Algebraic Program Analysis

Zachary Kincaid[1](✉), Thomas Reps[2](✉), and John Cyphert[2](✉)

[1] Princeton University, Princeton, NJ 08540, USA
zkincaid@cs.princeton.edu
[2] University of Wisconsin, Madison, WI 53706, USA
reps@cs.wisc.edu, jcyphert@wisc.edu

Abstract. This paper is a tutorial on algebraic program analysis. It explains the foundations of algebraic program analysis, its strengths and limitations, and gives examples of algebraic program analyses for numerical invariant generation and termination analysis.

1 Introduction

This tutorial provides an introduction to algebraic program analysis, focusing upon techniques for (numerical) invariant generation and termination analysis. By reading this paper, you will learn the answers to the following questions:

- How does one design an algebraic program analysis?
- What new opportunities does algebraic program analysis enable?
- What are the limitations and important open problems in algebraic program analysis?

The origin of algebraic program analysis is the algebraic approach to solving path problems in graphs [1,6,48,59]: (1) compute a regular expression recognizing a set of paths of interest, and (2) interpret that regular expression within an algebraic structure corresponding to the problem at hand. Various path problems (e.g., computing shortest paths, path-finding problems, and dataflow analysis) can be solved by using different algebraic structures to interpret regular expressions.

In the context of program analysis, the graph of interest is a control flow graph for a program, and the algebra defines a space of summaries (approximations of program behavior) and a means for composing them. The algebraic approach amounts to computing a summary for a program in "bottom-up" fashion, building summaries for larger and larger subprograms by applying the operators of the summary algebra.

The general pattern of an algebraic program analysis is: given a system of (recursive) equations defining the semantics of a program, (1) symbolically compute a closed-form solution, and then (2) interpret the closed form within an algebraic structure corresponding to the analysis. The algebraic approach can be contrasted with classical iterative abstract interpretation, which also starts with a system of (recursive) equations defining the semantics of a program. However, the iterative approach is to (a) interpret the operations in the equations in an abstract domain, and then (b) solve the equations over the abstract domain

A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 46–83, 2021.
https://doi.org/10.1007/978-3-030-81685-8_3

by successive approximation. Thus, the classical approach is one of “interpret and then solve,” whereas the algebraic approach is “solve and then interpret.”

The algebraic approach can be applied to various kinds of equations and algebraic structures. Three cases we consider in this article, and the corresponding kind of program-analysis problems they can be used to solve, are:

Section 2 (Non-recursive) program summarization: left-linear equations over regular algebras.
Section 4 Linearly-recursive procedure summarization: linear equations over tensor-product domains.
Section 5 Conditional termination analysis: right-linear equations over ω-regular algebras.

Why Algebraic Program Analysis? Algebraic program analysis is a general framework for understanding compositional program analyses. The principle of compositionality states that “the meaning of a complex expression is determined by its structure and the meanings of its constituents” [57]. A program analysis is compositional when the result of analyzing a composite program is a function of the results of analyzing its components. Compositionality enables program analyses to scale to large programs, to be parallelized, to be applied incrementally, and to be applied to incomplete programs [18]. Algebraic program analysis provides a structure in which to think about how to design such an analysis.

Insistence upon compositionality also demands a different perspective on program analysis, which can suggest solutions to problems that may otherwise not be apparent. We demonstrate this principle with a series of examples that illustrate a variety of different ideas that are enabled by thinking of program analysis in compositional terms.

Last, the algebraic framework enables a style of reasoning about the behavior of program analyses themselves. By exploiting compositionality, it is possible to design effective algebraic analyses that satisfy certain laws (e.g., monotonicity—“more information in yields more information out”). Analyses can be classified on the basis of algebraic laws that they satisfy, and we can reason how program transformation affects analysis using these laws.

Why Not Algebraic Program Analysis? While compositionality brings many desirable properties, it comes at the price of losing *context*. Compositionality requires that the analysis of a program component is a function of the source code of that component, and therefore *cannot* depend on the surrounding context in which the component appears in the program. Many program analysis techniques make essential use of context, for example:

– In an iterative abstract interpreter, which propagates information about reachable states from the program entry forwards, the analysis of a component depends on every component that may precede it in an execution.
– In a refinement-based software model checker, which inspects paths that go from entry to an error state, the analysis of a component depends on the whole program.

One of the main challenges of designing a good algebraic program analysis is to overcome this loss of contextual information.

Secondly, algebraic program analysis is less general than iterative program analysis, in the sense that any set of semantic (in)equations can be solved iteratively using the same basic algorithm, whereas each particular type of equation system requires a specialized algorithm. Some problems—e.g., resolving semantic equations of recursive procedures—have no known practical algebraic solutions.

2 Regular Algebraic Program Analysis

This section describes the algebraic approach to solving path problems in graphs [1,6,48,59]. The basic structure of the method is to use regular expressions to capture the set of paths of a graph, and then *interpret* these expressions to obtain a desired result. We illustrate the approach by considering the problem of computing shortest paths, and then show how it can be applied to numerical invariant generation.

First, we establish some basic definitions. The syntax of **regular expressions** over an alphabet Σ is as follows:

$$\begin{aligned} a &\in \Sigma \\ R \in \mathsf{RegExp}(\Sigma) &::= a \mid 0 \mid 1 \mid R_1 + R_2 \mid R_1 \cdot R_2 \mid R^* \end{aligned}$$

We will sometimes use juxtaposition R_1R_2 (rather than $R_1 \cdot R_2$) to denote concatenation.

The semantics of regular expressions over Σ is given by a **Σ-interpretation** $\mathscr{I} = \langle \mathbf{A}, f \rangle$, which consists of *regular algebra* $\mathbf{A}$ and a *semantic function* f. A **regular algebra** $\mathbf{A} = \left\langle A, 0^A, 1^A, +^A, \cdot^A, *^A \right\rangle$ is an algebraic structure consisting of a set A (called its *universe*) equipped with two distinguished elements $0^A, 1^A \in A$, two binary operations $+^A$ (*choice*) and $\cdot^A$ (*sequencing*), and a unary operation $(-)^{*^A}$ (*iteration*).[1] When the algebra is clear from context, we will drop the superscript. A **semantic function** $f : \Sigma \to A$ maps each letter in Σ to an element of $\mathbf{A}$'s universe.

A Σ-interpretation $\mathscr{I} = \langle \mathbf{A}, f \rangle$ assigns to each regular expression R over Σ to an element $\mathscr{I}[\![R]\!]$ of $\mathbf{A}$ by interpreting each letter according to the semantic function and each regular operator using its counterpart in $\mathbf{A}$:

$$\begin{aligned} &\mathscr{I}[\![0]\!] = 0^A & \mathscr{I}[\![R_1 \cdot R_2]\!] &= \mathscr{I}[\![R_1]\!] \cdot^A \mathscr{I}[\![R_2]\!] \\ &\mathscr{I}[\![1]\!] = 1^A & \mathscr{I}[\![R_1 + R_2]\!] &= \mathscr{I}[\![R_1]\!] +^A \mathscr{I}[\![R_2]\!] \\ &\mathscr{I}[\![a]\!] = f(a) \quad \text{For } a \in \Sigma & \mathscr{I}[\![R^*]\!] &= \mathscr{I}[\![R]\!]^{*^A} \end{aligned}$$

Notice that the interpretation is compositional: for any expression R, $\mathscr{I}[\![R]\!]$ is a function of the top-level operator in R and the interpretations of its subexpressions.

[1] Note that no particular laws are assumed to govern these operations. We will return to this issue in Sect. 3.

Example 1 (Standard interpretation). The *standard* interpretation of regular expressions is the language interpretation, $\mathscr{L} = \langle \mathbf{L}, \ell \rangle$ where $\mathbf{L}$ is the regular algebra of languages. The universe of the interpretation is the set of regular languages over Σ, $0 \triangleq \emptyset$ is the empty language, $1 \triangleq \{\epsilon\}$ is the singleton language containing the empty word, and the operators are

$$\begin{aligned} X + Y &\triangleq X \cup Y && \text{Union} \\ X \cdot Y &\triangleq \{xy : x \in X, y \in Y\} && \text{Concatenation} \\ X^* &\triangleq \{x_1 x_2 \dots x_n : x_1, \dots, x_n \in X\} && \text{Kleene closure} \end{aligned}$$

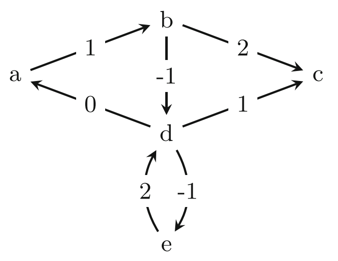
The semantic function ℓ maps each letter a to the singleton language $\{a\}$. For any regular expression R, $\mathscr{L}[\![R]\!]$ is the (regular) set of words recognized by R. ⌟

We now describe how *non-standard* interpretations can be used to solve problems over directed graphs. A **directed graph** $G = \langle V, E \rangle$ consists of a finite set of vertices V and a finite set of directed edges $E \subseteq V \times V$. A **path** in G is a finite sequence $e_1 e_2 \dots e_n$ with $e_i \in E$ such that for each i, the destination of e_i matches the source of e_{i+1}. A **path expression** (in G) is a regular expression over the alphabet of edges E that recognizes a set of paths in G. For any pair of vertices $u, v \in V$, there is a path expression $\mathit{PathExp}_G(u, v)$ that recognizes exactly the set of paths in G that begin at u and end at v. There are several ways to compute path expressions. The classical method is Kleene's algorithm [44] for computing a regular expression for a finite state automaton (thinking of G as an automaton over the alphabet E with start state u and final state v). For sparse graphs, there are more efficient alternatives to Kleene's algorithm, in particular Tarjan's algorithm [58]. The insight of the algebraic approach to path problems is that these algorithms can be re-used for multiple purposes: first use a path expression algorithm to find a regular expression recognizing a set of paths of interest, and then compute a problem-dependent (non-standard) interpretation of that expression.

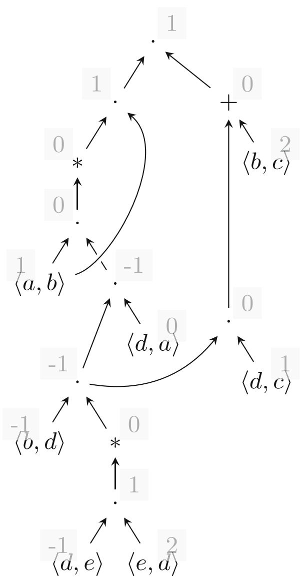
Example 2 (Shortest paths). Consider the integer-weighted graph depicted in Fig. 1a. Suppose that we wish to compute the length of the shortest path from a to c. We begin by computing a path expression recognizing all paths from a to c:

$$\left(\langle a,b\rangle \langle b,d\rangle \left(\langle d,e\rangle \langle e,d\rangle\right)^* \langle d,a\rangle\right)^* \langle a,b\rangle \left(\langle b,c\rangle + \langle b,d\rangle \left(\langle d,e\rangle \langle e,d\rangle\right)^* \langle d,c\rangle\right)$$

This path expression can be represented succinctly by the directed acyclic graph (DAG) pictured in Fig. 1b. Define the *distance interpretation* $\mathscr{D}$ where the semantic function maps each edge to its weight, and the algebra's universe consists of the integers along with $\pm\infty$, 0 is interpreted as ∞, 1 as 0, and the operators are as follows:



(a) An integer-weighted graph



(b) Path expression DAG

Fig. 1. An integer weighted graph and a path expression DAG representing the paths from a to c

$$\begin{aligned} d_1 + d_2 &\triangleq \min(d_1, d_2) && \text{Minimum} \\ d_1 \cdot d_2 &\triangleq d_1 + d_2 && \text{Addition} \\ d^* &\triangleq \begin{cases} -\infty & \text{if } d < 0 \\ 0 & \text{otherwise} \end{cases} && \text{Closure} \end{aligned}$$

The weight of the shortest weighted path from a to c is $\mathscr{D}[\![PathExp_G(a, c)]\!] = 1$, which can be calculated efficiently by interpreting the path expression DAG "bottom-up" (see gray labels in Fig. 1b). $\lrcorner$

Algebraic path-finding can be used to generate invariants by representing a program by a *control flow graph*, and interpreting path expressions within an algebra of program summaries. A **control flow graph** (CFG) $G = \langle V, E, r, C\rangle$ is a directed graph $\langle V, E\rangle$ with a distinguished root (or entry) vertex $r \in V$, and where each edge $e \in E$ is labeled by a command $C(e)$; see Fig. 2a for an example. In the remainder of this section, we give examples of interpretations that can be used to generate (numerical) program summaries.

