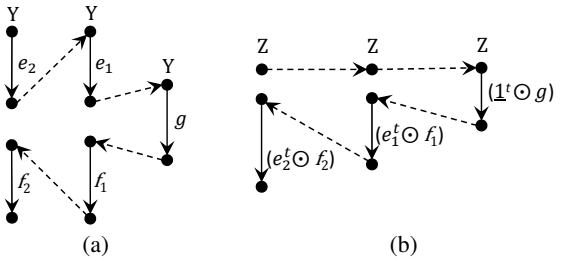

Figure 1: Graphical representations of (a) an unrolled LCFL path in the Y system of inequalities; and (b) the corresponding unrolled regular-language path in the system of Z inequalities. The paths have the values $e_2 \otimes e_1 \otimes g \otimes f_1 \otimes f_2$ and $(\underline{1}^t \odot g) \otimes (e_1^t \odot f_1) \otimes (e_2^t \odot f_2)$, respectively.

sum and product operations are *interpreted*. Also, NPA-TP requires the semiring to support a few additional operations—called "transpose" ($\cdot^t$), "tensor product" ($\odot$), and "detensor" ($\not\downarrow^{(t,\cdot)}$)—that one does not have with words. However, one does have such operations for predicate-abstraction problems (an important family of dataflow-analysis problems used in essentially all modern-day software model checkers). In predicate-abstraction problems,

- a semiring value is a square Boolean matrix
- the product operation is Boolean matrix multiplication
- the sum operation is pointwise "or"
- transpose is matrix transpose, and
- tensor product is Kronecker product.[4]

The key step in NPA-TP is to take each inequality of the form "$Y \gtrsim g \oplus (e \otimes Y \otimes f)$" in the linear model, and turn it into "$Z \gtrsim (\underline{1}^t \odot g) \oplus (Z \otimes (e^t \odot f))$," using the operation $\lambda a.\lambda b.(a^t \odot b)$ as a kind of pairing operator. The reason why this transformation helps is due to the following properties of $\cdot^t$ and $\odot$:

$$\begin{aligned}(a_1 \otimes a_2)^t &= a_2^t \otimes a_1^t \\ (a_1 \odot b_1) \otimes (a_2 \odot b_2) &= (a_1 \otimes a_2) \odot (b_1 \otimes b_2).\end{aligned}$$

These properties cause each regular-language path in the Z system to mimic an LCFL path in the Y system. For instance, the Z-system path shown in Fig. 1(b) has the value

$$\begin{aligned}(\underline{1}^t \odot g) \otimes (e_1^t \odot f_1) \otimes (e_2^t \odot f_2) &= (\underline{1}^t \otimes e_1^t \otimes e_2^t) \odot (g \otimes f_1 \otimes f_2) \\ &= \underbrace{(e_2 \otimes e_1 \otimes \underline{1})^t \odot (g \otimes f_1} \otimes f_2)\end{aligned}$$

which has the kind of mirrored symmetry that we need to properly track the symmetric correlations of values that arise in the Y system. More precisely, the detensor operation performs

$$\not\downarrow^{(t,\cdot)}(e^t \odot f) = e \otimes f,$$

[4] Each element R of a predicate-abstraction domain can be thought of as an $N \times N$ Boolean matrix

$$R = \begin{bmatrix} r_{1,1} & \cdots & r_{1,N} \\ \vdots & \ddots & \vdots \\ r_{N,1} & \cdots & r_{N,N} \end{bmatrix}.$$

The Kronecker product of a square Boolean matrix is defined as follows:

$$R \odot S = \begin{bmatrix} r_{1,1}S & \cdots & r_{1,N}S \\ \vdots & \ddots & \vdots \\ r_{N,1}S & \cdots & r_{N,N}S \end{bmatrix}$$

which is an $N^2 \times N^2$ binary matrix whose entries are

$$(R \odot S)[(a-1)N + b, (a'-1)N + b'] = R(a, a') \wedge S(b, b').$$

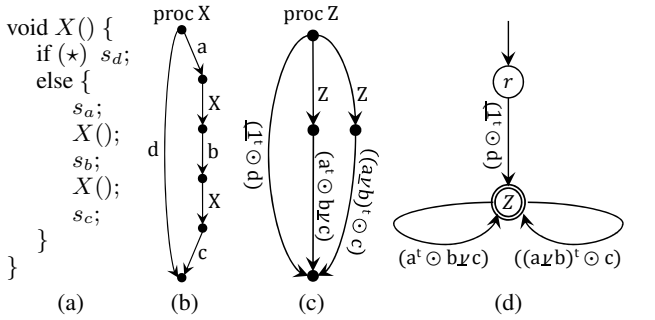

Figure 2: (a) Example program; (b) the program's interprocedural control-flow graph or, equivalently, the graphical representation of the corresponding semiring inequality; (c) graphical representation of the linearized inequality for Z obtained using NPA-TP; (d) finite-state machine for the valuations of all regular-language paths of the Z inequality, i.e., $(\underline{1}^t \odot d)((a^t \odot b \underline{\nu} c) \oplus ((a \underline{\nu} b)^t \odot c))^*$.

so that when the pair-value for the Z-system path is resolved into a product, it has the mirrored symmetry of the LCFL path in the Y system shown in Fig. 1(a); i.e.,

$$\not\downarrow^{(t,\cdot)}((e_2 \otimes e_1 \otimes \underline{1})^t \odot (g \otimes f_1 \otimes f_2)) = \underbrace{e_2 \otimes e_1 \otimes \underline{1} \otimes g \otimes f_1} \otimes f_2$$

To use this idea to solve an LCFL path problem precisely, there is one further requirement: the "detensor" operation must distribute over the (tensored) $\oplus$ operation. Because the Z system's paths encode all and only the LCFL paths of the Y system, the detensor of the $\oplus$-over-all-Z-paths equals the desired $\oplus$-over-all-Y-LCFL-paths. Reps et al. [29] showed that such a distributive detensor operation exists for Kronecker products of Boolean matrices, and thus all the pieces fit together for predicate-abstraction problems.

EXAMPLE 2.1. *To see how NPT-TP works end-to-end, consider the program scheme shown in Fig. 2(a), which has a single recursive procedure X that uses four program statements: s_a, s_b, s_c, and s_d. Suppose that we have a semiring that captures a suitable abstraction of the program's actions (such as a predicate-abstraction domain or the CRA abstract domain). Let a, b, c, and d denote the semiring elements that abstract statements s_a, s_b, s_c, and s_d, respectively. The (abstract) actions of X can be expressed as the recursive inequality given as Eqn. (2). The graphical representation of the inequality, shown in Fig. 2(b), can also be viewed as the program's interprocedural control-flow graph (ICFG).*

*The linear model of Eqn. (2) that would be created with NPA is the recursive linear inequality given as Eqn. (3). In NPA-TP, Eqn. (3) is then converted into the following **left-linear** inequality over tensored semiring elements:*

$$\begin{aligned} &(\underline{1}^t \odot d) \\ Z \gtrsim \oplus\ &Z \otimes ((a \underline{\nu} b)^t \odot c) \\ \oplus\ &Z \otimes (a^t \odot b \underline{\nu} c). \end{aligned} \tag{4}$$

The graphical representation of Eqn. (4) is shown in Fig. 2(c).

Because Eqn. (4) is left-linear, it corresponds to a regular language of paths, and hence we can obtain a solution for Z in closed form. One way to obtain a solution is to create a finite-state machine for the valuations of all regular-language paths of the Z inequality—see Fig. 2(d)—and then apply Tarjan's path-expression method [33] to obtain a regular expression for Z, namely,

$$Z = (\underline{1}^t \odot d)((a^t \odot b \underline{\nu} c) \oplus ((a \underline{\nu} b)^t \odot c))^*.$$

The above process is iterated by converting Z's value into the next approximant for $\underline{\nu}$ (and hence for X) by $\underline{\nu} = \not\downarrow^{(t,\cdot)}(Z)$. In

general, we want to find the least fixed-point of the inequality

$$\underline{\nu} \gtrsim \lambda^{(t,\cdot)}((\underline{1}^t \odot d)((a^t \odot b\underline{\nu}c) \oplus ((a\underline{\nu}b)^t \odot c))^*). \qquad (5)$$

The Newton-iteration sequence for X is thus the Kleene-iteration sequence for Eqn. (5).

$$\begin{aligned}
\underline{\nu}^{(0)} &= \underline{0} \\
\underline{\nu}^{(1)} &= \lambda^{(t,\cdot)}((\underline{1}^t \odot d)((a^t \odot b\underline{0}c) \oplus ((a\underline{0}b)^t \odot c))^*) \\
&= \lambda^{(t,\cdot)}((\underline{1}^t \odot d)(\underline{0})^*) \\
&= \lambda^{(t,\cdot)}(\underline{1}^t \odot d) \\
&= d \\
\underline{\nu}^{(2)} &= \lambda^{(t,\cdot)}((\underline{1}^t \odot d)((a^t \odot bdc) \oplus ((adb)^t \odot c))^*) \\
&\vdots
\end{aligned}$$

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.

Our initial goal was to use NPA-TP because it provides a way to harness Tarjan's path-expression method for interprocedural analysis. This property meshes well with the way CRA couples its recurrence-solving step with the interpretation of Kleene-star.

Unfortunately, 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 key insight that allowed us to devise an algorithm for ICRA is the following:

> The iteration operator of CRA can be "factored" through a simple abstract domain (its *iteration domain*), which has decidable equivalence and a widening operator. We can transform a system of inequalities into special form such that the Kleene-iteration sequence of the transformed system gives rise to a *derived sequence* in the iteration domain. We can then use equivalence checking in the derived sequence to detect convergence of the iteration sequence of the revised system, and use widening to ensure convergence in finite time.

