Compositional Recurrence Analysis Revisited

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

Jason Breck  
Ashkan Forouhi  
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:

- \textit{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);
- \textit{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 \(+, \cdot, \text{ and }^*\) as disjunction, sequential composition, and the generation and solving of an appropriate system of recurrence equations, respectively.

CRA is \textit{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 \(+, \cdot, \text{ and }^*\) [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 \textit{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 reduced 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 \textit{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 \textit{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.

2016/10/6
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:

1. 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).
2. 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.
3. 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 sequential ($\odot$), 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 $x$ denote a finite set of program variables, a transition formula $\phi(x, x')$ is a formula over the variables $x$ plus a set of primed copies $x'$, representing the values of the variables before and after executing a path. The sequencing operation is relational composition and choice is disjunction.

$\phi \odot \psi \overset{\text{def}}{=} \exists x''. \phi(x, x'') \land \psi(x'', x')$ \qquad $\phi \oplus \psi \overset{\text{def}}{=} \phi \lor \psi$

The heart of CRA is its iteration operator. Given as input a transition formula $\phi_{\text{body}}$ that summarizes the body of a loop, the iteration operator computes a formula $\phi_{\text{body}}$ that summarizes any number of iterations. The iteration operator works by using an SMT solver to extract a system of recurrences entailed by $\phi_{\text{body}}$, 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.)

```plaintext
while (i > 0)
  i := i - 1
if (x) 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:

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

The iteration operator extracts the following recurrence (in)equations from $\phi_{\text{body}}$, 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.
<table>
<thead>
<tr>
<th>Recurrence</th>
<th>Closed form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i' = i - 1$</td>
<td>$i^{(k)} = i^{(0)} - k$</td>
</tr>
<tr>
<td>$x' \geq x$</td>
<td>$x^{(k)} \geq x^{(0)}$</td>
</tr>
<tr>
<td>$y' \geq y - i$</td>
<td>$y^{(k)} \geq y^{(0)} - k + 1/2 + k_{i(0)}$</td>
</tr>
<tr>
<td>$x' \leq x + i$</td>
<td>$x^{(k)} \leq x^{(0)} + k + 1/2 + k_{i(0)}$</td>
</tr>
<tr>
<td>$y' \leq y$</td>
<td>$y^{(k)} \leq y^{(0)}$</td>
</tr>
<tr>
<td>$x' - y' = x - y + i$</td>
<td>$(k + 1)/2 + k_{i(0)}$</td>
</tr>
</tbody>
</table>

(where $i^{(k)}$ denotes the value of $i$ on the $k$th iteration of the loop).

Finally, to compute a transition formula representing any number of iterations of $\phi_{body}$, the iteration operator introduces an existentially quantified non-negative iteration variable $k$, and conjoins the closed form of every recurrence:

$$\phi_{body} \triangleq 3k. k \geq 0$$

$$\land i' = i - k$$

$$\land x' \geq x$$

$$\land y' \geq y + k + 1/2 + k_{i(0)}$$

$$\land x' \leq x + k + 1/2 + k_{i(0)}$$

$$\land y' \leq y$$

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

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 $\Lambda$). 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 $X_P \triangleright X_Q \otimes X_Q$, which is quadratic. If there are actions $a, b$, and $c$ before and after the calls, the inequality would be

$$X_P \triangleright a \otimes X_Q \otimes b \otimes X_Q \otimes c$$

(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

$$X_P \triangleright (a \otimes X_Q \otimes b \otimes (c \otimes X_Q \otimes d) + (e \otimes X_R \otimes f) \otimes (g \otimes X_P \otimes h))$$

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 $\otimes$ 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

$$X_P \triangleright (a \otimes X_Q \otimes b \otimes X_Q \otimes c) \otimes (a \otimes X_Q \otimes b \otimes X_Q \otimes c)$$

(2)

The associated linear model is the recursive linear inequality

$$Y \triangleright d \otimes (a \otimes Y \otimes b \otimes X \otimes c) \otimes (a \otimes Y \otimes b \otimes c \otimes e)$$

(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 “$e \otimes X \otimes c$” into “$e \otimes Y \otimes f$.” An inequality of the form “$Y \triangleright g \otimes e \otimes f \otimes \Lambda$” is similar to the right-linear grammar “$Y \rightarrow g \mid e \otimes f \otimes \Lambda$,” 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: “$Y \triangleright g \otimes e \otimes Y \otimes f$” cannot be rearranged into “$Y \triangleright g \otimes e \otimes f \otimes Y$.”

The inequality “$Y \triangleright g \otimes e \otimes Y \otimes f$” is similar to the linear context-free grammar “$Y \rightarrow g \mid e \otimes f$.” The latter describes the linear context-free language (LCFL) $L(Y) = (ef)^*$, 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 $a^* b^*$ 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
sum and product operations are interpreted. Also, NPA-TP requires the semiring to support a few additional operations—called “transpose” ($\mathcal{T}$), “tensor product” ($\otimes$), and “detensor” ($\mathcal{D}(\cdot, \cdot)$)—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.

