Numerical Invariants via Abstract Machines[⊛]

Zachary Kincaid

Princeton University

Abstract. This paper presents an overview of a line of recent work on generating non-linear numerical invariants for loops and recursive procedures. The method is compositional in the sense that it operates by breaking the program into parts, analyzing each part independently, and then combining the results. The fundamental challenge is to devise an effective method for analyzing the behavior of a loop given the results of analyzing its body. The key idea is to separate the problem into two: first we approximate the loop dynamics by an abstract machine, and then symbolically compute the reachability relation of the abstract machine.

1 Introduction

Compositional recurrence analysis (CRA) is a method for generating numerical invariant for loops [17, 25, 26]. The goal of CRA is to compute a *transition formula* that over-approximates the behavior of the program. CRA analyzes programs bottom-up, in the style of an *effective* denotational semantics: we syntactically decompose the program into parts, compute a transition formula for each part independently, and then compose the results. The composition operators for transition formulas correspond to the familiar regular expression operations of *sequencing*, *choice*, and *iteration*.

The essence of the analysis is the *iteration* operator. Given a transition formula that over-approximates the body of a loop, the iteration operator computes a transition formula that over-approximates any number of iterations of the loop. CRA accomplishes this by extracting recurrence relations from a transition formula using an SMT solver, and then computing the closed form of those recurrences. Using this strategy, CRA can compute rich numerical invariants, including polynomial and exponential equations and inequations.

This paper gives an alternate account for this strategy, which is based on extracting an abstract machine that simulates the loop body, and then computing a closed form for the reachability relation of that abstract machine (thus we replace "recurrence relations" with the broader notion of "abstract machine"). Seen in this light, CRA is an answer to the question *given some simple model of computation that admits a closed representation of the reachability relation, how can we make use of it in program analysis?*

Secondly, this paper describes how the compositional approach to program analysis can be used to analyze recursive procedures [25]. A key idea is to exploit

[⊛] This version corrects some errors that appeared in the SAS'18 paper.

the two-phase structure of the iteration operator: we can detect and enforce convergence of procedure summaries using widening and equivalence operations on abstract machines.

The remainder of the paper is organized as follows. §2 gives a short introduction to compositional program analysis. The technical core of the paper is §3, which gives a recipe for analyzing loops by computing the reachability relation of an abstract machine that simulates its body. §4 illustrates how abstract machines can be used to analyze programs with (recursive) procedures. §5 surveys related work, and §6 concludes.

2 Outline

We begin by defining a simple structured programming language:

$$\begin{aligned} x &\in \mathsf{Var} \\ e \in \mathsf{Expr} &::= \mathtt{x} \mid n \in \mathbb{Z} \mid e_1 + e_2 \mid e_1 e_2 \\ c \in \mathsf{Cond} &::= e_1 \leq e_2 \mid e_1 = e_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \neg c \\ P \in \mathsf{Program} &::= \mathtt{x\ :=\ } e \mid P_1 \mathtt{;} P_2 \mid \mathtt{if}\ c\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2 \mid \mathtt{while}\ c\ \mathtt{do}\ P \end{aligned}$$

Our goal is to compute, for any given program P, a transition formula $\mathbf{TF}[\![P]\!]$ that over-approximates its behavior. A transition formula is a logical formula over the program variables Var and a set of primed copies Var', representing the values of the program variables before and after executing a program. In the following, we will make use of several different languages for expressing transition formulas. For the sake of concreteness, we give a definition of polynomial arithmetic transition formulas, PolyTF, below:

$$\begin{aligned} s, t \in \mathsf{PolyTerm} &::= \mathtt{x} \in \mathsf{Var} \mid \mathtt{x}' \in \mathsf{Var}' \mid y \in \mathsf{BoundVar} \mid \lambda \in \mathbb{Q} \mid s + t \mid st \\ F, G \in \mathsf{PolyTF} &::= s \leq t \mid s = t \mid s < t \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \exists y \in \mathbb{N}.F \mid \exists y \in \mathbb{Z}.F \end{aligned}$$