Our solution, presented in §4.4, retains the flavor of NPA-TP: in particular, it adopts the idea of working with a simpler model of the system of inequalities, obtained by rearranging expressions using tensor product (Prop. 4.12). However, other aspects are modified in significant ways. For instance, the key step of our method has the flavor Gauss-Jordan elimination: it repeatedly carries out (i) a symbolic variation of NPA-TP's linearization step applied to a single inequality, and (ii) substitution of a closed-form expression for the linearized symbol into the other inequalities (Alg. 4.13).

4. Technical Details

4.1 Systems of Inequalities

The following definition encapsulates the properties of CRA's semantic algebra that we need to state the problem solved by the generalization of NPA-TP presented in this paper.

DEFINITION 4.1. *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 $\underline{1}$, 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 the closure operator $*$.*
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 over-approximates reflexive transitive closure, in the following sense. The combine operator of $\mathcal{D}$ defines a **natural preorder** $\lesssim$ on D, where for any a and b in D, $a \lesssim b$ if and only if $a \oplus b \equiv b$.*
   (a) *(Reflexivity) $\underline{1} \lesssim a^*$*
   (b) *(Transitivity) $a^* \otimes a \lesssim a^*$ and $a \otimes a^* \lesssim a^*$*

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

Note that we expect the $\oplus$, $\otimes$, and * operators of a quasi weight domain to be effective, but the equivalence relation need not be. The equivalence relation is used for developing the underlying theory of NPA-TP, but does not play a role in any algorithms.

The central question of interest in this paper is *how to compute solutions, over quasi weight domains, to systems of inequalities with regular right-hand sides*? The problem is formalized below.

DEFINITION 4.2. *Let $\mathcal{D} = \langle D, \equiv, \oplus, \otimes, *, \underline{0}, \underline{1}\rangle$ be a quasi weight domain and let $\mathcal{X} = \{X_1, ..., X_n\}$ be a finite set of variables. A **system of inequalities with regular right-hand sides** over $\mathcal{D}$ and $\mathcal{X}$ consists of one inequality for each variable*

$$X_1 \gtrsim R_1 \quad \cdots \quad X_n \gtrsim R_n$$

where each R_i is a regular expression over $\mathcal{D}$ and $\mathcal{X}$. (That is, R_i is a regular expression over an alphabet where each letter is either a variable X_i or a weight $w \in D$.) More compactly, we write

$$S : \{X_i \gtrsim R_i\}_{i=1}^n$$

to denote that S is the above system of inequalities with regular right-hand sides.

*A **solution** to S is any map $\sigma : \mathcal{X} \to \mathcal{D}$ such that for each $X_i \in \mathcal{X}$, we have $\sigma(X_i) \gtrsim \sigma(R_i)$, where $\sigma(R_i)$ denotes the natural extension of σ to regular expressions over $\mathcal{D}$ and $\mathcal{X}$.*

The standard method for solving systems of recursive inequalities is **Kleene iteration**. Given a system of inequalities with regular right-hand sides over $\mathcal{D}$ and $\mathcal{X}$, $S : \{X_i \gtrsim R_i\}_{i=1}^n$, the Kleene-iteration sequence $\langle \kappa^k : \mathcal{X} \to \mathcal{D}\rangle_{k \in \mathbb{N}}$ for S is defined by

$$\kappa^0(X_i) \stackrel{\text{def}}{=} \underline{0} \qquad \kappa^k(X_i) \stackrel{\text{def}}{=} \kappa^{k-1}(R_i)\,.$$

If this sequence *converges*—i.e., there is some k such that $\kappa^k(X_i) \equiv \kappa^{k-1}(X_i)$ for all X_i—then κ^k is a solution (in fact, the *least* solution) to S. However, Kleene iteration requires two conditions on $\mathcal{D}$ that do not hold in the general setting of quasi weight domains (and in particular, do not hold in the CRA quasi weight domain):

- *The ascending-chain condition*: to ensure that the Kleene-iteration sequence converges, we must have that every ascending chain in $\mathcal{D}$ is eventually constant.
- *Decidable equivalence*: to detect convergence of the Kleene-iteration sequence, we must be able to determine whether two weights in $\mathcal{D}$ are equivalent.

In what follows, we will describe an alternative method for solving systems of inequalities with regular right-hand sides. The goal of these methods is to exploit the iteration operator of the quasi weight domain. The iteration operator can be seen as a "built-in" method for solving a particularly simple class of systems—i.e., $X \mapsto a^*$ is a solution to the single-variable system $\{X \gtrsim 1 \oplus (X \otimes a)\}$. The question, then, is how we may use iteration operators to solve more general systems of inequalities. Our presentation of the method proceeds in three steps toward solving more general systems of inequalities: first *left-linear*, then *linear*, then finally culminating in *non-linear* systems.

4.2 Left-Linear Systems

The simplest class of systems of interest are *left-linear*.

DEFINITION 4.3. *Let $\mathcal{D} = \langle D, \equiv, \oplus, \otimes, *, \underline{0}, \underline{1}\rangle$ be a quasi weight domain and let $\mathcal{X} = \{X_1, ..., X_n\}$ be a finite set of variables. A system of inequalities with regular right-hand sides over $\mathcal{D}$ and $\mathcal{X}$, $S : \{X_i \gtrsim R_i\}_{i=1}^n$, is **left-linear** if each expression R_i $(1 \leq i \leq n)$ is of the form*

$$R_i = c_i \oplus \bigoplus_{j=1}^{n} X_j b_{ij},$$

where each c_i and each b_{ij} is a weight belonging to D.

There are two reasons why left-linear systems are interesting: first, because they correspond to *intra*procedural program-analysis problems, and second, because the systems can be solved in any quasi weight domain using Tarjan's path-expression algorithm [33]. The algorithm operates as follows: suppose that we are given a left-linear system of inequalities

$$S : \{X_i \gtrsim c_i \oplus \bigoplus_{j=1}^{n} X_j b_{ij}\}_{i=1}^n$$

over $\mathcal{D}$ and $\mathcal{X}$. First, we associate with S a directed labeled graph $G = \langle V, E\rangle$ where the set of vertices $V = \mathcal{X} \cup \{r\}$ consists of the variables $\mathcal{X}$ plus a designated *root vertex* $r \notin \mathcal{X}$. The set E contains one edge $r \xrightarrow{c_i} X_i$ for each variable X_i, and one edge $X_j \xrightarrow{b_{ij}} X_i$ for each pair of variables X_i and X_j. (See Eqn. (4) and Fig. 2(d) for a one-variable example.) Using Tarjan's path-expression algorithm, we may compute for each X_i a regular expression R_i over the alphabet of edge labels

$$\Sigma = \{c_i : 1 \leq i \leq n\} \cup \{b_{ij} : 1 \leq i, j \leq n\}$$

that recognizes exactly the set of paths in G from the root r to X_i. By interpreting the regular-expression operators in R_i using their counterparts in the quasi weight domain $\mathcal{D}$, we can evaluate R_i to a weight w_i. Letting σ denote the function $\mathcal{X} \to \mathcal{D}$ that maps each X_i to w_i, we have that σ is a solution to S.

4.3 Linear Systems

Intuitively, quasi weight domains are well-suited for solving left-linear systems: 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. gave a methodology that can be used to solve *linear* systems of inequalities by translating them into *left-linear* systems [29] over a *tensored* domain.

DEFINITION 4.4. *Let $\mathcal{D} = \langle D, \equiv, \oplus, \otimes, *, \underline{0}, \underline{1}\rangle$ be a quasi weight domain and let $\mathcal{X} = \{X_1, ..., X_n\}$ be a finite set of variables. A system of inequalities with regular right-hand sides over $\mathcal{D}$ and $\mathcal{X}$, $S : \{X_i \gtrsim R_i\}_{i=1}^n$, is **linear** if each expression R_i $(1 \leq i \leq n)$ is of the form*

$$R_i = c_i \oplus \bigoplus_{j,k} a_{ijk} \otimes X_j \otimes b_{ijk},$$

where each c_i, a_{ijk}, and b_{ijk} is a weight belonging to D.

DEFINITION 4.5. *A **tensor-product domain** $\mathcal{T} = \langle \mathcal{D}, \mathcal{D}_\mathcal{T}, {\cdot}^t, \odot, \natural^{(t,\cdot)}\rangle$ consists of two quasi weight domains $\mathcal{D}$ and $\mathcal{D}_\mathcal{T}$ along with the following operations.*

*A **transpose** operator, denoted by ${\cdot}^t : D \to D$, such that for all $a, b \in D$ the following properties hold:*

$$\begin{aligned} (a \oplus b)^t &= a^t \oplus b^t \\ (a \otimes b)^t &= b^t \otimes a^t \\ (a^t)^t &= a. \end{aligned} \tag{6}$$

and $\equiv$ is a congruence for ${\cdot}^t$ (if $a \equiv b$ then $a^t \equiv b^t$).

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

$$\begin{aligned} \underline{0} \odot a &\equiv_\mathcal{T} a \odot \underline{0} \equiv_\mathcal{T} \underline{0}_\mathcal{T} \\ a \odot (b \oplus c) &\equiv_\mathcal{T} (a \odot b) \oplus_\mathcal{T} (a \odot c) \\ (b \oplus c) \odot a &\equiv_\mathcal{T} (b \odot a) \oplus_\mathcal{T} (c \odot a) \\ (a_1 \odot b_1) \otimes_\mathcal{T} (a_2 \odot b_2) &\equiv_\mathcal{T} (a_1 \otimes a_2) \odot (b_1 \otimes b_2). \end{aligned} \tag{7}$$

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-transpose** operation $\natural^{(t,\cdot)} : D_\mathcal{T} \to D$, such that for all $a, b, c \in D$ and $p, q \in D_\mathcal{T}$ the following properties hold:*

$$\begin{aligned} \natural^{(t,\cdot)}(p \oplus_\mathcal{T} q) &\equiv \natural^{(t,\cdot)}(p) \oplus \natural^{(t,\cdot)}(q) \\ \natural^{(t,\cdot)}(p \otimes_\mathcal{T} (a \odot b)) &\equiv a^t \otimes \natural^{(t,\cdot)}(p) \otimes b^t \\ \natural^{(t,\cdot)}(\underline{1}_\mathcal{T}) &\equiv \underline{1} \end{aligned}$$

and for all $p \equiv_\mathcal{T} q$, we have $\natural^{(t,\cdot)}(p) \equiv \natural^{(t,\cdot)}(q)$.