The key step in NPA-TP is to take each inequality of the form “$Y \succeq g \circ (e \otimes f)$” in the linear model, and turn it into “$Z \succeq (1^T \otimes g) \circ (Z \otimes (e \otimes f))$,” using the operation $\mathcal{L}(a \circ b)$ as a kind of pairing operator. The reason why this transformation helps is due to the following properties of $\mathcal{T}$ and $\otimes$:

\[
\mathcal{T}(a \otimes b) = a \mathcal{T} \otimes b \mathcal{T},
\]

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

\[
\mathcal{D}(e \otimes f) = e \mathcal{D} \otimes f \mathcal{D}.
\]

\footnote{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:

\[
R \otimes S = \begin{bmatrix}
  r_{1,1}S & \cdots & r_{1,N}S \\
  \vdots & \ddots & \vdots \\
  r_{N,1}S & \cdots & r_{N,N}S
\end{bmatrix}
\]

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

\[
(R \otimes S)[(a-1)N + b, (a'-1)N + b'] = R(a, a') \land S(b, b').
\]

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 $((1^T \otimes g) \circ (e_1^T \otimes f_1) \circ (e_2^T \otimes f_2))$, respectively.

To use this idea to solve an LCFL path problem precisely, there is one further requirement: the “detensor” operation must distribute over the (tensored) $\otimes$ operation. Because the $Z$ system’s paths encode all and only the LCFL paths of the $Y$ system, the detensor of the $\circ$-over-all-$Z$-paths equals the desired $\circ$-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:

\[
(1^T \otimes d) \circ (Z \otimes ((a^T \otimes b) \circ c)) \circ Z = (a^T \otimes b) \circ c.
\]

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:

\[
Z = (1^T \otimes d) ((a^T \otimes b) \circ c). \tag{4}
\]

The above process is iterated by converting $Z$’s value into the next approximant for $\nu$ (and hence for $X$) by $\nu = \mathcal{D}(\cdot, \cdot)(Z)$.
general, we want to find the least fixed-point of the inequality
\[ \nu \gtrsim q^{(t-1)}((1^t \odot d)( (a^t \odot b a \odot c) )^*) \] (5)
The Newton-iteration sequence for \( X \) is thus the Kleene-iteration sequence for Eqn. (5).

\[ \begin{align*}
\rho^{(0)} &= \emptyset \\
\rho^{(1)} &= q^{(t-1)}((1^t \odot d)( (a^t \odot b a \odot c) )^*) \\
\rho^{(2)} &= q^{(t-1)}((1^t \odot d)( (a^t \odot b a \odot c) )^*) \\
&\vdots
\end{align*} \]

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 \( D = (D, \equiv, \oplus, \otimes, *, 0, 1) \) is a set \( D \), equipped with an equivalence relation \( \equiv \), a binary combine operator \( \oplus \), a binary extend operator \( \otimes \), a unary closure operator \(*\), and distinguished elements \( 0 \) and \( 1 \) 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 \( a_1 \otimes b_1 \equiv a_2 \otimes b_2 \) and \( a_1 \oplus b_2 \equiv a_2 \oplus b_2 \)). Note that \( \equiv \) is not necessarily a congruence with respect to the closure operator \(*\).

2. \((D, \equiv, \oplus, \otimes, *, 0, 1)\) is an idempotent semiring “up to equivalence,” meaning that for all \( a, b, c \in D \) we have
   - (a) (Associativity) \((a \oplus b) \otimes c \equiv a \otimes (b \oplus c) \) and \((a \otimes b) \otimes c \equiv a \otimes (b \oplus c) \)
   - (b) (Unit) \( a \otimes 1 \equiv 1 \otimes a \equiv a \) and \( 0 \otimes a \equiv 0 \)
   - (c) (Commutativity) \( a \otimes b \equiv b \otimes a \)
   - (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) \( a \otimes a \equiv a \)

3. The closure operator over-approximates reflexive transitive closure, in the following sense. The combine operator of \( D \) defines a natural preorder \( \preceq \) on \( D \), where for any \( a \) and \( b \) in \( D \), \( a \preceq b \) if and only if \( a \otimes b \equiv b \).

   (a) (Reflexivity) \( 1 \preceq a \)
   (b) (Transitivity) \( a \preceq a \preceq a \)

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: " \( \preceq \) \( \succeq \) \( \succeq \) \( \succeq \) \( \succeq \). We also sometimes use \( a \in D \) 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 \( D = (D, \equiv, \oplus, \otimes, *, 0, 1) \) be a quasi weight domain and let \( X = \{X_1, \ldots, X_n\} \) be a finite set of variables. A system of inequalities with regular right-hand sides over \( D \) and \( X \) consists of one inequality for each variable

\[ X_i \gtrsim R_i \quad \cdots \quad X_n \gtrsim R_n \]

where each \( R_i \) is a regular expression over \( D \) and \( X \). (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 \( \sigma : X \rightarrow D \) such that for each \( X_i \in X \), we have \( \sigma(X_i) \gtrsim \sigma(R_i) \), where \( \sigma(R_i) \) denotes the natural extension of \( \sigma \) to regular expressions over \( D \) and \( X \).