2.1 Transition-Formula Interpretations

Fix a finite set of variables, X, representing the variables of a program. A **transition formula** is a logical formula $F(X, X')$ whose free variables range over X

and a set of "primed copies" $X' \triangleq \{x' : x \in X\}$. For the purposes of this exposition, we further suppose that variables range over integers, and that transition formulas are expressed in the language of linear integer arithmetic. A transition formula can be interpreted as a binary relation $\rightarrow_F$ over states $\mathsf{State} \triangleq \mathbb{Z}^X$, where $s \rightarrow_F s'$ if and only if F is true when s is used to interpret the un-primed variables and s' is used to interpret the primed variables. For example, if F is the transition formula

$$F \triangleq x' = x + 1 \land y = y' \land x < y \ ,$$

then we have

$$s \rightarrow_F s' \iff s'(x) = s(x) + 1, s(y) = s'(y), \text{ and } s(x) < s(y) \ .$$

Suppose that $G = \langle V, E, r, C\rangle$ is a control flow graph, where commands range over assignments x `:=` e and assumptions `[`c`]`, where e is a linear integer term and c is a linear arithmetic formula. (An assumption `[`c`]` is a command that does not change the program state, but which can only be executed if the formula c holds.) We define a semantic function tf that maps each control flow edge into the universe of transition formulas by translating the command associated with the edge into logic:

$$\mathit{tf}(e) \triangleq \begin{cases} (x' = e) \land \left(\bigwedge_{y \neq x \in X} y' = y\right) & \text{if } C(e) \text{ is } x \texttt{ := } e \\ c \land \left(\bigwedge_{y \in X} y' = y\right) & \text{if } C(e) \text{ is } \texttt{[}c\texttt{]} \end{cases}$$

We define an algebra of transition formulas as follows:

$$\begin{aligned} 0 &\triangleq \mathit{false} && \text{Empty relation} \\ 1 &\triangleq \bigwedge_{x \in X} x' = x && \text{Identity relation} \\ F + G &\triangleq F \lor G && \text{Union} \\ F \cdot G &\triangleq \exists X''.F(X, X'') \land G(X'', X') && \text{Relational composition} \end{aligned}$$

Above and elsewhere, we use positional notation for substitution; e.g., $F(X, X'')$ denotes the formula obtained by replacing all the X' symbols with "double primed" symbols in X'' (and leaving the un-primed X symbols as they are). Intuitively, F^* should be interpreted as the reflective transitive closure of F. However, in general it is not possible to compute the reflexive transitive closure of a formula (nor even to *represent* it as a formula). Hence, we must be content with an over-approximate transitive closure operator. There are many different methods for over-approximating transitive closure, so we speak of the *family* of algebras of transition formulas, which have the same basic structure and differ only in the interpretation of the iteration operator. In the remainder of this section, we describe a selection of methods for implementing the iteration operator. *Disclaimer*: for each example, the presentation differs somewhat (sometimes substantially) from the cited source. The examples should be read as "how the cited analysis might be presented in the algebraic framework."

Example 3 (Transitive Predicate Abstraction [47]*).* Fix a set of variables X. Say that a transition formula $p(X, X')$ is

- *reflexive* if $\bigwedge_{x \in X} x = x' \models p(X, X')$
- *transitive* if $p(X, X') \land p(X', X'') \models p(X, X'')$

Let P be a finite set of *candidate* reflexive and transitive transition formulas. For example we might choose

$$P \triangleq \begin{array}{l} \{x \bowtie x' : x \in X, \bowtie \in \{\leq, \geq\}\} \\ \cup \{x \bowtie 0 \Rightarrow x' \bowtie 0 : x \in X, \bowtie \in \{\leq, \geq, <, >\}\} \end{array}$$

We can define an iteration operator that over-approximates the reflexive transitive closure of a formula F by the conjunction of the subset of P that is entailed by F:

$$F^* \triangleq \bigwedge \{p \in P : F \models p\} \ .$$

⌟

Example 4 (Interval analysis [51]*).* Let $F(X, X')$ be a transition formula. An *inductive interval invariant* for F assigns to each variable $x \in X$ a pair of integers $a_x, b_x \in \mathbb{Z}$ such that if s is a state such that $s(x) \in [a_x, b_x]$ for all $x \in X$ and $s \to_F s'$, then $s'(x) \in [a_x, b_x]$ for all $x \in X$. Monniaux showed that it is possible to determine optimal inductive interval invariants by posing the inductive-invariance condition symbolically and quantifying over the bounds [51].

Let $P = \{p_x : x \in X\}$ and $Q \triangleq \{q_x : x \in X\}$ be sets of fresh variables, which we use to the lower and upper bounds of intervals, respectively. The set of inductive interval invariants for a formula F can be represented by the formula

$$\mathit{Inv}(F, P, Q) \triangleq \forall X, X'. \left(F \land \bigwedge_{x \in X} p_x \leq x \leq q_x\right) \Rightarrow \left(\bigwedge_{x \in X} p_x \leq x' \leq q_x\right)$$

That is, the models of *Inv* (which assign integers to the lower and upper bound variables P and Q) are in one-to-one correspondence with the interval invariants of F. We may universally quantify over all inductive interval invariants to arrive at the following iteration operator:

$$F^* \triangleq \forall P, Q. \left(\mathit{Inv}(F, P, Q) \land \bigwedge_{x \in X} p_x \leq x \leq q_x\right) \Rightarrow \left(\bigwedge_{x \in X} p_x \leq x' \leq q_x\right)$$

In contrast to the typical iterative approach with classical widening and narrowing operators, this operator computes a formula that implies *all* (and therefore *most precise*) inductive interval invariants.[2] For example, for the

[2] Note that while the formula implies all interval invariants, it does not *itself* take the form of an interval invariant.

loop (**while** ($i \neq n$) **do** i := i + 1), this method yields the following over-approximation of the reflexive transitive closure of F:

$$F^* \equiv n' = n \land i \le i' \land (i \le n \Rightarrow i' \le n)$$

If we suppose that i is initially 0 and n is initially 100, then this formula implies the loop invariant that n is equal to 100, and i is in the interval $[0, 100]$. ⌟

Example 5 (Recurrence analysis [4,27]*).* Let $F(X, X')$ be a transition formula, and let $\mathbf{x}$ and $\mathbf{x}'$ denote vectors containing the variables X and X', respectively. A *linear recurrence inequation of* F is a formula of the form $\mathbf{a}^\intercal\mathbf{x}' \le \mathbf{a}^\intercal\mathbf{x} + b$ that is entailed by F. The idea behind recurrence analysis is to extract a set of linear recurrence inequations for a formula, $\{\mathbf{a}_i^\intercal\mathbf{x}' \le \mathbf{a}_i^\intercal\mathbf{x} + b_i\}_{i \in I}$, and to use the closed form of those recurrences to over-approximate the transitive closure of F:

$$F^* \triangleq \exists k. k \ge 0 \land \bigwedge_{i \in I} \mathbf{a}_i^\intercal\mathbf{x}' \le \mathbf{a}_i^\intercal\mathbf{x} + kb_i$$

For instance, consider the following loop:

```
while (x > 0) do
  if (y < 0) { x := x + y; y := y - 1 }
  else { x := x - 2; y := y - 3}
```

The loop exhibits the following recurrences

$$\begin{aligned}(2x' - y') &\le (2x - y) - 1\\ y' &\le y - 1\\ -y' &\le -y + 3\end{aligned} \quad \text{or in matrix form,} \begin{bmatrix}2 & -1\\0 & 1\\0 & -1\end{bmatrix}\begin{bmatrix}x'\\y'\end{bmatrix} \le \begin{bmatrix}2 & -1\\0 & 1\\0 & -1\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} + \begin{bmatrix}-1\\-1\\3\end{bmatrix}$$

which yields the following transition formula that summarizes the loop:

$$\exists k. k \ge 0 \land (2x' - y') \le (2x - y') - k \land y' \le y - k \land -y' \le -y + 3k\ .$$

The loop also exhibits other recurrences (such as $x' \le x - 1$); however, the three selected recurrences are complete in the sense that all implied recurrences are non-negative linear combinations of these three (e.g., $x' \le x - 1$ is obtained by adding 1/2-times the first and second recurrences).

Such a complete set of recurrences exists for any transition formula F, which can be computed as follows. First, observe that the set of linear recurrences of F,

$$\mathit{Rec}(F) \triangleq \{(\mathbf{a}, b) : F \models \mathbf{a}^\intercal\mathbf{x}' \le \mathbf{a}^\intercal\mathbf{x} + b\}$$

is closed under non-negative linear combinations (i.e., it is a *convex cone*). Our goal is to find a (finite) set of generators for $\mathit{Rec}(F)$—a finite set $\{(\mathbf{a}_i, b_i)\}_{i \in B}$ such that

$$\mathit{Rec}(F) = \left\{(0, \lambda_0) + \sum_{i \in B} \lambda_i(\mathbf{a}_i, b_i) : \lambda_0 \ge 0, \lambda_i \ge 0 \text{ for all } i \in B\right\}\ .$$

To compute generators for $\mathit{Rec}(F)$, we first introduce a fresh set of "difference" variables, $\{\delta_x\}_{x \in X}$ and form a formula

$$\Delta(F) \triangleq \exists X, X'. F \land \bigwedge_{x \in X} \delta_x = x' - x \ .$$

Observe that $(\mathbf{a}, b) \in \mathit{Rec}(F)$ if and only if $\Delta(F) \models \mathbf{a}^\intercal \delta \leq b$. Thus, a set of generators for $\mathit{Rec}(F)$ corresponds exactly to a half-space representation for the convex hull of $\Delta(F)$, which can be computed using the algorithm from [27].

The class of linear recurrence inequations considered in this example can be generalized in various ways to yield more powerful invariant generation procedures. In particular,

- [27] computes linear recurrences with polynomial closed forms
- [42] computes polynomial recurrences with polynomial and complex exponential closed forms.
- [41] computes polynomial recurrences with polynomial and rational exponential closed forms. $\lrcorner$

2.2 Weak Interpretations

Transition formulas are an appealing basis for algebraic program analysis, since all the operators (except the iteration operator) are *precise*—they simply encode the meaning of the program into logic. The significance of this is that transition formula algebras delay precision loss *as long as possible*, which helps to overcome loss of contextual information. However, there are algebraic analyses of interest that are defined on weak logical fragments that cannot precisely express union and/or relational composition.

Example 6 (Affine relation analysis [38]*).* An *affine relation* is a relation that corresponds to the set of models of a transition formula of the form $A\mathbf{x}' = B\mathbf{x}+c$. Define the algebra of affine transition relations to be the regular algebra where the universe is the set of affine transition relations, 0 is interpreted as the empty relation, 1 is interpreted as the identity relation, $+$ is interpreted as the affine hull of $R_1 \cup R_2$ (the smallest affine relation that contains both R_1 and R_2), $\cdot$ is interpreted as relational composition, and $*$ is interpreted as the operation that sends any affine relation R to the limit of the sequence $\{R_i\}_{i=0}^{\infty}$ defined by

$$R_0 = 0 \qquad\qquad R_{i+1} = R_i + (R_i \cdot R) \text{ for } i \geq 0$$

Since we have $R_0 \subseteq R_1 \subseteq \dots$ and if any R_{i+1} properly contains R_i the dimension of R_{i+1} is strictly greater than that of R_i, this sequence must stabilize in finite time, so the operation R^* is computable. $\lrcorner$

3 Semantic Foundations

This section presents a general view of algebraic program analysis, with the goal of elucidating its underlying principles so that they may be understood outside the setting of graphs and regular expressions. This sets the stage for Sect. 4 and Sect. 5, wherein we will develop program analysis schemes that follow the same general "recipe" that we lay out in this section, but deviate from the instance of this recipe that we saw in Sect. 2.

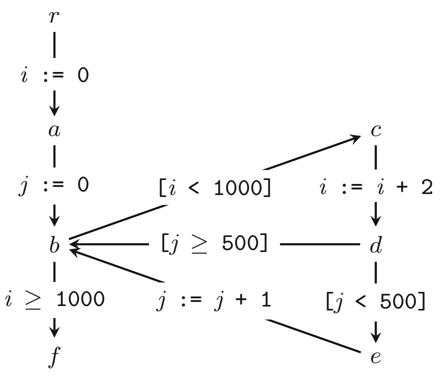
Following the theory of abstract interpretation [22], we begin with a *concrete semantics* that defines the meaning of a program. The concrete semantics is specified as the least (or greatest) solution to a system of recursive equations. The concrete semantics is not computable—the goal of a program analysis is to *approximate* it. The way that this is accomplished in an algebraic analysis is by symbolically computing a closed-form solution to the semantic equations (i.e., a non-recursive system of equations whose (unique) solution coincides with the concrete semantics), and then interpreting that closed-form solution in an algebraic structure that approximates the algebra of the concrete semantics.

3.1 Semantic Equations

Given a control flow graph G, we can syntactically derive a system of equations $E(G)$—see Fig. 2. For each vertex v, we introduce a variable X_v and an equation $(X_v = R_v)$ that relates that variable to the variables for v's predecessors. Notice that this system of equations can be viewed as a (left-)regular grammar, with each non-terminal symbol X_v recognizing the set of paths from the root r to the vertex v. This is an instance of the more general concept of a solution to a system of equations over an algebraic structure. A *solution* to the system of equations $E(G) = \{X_v = R_v\}_{v \in V}$ over a regular interpretation $\mathscr{I} = \langle \mathbf{A}, f \rangle$ is a function σ that maps each variable to an element of $\mathbf{A}$ such that each equation is satisfied: for each equation $(X_v = R_v)$ in $E(G)$, we have $\sigma(X) = \mathscr{I}_\sigma[\![R]\!]$, where $\mathscr{I}_\sigma$ is the interpretation obtained by extending the semantic function to variables by interpreting them according to σ.

The prototypical concrete semantics of interest in algebraic analysis is the relational semantics. The **relational semantics** of a program associates to every control flow vertex v a reachability relation R_v, which is the set of pairs $\langle s, s' \rangle$ such that if the program begins at r in state s, then it may reach v with state s'. The relational semantics may be obtained as the least solution to the system of semantic equations over the relational interpretation, which is defined as follows. The regular algebra of state relations, $\mathbf{R}$, has binary relations on states as its universe, 0 is interpreted as the empty relation $\emptyset$, 1 is interpreted as the identity relation $\{\langle s, s \rangle : s \in \mathsf{State}\}$, $\cdot$ is interpreted as relational composition, $+$ as union, and $*$ as reflexive, transitive closure. The **relational interpretation** $\mathscr{R}$ is the interpretation over the regular algebra of state relations where the semantic function maps each command to its associated transition relation; e.g., `i := i + 2` is associated with the set of all pairs $\langle s, s' \rangle$ such that $s'(i) = s(i) + 1$ and $s'(x) = s(x)$ for all $x \neq i$. The relational semantics of a CFG G is the least solution to $E(G)$ over the relational interpretation.