The operation to **couple** pairs of values from a tensor-product domain, denoted by $\mathcal{C} : D \times D \to D_\mathcal{T}$, is defined as follows:

$$\mathcal{C}(a, b) \stackrel{\text{def}}{=} a^t \odot b.$$

Note that by Eqns. (6) and (7),

$$\begin{aligned} \mathcal{C}(a_1, b_1) \otimes_\mathcal{T} \mathcal{C}(a_2, b_2) &= \left(a_1^t \odot b_1\right) \otimes_\mathcal{T} \left(a_2^t \odot b_2\right) \\ &\equiv_\mathcal{T} \left(a_1^t \otimes a_2^t\right) \odot (b_1 \otimes b_2) \\ &\equiv_\mathcal{T} (a_2 \otimes a_1)^t \odot (b_1 \otimes b_2) \\ &\equiv_\mathcal{T} \mathcal{C}(a_2 \otimes a_1, b_1 \otimes b_2) \end{aligned} \tag{8}$$

Note the order reversal in the first argument in the last line of Eqn. (8), $a_2 \otimes a_1$, vis à vis the order of the operands of $\otimes_\mathcal{T}$ in the first line.

PROPOSITION 4.6. *[29] Let $\mathcal{T} = \langle \mathcal{D}, \mathcal{D}_\mathcal{T}, {\cdot}^t, \odot, \natural^{(t,\cdot)}\rangle$ be a tensor-product domain, let $\mathcal{X} = \{X_1, ..., X_n\}$ be a finite set of variables, and let $S : \{X_i \gtrsim c_i \oplus \bigoplus_{j,k} a_{ijk} \otimes X_j \otimes b_{ijk}\}_{i=1}^n$ be a linear system of inequalities over $\mathcal{D}$ and $\mathcal{X}$. Let $\mathcal{Z} = \{Z_1, ..., Z_n\}$ be a set of variables (one for each X_i), and define the following left-linear system of inequalities over $\mathcal{D}_\mathcal{T}$ and $\mathcal{Z}$:*

$$S_\mathcal{T} : \{Z_i \gtrsim_\mathcal{T} (\underline{1}^t \odot c_i) \oplus_\mathcal{T} \bigoplus_{j}{}_\mathcal{T} \left(Z_j \otimes_\mathcal{T} \left(\bigoplus_{k}{}_\mathcal{T} a_{ijk}^t \odot b_{ijk}\right)\right)\}_{i=1}^n.$$

Let $\sigma_\mathcal{T}$ be any solution to $S_\mathcal{T}$. Then the map $\sigma : \mathcal{X} \to \mathcal{D}$ that sends each X_i to $\natural^{(t,\cdot)}(\sigma_\mathcal{T}(Z_i))$ is a solution to S.

The tensor-product domain of CRA. We now show how the CRA quasi weight domain can be extended to a tensor-product domain.

Recall that CRA weights are transition formulas $\phi(\mathbf{x}, \mathbf{x}')$ over some specified set of variables $\mathbf{x}$ and $\mathbf{x}'$. Making the program variables explicit, we use $\mathit{CRA}(\mathbf{x})$ to denote the quasi weight domain of transition formulas over $\mathbf{x}$ and $\mathbf{x}'$, with operators defined as in §2.1. The tensor-product domain $\mathscr{T}(\mathbf{x}) = \langle \mathcal{D}, \mathcal{D}_{\mathcal{T}}, {}^{\cdot t}, \odot, \curlywedge^{(t,\cdot)} \rangle$ of CRA is similarly parameterized by a set of variables $\mathbf{x}$. The weight domain $\mathcal{D}$ is taken to be $\mathit{CRA}(\mathbf{x})$, and the tensor weight domain $\mathcal{D}_{\mathcal{T}}$ is taken to be $\mathit{CRA}(\underline{\mathbf{x}}\overline{\mathbf{x}})$, where $\underline{\mathbf{x}}$ and $\overline{\mathbf{x}}$ denote disjoint copies of the set of program variables $\mathbf{x}$. That is, the tensor domain consists of transition formulas $\phi(\underline{\mathbf{x}}\overline{\mathbf{x}}, \underline{\mathbf{x}}'\overline{\mathbf{x}}')$ over doubled vocabularies; in total, there are *four* distinct copies of each variable. The models of such a transition formula can be interpreted in two ways:

1. As a set of transitions between states over the set of variables $\underline{\mathbf{x}} \cup \overline{\mathbf{x}}$, or
2. As a set of *pairs* of transitions between states over the set of variables $\mathbf{x}$.

It is helpful to use the first interpretation when thinking about the $\otimes_{\mathcal{T}}$, $\oplus_{\mathcal{T}}$, and ${}^{*\mathcal{T}}$ operations in $\mathit{CRA}(\underline{\mathbf{x}}\overline{\mathbf{x}})$, and the second interpretation when thinking about tensor product and detensor-transpose. Under the first interpretation, ${}^{*\mathcal{T}}$ is the operation for finding closed forms of recurrence equations described in §2.1, except that it operates over transitions from $\underline{\mathbf{x}} \cup \overline{\mathbf{x}}$ to $\underline{\mathbf{x}}' \cup \overline{\mathbf{x}}'$. The tensor product is simply the Cartesian product under the second interpretation. This operation can be defined syntactically with variable renaming and conjunction:

$$\phi(\mathbf{x}, \mathbf{x}') \odot \psi(\mathbf{x}, \mathbf{x}') \stackrel{\text{def}}{=} \phi(\underline{\mathbf{x}}, \underline{\mathbf{x}}') \wedge \psi(\overline{\mathbf{x}}, \overline{\mathbf{x}}') \ .$$

Detensor transpose operates on a pair of transitions by reversing the first transition and sequentially composing with the second. This operation can be defined syntactically with variable renaming and existential quantification:

$$\curlywedge^{(t,\cdot)}(\phi(\underline{\mathbf{x}}\overline{\mathbf{x}}, \underline{\mathbf{x}}'\overline{\mathbf{x}}')) \stackrel{\text{def}}{=} \exists \mathbf{x}''.\phi(\mathbf{x}''\mathbf{x}'', \mathbf{x}\mathbf{x}') \ .$$

Finally, the transpose operator simply reverses the transition formula via variable renaming:[5]

$$\phi(\mathbf{x}, \mathbf{x}')^t \stackrel{\text{def}}{=} \phi(\mathbf{x}', \mathbf{x}) \ .$$

4.4 Non-Linear Systems

This section describes a method for solving[6] general (non-linear) systems of inequalities with regular right-hand sides over a quasi weight domain that obeys certain properties. 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 inequalities, obtained by rearranging expressions using tensor product.

There are two fundamental challenges that our method overcomes:

- 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 defects). Note that Kleene iteration is not even an option for solving the original system of inequalities:

- When there are infinite ascending chains, Kleene iteration may not converge.
- When equivalence is undecidable, 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. 4.13).

As discussed in §3, our general 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 natural widening operator. By carefully re-arranging a system of inequalities into a special form, we can construct from the Kleene-iteration sequence a *derived sequence* in the iteration domain. We can use equivalence checking in the derived sequence to detect convergence of the original Kleene-iteration sequence, and use the widening operator to ensure that the sequence converges in finite time.

Before we define iteration domains in the abstract, we motivate them using the CRA domain. The iteration operator of CRA can be thought of as a two-phase process (cf. §2.1): 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 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 $\mathtt{x}$ 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. Polyhedra are a well-studied abstract domain, for which equivalence is decidable and widening operators are readily available. This two-phase structure of CRA's iteration operator is formalized abstractly by iteration domains:

DEFINITION 4.7. *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 $\langle a_r^\sharp \rangle_{r \in \mathbb{N}}$ in $D^\sharp$, the ascending chain defined as*

$$b_1^\sharp = a_1^\sharp \qquad b_{r+1}^\sharp = b_r^\sharp \triangledown a_{r+1}^\sharp$$

eventually stabilizes.

To motivate our approach, we show how an iteration domain can be used to solve a simple system of inequalities that has just one variable:

$$S : \{X \gtrsim d \oplus (a \otimes X \otimes b \otimes X \otimes c)\} \ . \tag{9}$$

By treating the first occurrence of X on the right-hand side of Eqn. (9) as a variable and the second occurrence as a symbolic con-

[5] In the CRA domain, transitions are actually guarded actions of the form $\langle \mathit{guard}(\mathbf{x}), \mathit{action}(\mathbf{x}, \mathbf{x}') \rangle$. Reversing a transition involves changing the guard to be a formula over the post-state variables $\mathbf{x}'$, which is accomplished by taking the strongest post-condition:

$$\mathit{guard}'(\mathbf{x}') \stackrel{\text{def}}{=} \exists \mathbf{x}.(\mathit{guard}(\mathbf{x}) \wedge \mathit{action}(\mathbf{x}, \mathbf{x}')).$$

In other words, $\langle \mathit{guard}(\mathbf{x}), \mathit{action}(\mathbf{x}, \mathbf{x}') \rangle^t \stackrel{\text{def}}{=} \langle \mathit{guard}'(\mathbf{x}'), \mathit{action}(\mathbf{x}', \mathbf{x}) \rangle$.

[6] In this section, *solving* refers to computing a variable assignment above the least solution. We will make this notion precise shortly.