For any given program P, a transition formula $\mathbf{TF}[\![P]\!]$ can be computed by recursion on syntax:

$$\begin{aligned} \mathbf{TF}[\![\mathtt{x\ :=\ } e]\!] &\triangleq \mathtt{x}' = e \wedge \bigwedge_{\mathtt{y} \neq \mathtt{x} \in X} \mathtt{y}' = \mathtt{y} \\ \mathbf{TF}[\![\mathtt{if}\ c\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2]\!] &\triangleq (c \wedge \mathbf{TF}[\![P_1]\!]) \vee (\neg c \wedge \mathbf{TF}[\![P_2]\!]) \\ \mathbf{TF}[\![P_1 \mathtt{;} P_2]\!] &\triangleq \exists X \in \mathbb{Z}.\mathbf{TF}[\![P_1]\!][\mathsf{Var}' \mapsto X] \wedge \mathbf{TF}[\![P_2]\!][\mathsf{Var} \mapsto X] \\ \mathbf{TF}[\![\mathtt{while}\ c\ \mathtt{do}\ P]\!] &\triangleq (c \wedge \mathbf{TF}[\![P]\!])^{\circledast} \wedge (\neg c[\mathsf{Var} \mapsto \mathsf{Var}']) \end{aligned}$$

where $(-)^{\circledast}$ is an iteration operator: a function that computes an approximation of the transitive closure of a transition formula. Thus, the essential problem involved in designing a program analysis in this style is to define the iteration operator.

3 Approximating Loops with Abstract Machines

This section outlines a general strategy for loop summarization which is based on decomposing the problem into two: (1) find an abstract machine that simulates the action of the transition formula, and (2) express the reachability relation of the abstract machine as a transition formula. We then describe compositional recurrence analysis as an instance of this strategy. We begin with an example.

Example 3.1 Consider the program P given below

```
      while (i < n) do
         i := i + 1
         if (y < z)
Body        y := y + i - 1
         else
            z := z + i - 1
```

Recall that $\mathbf{TF}[\![P]\!] = (\mathtt{i} < \mathtt{n} \land \mathbf{TF}[\![\mathit{Body}]\!])^{\circledast} \land \mathtt{n} \leq \mathtt{i}$, where $(-)^{\circledast}$ is an iteration operator (yet to be defined) and

$$\mathbf{TF}[\![\mathit{Body}]\!] \equiv \mathtt{i} < \mathtt{n} \land \mathtt{i}' = \mathtt{i} + 1 \land \begin{pmatrix} (\mathtt{y} < \mathtt{z} \land \mathtt{y}' = \mathtt{i} + 1 \land \mathtt{z}' = \mathtt{z}) \\ \lor (\mathtt{z} \leq \mathtt{y} \land \mathtt{y}' = \mathtt{y} \land \mathtt{z}' = \mathtt{z} + 1) \end{pmatrix} .$$

The formula $F \triangleq \mathtt{i} < \mathtt{n} \land \mathbf{TF}[\![\mathit{Body}]\!]$ defines a transition relation $R \subseteq \mathbb{Z}^4 \times \mathbb{Z}^4$ on the state space $\mathbb{Z}^4$, where each vector $\boldsymbol{u} = \begin{bmatrix} i \; y \; z \; n \end{bmatrix}^T$ corresponds to an assignment of values to the program variables `i`, `y`, `z`, and `n`. The behavior of F is difficult to analyze directly, so instead we will approximate by a simpler system that is more amenable to analysis. We observe that F is simulated by the affine transformation

$$f(\boldsymbol{x}) = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

where the correspondence between the state space of F (i.e., $\mathbb{Z}^4$) and the state space of f (i.e., $\mathbb{Q}^3$) is given by the linear transformation

$$S = \begin{array}{cl} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} & \begin{array}{l} \leftarrow \text{1st dimension corresponds to } \mathtt{i} \\ \leftarrow \text{2nd dimension corresponds to } \mathtt{y} + \mathtt{z} \\ \leftarrow \text{3rd dimension corresponds to } \mathtt{n} \end{array} \end{array}$$