The standard method for solving systems of recursive inequalities is Kleene iteration. Given a system of inequalities with regular right-hand sides over \( D \) and \( X \), \( S \), \( \{X_1 \gtrsim R_1\}_{i=1}^n \), the Kleene-iteration sequence (\( \kappa^k : X \rightarrow D \)\( k\in\mathbb{N} \)) for \( S \) is defined by

\[ \kappa^0(X_i) \equiv 0 \]
\[ \kappa^k(X_i) \equiv \kappa^{k-1}(R_i) \]

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 \( \kappa^k \) is a solution (in fact, the least solution) to \( S \). However, Kleene iteration requires two conditions on \( D \) 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 $D$ is eventually constant.

• Decidable equivalence: to detect convergence of the Kleene-iteration sequence, we must be able to determine whether two weights in $D$ 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., $X \mapsto a^i$ is a solution to the single-variable system $\{X \geq 1 \oplus (X \ominus a)\}$. 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, and 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 $D = \langle D, c, \oplus, \ominus, \odot, *, D_r, 0 \rangle$ be a quasi weight domain and let $X = \{X_1, \ldots, X_n\}$ be a finite set of variables. A system of inequalities with regular right-hand sides over $D$ and $X$, $S = \{X_i \geq R_i\}^n_{i=1}$, is **left-linear** if each expression $R_i (1 \leq i \leq n)$ is of the form

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

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

There are two reasons why left-linear systems are interesting: first, because they correspond to intraprocedural program-analysis problems, and second, because the solutions 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 $S = \{X_i \geq c_i \oplus \bigoplus_{j,k} X_j b_{ijk}\}^n_{i=1}$ over $D$ and $X$. First, we associate with $S$ a directed labeled graph $G = (V, E)$ where the set of vertices $V = X \cup \{r\}$ consists of the variables $X$ plus a designated root vertex $r \not\in X$. The set $E$ contains one edge $r \xrightarrow{c_i} X_i$ for each variable $X_i$, and one edge $X_j \xrightarrow{b_{ijk}} X_i$ 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

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

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 $D$, we can evaluate $R_i$ to a weight $w_i$. Letting $\sigma$ denote the function $X' \mapsto D$ that maps each $X_i$ to $w_i$, we have that $\sigma$ 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 interprocedural-analysis problems, however, these operators 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.

**Definition 4.4.** Let $D = \langle D, c, \oplus, \ominus, \odot, *, D_r, 0 \rangle$ be a quasi weight domain and let $X = \{X_1, \ldots, X_n\}$ be a finite set of variables. A system of inequalities with regular right-hand sides over $D$ and $X$, $S = \{X_i \geq R_i\}^n_{i=1}$, is **linear** if each expression $R_i (1 \leq i \leq n)$ is of the form

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

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

**Definition 4.5.** A **tensor-product domain** $\mathcal{F} = \langle D, D_r, \odot, \ominus, \oplus \rangle$ consists of two quasi weight domains $D$ and $D_r$ along with the following operations.

**A transpose operator**, denoted by $\odot : D \times D \rightarrow D_r$, such that for all $a, b \in D$ the following properties hold:

$$\begin{align*}
(a \odot b) & = a^i \odot b^j \\
(a \odot b)^i & = b^i \odot a^j \\
(a^i)^j & = a^{ij}
\end{align*}$$

and $\equiv$ is a congruence for $\odot$ (if $a \equiv b$ then $a^i \equiv b^j$).

**A tensor-product operator**, denoted by $\odot : D \times D \rightarrow D_r$, such that for all $a, b, c, a_1, b_1 \in D$, the following properties hold:

$$\begin{align*}
\odot & \equiv a \odot b \equiv \odot \equiv a \odot b \\
(a \odot b \odot c) & \equiv (a \odot (b \odot c)) \\
(b \odot a) & \equiv (a \odot (b \odot a)) \\
(a \odot b_1) \odot (a \odot b_2) & \equiv (a \odot (b_1 \odot b_2)) \equiv (a \odot b_2) \odot (b_1 \odot b_2).
\end{align*}$$

**A detensor-transpose operation** $\{t \} : D_r \rightarrow D$, such that for all $a, b, c \in D$ and $p, q \in D_r$ the following properties hold:

$$\{t \} (p \odot q) \equiv \{t \} (p) \odot \{t \} (q)$$

and for all $p \equiv q$, we have $\{t \} (p) \equiv \{t \} (q)$.