Having formulated the concrete semantics as the solution to a system of equations, we must now solve the system symbolically. The classical algorithm is a variation of Gaussian elimination, given in Algorithm 1. This algorithm is essentially Kleene's algorithm [44] for computing a regular expression for a



(a)

$$
\begin{aligned}
X_r &= 1\\
X_a &= X_r \cdot \langle r, a\rangle\\
X_b &= X_a \cdot \langle a, b\rangle\\
&\quad + X_d \cdot \langle d, b\rangle\\
&\quad + X_e \cdot \langle e, b\rangle\\
X_c &= X_b \cdot \langle b, c\rangle\\
X_d &= X_c \cdot \langle c, d\rangle\\
X_e &= X_d \cdot \langle d, e\rangle\\
X_f &= X_b \cdot \langle b, f\rangle
\end{aligned}
$$

(b)

$$
\begin{aligned}
X_r &= 1\\
X_a &= \langle r, a\rangle\\
X_b &= \langle r, a\rangle \langle a, b\rangle (\langle b, c\rangle \langle c, d\rangle (\langle d, b\rangle + \langle d, e\rangle\, e, b))^*\\
X_c &= \langle r, a\rangle \langle a, b\rangle (\langle b, c\rangle \langle c, d\rangle (\langle d, b\rangle + \langle d, e\rangle\, e, b))^* \langle b, c\rangle\\
X_d &= \langle r, a\rangle \langle a, b\rangle (\langle b, c\rangle \langle c, d\rangle (\langle d, b\rangle + \langle d, e\rangle\, e, b))^* \langle b, c\rangle \langle c, d\rangle\\
X_e &= \langle r, a\rangle \langle a, b\rangle (\langle b, c\rangle \langle c, d\rangle (\langle d, b\rangle + \langle d, e\rangle\, e, b))^* \langle b, c\rangle \langle c, d\rangle \langle d, e\rangle\\
X_f &= \langle r, a\rangle \langle a, b\rangle (\langle b, c\rangle \langle c, d\rangle (\langle d, b\rangle + \langle d, e\rangle\, e, b))^* \langle b, f\rangle
\end{aligned}
$$

(c)

Fig. 2. (a) A control flow graph; (b) the corresponding systems of equations; and (c) a closed-form solution.

finite state automaton, recast in the language of equations. The front-solving step eliminates variables one-by-one, at each step i producing a system of equation of equations that is equivalent to the original, but in which the variable X_i does not appear in the right-hand-side of any equations $X_j = R_j$ for $j \geq i$. The back-solving step eliminates all variable occurrences from right-hand-sides, at each step replacing X_i with its closed form R_i in each equation $X_j = R_j$ for $j < i$. An example illustrating the result of solving the system of equations in Fig. 2b symbolically appears in Fig. 2c. The significant difference to the familiar Gaussian elimination algorithm in linear algebra is the "loop-solving" step, which solves a single recursive equation $X_i = R_i$ symbolically by re-arranging R_i into the form $X_i A + B$ and taking BA^* to be the solution. The loop-solving step is justified under the relational interpretation, and more generally for any interpretation over a *Kleene algebra*.[3]

[3] The laws of Kleene algebra are not minimal in this regard.

```
Input   : Left-linear system of equations, $E = \{X_i = R_i\}_{i=1}^n$
Output : Closed-form solution to $E$
for $i = 1$ to $n$ do                                    /* Front-solving */
    Re-arrange $R_i$ in the form $X_i A + B$;
    $R_i \leftarrow BA^*$ ;                               /* “Loop-solving” */
    foreach $j > i$ do  $R_j \leftarrow R_j[X \mapsto R_i]$ ;
end
for $i = n$ to $2$ do                                    /* Back-solving */
    foreach $j < i$ do  $R_j \leftarrow R_j[X_i \mapsto R_i]$ ;
end
return $E$;
```

Algorithm 1: Gaussian elimination for left-linear systems of equations

Definition 1. *Let* $\mathbf{A} = \langle A, +, \cdot, *, 0, 1\rangle$ *be a regular algebra. We say that* $\mathbf{A}$ *is an **idempotent semiring** if it satisfies the following (for all* $a, b, c, \in A$*):*

$$\begin{array}{ccr} a + (b + c) = (a + b) + c & a(bc) = (ab)c & \textit{Associativity} \\ a(b + c) = ab + ac & (b + c)a = ba + ca & \textit{Distributivity} \\ a + 0 = a & 1a = a1 = a & \textit{Identity} \\ \multicolumn{2}{c}{a + b = b + a} & \textit{Commutativity of } + \\ \multicolumn{2}{c}{a + a = a} & \textit{Idempotence} \\ \multicolumn{2}{c}{a0 = 0a = 0} & \textit{Annihilation} \end{array}$$

*In any idempotent semiring, we may define a **natural order*** $\leq$*, where* $a \leq b$ *iff* $a + b = b$*. Note that* $+$ *is the least upper bound with respect to this order.*

We say that $\mathbf{A}$ *is a **Kleene algebra** if it is an idempotent semiring and the following hold (for all* $a, x \in A$*):*

$$\begin{array}{ccl} 1 + a(a^*) = a^* & 1 + (a^*)a = a^* & \textit{Unfolding} \\ ax \leq x \Rightarrow a^*x \leq x & xa \leq x \Rightarrow xa^* \leq x & \textit{Induction} \end{array}$$

Exercise 1. Show that in any Kleene algebra, the least solution to a (left-)linear recursive equation $X = a + Xb$ exists and is equal to ab^*

The sense in which Gaussian elimination computes a “closed-form solutions” to a system of left-linear equations E is that:

- (*closed form*) the right-hand sides do not refer to variables, and
- (*solution*) for any interpretation $\mathscr{I}$ over a Kleene algebra, for each equation $(X = R) \in E$, we have $\sigma(X) = \mathscr{I}[\![R]\!]$ where σ is the least solution to E over $\mathscr{I}$.

The connection between Gaussian elimination and graph algorithms like Floyd-Warshall inspired Tarjan's path-expression algorithm [58]. In the language of graphs, Tarjan's algorithm computes for each vertex v of a control flow graph G with root r a path expression $\mathit{PathExp}_G(r, v)$ that recognizes the set of paths from r to v; in the language of equations, it solves left-linear systems of equations symbolically. Tarjan's algorithm is preferred to Gaussian elimination in practice: is more efficient (nearly linear time for reducible flow graphs, compared to cubic time for Gaussian elimination) and produces simpler solutions. For expository purposes, we will continue to refer to Gaussian elimination for solving systems of equations, viewing Tarjan's method as an efficient variation.

3.2 Abstract Interpretation

Gaussian elimination can solve a system of left-linear equations over a Kleene algebra (e.g., relational semantics) symbolically. However, the solution cannot be interpreted in the concrete algebra, since operators are not effective (that is, they cannot be implemented by a machine). We approximate the concrete semantics by interpreting the closed-form solution in an effective *abstract* algebra (e.g., one of the transition-formula algebras from Sect. 2).

Following the theory of abstract interpretation [22], the correctness of this approach is justified by establishing a relationship between the "concrete" and "abstract" interpretations. In the algebraic framework, a natural way to express the relationship is via a *soundness relation* [24], which is a binary relation between two algebras that is preserved by the operations of the algebra. Membership of a (concrete, abstract) pair in the relation indicates that the concrete element is approximated by the abstract element.

Definition 2 (Soundness relation). *Given two Σ-interpretations $\mathscr{I}^\natural = \langle \mathbf{A}^\natural, f^\natural \rangle$ and $\mathscr{I}^\sharp = \langle \mathbf{A}^\sharp, f^\sharp \rangle$, $- \Vdash - \subseteq A^\natural \times A^\sharp$ is a* ***soundness relation*** *if $f^\natural(a) \Vdash f^\sharp(a)$ for all $a \in \Sigma$ and $\Vdash$ is a sub-algebra of the product algebra $\mathbf{A}^\natural \times \mathbf{A}^\sharp$; i.e., $0^\natural \Vdash 0^\sharp$, $1^\natural \Vdash 1^\sharp$, and for all $x_1 \Vdash y_1$ and $x_2 \Vdash y_2$ we have*

- $x_1 +^\natural x_2 \Vdash y_1 +^\sharp y_2$
- $x_1 \cdot^\natural x_2 \Vdash y_1 \cdot^\sharp y_2$
- $x_1^{*^\natural} \Vdash y_1^{*^\sharp}$

The definition of soundness relation generalizes to interpretations over other classes of algebraic structures in the natural way: it is a binary relation over two algebras of the same signature that is preserved by every operation in the signature.

Example 7 (Transition formula overapproximation). Let **R** denote the algebra of state relations and **TF** denote an algebra of transition formulas. The *over-approximation* relation is defined by

$$R \Vdash_O F \iff \forall \langle s, s' \rangle \in R, s \to_F s'.$$

Preservation of constants and the sequencing and choice operations is easily verified; to show that $\Vdash_O$ is a soundness relation, we need only to show that $R \Vdash_O F$ implies $R^{*^{\mathbf{R}}} \Vdash_O F^{*^{\mathbf{TF}}}$; i.e., $(-)^{*^{\mathbf{TF}}}$ over-approximates reflexive transitive closure. Of course, this proof depends on the particular implementation of the iteration operator.

The over-approximate soundness relation allows us to *verify* safety properties: if $R \Vdash_O F$ and F entails some property P, then R satisfies P. ⌟

Example 8 (Transition formula underapproximation). The *under-approximation* relation is defined by

$$R \Vdash_U F \iff \forall s, s'.s \to_F s' \Rightarrow \langle s, s' \rangle \in R,$$

Preservation of constants and the sequencing and choice operations is again easily verified; to show that $\Vdash_U$ is a soundness relation, we need only to show that $R \Vdash_O F$ implies $R^{*^{\mathbf{R}}} \Vdash_O F^{*^{\mathbf{TF}}}$; i.e., $(-)^{*^{\mathbf{TF}}}$ under-approximates reflexive transitive closure. The iteration operators in Sect. 2 are all over-approximate. An example of an under-approximate iteration operator is

$$F^* \triangleq \bigvee_{i=0}^{n} \underbrace{F \circ \cdots \circ F}_{i\ times}$$

(for some fixed choice of n) which corresponds to bounded model checking [9], with an unrolling bound of n.

The under-approximate soundness relation allows us to *refute* safety properties: if $R \Vdash_U F$ and F does not entail some property P, then R does not satisfy P. $\lrcorner$

The problem of "approximating the behavior of a program" can be formalized as follows:

> Given a system of semantic equations over a set of variables $\mathcal{X}$ describing the concrete semantics of a program (i.e., its least solution $\sigma^\natural$ over some interpretation $\mathscr{I}^\natural$), find some $\sigma^\sharp : \mathcal{X} \to \mathbf{A}^\sharp$ such that for each variable $X \in \mathcal{X}$, we have $\sigma^\natural(X) \Vdash \sigma^\sharp(X)$.

The algebraic approach to this problem is to compute for each variable X a closed form R_X (such that $\sigma^\natural(X) = \mathscr{I}^\natural(R_X)$), and define $\sigma^\sharp(X) \triangleq \mathscr{I}^\sharp(R_X)$. The correctness of this approach is justified by the following soundness lemma, which follows by induction on regular expressions.

Lemma 1 (Soundness). *Let Σ be an alphabet, let $\mathscr{I}^\natural = \langle \mathbf{A}^\natural, f^\natural \rangle$ and $\mathscr{I}^\sharp = \langle \mathbf{A}^\sharp, f^\sharp \rangle$ be Σ-interpretations, and let $\Vdash \subseteq A^\natural \times A^\sharp$ be a soundness relation. Then for any regular expression $R \in \mathsf{RegExp}(\Sigma)$, we have $\mathscr{I}^\natural[\![R]\!] \Vdash \mathscr{I}^\sharp[\![R]\!]$*

3.3 Discussion

A subtlety of algebraic program analysis is that most algebras of interest in program analysis are *not* Kleene algebras (for instance, none of the algebras in Sect. 2 are), and so in general, Gaussian elimination does not find solutions to systems of equations over "abstract" interpretations corresponding to program analyses. This technical difficulty is sidestepped by appealing to the concrete semantics (which typically *is* defined over a Kleene algebra, such as the algebra of state relations) to justify the use of path-expression algorithms, and a sound approximating algebra to interpret the resulting expressions. The fact that the abstract interpretation of the closed-form solution to the concrete system of equations does not yield a solution to the abstract system of equations is immaterial: our goal is to *over-approximate* the concrete rather than *solve* the abstract.

Formalizing a program analysis as an algebraic structure allows one to understand the behavior of program analyses in terms of algebraic laws, and use the language of algebra to reason about program analyses. For example, any transition formula algebra (in the family described in Sect. 2.1) is an idempotent semiring, and so any two $*$-free regular expressions that denote the same language have the same (up to logical equivalence) interpretation as a transition formula. While none of the iteration operators in Sect. 2.1 satisfy the *Unfolding* and *Induction* laws of Kleene algebra, they do satisfy weaker *pre-Kleene algebra* iteration laws:

$1 \leq a^*$	Reflexivity
$a \leq a^*$	Extensivity
$a^* a^* = a^*$	Transitivity
$a \leq b \Rightarrow a^* \leq b^*$	Monotonicity
For any n, $(a^n)^* \leq a^*$	Unrolling

A concrete use-case for these laws appears in [25], which develops regular expression transformation techniques that preserve concrete semantics but are guaranteed to produce (non-strictly) more precise abstract semantics.