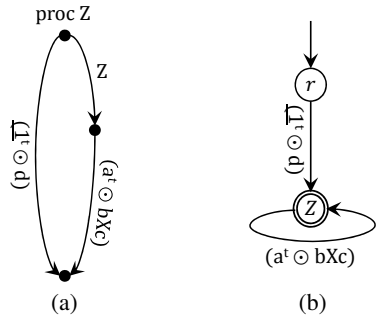

Figure 3: (a) Graphical representation of the linearized inequality given as Eqn. (10); (b) finite-state machine for the valuations of all regular-language paths of Eqn. (10), i.e., $(\underline{1}^t \odot d)(a^t \odot (bXc))^{*_\mathcal{T}}$.

stant (much like $\underline{\nu}$ was a stand-in for X in Eqn. (3)—see Ex. 2.1), we can create a "symbolically" left-linear system of inequalities over tensored weights:

$$S_\mathcal{T} : \{Z \gtrsim (\underline{1}^t \odot d) \oplus_\mathcal{T} (Z \otimes_\mathcal{T} (a^t \odot (b \otimes X \otimes c)))\}, \qquad (10)$$

where $X = \not{\iota}^{(t,\cdot)}(Z)$. The graphical representation of Eqn. (10) is shown in Fig. 3(a).

REMARK 4.8. *The transformation of Eqn. (9) into Eqn. (10) is different from the transformation used in Ex. 2.1 to turn Eqn. (2) into Eqn. (3) and then into Eqn. (4).*

- *The transformation used in Ex. 2.1 leaves us with a left-linear inequality in which there is at most one variable in each summand.*
- *The transformation of Eqn. (9) into Eqn. (10) produces a hybrid: it is left-linear in Z, but X can appear in a summand that also contains a Z.*

The advantage of this transformation will become apparent shortly. The transformation is formalized later in this section.

Applying Tarjan's algorithm to Eqn. (10) gives us an expression for Z in terms of X (see Fig. 3(a)),

$$Z \mapsto (\underline{1}^t \odot d) \otimes_\mathcal{T} (a^t \odot (b \otimes X \otimes c))^{*_\mathcal{T}}, \qquad (11)$$

from which we obtain the following expression for X in terms of X:

$$X \mapsto \not{\iota}^{(t,\cdot)}((\underline{1}^t \odot d) \otimes_\mathcal{T} (a^t \odot (b \otimes X \otimes c))^{*_\mathcal{T}}) \; .$$

Thus, the least solution of the system

$$\hat{S} : \{X \gtrsim \not{\iota}^{(t,\cdot)}((\underline{1}^t \odot d) \otimes_\mathcal{T} (a^t \odot (b \otimes X \otimes c))^{*_\mathcal{T}})\}$$

coincides with the least solution of S. The significance of this sequence of transformations is that in the system $\hat{S}$, every occurrence of a variable is "guarded" in the sense that it appears below a star. If we let $\langle \kappa^k : \mathcal{X} \to \mathcal{D} \rangle_{k \in \mathbb{N}}$ denote the Kleene-iteration sequence for $\hat{S}$, then we can define a "derived sequence" $\langle \beta^k \rangle_{k \in \mathbb{N}}$ in the iteration domain $\mathcal{D}^\sharp$:

$$\beta^k \stackrel{\text{def}}{=} \alpha(a^t \odot (b \otimes \kappa^k(X) \otimes c)) \; .$$

We observe that if the derived sequence converges (i.e., $\beta^k = \beta^{k-1}$ for some k), then so does the Kleene-interation sequence ($\kappa^{k+1}(X) \equiv \kappa^k(X)$). Moreover, convergence can be *enforced* by applying the widening operator of the iteration domain to the derived sequence.

We now show how this process can be carried out for an arbitrary system of inequalities over any number of variables. We begin by introducing a derived *detensor-product* operator that will aid in our symbolic manipulation of expressions.

DEFINITION 4.9. *Let $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_\mathcal{T}, \cdot^t, \odot, \not{\iota}^{(t,\cdot)} \rangle$ be a tensor-product domain. We define the **detensor-product** operator $\ltimes : D \times D_\mathcal{T} \to D$ as follows:*

$$a \ltimes p \stackrel{\text{def}}{=} \not{\iota}^{(t,\cdot)}((\underline{1}^t \odot a) \otimes_\mathcal{T} p) \; .^{7}$$

It is easy to show that for all $a, b \in D$ and all $p, q \in D_\mathcal{T}$ we have

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

and if $a \equiv b$ and $p \equiv_\mathcal{T} q$ then $a \ltimes p \equiv b \ltimes q$. In the remainder of this section, we will assume that the following property holds (as it does for the CRA tensor-product domain): for all $a \in D$ and $p, q \in D_\mathcal{T}$:

$$a \ltimes (p \otimes_\mathcal{T} q) \equiv (a \ltimes p) \ltimes q \; .$$

(Inclusion of this axiom makes $\langle \mathcal{D}, \ltimes \rangle$ a right $\mathcal{D}_\mathcal{T}$-module up to equivalence—i.e., $\ltimes$ acts like scalar multiplication in vector spaces, where the scalars are tensored weights and vectors are untensored weights.)

Next, we define (tensored) extended regular expressions, which correspond to the "expressions with symbolic constants" that we used in the construction of $\hat{S}$ above.

DEFINITION 4.10. *Let $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_\mathcal{T}, \cdot^t, \odot, \not{\iota}^{(t,\cdot)} \rangle$ be a tensor-product domain, and let $\mathcal{X} = \{X_1, ..., X_n\}$ be a finite set of variables. A **(tensored) extended regular expression** over $\mathscr{T}$ and $\mathcal{X}$ is an expression generated by the following grammar:*

$$\begin{aligned} E, F \in \mathit{Ext}(\mathscr{T}) \; &::= \; w \in \mathcal{D} \mid X_i \in \mathcal{X} \mid E \oplus F \mid E \otimes F \\ &\quad \mid E^* \mid E \ltimes E_\mathcal{T} \\ E_\mathcal{T}, F_\mathcal{T} \in \mathit{Ext}_\mathcal{T}(\mathscr{T}) \; &::= \; (E^t \odot F) \mid E_\mathcal{T} \oplus_\mathcal{T} F_\mathcal{T} \\ &\quad \mid E_\mathcal{T} \otimes_\mathcal{T} F_\mathcal{T} \mid E_\mathcal{T}^{*_\mathcal{T}} \end{aligned}$$

*An extended regular expression E over $\mathscr{T}$ is **normal** if for every subexpression of E of the form $E \ltimes E_\mathcal{T}$, $E_\mathcal{T}$ is of the form $F_\mathcal{T}^{*_\mathcal{T}}$.*

For extended regular expressions E and F, we write $E \simeq F$ to denote that for every function $\sigma : \mathcal{X} \to \mathcal{D}$, we have $\sigma(E) \equiv \sigma(F)$.

DEFINITION 4.11. *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**. Formally, for any (tensored) extended regular expression, its set of free variables is defined as follows:*

$$\begin{aligned} \mathit{free}(w) &\stackrel{\text{def}}{=} \emptyset \\ \mathit{free}(X_i) &\stackrel{\text{def}}{=} \{X_i\} \\ \mathit{free}(E \oplus F) &\stackrel{\text{def}}{=} \mathit{free}(E) \cup \mathit{free}(F) \\ \mathit{free}(E \otimes F) &\stackrel{\text{def}}{=} \mathit{free}(E) \cup \mathit{free}(F) \\ \mathit{free}(E^*) &\stackrel{\text{def}}{=} \emptyset \\ \mathit{free}(E \ltimes F_\mathcal{T}) &\stackrel{\text{def}}{=} \mathit{free}(E) \cup \mathit{free}_\mathcal{T}(F_\mathcal{T}) \\ \mathit{free}_\mathcal{T}(E \odot F) &\stackrel{\text{def}}{=} \mathit{free}(E) \cup \mathit{free}(F) \\ \mathit{free}_\mathcal{T}(E_\mathcal{T} \oplus_\mathcal{T} F_\mathcal{T}) &\stackrel{\text{def}}{=} \mathit{free}_\mathcal{T}(E_\mathcal{T}) \cup \mathit{free}_\mathcal{T}(F_\mathcal{T}) \\ \mathit{free}_\mathcal{T}(E_\mathcal{T} \otimes_\mathcal{T} F_\mathcal{T}) &\stackrel{\text{def}}{=} \mathit{free}_\mathcal{T}(E_\mathcal{T}) \cup \mathit{free}_\mathcal{T}(F_\mathcal{T}) \\ \mathit{free}_\mathcal{T}(E_\mathcal{T}^{*_\mathcal{T}}) &\stackrel{\text{def}}{=} \emptyset \end{aligned}$$

The next two propositions define the machinery that we will use to successively eliminate all free variables from the right-hand sides of a system of inequalities.

- Prop. 4.12 shows that tensor-product can be used to rewrite a right-hand side into $X \gtrsim F \oplus X \ltimes F_\mathcal{T}$, which is "symbolically" left-linear in X.

[7] Alternatively, we could have defined detensor-transpose in terms of detensor-product as $\not{\iota}^{(t,\cdot)}(p) \stackrel{\text{def}}{=} \underline{1} \ltimes p$.

- Alg. 4.13 successively invokes Prop. 4.12 to create $X \gtrsim F \oplus X \ltimes F_{\mathcal{T}}$, which is then rewritten to $X \gtrsim F \ltimes {F_{\mathcal{T}}}^{*_{\mathcal{T}}}$. The term $F \ltimes {F_{\mathcal{T}}}^{*_{\mathcal{T}}}$ is then substituted into the other inequalities to eliminate all free occurrences of X.

  This successive-elimination approach is related 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.

The resulting equation system—all of whose variables appear under * or $^{*_{\mathcal{T}}}$—is then solved by a successive-approximation method.