The operation to couple pairs of values from a tensor-product domain, denoted by $\odot : D \times D \rightarrow D_r$, is defined as follows:

$$C(a_1, b_1) \equiv a_1 \odot b_1.$$

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

$$C(a_1, b_1) \odot \odot C(a_2, b_2) = (a_1 \odot b_1) \odot (a_2 \odot b_2) \equiv (a_1 \odot a_2) \odot (b_1 \odot b_2) \equiv C(a_2 \odot a_1) \odot (b_1 \odot b_2) \equiv C(a_2 \odot a_1, b_1 \odot b_2).$$

Note the order reversal in the first argument in the last line of Eqn. (8). $a_2 \odot a_1$ is, in fact, the right order of the operands of $\odot$ in the first line.

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

$$S_r = \{Z_i \geq (c_i \oplus \bigoplus_{j,k} a_{ijk} \odot Z_j \ominus b_{ijk})\}^n_{i=1}.$$

Let $\sigma_r$ be any solution to $S_r$. Then the map $\sigma : X' \mapsto D$ that sends each $X_i$ to $\{t \} (\sigma_r(Z_i))$ is a solution to $S$. 

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.
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 $\phi(x, x')$ over some specified set of variables $x$ and $x'$. Making the program variables explicit, we use $\text{CRA}(x)$ to denote the quasi weight domain of transition formulas over $x$ and $x'$, with operators defined as in §2.1. The tensor-product domain $\mathcal{T}(x) = (D, D_\tau, \cdot, \otimes, \cdot, \tau, \cdot') \tau$ of CRA is similarly parameterized by a set of variables $x$. The weight domain $D$ is taken to be CRA($x$), and the tensor weight domain $D_\tau$ is taken to be $\text{CRA}(x \otimes x)$, where $x$ and $x$ denote disjoint copies of the set of program variables $x$. That is, the tensor domain consists of transition formulas $\phi(x \otimes x \otimes x \otimes x)$ 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 $x \cup x$, or
2. As a set of pairs of transitions between states over the set of variables $x$.

It is helpful to use the first interpretation when thinking about the $\otimes$ and $\cdot$ operations in CRA($x \otimes x$), and the second interpretation when thinking about tensor product and detensor-transpose. Under the first interpretation, $\tau$ is the operation for finding closed forms of recurrence equations described in §2.1, except that it operates over transitions from $x \cup x$ to $x \cup x$. The tensor product is simply the Cartesian product under the second interpretation. This operation can be defined syntactically with variable renaming and conjunction:

\[
\phi(x, x') \otimes \psi(x, x') \equiv \phi(x, x') \land \psi(x, x').
\]

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:

\[
\phi(x, x') \tau \equiv \exists x'' . \phi(x'', x'', xx').
\]

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

\[
\phi(x, x') \tau \equiv \phi(x', x).
\]

4.4 Non-Linear Systems

This section describes a method for solving 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:

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

\[
\text{guard}(x') \equiv \exists x . (\text{guard}(x) \land \text{action}(x, x')).
\]

In other words, $(\text{guard}(x), \text{action}(x, x')) \equiv (\text{guard}(x'), \text{action}(x', x))$.

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

- 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 $\phi$ representing the body of a loop, we (1) compute a set of recurrences by simulating the program by an SMT solver, and (2) compute closed forms for the recurrences to use as an approximation of the transitive closure of $\phi$. The loop-body formula $\phi$ is expressed in a rich assertion language, which includes disjunction, quantifiers, and non-linear terms. The recurrences computed by phase (1), on the other hand, are relatively simple: each recurrence (e.g., $x' = x + 1$, representing that $x$ is incremented in the loop) is a linear constraint, and so a set of recurrences can be represented by a convex polyhedron. Thus, we can express the iteration operator of CRA as the composition of (i) an abstraction function that computes a polyhedron (representing recurrences) from a transition formula, and (ii) an abstraction-closure function that computes a transition formula from a polyhedron by computing closed forms. Polyhedra are 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 $D = (D, \otimes, \oplus, \land, \lor, 0, 1)$ be a quasi weight domain. An iteration domain for $D$, $D_\tau = (D, \leq, \lor, \land, 0, 1)$, is a partially ordered set $(D^3, \leq)$ along with the following operations.

An abstraction operator $\alpha : D \rightarrow D^3$ and an abstraction-closure operator $\cl : D^3 \rightarrow D$ such that the following properties hold:

1. For all $a \in D$, $\cl(\alpha(a)) = a^*$
2. For all $a^2, b^2 \in D^3$ such that $a^2 \leq b^2$, we have $\cl(a^2) \leq \cl(b^2)$.

A widening operator $\lor : D^3 \times D^3 \rightarrow D^3$ such that the following properties hold:

1. For all $a^2, b^2 \in D^3$, $a^2 \lor b^2$ is an upper bound of $a^2$ and $b^2$ ($a^2 \leq a^2 \lor b^2$ and $b^2 \leq a^2 \lor b^2$)
2. For every infinite sequence $(a^2)_r \in D^3$, the ascending chain defined as

\[
b^2 = a^2_1 \rightarrow a^2_{r+1} = a^2_r \lor a^2_{r+1}
\]

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:

\[
S \{ x \geq d \oplus (a \otimes X \oplus b \otimes X \oplus e) \}.