That is, we have that for every $\boldsymbol{u}$ and $\boldsymbol{u}'$ in $\mathbb{Z}^4$ such that $\boldsymbol{u}$ may transition to $\boldsymbol{u}'$ via F, we have $S\boldsymbol{u}' = f(S\boldsymbol{u})$. Phrased differently, we have

$$F \models \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathtt{i}' \\ \mathtt{y}' \\ \mathtt{z}' \\ \mathtt{n}' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathtt{i} \\ \mathtt{y} \\ \mathtt{z} \\ \mathtt{n} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{or,}$$

$$F \models \mathtt{i}' = \mathtt{i} + 1 \land (\mathtt{y}' + \mathtt{z}') = (\mathtt{y}' + \mathtt{z}') + \mathtt{i} \land \mathtt{n}' = \mathtt{n} \ . \tag{1}$$

The analysis of affine systems is classical. We can compute the following symbolic representation of the transitive closure of the transition relation defined by f:

$$c\ell(f) = \exists k \in \mathbb{N}. x_1' = x_1 + kx_2 + \frac{k(k+1)}{2} \land x_2' = x_2 + k \land x_3' = x_3$$

Since f simulates the behavior of F, then $c\ell(f)$ simulates the behavior of any number of iterations of F. Thus, if we define

$$F^{\circledast} = \exists k \in \mathbb{N}. \mathtt{y}' + \mathtt{z}' = \mathtt{y} + \mathtt{z} + k\mathtt{i} + \frac{k(k+1)}{2} \land \mathtt{i}' = \mathtt{i} + k \land \mathtt{n}' = \mathtt{n} \tag{2}$$

then we may take $\mathbf{TF}[\![P]\!] = F^{\circledast} \land \mathtt{i}' \geq \mathtt{n}'$ to be a conservative over-approximation the behavior of P. ⌟

3.1 Approximating formulas by machines

Definition 1. *An* ***(m×n)-formula*** *is a formula whose free variables range over $m + n$ free variables $x_1, ..., x_m$ and $x_1', ..., x_n'$. For any $(m \times n)$-formula F, we use $\mathcal{R}[\![F]\!]$ to denote the relation that F represents:*

$$\mathcal{R}[\![F]\!] \triangleq \{(\boldsymbol{u}, \boldsymbol{v}) \in \mathbb{Q}^m \times \mathbb{Q}^n : \{x_1 \mapsto u_1, ..., x_n \mapsto u_n, x_1' \mapsto v_1, ..., x_n' \mapsto v_n\} \models F\}$$

*We call an $(n \times n)$-formula an n-**transition formula**. We use* **TF** *to denote the set of all transition formulas (for any n).*

If F is an $(m \times n)$-formula and $\boldsymbol{y} = y_1, ..., y_m$ and $\boldsymbol{z} = z_1, ..., z_n$ are vectors of variables of lengths m and n, we use $F(\boldsymbol{y}, \mathbf{z})$ to denote the result of replacing each x_i with y_i and each x_i' with z_i. If F is an $(\ell \times m)$-formula and G is an $(m \times n)$-formula, we use $F \odot G$ to denote the relational composition of F and G:

$$F \odot G \triangleq \exists \boldsymbol{y}. F(\boldsymbol{x}, \boldsymbol{y}) \land G(\boldsymbol{y}, \boldsymbol{x}') \ .$$

We use $\breve{F}$ to denote the reversal of F, the $(m \times \ell)$-formula defined by

$$\breve{F} \triangleq F[x_1 \mapsto x_1', ..., x_m \mapsto x_m', x_1' \mapsto x_1, ..., , x_n' \mapsto x_n]$$

Abstract machines Fix some class of abstract machines **M**, which can be understood as some kind of discrete dynamical system with a numerical state space. We suppose that we are given two functions that related abstract machines to transition formulas:

- $\gamma : \mathbf{M} \to \mathbf{TF}$, which maps each machine M to its *concretization* $\gamma(M)$, a transition formula that represents the action of one step of M.
- $c\ell : \mathbf{M} \to \mathbf{TF}$, which maps each machine M it its *closure* $c\ell(M)$, a transition formula that represents the action of any number of steps of M.