PROPOSITION 4.12. *Let $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_{\mathcal{T}}, \cdot^t, \odot, \not\downarrow^{(t,\cdot)} \rangle$ be a tensor-product domain, let $\mathcal{X} = \{X_1, ..., X_n\}$ be a finite set of variables, and let E be a normal extended regular expression over $\mathscr{T}$ and $\mathcal{X}$. For each variable $X \in \mathcal{X}$, there exist extended regular expressions F and $F_{\mathcal{T}}$ such that*
1. $E \simeq F \oplus (X \ltimes F_{\mathcal{T}})$,
2. F is normal, and
3. X is not free in F.

Proof: By structural induction on E.
- Case $w \in \mathcal{D}$: we have $w \simeq w \oplus (X \ltimes \underline{0}_{\mathcal{T}})$
- Case X: we have $X \simeq \underline{0} \oplus (X \ltimes \underline{1}_{\mathcal{T}})$
- Case $X_i \in \mathcal{X}$ for some $X_i \neq X$: we have $X_i \simeq X_i \oplus (X \ltimes \underline{0}_{\mathcal{T}})$
- Case $E \oplus E'$: by the induction hypothesis, there is some $F, F_{\mathcal{T}}, F', F'_{\mathcal{T}}$ such that $E \simeq F \oplus (X \ltimes F_{\mathcal{T}})$ and $E' \simeq F' \oplus (X \ltimes F'_{\mathcal{T}})$ Then

$$\begin{aligned} E \oplus E' &\simeq F \oplus (X \ltimes F_{\mathcal{T}}) \oplus F' \oplus (X \ltimes F'_{\mathcal{T}}) \\ &\simeq (F \oplus F') \oplus (X \ltimes (F_{\mathcal{T}} \oplus_{\mathcal{T}} F'_{\mathcal{T}})) \end{aligned}$$

- Case $E \otimes E'$: by the induction hypothesis, there is some $F, F_{\mathcal{T}}, F', F'_{\mathcal{T}}$ such that $E \simeq F \oplus (X \ltimes F_{\mathcal{T}})$ and $E' \simeq F' \oplus (X \ltimes F'_{\mathcal{T}})$. Then:

$$\begin{aligned} E \otimes E' &\simeq E \otimes (F' \oplus (X \ltimes F'_{\mathcal{T}})) \\ &\simeq (E \otimes F') \oplus (E \otimes (X \ltimes F'_{\mathcal{T}})) \\ &\simeq (E \otimes F') \oplus (X \ltimes (F'_{\mathcal{T}} \otimes_{\mathcal{T}} (E^t \odot \underline{1}))) \\ &\simeq ((F \oplus (X \ltimes F_{\mathcal{T}})) \otimes F') \oplus (X \ltimes (F'_{\mathcal{T}} \otimes_{\mathcal{T}} (E^t \odot \underline{1}))) \\ &\simeq (F \otimes F') \oplus ((X \ltimes F_{\mathcal{T}}) \otimes F') \\ &\qquad \oplus (X \ltimes (F'_{\mathcal{T}} \otimes_{\mathcal{T}} (E^t \odot \underline{1}))) \\ &\simeq (F \otimes F') \oplus (X \ltimes (F_{\mathcal{T}} \otimes_{\mathcal{T}} (\underline{1} \odot F'))) \\ &\qquad \oplus (X \ltimes (F'_{\mathcal{T}} \otimes_{\mathcal{T}} (E^t \odot \underline{1}))) \\ &\simeq (F \otimes F') \oplus \left(X \ltimes \begin{pmatrix} F_{\mathcal{T}} \otimes_{\mathcal{T}} (\underline{1} \odot F') \\ \oplus_{\mathcal{T}} \; F'_{\mathcal{T}} \otimes_{\mathcal{T}} (E^t \odot \underline{1}) \end{pmatrix} \right) \end{aligned}$$

- Case E^*: we have $E^* \simeq E^* \oplus (X \ltimes \underline{0}_{\mathcal{T}})$
- Case $E \ltimes E_{\mathcal{T}}$: by the induction hypothesis, there is some $F, F_{\mathcal{T}}$ such that $E \simeq F \oplus (X \ltimes F_{\mathcal{T}})$. Then

$$\begin{aligned} E \ltimes E_{\mathcal{T}} &\simeq (F \oplus (X \ltimes F_{\mathcal{T}})) \ltimes E_{\mathcal{T}} \\ &\simeq (F \ltimes E_{\mathcal{T}}) \oplus ((X \ltimes F_{\mathcal{T}}) \ltimes E_{\mathcal{T}}) \\ &\simeq (F \ltimes E_{\mathcal{T}}) \oplus (X \ltimes (F_{\mathcal{T}} \otimes_{\mathcal{T}} E_{\mathcal{T}})) \end{aligned}$$

The last line of each case of the proof can be used to create a recursive function that implements the transformation.

ALGORITHM 4.13 (Free-Variable Elimination). *The input is a tensor-product domain $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_{\mathcal{T}}, \cdot^t, \odot, \not\downarrow^{(t,\cdot)} \rangle$ and a system of inequalities $S : \{X_i \gtrsim R_i\}_{i=1}^n$ with regular right-hand sides, over $\mathcal{D}$ and a finite set of variables $\mathcal{X} = \{X_1, ..., X_n\}$. The output is a system of inequalities $\hat{S} : \{X_i \gtrsim \hat{E}_i\}_{i=1}^n$ where the right-hand sides are normal extended regular expressions over $\mathscr{T}$ and $\mathcal{X}$ with no free variables, and such that the least solution to $\hat{S}$ coincides with the least solution to S.*

We transform S into $\hat{S}$ by eliminating variables one at a time, in a style reminiscent of Gauss-Jordan elimination. We use $S_k : \{X_i \gtrsim E_{i,k}\}_{i=1}^n$ to denote the system of inequalities 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 an extended regular expression $E_{i,0}$).
1. Repeat for each $k = 1$ to n:
  (a) Rewrite $E_{k,k}$ as $E_{k,k} \simeq F \oplus (X_k \ltimes F_{\mathcal{T}})$ (cf. Prop. 4.12).
  (b) Define $S_{k+1} : \{X_i \gtrsim E_{i,k+1}\}_{i=1}^n$ 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 $\hat{S} \stackrel{\text{def}}{=} S_{n+1}$.

Correctness of Alg. 4.13. We would like to claim, "Given a system of inequalities S, Alg. 4.13 computes a system of inequalities $\hat{S}$ (of a particular form) such that the *least solution* of S coincides with the *least solution* of $\hat{S}$." However, this statement does not hold, because a system of inequalities need not have a least solution over $\mathcal{D}$ (i.e., a solution μ to the system such that for every other solution σ, $\mu(X_i) \lesssim \sigma(X_i)$ for all variables X_i). However, any quasi weight domain can be embedded into its *completion*, in which a least solution always exists. (The least solution of a system of inequalities in the completion is analogous to the combine-over-paths solution of a dataflow-analysis problem.)

DEFINITION 4.14 (Completion of a quasi weight domain). *Let $\mathcal{D} = \langle D, \equiv, \oplus, \otimes, *, \underline{0}, \underline{1} \rangle$ be a quasi weight domain. The **completion** of $\mathcal{D}$, $\mathbf{C}(\mathcal{D}) = \langle D_C, \equiv_C, \oplus_C, \otimes_C, \underline{0}_C, \underline{1}_C \rangle$, is the quasi weight domain in which the weights $D_C = \{A \in 2^D : \forall a \in A \forall b \in D. a \equiv b \Rightarrow b \in A\}$ are sets of weights from D (closed under equivalence), the equivalence relation $\equiv_C$ is equality, $\oplus_C$ is union, $\otimes_C$ is pointwise $\otimes$ (i.e., $A \otimes_C B \stackrel{\text{def}}{=} \{a \otimes b : a \in A, b \in B\}$), $A^{*_C} \stackrel{\text{def}}{=} \bigcup_{k \in \mathbb{N}} A^n$, $\underline{0}_C \stackrel{\text{def}}{=} \emptyset$ is the empty set, and $\underline{1} \stackrel{\text{def}}{=} \{\underline{1}_C\}$ is the singleton set containing $\underline{1}$.*

Completion extends to tensor-product domains in the natural way: for a tensor-product domain $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_{\mathcal{T}}, \cdot^t, \odot, \not\downarrow^{(t,\cdot)} \rangle$ its completion domain $\mathbf{C}(\mathscr{T})$ is $\langle \mathbf{C}(\mathcal{D}), \mathbf{C}(\mathcal{D}_{\mathcal{T}}), \cdot^{t_C}, \odot_C, \not\downarrow_C^{(t,\cdot)} \rangle$ where $\cdot^{t_C}$, $\odot_C$, and $\not\downarrow_C^{(t,\cdot)}$ are defined pointwise.

Because the completion of a quasi weight domain $\mathcal{D}$ is a complete lattice and all of its operations are monotonic, any system of inequalities over $\mathcal{D}$ has a unique least solution in $\mathbf{C}(\mathcal{D})$. It is *this* least solution that is preserved by the transformations in Alg. 4.13.

Having now formalized the correctness statement of Alg. 4.13, we will prove it. Let $S : \{X_i \gtrsim R_i\}_{i=1}^n$ be a system of inequalities over $\mathcal{D}$ and a finite set of variables $\mathcal{X} = \{X_1, ..., X_n\}$.