\]

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-
stant (much like ρ 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:

\[ S_T : \{ Z \geq \left( 1^t \odot d \right) \oplus_T (a^t \odot (b \otimes X \otimes c)) \} \]  

where \( X = \{ t \} \) (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 X in terms of X (see Fig. 3(a)),

\[ Z \mapsto \left( 1^t \odot d \right) \oplus_T (a^t \odot (b \otimes X \otimes c))^T, \]

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

\[ X \mapsto \left( 1^t \odot d \right) \oplus_T (a^t \odot (b \otimes X \otimes c))^T. \]

Thus, the least solution of the system

\[ \hat{S} : \{ X \geq \left( 1^t \odot d \right) \oplus_T (a^t \odot (b \otimes X \otimes c))^T \} \]

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 \( \{ \kappa^k : X \to D \}_{k \in \mathbb{N}} \) denote the Kleene-iteration sequence for \( \hat{S} \), then we can define a “derived sequence” \( \{ \beta^k \}_{k \in \mathbb{N}} \) in the iteration domain \( D^T \):

\[ \beta^k \equiv \alpha(a^t \odot (b \otimes \kappa^k(X) \otimes c)). \]

We observe that if the derived sequence converges (i.e., \( \beta^k = \beta^{k+1} \) for some \( k \)), then so does the Kleene-iteration sequence \( \{ \kappa^k(X) \equiv \kappa^k(X) \} \). 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 \( \mathcal{F} = \{ D, D_T, \cdot, \odot, \{ t, ◁ \} \} \) be a tensor-product domain. We define the detensor-product operator \( \times : D \times D_T \to D \) as follows:

\[ a \times p \equiv \frac{1}{r}(1^t \odot a) \otimes_T p. \]

It is easy to show that for all \( a, b \in D \) and all \( p, q \in D_T \) we have

\[ a \times (p \oplus_T q) \equiv (a \times p) \oplus_T (a \times q) \]

\[ a \times (p \odot_T (b \otimes c)) \equiv b \odot_T (a \times p) \otimes_T c \]

and if \( a \equiv b \equiv p \equiv q \) then \( a \times p \equiv b \times q \). 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 \( p, q \in D_T \):

\[ a \times (p \oplus_T q) \equiv (a \times p) \odot_T q. \]

(Inclusion of this axiom makes \( \langle D, \times \rangle \) a right \( D_T \)-module up to equivalence—i.e., \( \times \) 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 \( \mathcal{F} = \{ D, D_T, \cdot, \odot, \{ t, ◁ \} \} \) be a tensor-product domain, and let \( X = \{ X_1, \ldots, X_n \} \) be a finite set of variables. A (tensored) extended regular expression over \( \mathcal{F} \) and \( X \) is an expression generated by the following grammar:

\[ E, F \in \text{Ext}(\mathcal{F}) \quad ::= \quad w \in D \mid X_i \in X \mid E \oplus F \mid E \odot F \mid E \oplus_T F \mid | E^* \mid | E \otimes_{T} F \]

An extended regular expression \( E \) over \( \mathcal{F} \) is normal if for every subexpression of \( E \), the form \( E \otimes_{T} F \) is of the form \( F \otimes_{T} a \).

For extended regular expressions \( E \) and \( F \), we write \( E \simeq F \) to denote that for every function \( \sigma : X \to D \), 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 unguarded occurrence is called free. Formally, for any (tensored) extended regular expression, its set of free variables is defined as follows:

\[ \text{free}(w) \equiv \emptyset \]

\[ \text{free}(X_i) \equiv \{ X_i \} \]

\[ \text{free}(E \oplus F) \equiv \text{free}(E) \cup \text{free}(F) \]

\[ \text{free}(E \odot F) \equiv \text{free}(E) \cup \text{free}(F) \]

\[ \text{free}(E^*) \equiv \emptyset \]

\[ \text{free}(E \times F) \equiv \text{free}(E) \cup \text{free}_{\otimes T}(F) \]

\[ \text{free}_{\otimes T}(E \otimes F) \equiv \text{free}_{\otimes T}(E) \cup \text{free}_{\otimes T}(F) \]

\[ \text{free}_{\otimes T}(E \otimes T \otimes F) \equiv \text{free}_{\otimes T}(E) \cup \text{free}_{\otimes T}(F) \]

\[ \text{free}_{\otimes T}(E \otimes ^* T \otimes F) \equiv \emptyset \]

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 \( X \simeq F \otimes X \otimes FR \), which is “symbolically” left-linear in \( X \).