We assume that for any machine M in $\mathbf{M}$, we have $\mathcal{R}[\![\gamma(M)]\!]^* = \mathcal{R}[\![c\ell(M)]\!]$.[1]

Example 3.2 Let **1-LT** denote the set of affine transformations of the form $f(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$, where A is a lower triangular matrix with 1's on the diagonal (e.g., the function f in Example 1). For any $f(\boldsymbol{x}) = A\boldsymbol{x} + \boldsymbol{b}$ in **1-LT**, define the concretization of f simply as $\gamma(f) \triangleq \boldsymbol{x}' = A\boldsymbol{x} + \boldsymbol{b}$. The readability relation of an affine transformation in **1-LT** can be expressed in polynomial arithmetic (i.e, PolyTF) and computed in polytime. The procedure is a specialization of the classical one for computing the reachability relation of an affine transformation (which in general does not have a closed form in PolyTF)—see [17, §III.B] for details. ⌟

Simulation Simulation relations are a standard approach to relating the behavior of dynamical systems [31]. Below we specialize the theory to our setting.

Definition 2. *Let F be an m-transition formula and let G be an n-transition formula. A* ***simulation formula*** *is an $(m \times n)$-formula S such that for all $(\boldsymbol{u}, \boldsymbol{v}) \in \mathcal{R}[\![S]\!]$, for every $\boldsymbol{u'}$ such that $(\boldsymbol{u}, \boldsymbol{u'}) \in \mathcal{R}[\![F]\!]$, there exists some $\boldsymbol{v'}$ such that $(\boldsymbol{v}, \boldsymbol{v'}) \in \mathcal{R}[\![G]\!]$ and $(\boldsymbol{u'}, \boldsymbol{v'}) \in \mathcal{R}[\![S]\!]$. Diagrammatically,*

$$\begin{tikzcd}
\forall\, \boldsymbol{u} \arrow[r, "F"] \arrow[d, dash, "S"'] & \boldsymbol{v} \arrow[d, dash, dashed, "S"] \\
\boldsymbol{u'} \arrow[r, dashed, "G"'] & \boldsymbol{v'}\ \exists
\end{tikzcd}$$

We use $S : F \Vdash G$ to denote that S is simulation formula from F to G.

Example 3.3 Consider Example 1. For ease of reading, we will refer to original variables i, y, z, n of the system rather than their canonical names x_1, x_2, x_3, x_4. The simulation between relation between the transition formula F and the affine map f is

$$S \triangleq x_1' = y + z \wedge x_2' = i \wedge x_3' = n\ .$$

This is a special simulation in that it is functional (each state of the program is related to exactly one state of the affine system), but this need not be the case. For example, suppose that we know (perhaps by running a sign analysis on the program P) that `i` is non-negative. Let $G = F \wedge \mathtt{i} \geq 0$. While we cannot understand the effect of the loop on `y` and `z` as an affine transformation, we can so understand lower and upper bounds on them: `y` and `z` are incremented by at least 0 and at most `i`. This abstraction can be realized by the function $g \in$ **1-LT**

[1] For our purposes, the weaker hypothesis $\mathcal{R}[\![\gamma(M)]\!]^* \subseteq \mathcal{R}[\![c\ell(M)]\!]$ is sufficient. We use equality to emphasize that $\mathbf{M}$ is expected to be a class of machines that is easy to analyze.

and simulation T defined by

$$g(\boldsymbol{x}) = \begin{bmatrix} 1\,0\,0\,0\,0\,0\,0 \\ 1\,1\,0\,0\,0\,0\,0 \\ 0\,0\,1\,0\,0\,0\,0 \\ 0\,1\,0\,1\,0\,0\,0 \\ 0\,1\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,0\,1\,0 \\ 0\,0\,0\,0\,0\,0\,1 \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad T = \begin{pmatrix} x_1 = \mathtt{i} \\ \land\, x_2 = \mathtt{y} + \mathtt{z} \\ \land\, x_3 = \mathtt{n} \\ \land\, x_4 \geq \mathtt{y} \\ \land\, x_5 \geq \mathtt{z} \\ \land\, x_6 \geq -\mathtt{y} \\ \land\, x_7 \geq -\mathtt{z} \end{pmatrix}$$