Such laws can also be useful for users of program analysis tools. For example, since all operations are monotone (as a consequence of the monotonicity and idempotent-semiring laws), a user can rely on the principle that "more information in yields more information out." If a user alters a program P by adding additional *assume* commands to get a program P' (e.g., expressing invariants that are found by some other automated invariant generation technique, user-provided hints, etc.), monotonicity means that they may rely on the fact that the analysis will produce summaries for P' that are at least as precise as those for P.

A Recipe for Algebraic Program Analysis. We conclude this section by presenting a general view of algebraic program analysis, abstracted from the language of graphs and regular expressions:

1. *(Modeling)* Express the concrete semantics as the least (or greatest) solution to a system of recursive equations (e.g., relational semantics as the least solution to the left-linear system of equations corresponding to a control flow graph).
2. *(Closed forms)* Design a suitable language of "closed-form solutions" and an algorithm for computing them (e.g., regular expressions and path-expression algorithms).
3. *(Interpretation)* Design an abstract interpretation of the language of closed forms and a soundness relation connecting the concrete and abstract interpretations (e.g., transition-formula algebras (Sect. 2.1) and the over-approximate soundness relation (Ex. 7)).

Section 4 and Sect. 5 give two more instances of this generic recipe, generalizing beyond left-linear equations and regular-expressions as closed forms. Section 4 considers linear equations (and an appropriate language of closed forms); Sect. 5 considers another form of equation with ω-regular expressions as closed forms.

4 Interprocedural Analysis

Algebraic program analyses are oriented around computing summaries for program fragments, and are naturally suited to analyzing programs with procedures. Following Cousot & Cousot [23] and Sharir & Pnueli [56], the idea is to structure the analysis in two phases:

Phase I: compute for each procedure X a summary that approximates the behavior of X (including the actions of all procedures called transitively from X).

Phase II: analyze whole-program paths from the start of the main procedure, using the summaries to interpret procedure calls.

An example of a program with procedures is given in Fig. 3(a). The CFGs for its procedures are shown in Fig. 3(b) along with a set of equations corresponding to the CFGs (Fig. 3(c)). For Phase I, it is also useful to consider the following equations in which we have eliminated all variables except for those of the form $X_{s,x}$, which represent the procedure summaries.

$$\begin{aligned} X_{s_1,x_1} &= (\langle s_1, a\rangle \cdot X_{s_2,x_2} + \langle s_1, b\rangle) \cdot X_{s_2,x_2} \\ X_{s_2,x_2} &= X_{s_3,x_3} \cdot X_{s_3,x_3} \\ X_{s_3,x_3} &= \langle s_3, x_3\rangle \end{aligned} \tag{1}$$

This system of equations can be obtained either by a process of successively eliminating variables from Fig. 3(c), or they can be read off directly from each control-flow graph: sequential composition corresponds to $\cdot$, and branching corresponds to $+$.

We can also construct a graph of the dependencies among the variables in the equation system. In this case, we would have

$$X_{s_3,x_3} \longrightarrow X_{s_2,x_2} \longrightarrow X_{s_1,x_1} \tag{2}$$

(which is also isomorphic to the program's call graph). Note that the equations in Eq. (1) are *not* left-linear. However, by eliminating variables in a topological order of Eq. (2), these systems can still be solved using Gaussian elimination (Algorithm 1).

$$\begin{aligned} X_{s_3,x_3} &= \langle s_3, x_3\rangle \\ X_{s_2,x_2} &= \langle s_3, x_3\rangle \cdot \langle s_3, x_3\rangle \\ X_{s_1,x_1} &= (\langle s_1, a\rangle \cdot \langle s_3, x_3\rangle \cdot \langle s_3, x_3\rangle + \langle s_1, b\rangle) \cdot \langle s_3, x_3\rangle \cdot \langle s_3, x_3\rangle \end{aligned} \tag{3}$$

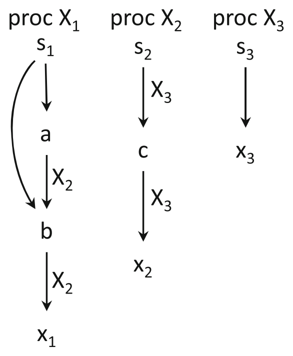
Unfortunately, this strategy breaks down for programs with recursive procedures: the essential difficulty is in computing the summaries of procedures that are directly recursive or part of a set of mutually recursive procedures. We will return to this issue shortly, after a brief discussion of Phase II, which can be addressed via algebraic program analysis, regardless of whether the original equation system contains recursion.

```
X1() {
    if (*) X2();
    X2();
}

X2() {
    X3();
    X3();
}

X3() {
    return;
}
```

(a)



(b)

$$
\begin{aligned}
X_{s_1,s_1} &= 1\\
X_{s_1,a} &= X_{s_1,s_1} \cdot \langle s_1, a\rangle\\
X_{s_1,b} &= \begin{array}{l} X_{s_1,a} \cdot X_{s_2,x_2} \\ + \; X_{s_1,s_1} \cdot \langle s_1, b\rangle \end{array}\\
X_{s_1,x_1} &= X_{s_1,b} \cdot X_{s_2,x_2}\\
\\
X_{s_2,s_2} &= 1\\
X_{s_2,c} &= X_{s_2,s_2} \cdot X_{s_3,x_3}\\
X_{s_2,x_2} &= X_{s_2,c} \cdot X_{s_3,x_3}\\
\\
X_{s_3,s_3} &= 1\\
X_{s_3,x_3} &= X_{s_3,s_3} \cdot \langle s_3, x_3\rangle
\end{aligned}
$$

(c)

Fig. 3. (a) A three-procedure program scheme. (b) Control-flow graphs for program (a). The edges labeled "X_2" and "X_3" represent calls to the respective procedures. (c) A system of equations corresponding to (b).

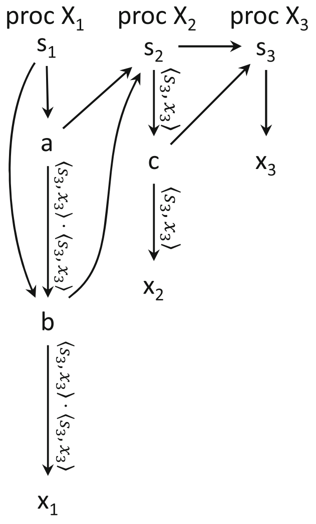


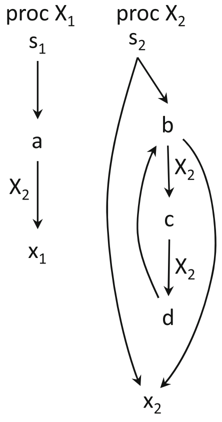
Fig. 4. Graph corresponding to the equation system used for Phase II for the program from Fig. 3.

With closed-form solutions for the procedure summaries in hand, Phase II can be addressed with Gaussian elimination. (Note that for a program with recursive procedures, the transformed Phase II system is still recursive. However, it is *left*-recursive, and so can be handled with regular expressions, and analyzed using the transition-formula interpretations of Sect. 2—the "loops" in Phase II correspond to sequences of recursive calls). Figure 4 shows the equation system

```
$X_1()$ {
    $C_{\langle s_1,a\rangle}$;
    $X_2()$
}

$X_2()$ {
    if (*) $C_{\langle s_2,x_2\rangle}$;
    $C_{\langle s_2,b\rangle}$;
    while (*) {
        $X_2()$;
        $X_2()$;
    }
    $C_{\langle b,x_2\rangle}$;
}
```

(a)



(b)

$$
\begin{aligned}
X_{s_1,s_1} &= 1\\
X_{s_1,a} &= X_{s_1,s_1}\cdot\langle s_1,a\rangle\\
X_{s_1,x_1} &= X_{s_1,a}\cdot X_{s_1,x_2}
\end{aligned}
$$

$$
\begin{aligned}
X_{s_2,s_2} &= 1\\
X_{s_2,b} &= \begin{array}{l} \quad X_{s_2,s_2}\cdot\langle s_2,b\rangle\\ +\ X_{s_2,d}\cdot\langle d,b\rangle\end{array}\\
X_{s_2,c} &= X_{s_2,b}\cdot X_{s_2,x_2}\\
X_{s_2,d} &= X_{s_2,c}\cdot X_{s_2,x_2}\\
X_{s_2,x_2} &= \begin{array}{l} \quad X_{s_2,s_2}\cdot\langle s_2,x_2\rangle\\ +\ X_{s_2,b}\cdot\langle b,x_2\rangle\end{array}
\end{aligned}
$$

(c)

Fig. 5. (a) A two-procedure program scheme, where X_1 represents the main procedure, X_2 represents a recursive subroutine, and $C_{\langle s_1,a\rangle}$, $C_{\langle s_2,x_2\rangle}$, $C_{\langle s_2,b\rangle}$, and $C_{\langle b,x_2\rangle}$ represent four program statements. (b) Control-flow graphs for program (a). The three edges labeled "X_2" represent calls to procedure X_2. (c) A system of equations corresponding to (b).

used for Phase II for the program from Fig. 3 in graphical form. The graph is similar to Fig. 3(b) with (i) additional edges from each call-site to the start node of the called procedure, and (ii) the edges previously labeled with "X_2" and "X_3" are now labeled with the values from Eq. (3) for the corresponding procedure summaries: $\langle s_3,x_3\rangle\cdot\langle s_3,x_3\rangle$ and $\langle s_3,x_3\rangle$, respectively.

The remainder of this section focuses on Phase I: computing procedure summaries. Consider the two-procedure program shown in Fig. 5(a). CFGs for its procedures are shown in Fig. 5(b) along with a set of recursive equations corresponding to the interprocedural CFG. Unfortunately, equations like those in Fig. 5(c) do not fit naturally with the recipe given in Sect. 3.3. The essential difficulty is with item 3.3: "Design a suitable language of 'closed-form solutions' and an algorithm for computing them." In particular, we cannot use regular expressions and path-expression algorithms because the equations in Fig. 5(c) are not left-linear (and they cannot be put in left-linear form).

Two ideas are involved in using algebraic program analysis to summarize recursive procedures:

1. The generalization by Esparza et al. [26] of Newton's method—the classical numerical-analysis algorithm for finding roots of real-valued functions—to a method for solving a system of equations over a semiring $\mathcal{S}$, called *Newtonian Program Analysis* (NPA). As in its real-valued counterpart, each iteration of NPA solves a simpler "linearized" problem. (See Sect. 4.1.)
2. The technique of Reps et al. [53] for applying the algebraic-program-analysis recipe to the linearized problems that arise in NPA. (See Sect. 4.2.)

4.1 Motivation: Newtonian Program Analysis

To motivate why we are interested in the special case of linear equations (Sect. 4.2), this section provides a brief overview of how linear equations arise in NPA. Let $E = \{X_i = R_i\}_{i=1}^n$ be a system of equations, and fix an interpretation $\mathscr{I}$ over some algebra $\mathbf{A}$. Define a function $\mathbf{f} : A^n \to A^n$ by $\mathbf{f}(\sigma) = (\mathscr{I}_\sigma[\![R_1]\!], \ldots, \mathscr{I}_\sigma[\![R_n]\!])$ (i.e., the n-tuple of interpreted right-hand-sides, where variables are interpreted according to σ). NPA is an iterative method for program analysis that solves the following sequence of problems for ν:

$$\boxed{\nu^{(0)} = \mathbf{f}(\mathbf{0}) \qquad \nu^{(i+1)} = \mathbf{Y}^{(i)}} \tag{4}$$

where $\mathbf{Y}^{(i)}$ is the value of $\mathbf{Y}$ in the least solution of

$$\boxed{\mathbf{Y} = \mathbf{f}(\nu^{(i)}) + \text{LinearCorrectionTerm}(E, \nu^{(i)}, \mathbf{Y})} \tag{5}$$

Thus, NPA is similar to Kleene iteration, except that on each iteration, $\mathbf{f}(\nu^{(i)})$ is "corrected" by an amount controlled by LinearCorrectionTerm$(E, \nu^{(i)}, \mathbf{Y})$—a function of $\mathbf{f}$, the current approximation $\nu^{(i)}$, and (vector) variable $\mathbf{Y}$—which nudges the next approximation $\nu^{(i+1)}$ in the right direction at each step.

The linear correction term is the result of replacing each right-hand side $R_i = \sum_j R_j$ with a sum $\sum_{i=0}^n R_{i,j,k}$, where each $R_{i,j,k}$ is obtained from $R_{i,j}$ by replacing all variables, except possibly one, with its interpretation in ν. (The formal definition can be found elsewhere [26, §3.2].) For example, consider the system of equations below, a simplified variant of Fig. 5(c) that is obtained by eliminating all variables except $X_{s_1,x_1}, X_{s_2,b}, X_{s_2,x_2}$:

$$\begin{aligned} X_{s_1,x_1} &= \langle s_1, a\rangle\, X_{s_1,x_2} \\ X_{s_2,b} &= \langle s_2, b\rangle + X_{s_2,b} \cdot X_{s_2,x_2} \cdot X_{s_2,x_2} \cdot \langle d, b\rangle \\ X_{s_2,x_2} &= \langle s_2, x_2\rangle + X_{s_2,b}\, \langle b, x_2\rangle \end{aligned} \tag{6}$$

The transformation results in the following system (for brevity, we denote $Y_{s_1,x_1}, Y_{s_2,b}, Y_{s_2,x_2}$ by Y_1, Y_2, Y_3):

$$\begin{aligned} Y_1 &= \langle s_1, a\rangle \cdot Y_3 \\ Y_2 &= \langle s_2, b\rangle + Y_2 \cdot \nu_3 \cdot \nu_3 \cdot \langle d, b\rangle + \underline{\nu_2 \cdot Y_3 \cdot \nu_3 \cdot \langle d, b\rangle} + \underline{\nu_2 \cdot \nu_3 \cdot Y_3 \cdot \langle d, b\rangle} \\ Y_3 &= \langle s_2, x_2\rangle + Y_2 \cdot \langle b, x_2\rangle \end{aligned} \tag{7}$$

Note that the two underlined summands are both truly *linear*: they are linear, but not left-linear nor right-linear.