Let $\hat{\sigma}_C : \mathcal{X} \to \mathbf{C}(\mathcal{D})$ denote the least solution of $\hat{S}$. We argue that $\hat{\sigma}_C$ is the least solution to every intermediate system of inequalities S_k by induction, working backward from $S_{n+1} = \hat{S}$ to $S_0 = S$. For the induction step, we suppose that $\hat{\sigma}_C$ is the least solution to S_{k+1} and prove that it is the least solution to S_k. Rewrite $E_{k,k}$ as $E_{k,k} \simeq F \oplus (X_k \ltimes F_{\mathcal{T}})$ (cf. Prop. 4.12). Because $\hat{\sigma}_C$ is the least solution to S_{k+1}, we must have

$$\hat{\sigma}_C(X_k) = \hat{\sigma}_C(E_{k,k+1}) = \hat{\sigma}_C(F \ltimes F_{\mathcal{T}}^{*_{\mathcal{T}}}) \; .$$

We then have:

$$\begin{aligned}
\hat{\sigma}_C(X_k) &= \hat{\sigma}_C(F \ltimes F_{\mathcal{T}}^{*\mathcal{T}}) \\
&= \hat{\sigma}_C(F) \ltimes_C \hat{\sigma}_C(F_{\mathcal{T}}^{*\mathcal{T}}) \\
&= \hat{\sigma}_C(F) \ltimes_C \hat{\sigma}_C(\underline{1}_C \oplus_{\mathcal{T}} (F_{\mathcal{T}}^{*\mathcal{T}} \otimes_{\mathcal{T}} F_{\mathcal{T}})) \\
&= \hat{\sigma}_C(F) \ltimes_C (\underline{1}_C \oplus_{C(\mathcal{T})} (\hat{\sigma}_C(F_{\mathcal{T}}^{*\mathcal{T}}) \otimes_{C(\mathcal{T})} \hat{\sigma}_C(F_{\mathcal{T}}))) \\
&= \begin{array}{l} (\hat{\sigma}_C(F) \ltimes_C \underline{1}_C) \\ \oplus_C (\hat{\sigma}_C(F) \ltimes_C (\hat{\sigma}_C(F_{\mathcal{T}}^{*\mathcal{T}}) \otimes_{C(\mathcal{T})} \hat{\sigma}_C(F_{\mathcal{T}}))) \end{array} \\
&= \hat{\sigma}_C(F) \oplus_C \left(\hat{\sigma}_C(F) \ltimes_C \left(\hat{\sigma}_C(F_{\mathcal{T}}^{*\mathcal{T}}) \otimes_{C(\mathcal{T})} \hat{\sigma}_C(F_{\mathcal{T}})\right)\right) \\
&= \hat{\sigma}_C(F) \oplus_C \left(\left(\hat{\sigma}_C(F) \ltimes_C \hat{\sigma}_C(F_{\mathcal{T}}^{*\mathcal{T}})\right) \ltimes_C \hat{\sigma}_C(F_{\mathcal{T}})\right) \\
&= \hat{\sigma}_C(F) \oplus_C \left(\left(\hat{\sigma}_C(F \ltimes F_{\mathcal{T}}^{*\mathcal{T}})\right) \ltimes_C \hat{\sigma}_C(F_{\mathcal{T}})\right) \\
&= \hat{\sigma}_C(F) \oplus_C (\hat{\sigma}_C(X_k) \ltimes_C \hat{\sigma}_C(F_{\mathcal{T}})) \\
&= \hat{\sigma}_C(F \oplus (X_k \ltimes F_{\mathcal{T}})) \\
&= \hat{\sigma}_C(E_{k,k})
\end{aligned}$$

and thus $\hat{\sigma}_C$ solves the constraint $X_k \gtrsim E_{k,k}$.

For every $j \neq k$, the right-hand-sides $E_{j,k}$ and $E_{j,k+1}$ differ only by substitution of $F \ltimes F_{\mathcal{T}}^{*\mathcal{T}}$ for X_k, so because $\hat{\sigma}_C(X_k) = \hat{\sigma}_C(F \ltimes F_{\mathcal{T}}^{*\mathcal{T}})$ we have $\hat{\sigma}_C(E_{j,k}) = \hat{\sigma}_C(E_{j,k+1})$. It follows from the fact that $\hat{\sigma}_C(X_j) \gtrsim \hat{\sigma}_C(E_{j,k+1})$ that $\hat{\sigma}_C(X_j) \gtrsim \hat{\sigma}_C(E_{j,k})$. We have thus shown that $\hat{\sigma}_C$ is a solution to S_k.

It remains to show that $\hat{\sigma}_C$ is the *least* solution to S_k. Suppose that σ_C is the least solution to S_k. We will prove that σ_C is a solution to S_{k+1} so that (by virtue of $\hat{\sigma}_C$ being the *least* solution to S_{k+1} and σ_C being *a* solution to S_{k+1}, and the symmetric argument) σ_C and $\hat{\sigma}_C$ must coincide.

We begin by showing that σ_C solves the constraint $X_k \gtrsim E_{k,k+1}$ (in fact, $\sigma_C(X_k) = \sigma_C(E_{k,k+1})$). Observe that

$$\begin{aligned}
\sigma_C(E_{k,k+1}) &= \sigma_C(F \ltimes (F_{\mathcal{T}})^{*\mathcal{T}}) \\
&= \sigma_C(F) \ltimes_C (\sigma_C(F_{\mathcal{T}}))^{*\mathcal{T}} \\
&= \textstyle\bigcup_j \sigma_C(F) \ltimes_C (\sigma_C(F_{\mathcal{T}}))^j \ .
\end{aligned}$$

So to show that $\sigma_C(X_k) \supseteq \sigma_C(E_{k,k+1})$, it is sufficient to show that for all j we have

$$\sigma_C(X_k) \supseteq \sigma_C(F) \ltimes_C (\sigma_C(F_{\mathcal{T}}))^j \ . \tag{12}$$

Note that because σ_C is the least solution to S_k, we have

$$\sigma_C(X_k) = \sigma_C(E_{k,k}) = \sigma_C(F) \oplus_C \sigma_C(X_k) \ltimes_C \sigma_C(F_{\mathcal{T}})$$

and thus

$$\sigma_C(X_k) \supseteq \sigma_C(F) \tag{13}$$

$$\sigma_C(X_k) \supseteq \sigma_C(X_k) \ltimes_C \sigma_C(F_{\mathcal{T}}) \tag{14}$$

We prove Eqn. (12) by induction on j.

- Base case: immediate from Eqn. (13)
- Induction step: Suppose that $\sigma_C(X_k) \supseteq \sigma_C(F) \ltimes_C (\sigma_C(F_{\mathcal{T}}))^j$. From Eqn. (14) and monotonicity of $\ltimes$, we have

$$\begin{aligned}
\sigma_C(X_k) &\supseteq \sigma_C(X_k) \ltimes_C \sigma_C(F_{\mathcal{T}}) \\
&\supseteq (\sigma_C(F) \ltimes_C (\sigma_C(F_{\mathcal{T}}))^j) \ltimes_C \sigma_C(F_{\mathcal{T}}) \\
&= \sigma_C(F) \ltimes_C ((\sigma_C(F_{\mathcal{T}}))^j \otimes_{C(\mathcal{T})} \sigma_C(F_{\mathcal{T}})) \\
&= \sigma_C(F) \ltimes_C (\sigma_C(F_{\mathcal{T}}))^{j+1}
\end{aligned}$$

So from the above we have that

$$\sigma_C(X_k) \supseteq \bigcup_j \sigma_C(F) \ltimes_C (\sigma_C(F_{\mathcal{T}}))^j = \sigma_C(E_{k,k+1}). \tag{15}$$

We now need to show that $\sigma_C(X_k) \subseteq \sigma_C(E_{k,k+1})$. Let σ'_C be the map that sends X_k to $\sigma_C(E_{k,k+1})$, and every other variable X_i to $\sigma_C(X_i)$. It is easy to check that σ'_C is a solution to S_k. By virtue of σ_C being the *least* solution to S_k, it follows that $\sigma_C(X_k) \subseteq \sigma'_C(X_k) = \sigma_C(E_{k,k+1})$. Thus, from Eqn. (15), we have $\sigma_C(X_k) = \sigma_C(E_{k,k+1})$.

For every $j \neq k$, the right-hand-sides $E_{j,k}$ and $E_{j,k+1}$ differ only by substitution of $F \ltimes F_{\mathcal{T}}^{*\mathcal{T}}$ for X_k, so because $\sigma_C(X_k) = \sigma_C(F \ltimes F_{\mathcal{T}}^{*\mathcal{T}})$ we have $\sigma_C(E_{j,k}) = \sigma_C(E_{j,k+1})$. It follows from the fact that $\sigma_C(X_j) \gtrsim \sigma_C(E_{j,k})$ that $\sigma_C(X_j) \gtrsim \sigma_C(E_{j,k+1})$. We have thus shown that σ_C is a solution to S_{k+1}.

Because σ_C and $\hat{\sigma}_C$ are both solutions to S_k and S_{k+1}, σ_C is the least solution to S_k, and $\hat{\sigma}_C$ is the least solution to S_{k+1}, we have $\sigma_C = \hat{\sigma}_C$, and $\hat{\sigma}_C$ is the least solution to S_k. This observation completes our argument that the least solution to $\hat{S}$ coincides with the least solution to S.

Solving systems of inequalities with extended-regular-expression right-hand sides. Let $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_{\mathcal{T}}, \cdot^t, \odot, \frac{1}{2}^{(t,\cdot)} \rangle$ be a tensor-product domain, let $\mathcal{X} = \{X_1, ..., X_n\}$ be a finite set of variables, and let $S : \{X_i \gtrsim E_i\}_{i=1}^n$ be a system of inequalities in which each E_i is an extended regular expression over $\mathscr{T}$ and $\mathcal{X}$ with no free variables.