\footnote{Alternatively, we could have defined detensor-transpose in terms of detensor-product as \( \{ t, ◁ \}(p) \equiv 1^t \times p. \)}
Proposition 4.13. Let $\mathcal{F} = (\mathcal{D}, \mathcal{D}, \mathcal{F}, \mu, \tau, \rho, \delta)$ be a tensor-product domain, let $X = \{X_1, \ldots, X_n\}$ be a finite set of variables, and let $E$ be a normal extended regular expression over $\mathcal{F}$ and $X$. For each variable $X \in X$, there exist extended regular expressions $F$ and $F'$ such that

1. $E \simeq F \cup (X \times F')$.
2. $F$ is normal, and
3. $X$ is not free in $E$.

Proof: By structural induction on $E$.

- Case $w \in \mathcal{D}$: we have $w \simeq w \cup (X \times \emptyset)$.
- Case $X$: we have $X \simeq \emptyset \cup (X \times \emptyset)$.
- Case $X \in X$: for some $X \in X$, we have $X \simeq X \cup (X \times \emptyset)$.

The last line of each case of the proof can be used to create a 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.

PROPOSITION 4.12. Let $E = (\mathcal{D}, \mathcal{D}, \mathcal{F}, \mu, \tau, \rho, \delta)$ be a tensor-product domain, let $X = \{X_1, \ldots, X_n\}$ be a finite set of variables, and let $E$ be a normal extended regular expression over $\mathcal{F}$ and $X$. For each variable $X \in X$, there exist extended regular expressions $F$ and $F'$ such that

- $E \simeq F \cup (X \times F')$.
- $F$ is normal, and
- $X$ is not free in $E$.

Proof: By structural induction on $E$.

- Case $w \in \mathcal{D}$: we have $w \simeq w \cup (X \times \emptyset)$.
- Case $X$: we have $X \simeq \emptyset \cup (X \times \emptyset)$.
- Case $X \in X$: for some $X \in X$, we have $X \simeq X \cup (X \times \emptyset)$.

Then:

$$E \simeq F \cup (X \times F')$$

Case $E \cup E'$: by the induction hypothesis, there is some $F, F', F''$ such that $E \simeq F \cup (X \times F')$ and $E' \simeq F' \cup (X \times F'')$.

Then:

$$E \cup E' \simeq F \cup (X \times F') \cup F' \cup (X \times F'')$$

Case $F \cup F'$: by the induction hypothesis, there is some $F, F'$ such that $E \simeq F \cup (X \times F')$ and $E' \simeq F' \cup (X \times F'')$.

Completion extends to tensor-product domains in the natural way: for a tensor-product domain $\mathcal{F} = (\mathcal{D}, \mathcal{D}, \mathcal{F}, \mu, \tau, \rho, \delta)$ its completion domain $\mathcal{C}(\mathcal{F})$ is $(\mathcal{C}(\mathcal{D}), \mathcal{C}(\mathcal{D}), \mathcal{C}(\mathcal{F}), \mu, \tau, \rho, \delta)$ where $\mathcal{C}(\mathcal{D})$ and $\mathcal{C}(\mathcal{F})$ are defined pointwise.

Let $\delta_{\mathcal{C}} : \mathcal{X} \to \mathcal{C}(\mathcal{D})$ denote the least solution of $\hat{S}$. We argue that $\delta_{\mathcal{C}}$ 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 $\delta_{\mathcal{C}}$ is the least solution to $S_k$ and prove that it is the least solution to $S_k$. Rewrite $E_{k,k}$ as $E_{k,k} \simeq F \cup (X_k \times F')$ (cf. Prop. 4.12). Because $\delta_{\mathcal{C}}$ is the least solution to $S_{k+1}$, we must have

$$\delta_{\mathcal{C}}(X_k) = \delta_{\mathcal{C}}(E_{k,k+1}) \simeq \delta_{\mathcal{C}}(F \cup F' \tau)$$.
We then have:
\[
\sigma_C(X_k) = \sigma_C(F \times F_T^{\tau T}) = \sigma_C(F) \times C(F_T^{\tau T}) = \sigma_C(F) \times \sigma_C(F_T^{\tau T}) \times C(T) \sigma_C(F_T^{\tau T}) \times C(T)
\]
\[
= (\sigma_C(F) \times C(F_T^{\tau T}) \times C(T)) \sigma_C(F_T^{\tau T}) \times C(T)
\]
and thus \(\sigma_C\) solves the constraint \(X_k \geq E_{k,k}\).

For every \(j \neq k\), the right-hand-sides \(E_{j,k}\) and \(E_{j,k+1}\) differ only by substitution of \(F \times F_T^{\tau T}\) for \(X_k\), so because \(\sigma_C(X_k) = \sigma_C(F \times F_T^{\tau T})\) we have \(\sigma_C(E_{j,k}) = \sigma_C(E_{j,k+1})\). It follows from the fact that \(\sigma_C(X_k) \geq \sigma_C(E_{j,k})\) that \(\sigma_C(X_k) \geq \sigma_C(E_{j,k+1})\).