The process of solving Eqs. (4) and (5) for $\nu^{(i+1)}$, given $\nu^{(i)}$, is called one *Newton round*. On the initial Newton round, we set $\langle \nu_1^{(0)}, \nu_2^{(0)}, \nu_3^{(0)}\rangle \leftarrow \langle 0, \mathscr{I}[\![\langle s_2, x_2\rangle]\!], \mathscr{I}[\![\langle s_3, x_3\rangle]\!]\rangle$. On round $i + 1$, we solve Eq. (7) for $\langle Y_1, Y_2, Y_3\rangle$ with $\langle \nu_1, \nu_2, \nu_3\rangle$ set to the value $\langle \nu_1^{(i)}, \nu_2^{(i)}, \nu_3^{(i)}\rangle$ obtained on round i, and then set $\langle \nu_1^{(i+1)}, \nu_2^{(i+1)}, \nu_3^{(i+1)}\rangle \leftarrow \langle Y_1, Y_2, Y_3\rangle$.

Operationally, the linearization transformation imposes a particular protocol for sampling the program's space of behaviors. For instance, in Fig. 5(b), the procedure X_2 has two call-sites along the loop through b. In Eq. (7), each right-hand-side summand in the equation for Y_2 has at most one variable: the transformation inserted ν_2 or ν_3 at various call-sites (considering $X_{s_2,b}$ as a pseudo-call-site corresponding to tail recursion), and left at most one variable Y_i in each summand. In essence, during a given Newton round, the analyzer samples the behavior of $\mathbf{f}$ by taking the $+$ of various paths through the transformation of $\mathbf{f}$. Along each path through a (transformed) right-hand side, the summary for each pseudo-call-site X_i encountered is held fixed at ν_i, except for possibly one pseudo-call-site on the path, which is explored by visiting (the linearized version of) the called procedure. The summaries ν_1, ν_2, ν_3 are updated according to the result of this exploration, and the algorithm performs the next Newton round.

The analogy between NPA and Newton's method in numerical analysis is that in both cases one creates a linear approximation of $\mathbf{f}(\mathbf{X})$ around the "point" $(\nu^{(i)}, \mathbf{f}(\nu^{(i)}))$; the solution of the linear system is the next approximation of $\mathbf{X}$.

4.2 Algebraic Program Analysis for Linear Equations

In this section, we instantiate the recipe for algebraic program analysis from Sect. 3.3 to solve a system of linear equations, such as the linearized problems that arise as Eq. (5) [53]. This goal may seem out of reach because item 3.3 of the recipe requires us to "design a suitable language of 'closed-form solutions' and an algorithm for computing them."

What is a suitable language of closed-form solutions of linear equations? Clearly the regular expressions and path-expression algorithms used in Sect. 2 and Sect. 3 will not do, because the least solution under the language interpretation to the (truly) linear equation $X = aXb + 1$ is $\left\{a^i b^i : i \geq 0\right\}$, which is the canonical example of a linear-context-free language that is not regular. However, over fifty years ago, formal-language theorists established that linear-context-free languages have certain similarities to regular languages [17,34,61], and we can make use of this property to design a language of closed forms for linear equations. Intuitively, $\left\{a^i b^i : i \geq 0\right\}$ can be obtained by (i) introducing *paired alphabet symbols*, such as (a, b), (ii) defining *concatenation of paired symbols* as $(a, b) \cdot (c, d) \stackrel{\text{def}}{=} (ca, bd)$, (iii) defining Kleene-star in the natural way over paired-symbol concatenation, so $(a, b)^*$ is the language of paired words $\left\{(a^i, b^i) : i \geq 0\right\}$, and (iv) applying an operation that concatenates the left word and right word of each paired word: $\left\{(a^i, b^i) : i \geq 0\right\} \mapsto \left\{a^i b^i : i \geq 0\right\}$.

For the purpose of algebraic program analysis, this idea can be formalized by introducing **tensored regular expressions** over an alphabet Σ, whose syntax is defined as follows:[4]

$$\begin{aligned}
& a \in \Sigma \\
R \in \mathsf{RegExp}(\Sigma) \ & ::= a \mid 0 \mid 1 \mid R_1 + R_2 \mid R_1 \cdot R_2 \mid R^* \mid S^{\lightning} \\
S \in \mathsf{RegExp}_{\mathcal{T}}(\Sigma) \ & ::= R_1 \otimes R_2 \mid \underline{0} \mid \underline{1} \mid S_1 \oplus S_2 \mid S_1 \odot S_2 \mid S^{\circledast}
\end{aligned}$$

We can now follow the pattern of Sect. 2, and define algebras suitable for interpreting tensored regular expressions.

Definition 3. *A* ***tensor-product algebra*** $\mathcal{T} = \langle \mathbf{A}, \mathbf{T}, \otimes, \lightning \rangle$ *consists of two regular algebras* $\mathbf{A}$ *and* $\mathbf{T}$ *along with an operation* $\otimes : A \times A \to T$*, called* tensor product*, and an operation* $\lightning : T \to A$*, called* detensor*.*

Example 9 (Standard interpretation). The standard interpretation from Example 1 can be extended to tensored regular expressions by defining a universe of languages over word pairs ("tensored words") $T = 2^{\Sigma^* \times \Sigma^*}$, whose operators are given by:

$$\begin{aligned}
X \otimes Y &\triangleq \{\langle x, y\rangle : x \in X, y \in Y\} \\
Z^{\lightning} &\triangleq \{\underline{z}\overline{z} : \langle \underline{z}, \overline{z}\rangle \in Z\} \\
Z_1 \odot Z_2 &\triangleq \{\langle \underline{z}_2\underline{z}_1, \overline{z}_1\overline{z}_2\rangle : \langle \underline{z}_1, \overline{z}_1\rangle \in Z_1, \langle \underline{z}_2, \overline{z}_2\rangle \in Z_2\} \\
Z_1 \oplus Z_2 &\triangleq Z_1 \cup Z_2 \\
Z^{\circledast} &\triangleq \bigcup_{i \in \mathbb{N}} Z^i
\end{aligned}$$

Note that this interpretation allows tensored regular expressions to be used to capture linear context-free languages. For instance, the equation $X = aXb + 1$, whose least solution is $\left\{a^i b^i : i \geq 0\right\}$ can be written in closed form as $X = ((a \otimes b)^{\circledast})^{\lightning}$, and the equation $X = aXa + bXb + 1$, whose least solution is the language of even-length palindromes over $\{a, b\}$, can be written as $X = (((a \otimes a) \oplus (b \otimes b))^{\circledast})^{\lightning}$. ⌟

Example 10 (Relational interpretation). The relational interpretation can be extended to tensored regular expressions by defining an algebra of binary state-pair relations, as follows.[5] The universe is the set of relations on $\mathsf{State} \times \mathsf{State}$ (i.e., an element of the universe is a subset of $\mathsf{State} \times \mathsf{State} \times \mathsf{State} \times \mathsf{State}$). Comparing with the standard interpretation, (in which an element $\langle p_1, p_2\rangle$ consists of a "backwards path" p and a "forwards continuation") we may think of

[4] A warning about notation: in our previous papers, we used $\oplus$ and $\otimes$ for the two semiring operations, $\odot$ for tensor product, and $\oplus_{\mathcal{T}}$ and $\otimes_{\mathcal{T}}$ for the two tensored-semiring operations. In this paper, we use $+$ and $\cdot$ for the semiring operations, with circles around them for the tensored-semiring versions: $\oplus$ and $\odot$. We use $\otimes$ for tensor product, which is consistent with usual mathematical notation.

[5] That is, an element of the algebra is a pair of pairs of states.

an element $\left\langle \begin{pmatrix} s_1' \\ s_2 \end{pmatrix}, \begin{pmatrix} s_1 \\ s_2' \end{pmatrix} \right\rangle$ of a state-pair relation as consisting of two pre/post state pairs: a "backwards" pair $s_1' \;{}^*\!\!\leftarrow s_1$ and a "forwards" pair $s_2 \rightarrow^* s_2'$. In the algebra of state-pair relations, 0 is interpreted as the empty relation, 1 as the identity relation, and + as union. The remaining operators are given by:

$$\begin{aligned}
R_1 \otimes R_2 &= \left\{ \left\langle \begin{pmatrix} s_1' \\ s_2 \end{pmatrix}, \begin{pmatrix} s_1 \\ s_2' \end{pmatrix} \right\rangle : \langle s_1, s_1' \rangle \in R_1, \langle s_2, s_2' \rangle \in R_2 \right\} \\
T^{\lightning} &= \left\{ \langle s, s' \rangle : \exists s'', s''. \left\langle \begin{pmatrix} s'' \\ s''' \end{pmatrix}, \begin{pmatrix} s \\ s' \end{pmatrix} \right\rangle \in T \wedge s'' = s''' \right\} \qquad (8) \\
T_1 \odot T_2 &= \left\{ \left\langle \begin{pmatrix} s_1 \\ s_2 \end{pmatrix}, \begin{pmatrix} s_1' \\ s_2' \end{pmatrix} \right\rangle : \left\langle \begin{pmatrix} s_1 \\ s_2 \end{pmatrix}, \begin{pmatrix} s_1'' \\ s_2'' \end{pmatrix} \right\rangle \in T_1 \wedge \left\langle \begin{pmatrix} s_1'' \\ s_2'' \end{pmatrix}, \begin{pmatrix} s_1' \\ s_2' \end{pmatrix} \right\rangle \in T_2 \right\} \\
T^{\circledast} &= \bigcup_{i=0}^{\infty} \underbrace{T \odot \ldots \odot T}_{i \text{ times}}
\end{aligned}$$

Note that the tensored sequencing operation is just a form of relational composition (over tuples of stacked elements); similarly, tensored iteration is a form of reflexive transitive closure. ⌟

Example 11 (Transition-formula interpretation). Transition formulas can be used to interpret tensored regular expression in a way analogous to the relational interpretation (as one should expect, because there must be a soundness relation between them!). A tensored transition formula T is a formula over four vocabularies, representing the value of the variables before and after a pair of computations. The tensor and detensor operations are essentially the same as those from the relational interpretation, translated into logic:

$$\begin{aligned}
(F_1 \otimes F_2) \left(\begin{pmatrix} X_1' \\ X_2 \end{pmatrix}, \begin{pmatrix} X_1 \\ X_2' \end{pmatrix} \right) &\triangleq F_1(X_1, X_1') \wedge F_2(X_2, X_2') \qquad (9) \\
T^{\lightning}(X, X') &\triangleq \exists \begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix} . T \left(\begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix}, \begin{pmatrix} X \\ X' \end{pmatrix} \right) \wedge Y_1 = Y_2
\end{aligned}$$

In the Eq. (9), the vocabularies X_1, X_1', X_2, and X_2' track the original role of the respective vocabulary in F_1 or F_2. The "stacked" notation is intended to be suggestive of an interpretation of a tensored transition formula over a doubled vocabulary, where the variables are $X_1' \cup X_2$ and their "primed copies" are $X_1 \cup X_2'$. To make the connection with Sect. 2.1 more apparent, we shall define $W_1 = X_1'$, $W_2 = X_2$, $W_1' = X_1$, $W_2' = X_2'$. With this notation, the product operation can be defined as:

$$(T_1 \odot T_2) \left(\begin{pmatrix} W_1 \\ W_2 \end{pmatrix}, \begin{pmatrix} W_1 \\ W_2 \end{pmatrix}' \right) \triangleq \exists \begin{pmatrix} W_1 \\ W_2 \end{pmatrix}'' . T_1 \left(\begin{pmatrix} W_1 \\ W_2 \end{pmatrix}, \begin{pmatrix} W_1 \\ W_2 \end{pmatrix}'' \right) \wedge T_2 \left(\begin{pmatrix} W_1 \\ W_2 \end{pmatrix}'', \begin{pmatrix} W_1 \\ W_2 \end{pmatrix}' \right)$$

As with the relational interpretation, the product operation is just a form of relational composition (over tuples of stacked elements).

Remarkably, the algebra of tensored transition formulas is the same as the algebra of untensored transition formulas, just over an extended set of variables. In particular, the iteration operators from Sect. 3 can be used to implement $\circledast$. For instance, consider the recursive procedure

```
foo(): if (*) then a := a + 1; foo(); b := b + 1
```

The path to the recursive call of *foo* and the path from the recursive call to exit can be modeled by the transition formulas F and G, respectively:

$$F \triangleq a' = a + 1 \wedge b' = b$$
$$G \triangleq b' = b + 1 \wedge a' = a$$

A procedure summary for *foo* can be calculated by evaluating $((F \otimes G)^{\circledast})^{\lightning}$, using recurrence analysis (Example 5) to implement the $\circledast$ operator:

$$\begin{aligned} F \otimes G &\triangleq a_1 = a_1' - 1 \wedge b_1 = b_1' \wedge b_2' = b_2 + 1 \wedge a_2' = a_2 \\ (F \otimes G)^{\circledast} &\triangleq \exists k. k \geq 0 \wedge a_1 = a_1' - k \wedge b_1 = b_1' \wedge b_2' = b_2 + k \wedge a_2' = a_2 \\ ((F \otimes G)^{\circledast})^{\lightning} &\triangleq \exists k. k \geq 0 \wedge a' = a + k \wedge b' = b + k \end{aligned}$$

┘

We now show how to compute closed forms for linear equations. First, we perform a *regularizing transformation*, which takes a system of linear equations E_{Lin} and converts it into a system of left-linear equations E_{LeftLin}. The transformation takes each right-hand-side term of the form $a \cdot Y \cdot b$ and converts it to $Z \odot (a \otimes b)$, where Y and Z are variables whose values are elements of the regular algebras $\mathbf{A}$ and $\mathbf{T}$ of a tensor-product algebra $\langle \mathbf{A}, \mathbf{T}, \otimes, \lightning \rangle$.