Define a sequence of functions $\langle \sigma^k : \mathcal{X} \to \mathcal{D} \rangle_{k \in \mathbb{N}}$ as follows. The function σ^0 is defined to be the constant $\underline{0}$ function. For each $k \geq 1$, define

$$\sigma^k(X_i) \stackrel{\text{def}}{=} \mathit{eval}^k(E_i)$$

where

$$\begin{aligned}
\mathit{eval}^k(w) &= w \\
\mathit{eval}^k(X_i) &= \sigma^{k-1}(X_i) \\
\mathit{eval}^k(E \oplus F) &= \mathit{eval}^k(F) \oplus \mathit{eval}^k(F) \\
\mathit{eval}^k(E \otimes F) &= \mathit{eval}^k(E) \otimes \mathit{eval}^k(E) \\
\mathit{eval}^k(E^*) &= \mathit{cl}\big(\mathit{body}^k(E)\big) \\
\mathit{eval}^k(E \ltimes E_{\mathcal{T}}) &= \mathit{eval}^k(E) \ltimes \mathit{eval}^k_{\mathcal{T}}(E_{\mathcal{T}}) \\
\mathit{eval}^k_{\mathcal{T}}(E^t \odot F) &= \mathit{eval}^k_{\mathcal{T}}(E)^t \odot \mathit{eval}^k_{\mathcal{T}}(F) \\
\mathit{eval}^k_{\mathcal{T}}(E_{\mathcal{T}} \oplus_{\mathcal{T}} F_{\mathcal{T}}) &= \mathit{eval}^k_{\mathcal{T}}(E_{\mathcal{T}}) \oplus_{\mathcal{T}} \mathit{eval}^k_{\mathcal{T}}(F_{\mathcal{T}}) \\
\mathit{eval}^k_{\mathcal{T}}(E_{\mathcal{T}} \otimes_{\mathcal{T}} F_{\mathcal{T}}) &= \mathit{eval}^k_{\mathcal{T}}(E_{\mathcal{T}}) \otimes_{\mathcal{T}} \mathit{eval}^k_{\mathcal{T}}(F_{\mathcal{T}}) \\
\mathit{eval}^k_{\mathcal{T}}(E^{*\mathcal{T}}_{\mathcal{T}}) &= \mathit{cl}_{\mathcal{T}}(\mathit{body}^k_{\mathcal{T}}(E_{\mathcal{T}}))
\end{aligned}$$

and

$$\begin{aligned}
\mathit{body}^k(E) &= \mathit{body}^{k-1}(E) \triangledown \alpha(\sigma^{k-1}(E)) \\
\mathit{body}^k_{\mathcal{T}}(E_{\mathcal{T}}) &= \mathit{body}^{k-1}_{\mathcal{T}}(E_{\mathcal{T}}) \triangledown_{\mathcal{T}} \alpha(\sigma^{k-1}(E_{\mathcal{T}})) \ .
\end{aligned}$$

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 subexpresssion 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 4.15. *If S stabilizes at k, then σ^k is a solution to S.*

Proof: To show that σ^k is a solution to S, we must demonstrate that $\sigma^k(X_i) \gtrsim \sigma^k(E_i)$ for all i. The essence of the argument is to show that the following sequence of (in)equalities is valid:

$$\sigma^k(X_i) = \mathit{eval}^k(E_i) \gtrsim \sigma^{k-1}(E_i) = \sigma^k(E_i) \ .$$

The first equality in the sequence holds by the definition of σ^k. The middle inequality we prove by induction on E_i. We prove validity of the last equation by showing that the assumption that S stabilizes at k (i.e., the fact that $\mathit{body}^k(E) = \mathit{body}^{k-1}(E)$ for every "loop-body" expression E) is sufficient to prove that $\sigma^{k-1} = \sigma^k$.

First, we show that $\mathit{eval}^k(E_i) \gtrsim \sigma^{k-1}(E_i)$ holds. We prove that this property holds for all sub-expressions of E_i by structural induction.

- Case $w \in \mathcal{D}$, $X_j \in \mathcal{X}$: trivial.
- Case $E \oplus F$, $E \otimes F$, $E \ltimes E_{\mathcal{T}}$, $E^t \odot F$, $E_{\mathcal{T}} \oplus_{\mathcal{T}} F_{\mathcal{T}}$, $E_{\mathcal{T}} \otimes_{\mathcal{T}} F_{\mathcal{T}}$: from the induction hypothesis and monotonicity.
- Case E^* (Case $E^{*\mathcal{T}}_{\mathcal{T}}$ is similar): First, observe that by property 1 of the widening operator, we have

$$\mathit{body}^{k-1}(E) \triangledown \alpha(\sigma^{k-1}(E)) \geq^\sharp \alpha(\sigma^{k-1}(E)) \ .$$

It follows that:

$$\begin{aligned}
\mathit{eval}^k(E^*) &= \mathit{cl}(\mathit{body}^k(E)) && \text{Definition}\\
&= \mathit{cl}(\mathit{body}^{k-1}(E) \triangledown \alpha(\sigma^{k-1}(E))) && \text{Definition}\\
&\gtrsim \mathit{cl}(\alpha(\sigma^{k-1}(E))) && \text{Monotonicity}\\
&= (\sigma^{k-1}(E))^* && \text{Closure axiom}\\
&= \sigma^{k-1}(E^*) && \text{Definition}
\end{aligned}$$

Second, we prove that $\sigma^{k-1} = \sigma^k$ holds. Let X_i be a variable. We have $\sigma^k(X_i) = \mathit{eval}^k(E_i)$ by definition, so it is sufficient to prove that $\mathit{eval}^k(E_i) = \mathit{eval}^{k-1}(E_i)$. We prove that this equality holds for all closed sub-expressions of E_i—i.e., subexpressions without free variables—by structural induction.

- Case $w \in \mathcal{D}$: trivial.
- Case $E \oplus F$, $E \otimes F$, $E \ltimes E_{\mathcal{T}}$, $E^t \odot F$, $E_{\mathcal{T}} \oplus_{\mathcal{T}} F_{\mathcal{T}}$, $E_{\mathcal{T}} \otimes_{\mathcal{T}} F_{\mathcal{T}}$: immediately from the induction hypothesis.
- Case E^* (Case $E^{*_{\mathcal{T}}}_{\mathcal{T}}$ is similar): Because S stabilizes at k, we have $\mathit{body}^k(E) = \mathit{body}^{k-1}(E)$. It follows that $\mathit{cl}(\mathit{body}^k(E)) = \mathit{cl}(\mathit{body}^{k-1}(E))$, and thus $\mathit{eval}^k(E^*) = \mathit{eval}^{k-1}(E^*)$.

PROPOSITION 4.16. *There exists a k such that S stabilizes at k.*

Proof: Let E^* be any starred subexpression that appears in some E_i (the case for $E^{*_{\mathcal{T}}}_{\mathcal{T}}$ is similar). We must show that the sequence $\langle \mathit{body}^r(E)\rangle_{r\in\mathbb{N}}$ eventually stabilizes. Consider the infinite sequence $\langle \alpha(\sigma^r(E))\rangle_{r\in\mathbb{N}}$ in $\mathcal{D}^\sharp$. Because $\triangledown$ is a widening operator, the sequence

$$b^\sharp_1 = \alpha(\sigma^0(E)) \qquad b^\sharp_{r+1} = b^\sharp_r \triangledown \alpha(\sigma^r(E))$$

eventually stabilizes. Finally, observe that the sequence $\langle b^\sharp_r\rangle_{r\in\mathbb{N}}$ coincides with the sequence $\langle \mathit{body}^r(E)\rangle_{r\in\mathbb{N}}$.

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

ALGORITHM 4.17 (NPA-TP-GJ). *The input is a tensor-product domain $\mathscr{T} = \langle \mathcal{D}, \mathcal{D}_{\mathcal{T}}, \cdot^t, \odot, \curlywedge^{(t,\cdot)}\rangle$ and a system of inequalities $S : \{X_i \gtrsim R_i\}_{i=1}^n$ with regular right-hand sides, over $\mathcal{D}$ and a finite set of variables $\mathcal{X} = \{X_1, ..., X_n\}$. The output is a mapping $\sigma : \mathcal{X} \to \mathcal{D}$ such that if μ is the least solution to S, then for all i and $a \in \mu(X_i)$, $\sigma(X_i) \gtrsim a$.*

1. *Perform Alg. 4.13 to transform S into a system of inequalities $\hat{S} : \{X_i \gtrsim \hat{E}_i\}_{i=1}^n$, in which each $\hat{E}_i$ is an extended regular expression over $\mathscr{T}$ and $\mathcal{X}$ that has no free variables.*
2. *$k \leftarrow 0$; $\sigma^0 \leftarrow \lambda x.\underline{0}$*
3. *Repeat*
   (a) $k \leftarrow k + 1$
   (b) $\sigma^k(X_i) \leftarrow \mathit{eval}^k(\hat{E}_i)$
   until $\hat{S}$ stabilizes
4. *Return σ^k*

In the degenerate case when S is linear, left-linear, or right-linear, the call on Alg. 4.13 in step (1) returns a solution to S (or, more precisely, a system of inequations $\hat{S}$ in which the right-hand-sides do not contain *any* variables). In this case, the evaluation loop stabilizes on the first pass.

Discussion. The problem that NPR-TP-GJ addresses is how to both detect and enforce convergence when working with an abstract domain that has neither decidable equivalence nor the ascending chain condition. Our first (flawed) attempt at a solution was to work with NPA-TP, but to introduce the iteration domain so that we could check that each expression under a Kleene-star converged. However, that approach is unsound because variables *not* under a Kleene-star may not have received a sound final value when using that convergence condition. In other words, the iteration-domain trick does not work for the NPA-TP linearization. Consequently, we introduced the Gauss-Jordan-like linearization, which always obtains a sound result.

5. Implementation and Experiments

NPA-TP-GJ is implemented on top of the WALi [22] system for weighted pushdown systems. In particular, it uses the implementation of Tarjan's method from the FWPDS solver [24] of WALi to create the initial system of inequalities with regular right-hand sides from a specification of the problem to be solved as a weighted pushwdown system. We instantiated NPA-TP-GJ for ICRA by coupling WALi to the CRA abstract domain[8] [13], and augmenting the implementation of CRA (which, in turn, makes use of APRON [1] and Z3 [10]) with $\cdot^t$, $\odot$, and $\curlywedge^{(t,\cdot)}$ to create a CRA tensor-product domain.

The implementation supports programs with multiple procedures, including recursion and mutual recursion. It supports local variables via the method explained in [29, §8]. The merge function used when returning from a prccedure call is based on [29, 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 [6]?