We have thus shown that \(\sigma_C\) is a solution to \(S_{k+1}\).

Because \(\sigma_C\) and \(\sigma_C\) are both solutions to \(S_k\) and \(S_{k+1}\), \(\sigma_C\) is the least solution to \(S_k\), and \(\sigma_C\) is the least solution to \(S_{k+1}\). We have \(\sigma_C = \sigma_C\), and \(\sigma_C\) 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 \(\mathcal{F} = \langle \mathcal{D}, \mathcal{D}_T, \mathcal{I}^1, \mathcal{I}^2 \rangle\) be a tensor-product domain, let \(X = \{X_1, \ldots, X_n\}\) be a finite set of variables, and let \(S : \{X_i \geq E_i\}_{i=1}^n\) be a system of inequalities in which each \(E_i\) is an extended regular expression over \(\mathcal{F}\) and \(\mathcal{X}\) with no free variables.

Define a sequence of functions \(\sigma^k : X \to \mathcal{D}\) as follows. The function \(\sigma^0\) is defined to be the constant \(0\) function. For each \(k \geq 1\), define
\[
\sigma^k(X_i) = \text{eval}^k(E_i)
\]
where
\[
\text{eval}^0(w) = w
\]
\[
\text{eval}^k(E_i) = \sigma^{k-1}(X_i)
\]
and
\[
\text{eval}^{k}(E\times F) = \text{eval}^{k}(E) \times \text{eval}^{k}(F)
\]
\[
\text{eval}^{k}(E \ast F) = \text{eval}^{k}(E) \ast \text{eval}^{k}(F)
\]
\[
\text{eval}^{k}(E \ast T) = \text{eval}^{k}(E) \ast T_{\text{eval}^{k}(F)}
\]
and
\[
\text{body}^{k}(E) = \text{body}^{k-1}(E) \vee \alpha^{k-1}(E)
\]
\[
\text{body}^{k}_{\land}(E) = \text{body}^{k-1}_{\land}(E) \land \alpha^{k-1}(E).
\]
We say that \(S\) stabilizes at \(k\) if for every subexpression of the form \(E^*\) that appears in any \(E_i\), we have
\[
\text{body}^{k}_{\land}(E) = \text{body}^{k-1}_{\land}(E)
\]
and for every subexpression of the form \(F_T^{\tau T}\) that appears in any \(E_i\), we have
\[
\text{body}^{k}_{\land}(E) = \text{body}^{k-1}_{\land}(E).
\]

THEOREM 4.15. If \(S\) stabilizes at \(k\), then \(\sigma^k\) is a solution to \(S\).

Proof: To show that \(\sigma^k\) is a solution to \(S\), we must demonstrate that \(\sigma^k(X_i) \geq \sigma^k(E_i)\) for all \(i\). The essence of the argument is to show that the following sequence of (in)equalities is valid:
\[
\sigma^k(X_i) = \text{eval}^k(E_i) \geq \sigma^{k-1}(E_i) = \sigma^k(E_i).
\]
The first equality in the sequence holds by the definition of \(\sigma^k\). The middle inequality we prove by induction on \(E_i\). We prove validity of the last equation by showing that the assumption that \(S\) stabilizes at \(k\) (i.e., the fact that \(\text{body}^{k}_{\land}(E) = \text{body}^{k-1}_{\land}(E)\) for every “loopy” expression \(E\)) is sufficient to prove that \(\sigma^{k-1} = \sigma^k\).

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

Case \(w \in \mathcal{D}\), \(X_i \in \mathcal{X}\): trivial.

Case \(E \oplus F\), \(E \ast F\), \(E \times F_T\), \(E T_F\), \(E T_F : F\): from the induction hypothesis and monotonicity.

Case \(E^*\) (\(E_T^{\tau T}\) is similar): First, observe that by property 1 of the widening operator, we have
\[
\text{body}^{k-1}_{\land}(E) \lor \alpha^{k-1}(E) \geq \alpha^{k-1}(E).
\]
It follows that:
\[
\text{eval}^k(E^*) = cl(body^k(E)) \quad \text{Definition}
\]
\[
= cl(body^{k-1}(E) \cup \alpha(k^{-1}(E))) \quad \text{Definition}
\]
\[
\geq cl(\alpha(k^{-1}(E))) \quad \text{Monotonicity}
\]
\[
= \sigma^{k-1}(E^*) \quad \text{Closure axiom}
\]
\[
= \sigma^{k-1}(E^t) \quad \text{Definition}
\]

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

- Case \(w \in \mathcal{T}\): trivial.
- Case \(E \oplus F\): \(E \circ F\), \(E \times F\), \(E^i \circ F\), \(E^i \odot F\), \(E^i \oplus F\), \(F^i \oplus F\), \(F^i \odot F\): immediately from the induction hypothesis.