Definition 4. *Given a linear equation system E_{Lin} over the regular algebra $\mathbf{A}$ of a tensor-product algebra $\mathcal{T} = \langle \mathbf{A}, \mathbf{T}, \otimes, \lightning \rangle$, the* ***regularizing transformation*** *τ_{Reg} creates a left-linear equation system $E_{LeftLin} = \tau_{Reg}(E_{Lin})$ over $\mathbf{T}$ by transforming each equation of E_{Lin} as follows:*

$$\frac{Y_j = c_j + \sum_{i,k} (a_{i,j,k} \cdot Y_i \cdot b_{i,j,k})}{Z_j = (1 \otimes c_j) \oplus \bigoplus_{i,k} (Z_i \odot (a_{i,j,k} \otimes b_{i,j,k}))} \tau_{\text{Reg}}$$

where Z_i and Z_j are variables that take on values from $\mathbf{T}$.

For instance, if the regularizing transformation is applied to the linear system of equations in Fig. 6a, the result is the system of equations Fig. 6b. Because Fig. 6b is left-linear, we can now use the approach from Sect. 2 and Sect. 3—that is, create a closed-form solution for each variable Z_i by finding a path expression for the variable in the graph Fig. 6c. Finally, one gives a closed-form solution for each variable Y_i for the linear equation system in Fig. 6a by applying $(-)^{\lightning}$ to each path expression—see Fig. 6d. This algorithm for computing closed-form solutions to linear equations is justified in the tensored-relational interpretation, and more generally, in any interpretation whose algebra forms what we dub a Kronecker algebra, defined as follows:

$$\begin{aligned} Y_1 &= aY_2b + cY_2d \\ Y_2 &= e + fY_1g \end{aligned}$$

(a)

$$\begin{aligned} Z_1 &= (Z_2 \odot (a \otimes b)) \oplus (Z_2 \odot (c \otimes d)) \\ Z_2 &= (1 \otimes e) \oplus (Z_1 \odot (f \otimes g)) \end{aligned}$$

(b)

$$\begin{tikzcd} & Z_1 \arrow[d, bend left, "(f \otimes g)"] \\ \circ \arrow[r, "(1 \otimes e)"] & Z_2 \arrow[u, bend left, "(a \otimes b) \oplus (c \otimes d)"] \end{tikzcd}$$

(c)

$$\begin{aligned} Y_1 &= \begin{pmatrix} (1 \otimes e) \\ \odot\ (((a \otimes b) \oplus (c \otimes d)) \odot (f \otimes g))^{\circledast} \\ \odot\ ((a \otimes b) \oplus (c \otimes d)) \end{pmatrix}^{\lightning} \\ Y_2 &= \big((1 \otimes e) \odot (((a \otimes b) \oplus (c \otimes d)) \odot (f \otimes g))^{\circledast}\big)^{\lightning} \end{aligned}$$

(d)

Fig. 6. (a) A linear system of equations; (b) its regularization; (c) the graph corresponding to (b); (d) a closed-form solution for (a).

Definition 5. *A* ***Kronecker algebra*** $\mathbf{Kr} = \langle\langle A, +, \cdot, *, 0, 1\rangle, \langle T, \oplus, \odot, \circledast, \underline{0}, \underline{1}\rangle, \otimes, \lightning\rangle$ *is a tensor-product algebra that consists of two Kleene algebras* $\langle A, +, \cdot, *, 0, 1\rangle$ *and* $\langle T, \oplus, \odot, \circledast, \underline{0}, \underline{1}\rangle$ *such that (i) the natural order forms a complete lattice (i.e., both algebras have all infinite sums), and (ii) the following properties hold:*

1. $0 \otimes 0 = \underline{0}$
2. $1 \otimes 1 = \underline{1}$
3. $(a \otimes b)^{\lightning} = a \cdot b$, *for all* $a, b \in A$
4. $(a_1 \otimes b_1) \odot (a_2 \otimes b_2) = (a_2 \cdot a_1) \otimes (b_1 \cdot b_2)$, *for all* $a_1, a_2, b_1, b_2 \in A$
5. $(t_1 \oplus t_2)^{\lightning} = t_1^{\lightning} + t_2^{\lightning}$, *for all* $t_1, t_2 \in T$

We assume that all distributivity properties of A *and* T*, as well as item 5, hold for infinite sums. In particular, for item 5, we have*

$$\left(\bigoplus_{i \in I} t_i\right)^{\lightning} = \sum_{i \in I} t_i^{\lightning} \tag{10}$$

4.3 Discussion

The Instantiation of the Recipe. Returning to the recipe from Sect. 3.3, what we have done for a system of linear equations E_{Lin} is to instantiate the recipe as follows:

1. *(Modeling).* The concrete semantics is the least solution of E_{Lin} interpreted in relational semantics.
2. *(Closed forms).* Each variable of E_{Lin} is expressed as the detensor $((-)^{\lightning})$ of a tensored regular expression. Closed forms are computed from the closed-forms of the left-linear system of equations $\tau_{\mathrm{Reg}}(E_{\mathrm{Lin}})$ that results from the regularizing transformation (e.g., see Fig. 6).

3. *(Interpretation).* Tensored regular expressions can be interpreted as tensored transition formulas (Example 11), which are simply transition formulas over a "doubled" vocabulary.

Two Lessons. We would like to mention two lessons that we learned while working on this material over the years.

1. For the problems that arise in NPA, we must solve an equation system that is *truly linear*, not left-linear or right-linear. A reasonable sanity check might go as follows:
   - Algebraic program analysis à la Sect. 2 solves a left-linear (or right-linear) system of equations using methods based on regular expressions.
   - NPA repeatedly creates a system of linear equations that needs to be solved. Such linear equations are related to linear context-free languages, such as the language $\{a^i b^i\}$, which is not regular.
   - Ergo, it is a non-starter to attempt to apply algebraic program analysis to the equations that arise on each round of NPA.

   However, as shown in this section, it was possible to side-step this fundamental mismatch, by extending algebraic program analysis to systems of linear equations using Kronecker algebras, which have additional operations, such as tensor product and detensor.
   Thus, beyond the technical details, perhaps a more important takeaway is "be careful how you apply sanity checks." There is a risk that a plausible-sounding sanity check could cause you to discard an idea that is worth pursuing.
2. In some sense, the solution using Kronecker algebras goes against the grain of what computer scientists typically preach, namely, create appropriate abstractions (in the sense of abstract data-types) for a problem at hand, and then program your solution, thinking of the chosen abstractions as the operations of an abstract machine. This style of thinking is considered central to managing complexity in computer science, and it is generally considered heresy to break an abstraction.
   For algebraic program analysis, the abstraction is regular algebra, used with interpretations that are abstractions (in the sense of abstract interpretation [22]) of a program's concrete transition relations. However, the introduction of tensor product and detensor *breaks* that abstraction! To understand what we mean, consider the definition of $F \cdot G$ for transition relations in Boolean programs, i.e.,

$$(F \cdot G)(W, Z) \triangleq \exists X, Y. F(W, X) \wedge G(Y, Z) \wedge (X = Y),$$

   and the definitions of $F \otimes G$ and $T^{\lightning}$,[6] namely,

$$\begin{aligned}(F \otimes G)(W, X, Y, Z) &\triangleq F(W, X) \wedge G(Y, Z)\\ T(W, X, Y, Z)^{\lightning} &\triangleq \exists X, Y. T(W, X, Y, Z) \wedge (X = Y)\end{aligned}$$

6 Because we are trying to relate these operations to the untensored product operation $\cdot$, we do not make use of the stacked notation from Sect. 4.2.

The product operation $F \cdot G$ has three distinct steps: (i) conjoin $F(W, X)$ and $G(Y, Z)$; (ii) conjoin the equality $X = Y$; and (iii) project out vocabularies X and Y. In essence, tensor product and detensor break the abstraction of $\cdot$ as an indivisible operation: $\cdot$ is decomposed into two more-granular operations, $\otimes$ and $\lightning$. By performing $F \otimes G$, we perform just the first step of $\cdot$, and only later, when $\lightning$ is performed, do we "finish up" by applying the second and third steps of $\cdot$. The advantage is that we can operate on tensored values for some number of steps before "finishing" some earlier $\cdot$.

Again, beyond the technical details, the takeaway may be the *process* that we went through, which may be of value as a conceptual tool in other contexts:

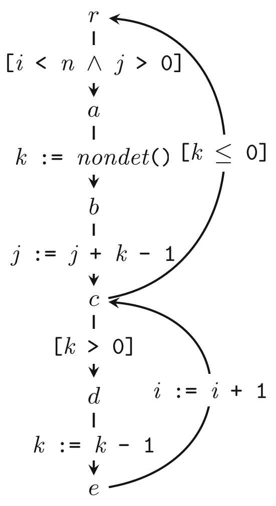
– The insight on how to break the abstraction—both as presented here and as occurred during our research seven or eight years ago—came from thinking about one *specific* interpretation of Kleene algebra: transition relations for Boolean programs.
– The algebraic properties of the new, finer-granularity operations allowed us to abstract out a new algebra, dubbed in this paper Kronecker algebra.
– The ideas could now be applied in other contexts by finding other interpretations of Kronecker algebra (or, because we are interested in program analysis, by finding interpretations that over-approximate Kronecker algebra).

5 Termination Analysis

This section describes how algebraic program analysis can be applied to termination analysis, based on the approach of [63]. The goal of termination analysis is to prove that a program has no infinite executions. Our high-level strategy is to exploit compositionality: we prove that a loop terminates by first computing a summary (e.g., a transition formula) for its body, and then finding a termination argument for the summary.

Following Sect. 3, we first formalize a concrete semantics as the (greatest) solution of a system of semantic equations. An appropriate notion of concrete semantics for termination analysis is the set of *non-terminating* states of the program (from which there exists an infinite execution)—the program terminates exactly when none of the program's initial states belong to this set. As in Sect. 3, this system of equations can be derived syntactically from a program's control flow graph—see Fig. 7 for an example. The non-terminating states of the program are the greatest solution to this system of equations over the algebra where the universe is the set of states, $\boxplus$ is interpreted as union (a state is non-terminating if it has at least one infinite execution) and $\boxdot$ is interpreted as preimage (a state is non-terminating iff it can reach a non-terminating state).[7]

[7] Despite the fact that this system of equations is right-linear, the method of Sect. 2 does not apply because the system of equations has two sorts instead of one; in particular, $\boxdot$ has type $\boxdot : 2^{\mathsf{State} \times \mathsf{State}} \times 2^{\mathsf{State}} \to 2^{\mathsf{State}}$, and so is not a binary operation on a set.



(a)

```
while (i < n ∧ j > 0) do
  k := nondet()
  j := j + k - 1
  while (k > 0) do
    k := k - 1
    i := i + 1
```

(b)

$$
\begin{aligned}
X_r &= \langle r, a \rangle \boxdot X_a \\
X_a &= \langle a, b \rangle \boxdot X_b \\
X_b &= \langle b, c \rangle \boxdot X_c \\
X_c &= (\langle c, r \rangle \boxdot X_r) \boxplus (\langle c, d \rangle \boxdot X_d) \\
X_d &= \langle d, e \rangle \boxdot X_e \\
X_e &= \langle e, c \rangle \boxdot X_c
\end{aligned}
$$

(c)

Fig. 7. A program represented by a control flow graph (a), abstract syntax tree (b), and system of equations (c).

A suitable language of "closed-form solutions" for the system of equations that arise in termination analysis is ω-regular expressions. The syntax of **ω-regular expressions** over an alphabet Σ is as follows:

$$
\begin{aligned}
a &\in \Sigma \\
R \in \mathsf{RegExp}(\Sigma) &::= a \mid 0 \mid 1 \mid R_1 + R_2 \mid R_1 \cdot R_2 \mid R^* \\
S \in \omega\text{-}\mathsf{RegExp}(\Sigma) &::= R^\omega \mid S_1 \boxplus S_2 \mid R \boxdot S
\end{aligned}
$$

The semantics of a (ω)-regular expressions is given by an interpretation over an ω-algebra and a regular algebra.