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

Timings were taken on a virtual machine (using Oracle VirtualBox), with a guest OS of Ubuntu 14.04, and host OS of Red Hat Enterprise Linux 6. The host CPU was a dual-core Intel Pentium G3240 running at 3.1 GHz.

5.1 Assertion Checking

To assess the suitability of ICRA for assertion checking, we ran ICRA on (i) the benchmarks in the *Loops* and *Recursive* subcategories of the *Integers and Control Flow* category from SV-COMP16 [32]; (ii) the suite of programs from C4B [7, App. A], annotated with upper-bound resource assertions of the bounds reported by C4B;[9] and (iii) a miscellaneous collection of recursive programs. We also ran two state-of-the-art software model checkers, CPAchecker [3] and Ultimate Automizer [19], on the same programs. Specifically, we used CPAchecker version 1.6.1-unix with the configuration file "sv-comp16.properties", and the version of UAutomizer submitted to SV-COMP16. The data from the experimental runs made with the three tools, as well as links to the tools' outputs, are available in the anonymous supplemental material, in the file "assertion-checking-results.html."

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.[10]

(2) ICRA handled more assertions than the other two tools.

All three tools have different success/failure-versus-timeout profiles, and there are examples on which one tool succeeds quickly

[8] We implemented a subclass of WALi's semiring class/interface by making appropriate calls to operations of the CRA abstract domain. The main complication was reconciling CRA's OCaml implementation with WALi's C++ implementation.

[9] C4B only obtains bounds for 34 of the 35 programs. The assertion in the 35th program is the bound from SPEED [18].

[10] We used a timeout limit of 300 seconds; at SV-COMP16, the timeout limit was 900 seconds.

Benchmark Suite	Total #A	ICRA Time	ICRA #A	UAut. Time	UAut. #A	CPA Time	CPA #A
recursive	18/7	**104.89**	7/**7**	2,013.45	8/**7**	1,879.68	**10/7**
rec.-simple	36/38	**188.98**	21/**38**	7,415.99	27/28	3,046.76	**32**/33
Rec. (tot.)	54/45	**293.87**	28/**45**	9,429.44	35/35	4,926.44	**42**/40
loop-accel.	19/16	**20.58**	**13/16**	6,719.06	7/4	4,807.11	**13**/8
loop-invgen	18/1	**56.21**	**16/1**	1,901.32	7/**1**	4,929.20	2/**1**
loop-lit	15/1	**323.43**	**12/1**	2,744.02	4/**1**	2,758.87	6/**1**
loop-new	8/0	**6.40**	**7**/0	2,151.40	1/0	1,874.28	3/0
loops	34/32	**190.53**	19/**31**	4,091.96	20/19	4,605.55	**27**/27
Loops (tot.)	94/50	**597.15**	**67/49**	17,607.76	39/25	18,975.01	51/37
C4B	35/0	**31.68**	**29**/0	6,182.25	1/0	8,072.61	2/0
Misc.-Rec.	10/4	**325.23**	**9/4**	525.06	8/**4**	344.45	7/1

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 columns 4, 6, and 8, "X/Y" means that (all) the valid assertions in X programs were established, and (all) the invalid assertions in Y programs were (correctly) not established. A timeout is scored in neither category.

but the other tools time out. For instance, ICRA succeeds in proving the assertion in the count_up_down_true-unreach-call_true-termination benchmark (shown below) in .55 seconds, whereas both CPAchecker and UAutomizer time out:

```
int main() {
  unsigned int n = __VERIFIER_nondet_uint();
  unsigned int x=n, y=0;
  while(x>0) {
    x--; y++;
  }
  __VERIFIER_assert(y==n);
}
```

In contrast, for the the underapprox_true-unreach-call1 benchmark (see below), both CPAchecker and UAutomizer were able to establish the assertion, but ICRA was not.

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

We were somewhat surprised that ICRA did not perform better on the recursive examples. However, it should be noted that the SV-COMP16 *Recursive* subcategory is probably not representative, and many of the recursive benchmarks match up poorly with ICRA's current capabilities. In particular, many are merely copies of the Fibonacci function with an assertion that the function computes a particular output value for a particular input. (There are 35 variants of Fibonacci and 25 variants of the identity function.) These benchmarks are intended to be used with a software model-checker, which will unroll the recursion down to a sufficient depth to prove the assertion. In contrast, ICRA is designed to find invariants, and the invariant of the Fibonacci function is beyond our implementation's current capabilities, although we are actively working on enhanced recurrence-solving techniques that would handle it.

The test suite listed as *Misc.-Rec.* consists of a few newly-written recursive programs, as well as a few recursive programs drawn from the test suite of the CRA static-analysis tool [13].

5.2 Applications in Resource-Bound Analysis

For the experiments discussed in §5.1, 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.)

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

(a)

```
void start(int n) {
  int flag = 1;
  while (flag > 0) {
    flag = 0;
    while (n > 0) {
      tick(1);
      n = n - 1;
      flag = 1;
} } }
```

(b)

Figure 4: (a) Recursive function used to illustrate upper and lower bounds on resource usage. (b) The speed_pldi10_ex4.c benchmark from the C4B suite.

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).

Given a program, ICRA can compute terms that upper-bound and lower-bound both the *final* resource usage and the *high-water mark* of resource usage.[11] 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 [7]), 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)
$\mathit{final} \geq 4\|[0,n]\| + 4$	$1 \to 2 \to 4 \to$ `perform(n-1)` $\to 5$
$\mathit{final} \leq 6\|[0,n]\| + 6$	$1 \to 2 \to 3 \to 4 \to$ `perform(n-1)` $\to 5$
$\mathit{hwm} \geq 7\|[0,n]\| + 7$	$1 \to 2 \to 4 \to$ `perform(n-1)` $\to 5$
$\mathit{hwm} \leq 9\|[0,n]\| + 9$	$1 \to 2 \to 3 \to 4 \to$ `perform(n-1)` $\to 5$

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

On the C4B test suite, ICRA is able to generate upper bounds on resource usage for 25 of the 35 programs, and nontrivial lower bounds on resource usage for 25 of the programs (but not the same set of 25). For 3 programs, ICRA obtains a tighter upper-bound than C4B. For example, given the procedure shown in Fig. 4(b), ICRA generates the following bounds (the first of which improves on the C4B (upper) bound of $\mathit{final} \leq 1 + 2|[0,n]|$):

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

The source files and the outputs of the bounds-generation experiments on the C4B suite are available in the anonymous supplemental material in the file "bounds-generation-results.html."

6. Related Work

The NPA-TP-GJ algorithm and its instantiation for ICRA rely heavily on prior work on CRA [13], NPA [12], and NPA-TP [29].

- The NPA-TP-GJ algorithm adopts NPA-TP's tensor-product operation so that various terms in a non-linear inequality can be rewritten into the "symbolically" left-linear form $X \gtrsim F \oplus X \ltimes F_{\mathcal{T}}$, and then further rewritten as $X \gtrsim F \ltimes {F_{\mathcal{T}}}^{*_{\mathcal{T}}}$.
- ICRA relies on the fact that the logic fragment that CRA employs at loops to solve recurrence equations supports effective equivalence. That property—along with the widening performed during

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

the *body* subroutine of *eval*, and the stabilization result given in Thm. 4.15—ensures that ICRA will always terminate when it is applied to a program that uses recursion.

The elimination step in NPA-TP-GJ is related to Gauss-Jordan elimination, in that variables are eliminated in some order. However, 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 Tarjan's path-expression algorithm [33], Gondran & Minoux [15, §4.5], Rote [30, §4], Carré [8, §7], and Backhouse & Carré [2, §4]. The techniques developed in the present paper may open up the possibility of using those methods—or perhaps variants that only eliminate "free variables"—for interprocedural program analysis.

Recently, a new iteration scheme was proposed by Meyer and Muskalla [26], called Munchausen Iteration (MI). MI is based on an iteration step that (a) works on a vector of expressions, and (b) substitutes the current approximant into itself. Both aspects have some similarities with NPA-TP-GJ, but the substitution step in MI is more aggressive. The benefit of MI is that, in principle, MI takes exponentially fewer rounds than NPA; however, Meyer and Muskalla indicate that MI is not yet an algorithm, ". . . so far, we do not know how to extract information from the [structures that represent answers]." Also, when the ascending-chain condition does not hold, it is not clear how widening would be performed on the symbolic structures that MI employs.

Although the performance of ICRA in the tool comparison reported in §5.1 is encouraging, it is important to note that ICRA is an invariant generator, while CPAchecker and UAutomizer are software model checkers. The standard trade-offs apply. CPAchecker and UAutomizer are able to prove challenging program properties by employing abstraction refinement to tailor abstractions to the property at hand, but may refine abstractions indefinitely and fail to terminate. ICRA can be used to compute rich program invariants (rather than just answer verification queries) but is not currently able to compute counter-example witnesses to assertion failures. (We illustrated one use of ICRA's facility for invariant generation in §5.2 by employing ICRA to compute upper and lower bounds on resource usage.)

The success of compositional recurrence analysis is due to the power of the iteration operator to summarize loops. There are a number of related techniques for loop summarization [4, 17, 23] and abstract loop-acceleration [16, 21, 25] (which computes post-images of loops rather than summaries). Loop summarization in CRA is based on computing 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.

7. Conclusion

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 method presented in the paper adopts, but adapts in significant ways, ideas used in the NPA-TP framework [29].

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

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

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

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

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

[9] 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.

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

[11] J. Esparza, S. Kiefer, and M. Luttenberger. Newton's method for omega-continuous semirings. In *ICALP*, 2008.

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

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

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

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

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

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

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

[19] 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.

[20] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *TOPLAS.*, 34(3), 2012.

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

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

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

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

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

[26] R. Meyer and S. Muskalla. Munchausen iteration. *CoRR (arXiv)*, abs/1605.00422, 2016. URL `http://arxiv.org/abs/1605.00422`.

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

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

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

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

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

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

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