- Case \(E^*\) (Case \(E^{*T}\) is similar): Because \(S\) stabilizes at \(k\), we have \(body^k(E) = body^{k-1}(E)\). It follows that \(cl(body^k(E)) = cl(body^{k-1}(E))\), and thus \(\text{eval}^k(E^*) = \text{eval}^{k-1}(E^*)\).

**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 \(E^{*T}\) is similar). We must show that the sequence \(\langle body^i(E) \rangle_{i \in \mathbb{N}}\) eventually stabilizes. Consider the infinite sequence \(\langle \alpha(s^i(E)) \rangle_{i \in \mathbb{N}}\) in \(D^2\). Because \(\cup\) is a widening operator, the sequence
\[
b_t \geq \alpha(s^0(E)) \quad b_t \geq \alpha(s^0(E))
\]
evitably stabilizes. Finally, observe that the sequence \(\langle b^i \rangle_{i \in \mathbb{N}}\) coincides with the sequence \(\langle body^i(E) \rangle_{i \in \mathbb{N}}\).

**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 \(\mathcal{T} = (\mathcal{D}, \mathcal{D}_r, \cdot, \cdot, \cdot, \cdot, \cdot)\) and a system of inequalities \(S: \{X_i \geq R_i\}_{i=1}^n\) with regular right-hand sides, over \(\mathcal{D}\) and a finite set of variables \(\mathcal{X} = \{X_1, \ldots, X_n\}\). The output is a mapping \(\sigma: \mathcal{X} \rightarrow \mathcal{D}\) such that if \(\mu\) is the least solution to \(S\), then for all \(i\) and \(a \in \mu(X_i), \sigma(X_i) \geq a\).

1. Perform Alg. 4.13 to transform \(S\) into a system of inequalities \(S: \{X_i \geq E_i\}_{i=1}^n\), in which each \(E_i\) is an extended regular expression over \(\mathcal{T}\) and \(\mathcal{X}\) that has no free variables.
2. \(k \leftarrow 0; \sigma^0 \leftarrow \text{ar} \emptyset\)
3. Repeat
   \(a\) \(k \leftarrow k + 1\)
   \(b\) \(\sigma^k(X_i) \leftarrow \text{eval}^k(E_i)\)
   until \(S\) stabilizes
4. Return \(\sigma^k\)

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 \(S\) 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 pushdown system. We instantiated NPA-TP-GJ for ICRA by coupling WALi to the CRA abstract domain\(^9\) [13], and augmenting the implementation of CRA (which, in turn, makes use of APRON [1] and Z3 [10]) with \(\cdot, \odot\), and \(\otimes(\cdot, \cdot)\) 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 procedure 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. A1], 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

\(^9\) 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.

\(^10\) We used a timeout limit of 300 seconds; at SV-COMP16, the timeout limit was 900 seconds.
### 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.)

The procedure in Fig. 4(a) illustrates the application of ICRA to resource-bound analysis. Statements of the form \( \text{tick}(k) \) represent manipulations of a resource such as memory or time, by consuming some of the 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. 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 \( \max(x, 0) \) 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 \( \text{perform} \), ICRA computes the following four bounds, each of which is achieved by following paths of a particular form:

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

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

On the C4B test suite, ICRA is able to generate upper bounds on resource usage for 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 \( \text{final} \leq 1 + 2[0, n] \)):

\[
\text{final} \leq [0, n] \quad \text{final} \geq [0, n]
\]

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 \( X \preceq F \otimes X \times F \), and then further rewritten as \( X \preceq F \times F \). 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

---

**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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>CRACK</th>
<th>UAutomizer</th>
<th>CPA</th>
<th>Time (s)</th>
<th>Time (s)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>122</td>
<td>9</td>
<td>2</td>
<td>11</td>
<td>1,489.72</td>
<td>1,172.75</td>
<td>1,800.45</td>
</tr>
<tr>
<td>Rec.(tot.)</td>
<td>173</td>
<td>150</td>
<td>20</td>
<td>13</td>
<td>2,587.27</td>
<td>1,282.44</td>
<td>4,324.60</td>
</tr>
<tr>
<td>loops (tot.)</td>
<td>391</td>
<td>377</td>
<td>14</td>
<td>10</td>
<td>4,749.78</td>
<td>3,307.88</td>
<td>8,007.61</td>
</tr>
<tr>
<td>Misc.-Rec.</td>
<td>198</td>
<td>176</td>
<td>22</td>
<td>10</td>
<td>3,025.38</td>
<td>1,304.65</td>
<td>4,324.60</td>
</tr>
</tbody>
</table>

*Figure 4:* (a) Recursive function used to illustrate upper and lower bounds on resource usage. (b) The speed_pld10_ex4.c benchmark from the C4B suite.

---
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 ? 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 postimages 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 ?. Finally, it would be desirable to be able to perform witness tracing for ICRA.

References