Definition 6. *An **ω-algebra** over a regular algebra* $\mathbf{A}$ *is 4-tuple* $\mathbf{B} = \left\langle B, \boxdot^B, \boxplus^B, ^{\omega^B} \right\rangle$ *consisting of a universe* B*, an operation* $\boxdot^B : A \times B \to B$*, an operation* $\boxplus^B : B \times B \to B$*, and an operation* $(-)^{\omega^B} : A \to B$*.*

Example 12 (Standard interpretation). In the *standard interpretation* of ω-regular expressions, the universe consists of sets of infinite sequences over the alphabet Σ, and the operations are

$$
\begin{aligned}
W_1 \boxplus W_2 &\triangleq W_1 \cup W_2 && \text{Union} \\
X \boxdot W &\triangleq \{xw : x \in X, w \in W\} && \text{Concatenation} \\
X^\omega &\triangleq \{x_1 x_2 \cdots : x_1, x_2, \cdots \in X\} && \text{Infinite repetition}
\end{aligned}
$$

For example, an ω-regular expression that recognizes all infinite paths in Fig. 7a starting at r is:

$$\overbrace{(\langle r,a\rangle\,\langle a,b\rangle\,\langle b,c\rangle\,(\langle c,d\rangle\,\langle d,e\rangle\,\langle e,c\rangle)^*\,\langle c,r\rangle)^\omega}^{\text{Outer loop}}$$
$$\boxplus\big((\langle r,a\rangle\,\langle a,b\rangle\,\langle b,c\rangle\,(\langle c,d\rangle\,\langle d,e\rangle\,\langle e,c\rangle)^*\,\langle c,r\rangle)^*\,\langle r,a\rangle\,\langle a,b\rangle\,\langle b,c\rangle\big)\boxdot\underbrace{(\langle c,d\rangle\,\langle d,e\rangle\,\langle e,c\rangle)^\omega}_{\text{Inner loop}}$$

⌟

Example 13 (Nonterminating state interpretation). The non-terminating state algebra is an ω-algebra over the algebra of state relations. Its universe consists of sets of states. The operators are

$$R\boxdot S \triangleq \{x : \exists y.\, \langle x,y\rangle \in R \land y \in S\} \qquad \text{Preimage}$$
$$S_1 \boxplus S_2 \triangleq S_1 \cup S_2 \qquad \text{Union}$$
$$R^\omega \triangleq \left\{x_0 \in \mathsf{State} : \begin{array}{l}\exists x_1, x_2, \ldots\\ \forall i.\, \langle x_i, x_{i+1}\rangle \in R\end{array}\right\} \qquad \text{Non-terminating states of } R$$

⌟

Tarjan's path expression algorithm can be adapted to compute an ω-regular expression that recognizes the set of infinite paths in a graph beginning at a particular node [63]. The equational view of this algorithm is that it computes closed-form solutions to right-linear equations over *Büchi algebras* (e.g., the algebra of non-terminating states).

Definition 7 (Büchi algebra). *A Büchi algebra is an ω-algebra over a Kleene algebra satisfying the following:*

$$\begin{aligned}
S_1 \boxplus (S_2 \boxplus S_3) &= (S_1 \boxplus S_2) \boxplus S_3 && \textit{Associativity}\\
S_1 \boxplus S_2 &= S_2 \boxplus S_1 && \textit{Commutativity}\\
S \boxplus S &= S && \textit{Idempotence}\\
((R_1 \cdot R_2) \boxdot S) &= R_1 \boxdot (R_2 \boxdot S) && \textit{Compatibility}\\
((R_1 + R_2) \boxdot S) &= (R_1 \boxdot S) \boxplus (R_2 \boxdot S) && \textit{Right-distributivity}\\
R \boxdot (S_1 \boxplus S_2) &= (R \boxdot S_1) \boxplus (R \boxdot S_2) && \textit{Left-distributivity}\\
R^\omega &= R \boxdot R^\omega && \textit{Unfold}\\
S_1 \preceq (R \boxdot S_1) \boxplus S_2 &\Rightarrow S_1 \preceq R^\omega \boxplus (R^* \boxdot S_2) && \textit{Coinduction}
\end{aligned}$$

where $\preceq$ is the order defined by $a \preceq b$ iff $a \boxplus b = b$.

Exercise 2. Show that in any Büchi algebra, the greatest solution to the equation $X = (a \boxdot X) \boxplus z$ exists and is equal to $X = a^\omega \boxplus (a^* \boxdot z)$.

Summarizing: we have modeled a program's non-terminating states as the greatest solution to a system of semantic equations, devised a language of "closed form solutions", and identified an algorithm for computing closed form solutions to the equations. It remains only to develop abstract interpretations of the language of closed forms which implements termination analysis.

5.1 Non-terminating State-Formula Interpretations

Just as transition formulas (over variables X and X') can be used to represent state relations, state formulas (over the variables X) can be used to represent sets of (non-terminating) states. We can extend an algebra of transition formulas to an algebra of non-terminating state formulas by defining

$$\begin{aligned} F \boxdot P &\triangleq \exists X'.F(X,X') \land P(X') && \text{Preimage} \\ P_1 \boxplus P_2 &\triangleq P_1 \lor P_2 && \text{Union} \end{aligned}$$

Intuitively, the ω operator should compute the set of non-terminating states of a transition formula. Analogously to the $*$ operator in Sect. 2, this set is uncomputable, and we must be satisfied with an over-approximation (i.e., we aim to compute a state formula that contains all non-terminating states—the soundness relation of interest is the one defined by $N \Vdash S \iff \forall s \in N.s \models S$). There are many ways of doing this, so we speak of the family of non-terminating state formula interpretations. In the remainder of this section, we give examples of ω-operators.

Example 14 (Linear-lexicographic ranking functions [32]*).* Let $F(X,X')$ be a transition formula. A *linear lexicographic ranking function* (LLRF) for F is a sequence of linear terms $t_1,\dots,t_n$ over X such that for any states s and s' such that $s \to_F s'$, each t_i evaluates to a non-negative integer in s, and the integer n-tuple decreases in lexicographic order going from s to s'. Since there are no infinite strictly descending chains of non-negative n-tuples of integers with respect to the lexicographic order, if F has an LLRF, then F has no non-terminating states. For example, the inner loop of Fig. 7 has a 1-dimensional LLRF $\langle k \rangle$, and the outer loop has a 2-dimensional LLRF $\langle n-i, j \rangle$.

The problem of determining whether a linear integer arithmetic formula has an LLRF is decidable [32]. If a formula does *not* have an LLRF, then we can use a coarse over-approximation of the non-terminating states of a formula (e.g., the set of states that have at least one outgoing transition). This yields the following interpretation of the ω operator:

$$F^\omega \triangleq \begin{cases} \mathit{false} & \text{if there is an LLRF for } F \\ \exists X'.F(X,X') & \text{otherwise} \end{cases}$$

For Fig. 7, using recurrence analysis to implement the $*$ operator (Example 5), we get that every non-terminating state must satisfy *false*—the program terminates from any initial state. ⌟

Example 15 (Unbounded trajectories [63]*).* Let $F(X,X')$ be a transition formula. A necessary (but not sufficient) condition for a state s to be a non-terminating for a transition formula F is that there is a computation of F starting from s for every possible length. This condition is undecidable, but it can be approximated using an approximate transitive-closure operator such as the ones in Sect. 2.1. Suppose that $(-)^*$ is an over-approximate transitive-closure operator. Letting k

and k' be symbols that do not appear in F, we can create a transition formula $\exp(F)$ in one parameter k' such that for any k', if there exists a sequence $s_1 \to_F s_2 \to_F \cdots \to_F s_{k'}$, then $s_1 \to_{\exp(F)} s_{k'}$:

$$\exp(F) \triangleq (F \wedge k' = k + 1)^*[k \mapsto 0]$$

The states s for which there exists a computation $s \to_{\exp(F)} s' \to s''$ for all choices of the parameter k' over-approximates the set of non-terminating states of F:

$$F^\omega \triangleq \forall k' \geq 0. \exists X', X''. \exp(F) \wedge F(X', X'')$$

For example, if $*$ is instantiated to recurrence analysis (Example 5), then on the transition formula

$$F \triangleq i \neq n \wedge i' = i + 2 \wedge n' = n$$

(corresponding to the program **while** $(i \neq n)$ **do** i := i + 2), we have

$$F^\omega = i > n \vee (n - i) \bmod 2 = 1$$ ⌟

Additional examples of termination analyses in the algebraic framework appear in [63] and [62].

5.2 The Instantiation of the Recipe

The recipe from Sect. 3.3 is instantiated for termination analysis as follows:

1. *(Modeling).* The concrete semantics is the set of non-terminating states, which is the greatest solution to a system of right-linear equations.
2. *(Closed forms).* The language of closed-forms is given by ω-regular expressions; they can be computed by a variation of Tarjan's algorithm [63].
3. *(Interpretation).* An ω-regular expression can be interpreted as a state formula representing a set of *possibly non-terminating states*, while regular expressions are interpreted as transition formulas (Sect. 2). The soundness relation is over-approximate: we can prove that a program terminates by finding an unsatisfiable pre-condition, but the analysis cannot prove non-termination.

6 Recap

This section contains a few remarks about commonalities among the three kinds of problems and the techniques we have presented for applying algebraic program analysis to them. The paper has been structured around the three-part recipe for algebraic program analysis given in Sect. 3.3. Table 1 recaps how the recipe has been instantiated for the three kinds of problems considered.

Within this paper, all methods for computing closed-form solutions can be understood as some variation of Gaussian elimination, Algorithm 1 (in practice, they are variations of Tarjan's path-expression algorithm). The essential

Table 1. Instantiations of the recipe for algebraic program analysis from Sect. 3.3.

	Section 3.3	Section 4.3	Section 5.2
Analysis type	Intraprocedural	Linear interprocedural	Termination
Modeling	LFP of left-linear equations	LFP of linear equations	GFP of right-linear equations
Closed-form solution	A regular expression (path expression over the CFG)	Detensor of a tensored path expression	An omega-regular expression
Interpretation (concrete)	A Kleene algebra (Definition 1), e.g., transition relations (Sect. 3.1)	A Kronecker algebra (Definition 5), e.g., tensored transition relations (Example 10)	A Büchi algebra (Definition 7), e.g., non-terminating states (Example 13)
Interpretation (abstract)	A regular algebra (Sect. 2), e.g., a transition-formula interpretation (Sect. 2.1)	A tensor-product algebra (Definition 3), e.g., a tensored transition-formula interpretation (Example 11)	An ω algebra (Definition 6), e.g., a non-terminating state-formula interpretation (Sect. 5.1)

Table 2. "Loop-solving" steps.

Equation type	Form of "loop"		Closed form for X
Left-linear	$X = a + Xb$	$\rightsquigarrow$	ab^*
Linear	$X = a + \sum_{i=1}^{m} b_i X c_i$	$\rightsquigarrow$	$((1 \otimes a) \odot (\bigoplus_{i=1}^{m} b_i \otimes c_i)^{\circledast})^{\lightning}$
Right-linear	$X = (b \boxdot X) \boxplus z$	$\rightsquigarrow$	$a^\omega \boxplus (b^* \boxdot z)$

difference between Sect. 2, Sect. 4, and Sect. 5 is the "loop-solving" step. Each requires the right-hand-side expression R to be in a particular form (left-linear, linear, right-linear), and each requires a different language of expressions in which to express closed forms (regular, tensored regular, ω-regular). Table 2 shows the respective "loop-solving" steps for computing a closed form. Note that in Table 2, the letters a, b_i, c_i, z range over expressions (which may involve variables other than X). For example, to apply the left-linear rule to the equation $X = Xp + Xq + Yr + Z$, we first re-arrange terms on the right-hand side as $X(p + q) + (Yr + Z)$ and then compute the "closed-form" $(Yr + Z)(p + q)^*$.

7 Related Work

Abstracting States Versus State Changes. Classically, invariant generation is conceived as the problem of over-approximating the reachable states of a program. Computing invariants involves solving a system of equations of the form

$$\begin{array}{lll} X[r] = v_r & & r \in \mathit{Nodes}, \text{ the root node} \\ X[n] = \displaystyle\sum_{e_{m,n} \in \mathit{Edges}} \mathscr{I}[\![e_{m,n}]\!](X[m]) & & n \in \mathit{Nodes} - \{r\} \end{array} \tag{11}$$

for the unknowns $X[n]$, $n \in \mathit{Nodes}$, where v_r represents the set of initial states and $\mathscr{I}[\![-]\!]$ provides an interpretation of each CFG edge as a state transformer.

In a solution, $X[n]$ holds a descriptor that represents a superset of the set of program states that can arise at program point n. Note that in Eq. (11), the function $\mathscr{I}[\![e_{m,n}]\!]$ on edge $e_{m,n}$ is *applied* to the value $X[m]$ on node m.

Algebraic program analyses, in contrast, concern dynamics—state *changes*—rather than states. The reason is that algebraic analyses are compositional: states do not compose, but state changes do.

A first step towards abstracting state *changes* was taken by Graham & Wegman [33], who gave a method to solve dataflow equations via *composition* of the state transformers on CFG edges. That is, their basic primitives were (i) composition of functions, and (ii) union of functions. If we adopt this outlook and define $r_1 \cdot r_2$ to be $r_2 \circ r_1$, $r_1 + r_2$ to be the union of r_1 and r_2, and 1 to be the identity function, instead of Eq. (11), the goal would be to solve the following equation system:

$$\begin{aligned} X[r] &= 1 && r \in \mathit{Nodes}, \text{ the root node} \\ X[n] &= \sum_{e_{m,n} \in \mathit{Edges}} X[m] \cdot \mathscr{I}[\![e_{m,n}]\!] && n \in \mathit{Nodes} - \{r\} \end{aligned} \tag{12}$$

where the unknowns $X[n]$ are now function-valued. Note that the function $\mathscr{I}[\![e_{m,n}]\!]$ on edge $e_{m,n}$ is *composed* with the value $X[m]$ on node m. From here—because one is working over function-valued quantities—it is now natural to formulate interprocedural program-analysis problems by means of equations over unknowns that denote procedure summaries, as was done by Cousot and Cousot [23] and Sharir and Pnueli [56].

"Interpret, Then Solve" Versus "Solve, Then Interpret." The systems in Eqs. (11) and (12) are *interpreted*, in the sense that they are understood as semantic equations valued over a particular abstract domain, say D. Such a system $E = \{X_i = R_i\}_{i \in I}$ can be solved by an iterative method: compute a sequence $\sigma_0, \sigma_1, \cdots \in \{X_i\}_{i \in I} \to D$ of assignments abstract domain values to variables

$$\begin{aligned} \sigma_0(X_i) &\triangleq 0 && \text{for all } i \in I \\ \sigma_{n+1}(X_i) &\triangleq \mathscr{I}_{\sigma_n}[\![R_i]\!] && \text{for all } n \geq 0 \text{ and all } i \in I \end{aligned}$$

Eventually this process converges—typically with the aid of widening to extrapolate to the limit—upon an assignment that over-approximates the least solution to E.

In algebraic program analysis, we think of a system of equations as an uninterpreted (syntactic) object. Equations are solved symbolically and then the solutions are interpreted in an algebraic structure to obtain an analysis result. The key step in this direction was made by Tarjan [59], who observed that once a solution to the path-expression problem was in hand, multiple dataflow-analysis problems could be solved merely by reinterpreting the alphabet symbols and operators of regular expressions in different algebras—i.e., "solve and then interpret."

Whereas the iterative framework for program analysis has a "built-in" algorithm for analyzing loops and recursive behavior (by computing the limit of a sequence), the algebraic framework does not prescribe any particular method, and it is up to the analysis designer to devise one. This obligation places an additional burden on the analysis designer, but also provides flexibility: the analysis designer may analyze loops in ways that may (Example 6) or may not (Examples 5 and 4) resemble iterative fixpoint computation.

Iteration Operators and Loop Summarization. In the computer-aided-verification community, there is a body of literature on loop summarization (or "loop leaping") and acceleration. Summarization aims to compute or approximate the behavior of (certain) loops, while acceleration aims to approximate the postimage of a set of states under a loop. These techniques have been incorporated into iterative abstract interpretation [28,31], abstraction-refinement-based software model checking [19,37], termination analysis [7,20,60], and resource bound analysis [10,64]. The most closely related techniques to algebraic program analysis are those that build summaries for whole programs in "bottom-up" fashion. Such analyses have been formalized in various ways, including: recursion on the abstract syntax tree (AST) of a program [51], AST rewriting [8], and graph rewriting [47,60]. Algebraic program analysis provides a unifying foundation for such analyses, in the same way that dataflow analysis [39] and (iterative) abstract interpretation [22] provide a unifying foundation for iterative program analyses.

There are several methods for loop summarization, based on finite-monoid affine transformations [11,12,29], difference-bound relations [15,21], octagonal relations [13,14,45], integer vector addition systems [35], fragments of the theory of arrays [2]. For the most part, these summarization methods are non-uniform in the sense that their input language differs from their output language (e.g., [13] takes as input an octagonal relation and produces as output a Presburger formula). This non-uniformity is the essential barrier that must be overcome to use such techniques to implement the iteration operator of an algebraic program analysis (e.g., we can define an iteration operator by using optimization modulo theories [55] to extract the octagonal hull of a Presburger formula, then use [13] to compute a Presburger formula representing its transitive closure).

Elimination-Based Dataflow Analysis. Elimination-based dataflow analysis is a family of dataflow analyses that computes analysis results using methods that resemble Gaussian elimination [3,33,36] (see [54] for a survey). Early methods were specialized to reducible control flow graphs, but operated faster than general Gaussian elimination. Tarjan's algorithm [58] is an elimination method with fast operation on reducible (and "nearly reducible") control flow graphs, but is applicable to arbitrary graphs.

Weighted Graphs. There is a vast literature on solving path problems on weighted graphs where the weights are drawn from a semiring [1,30,50]. Path problems can also be solved on semiring-weighted pushdown systems, which has applications to interprocedural dataflow analysis [52]. This work focuses on *iterative* techniques for solving path problems.

(Non-iterative) algorithms for path problems over algebraic structures with an explicit iteration operator were considered by Aho et al. [1], Backhouse & Carré [5], and Lehmann [48], and was implicit in previous work by Kleene [44], and McNaughton & Yamada [49]. Tarjan connected this line of work with program analysis [58,59].

8 Open Problems

We conclude with a list of challenges suggested by algebraic program analysis.

Scaling SMT-Based Algebraic Program Analysis. The bottom-up interpretation step of a closed-form expression is efficient, in that it operates in linear time and space in the size of the expression DAG in a model where each algebraic operation has unit cost. For logic-based interpretations, however, algebraic operations do *not* have unit cost: operators manipulate formulas, and the size of those formulas may grow as operators are applied. For example, the regular expression a^{2^n} can be represented by an expression DAG with $n+1$ nodes, with the following shape:

$$\cdot \rightrightarrows \cdot \rightrightarrows \cdot \rightrightarrows \cdots \rightrightarrows \cdot \rightrightarrows a$$

If the letter a is interpreted as the transition formula $x' = x + 1$ and $\cdot$ as relational composition, then the transition-formula interpretation of a^{2^n} has size $O(2^n)$. Scaling SMT-based algebraic program analysis to large programs requires techniques for generating *succinct* summaries, and/or efficient reasoning about compact formula representations involving λ-expressions.

Recursive Procedures. Section 4.2 shows how the algebraic approach can be applied to summarize *linearly* recursive procedures. But to compute summaries for generally recursive procedures, current-generation algebraic-program-analysis tools fall back on another non-algebraic scheme (such as hybrid iterative/algebraic, like Kleene or Newton iteration [40,53], or the template-based approach of [16]). This raises the question: is there a practical algebraic method for analyzing general recursion? The essential challenge is in devising a language of "closed forms" that (1) can represent arbitrary context-free languages, and (2) is amenable to an effective interpretation in logic.

Beyond Numerical Domains. To date, all algebraic program analyses have been numerical in nature—they abstract away aspects of program behavior that cannot be captured by integer variables. It remains to be seen whether the algebraic approach can yield practical analyses for reasoning about features like strings, arrays, and the heap. Reasoning about memory manipulation is particularly challenging in a compositional setting, since we cannot rely on the context of a program fragment to resolve aliasing relationships. One possible avenue is to incorporate abductive reasoning to make educated guesses about the shape of memory, as in [18].

Property Refutation. Algebraic program analysis is typically conceived as a method for generating over-approximate summaries. The nature of over-approximation is that the summaries can be used to verify that a program *does* satisfy a property of interest, but not prove that it *doesn't.* An interesting direction for future work is to devise methods by which algebraic program analyses can refute properties, perhaps based on bounded model checking [9], under-approximate loop summarization [46], or symbolic execution [43].

Acknowledgments. Supported, in part, by a gift from Rajiv and Ritu Batra; by a Facebook Research Award; by NSF under grant number 1942537, and by ONR under grants N00014-17-1-2889 and N00014-19-1-2318. 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 entities.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1974)
2. Alberti, F., Ghilardi, S., Sharygina, N.: Definability of accelerated relations in a theory of arrays and its applications. In: Fontaine, P., Ringeissen, C., Schmidt, R.A. (eds.) FroCoS 2013. LNCS (LNAI), vol. 8152, pp. 23–39. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40885-4_3
3. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Commun. ACM **19**(3), 137 (1976)
4. Ancourt, C., Coelho, F., Irigoin, F.: A modular static analysis approach to affine loop invariants detection. Electr. Notes Theor. Comp. Sci. **267**(1), 3–16 (2010)
5. Backhouse, R., Carré, B.: Regular algebra applied to path-finding problems. J. Inst. Math. Appl. **15**, 161–186 (1975)
6. Backhouse, R.C., Carré, B.A.: Regular algebra applied to path-finding problems. IMA J. Appl. Math. **15**(2), 161–186 (1975)
7. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.: Variance analyses from invariance analyses. In: POPL, pp. 211–224 (2007)
8. Biallas, S., Brauer, J., King, A., Kowalewski, S.: Loop leaping with closures. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 214–230. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33125-1_16
9. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
10. Blanc, R., Henzinger, T.A., Hottelier, T., Kovács, L.: ABC: algebraic bound computation for loops. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 103–118. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_7
11. Boigelot, B.: On iterating linear transformations over recognizable sets of integers. Theor. Comput. Sci. **309**(1), 413–468 (2003)
12. Boigelot, B., Wolper, P.: Symbolic verification with periodic sets. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 55–67. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58179-0_43

13. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 337–351. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_29
14. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_23
15. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 577–588. Springer, Heidelberg (2006). https://doi.org/10.1007/11787006_49
16. Breck, J., Cyphert, J., Kincaid, Z., Reps, T.: Templates and recurrences: better together. In: PLDI, pp. 688–702 (2020)
17. Brzozowski, J.A.: Regular-like expressions for some irregular languages. In: SWAT (FOCS), pp. 278–286 (1968)
18. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM **58**(6), 1–66 (2011)
19. Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 428–442. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_32
20. Chen, H., David, C., Kroening, D., Schrammel, P., Wachter, B.: Bit-precise procedure-modular termination analysis. TOPLAS **40**(1), 1:1-1:38 (2018)
21. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028751
22. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
23. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: Neuhold, E. (ed.) Formal Descriptions of Programming Concepts, (IFIP WG 2.2, St. Andrews, Canada, August 1977), pp. 237–277. North-Holland (1978)
24. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. **2**(4), 511–547 (1992)
25. Cyphert, J., Breck, J., Kincaid, Z., Reps, T.W.: Refinement of path expressions for static analysis. Proc. ACM Program. Lang. **3**(POPL), 45:1–45:29 (2019)
26. Esparza, J., Kiefer, S., Luttenberger, M.: Newtonian program analysis. J. ACM **57**, 6 (2010)
27. Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: FMCAD, pp. 57–64 (2015)
28. Feautrier, P., Gonnord, L.: Accelerated invariant generation for C programs with aspic and c2fsm. Electr. Notes Theor. Comput. Sci. **267**(2), 3–13 (2010)
29. Finkel, A., Leroux, J.: How to compose Presburger-accelerations: applications to broadcast protocols. In: Agrawal, M., Seth, A. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36206-1_14
30. Gondran, M., Minoux, M.: Graphs, Dioids and Semirings: New Models and Algorithms. ORCS, vol. 41, 1st edn. Springer, Boston (2008). https://doi.org/10.1007/978-0-387-75450-5
31. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006). https://doi.org/10.1007/11823230_10

32. Gonnord, L., Monniaux, D., Radanne, G.: Synthesis of ranking functions using extremal counterexamples. SIGPLAN Not. **50**(6), 608–618 (2015)
33. Graham, S.L., Wegman, M.N.: A fast and usually linear algorithm for global flow analysis. J. ACM **23**(1), 172–202 (1976)
34. Gruska, J.: Some classifications of context-free languages. Inf. Control **14**(2), 152–179 (1969)
35. Haase, C., Halfon, S.: Integer vector addition systems with states. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) RP 2014. LNCS, vol. 8762, pp. 112–124. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11439-2_9
36. Hecht, M.S., Ullman, J.D.: Analysis of a simple algorithm for global data flow problems. In: POPL, pp. 207–217 (1973)
37. Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, pp. 187–202. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33386-6_16
38. Karr, M.: Affine relationship among variables of a program. Acta Inf. **6**, 133–151 (1976)
39. Kildall, G.: A unified approach to global program optimization. In: POPL (1973)
40. Kincaid, Z., Breck, J., Boroujeni, A.F., Reps, T.W.: Compositional recurrence analysis revisited. In: PLDI, pp. 248–262 (2017)
41. Kincaid, Z., Breck, J., Cyphert, J., Reps, T.W.: Closed forms for numerical loops. Proc. ACM Program. Lang. **3**(POPL), 55:1–55:29 (2019)
42. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.W.: Non-linear reasoning for invariant synthesis. Proc. ACM Program. Lang. **2**(POPL), 54:1–54:33 (2018)
43. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
44. Kleene, S.: Representation of events in nerve nets and finite automata. In: Shannon, C., McCarthy, J. (eds.) Automata Stud., pp. 3–40. Princeton University Press, Princeton (1956)
45. Konečný, F.: PTIME computation of transitive closures of octagonal relations. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 645–661. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_42
46. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 381–396. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_26
47. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: Cha, S.S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 111–125. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88387-6_10
48. Lehmann, D.J.: Algebraic structures for transitive closure. Theoret. Comput. Sci. **4**(1), 59–76 (1977)
49. McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. IRE Trans. Electron. Comput. **9**(1), 39–47 (1960)
50. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. J. Autom. Lang. Comb. **7**(3), 321–350 (2002)
51. Monniaux, D.: Automatic modular abstractions for linear constraints. In: POPL, pp. 140–151 (2009)
52. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. SCP **58**(1–2), 206–263 (2005)
53. Reps, T., Turetsky, E., Prabhu, P.: Newtonian program analysis via tensor product. TOPLAS **39**(2), 9:1–9:72 (2017)

54. Ryder, B.G., Paull, M.C.: Elimination algorithms for data flow analysis. ACM Comput. Surv. (CSUR) **18**(3), 277–316 (1986)
55. Sebastiani, R., Tomasi, S.: Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ cost functions. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS (LNAI), vol. 7364, pp. 484–498. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31365-3_38
56. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications. Prentice-Hall (1981)
57. Szabó, Z.: Compositionality (2020). https://plato.stanford.edu/entries/compositionality/
58. Tarjan, R.E.: Fast algorithms for solving path problems. J. ACM **28**(3), 594–614 (1981)
59. Tarjan, R.E.: A unified approach to path problems. J. ACM **28**(3), 577–593 (1981)
60. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop summarization and termination analysis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 81–95. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19835-9_9
61. Yntema, M.: Inclusion relations among families of context-free languages. Inf. Control **10**, 572–597 (1967)
62. Zhu, S., Kincaid, Z.: Reflections on termination of linear loops. In: CAV (2021)
63. Zhu, S., Kincaid, Z.: Termination analysis without the tears. In: PLDI (2021)
64. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound analysis of imperative programs with the size-change abstraction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 280–297. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_22