⌟

The last ingredient we need to be able to define (although not yet *compute*) an approximate transitive closure operator is a way of understanding an $(n \times m)$ simulation formula and a m-transition formula as an n-transition formula. This is given by conjugation:

Definition 3. *Let F be an $(m \times m)$-formula and let S be an $(n \times m)$-formula. The **conjugation of** F **by** S, $S \triangleright F$, is the $(n \times n)$-formula defined by*

$$S \triangleright F \triangleq \forall \boldsymbol{y}.\exists \boldsymbol{y}'.S(\boldsymbol{x}, \boldsymbol{y}) \Rightarrow (F(\boldsymbol{y}, \boldsymbol{y}') \land S(\boldsymbol{x}', \boldsymbol{y}'))$$

Example 3.4 In Example 1, conjugation of f by S yields the formula in Eq (1), and conjugation of $c\ell(f)$ by S yields the formula in Eq (2). ⌟

Observe that $S \triangleright F$ is the *strongest* among all n-transition formulas G such that S is a simulation from G to F. That is, we have

1. $S : (S \triangleright F) \Vdash F$
2. For all n-transition formulas G, we have $S : G \Vdash F$ if and only if $G \models S \triangleright F$.

Moreover, note that $R \triangleright (S \triangleright F) \models (R \circ S) \triangleright F$ and $id \triangleright F \equiv F$, where id denotes an identity relation of appropriate dimension.

Finally, we arrive at the central observation underlying our approach:

Observation 1. *Let F be a transition formula, let M be an abstract machine, and let S be a simulation formula such that $S : F \Vdash M$. Let $F^{\circledast} = S \triangleright c\ell(M)$. Then $\mathcal{R}[\![F]\!]^* \subseteq \mathcal{R}[\![F^{\circledast}]\!]$.*

That is: provided we can compute for any transition formula F an abstract machine M and a simulation S such that $S : F \Vdash M$, we can over-approximate the transitive closure of F with the transition formula $S \triangleright c\ell(M)$.

3.2 Computing (best) abstractions

We now turn to the question of what it means for an abstract machine to be a *best* abstraction of a transition formula. Consider again Example 1: the function f is *an* affine transformation in **1-LT** that simulates the given loop, but might there be a better one, whose closure yields more precise information about the loop?

To investigate this problem, it is convenient to use the language of category theory. Fix some class of simulation formulas $\mathcal{S}$, which is quotiented by logical equivalence, contains all identity relations, and is closed under composition (e.g., $\mathcal{S}$ might be the class of simulation formulas that correspond to linear transformations as in Example 1). We construct a category $\mathbf{TF}_{\mathcal{S}}$ where the objects are transition formulas (or perhaps transition formulas of a certain kind, e.g., formulas expressed in Presburger arithmetic), and the morphisms $S : F \to G$ are simulations belonging to $\mathcal{S}$ such that $S : F \Vdash G$. Similarly, we may construct a category of abstract machines $\mathbf{M}_{\mathcal{S}}$ similarly where the objects are machines and the morphisms are $\mathcal{S}$-simulations. The concretization function γ can now be extended to a functor $\gamma : \mathbf{M}_{\mathcal{S}} \to \mathbf{TF}_{\mathcal{S}}$, which maps each machine M in $\mathbf{M}$ to its concretization $\gamma(M)$, and maps each simulation $S : M \to M'$ between $\mathbf{M}$-machines to the same simulation $S : \gamma(M) \to \gamma(M')$ between their associated transition formulas. The closure function $c\ell$ can likewise be extended to a functor. The question of whether the transition formulas in $\mathbf{TF}$ have best abstractions in $\mathbf{M}$ with respect to simulations in $\mathcal{S}$ can now be phrased as: *does the functor γ have a left-adjoint?*

Recall that a functor $\alpha : \mathbf{TF}_{\mathcal{S}} \to \mathbf{M}_{\mathcal{S}}$ is left-adjoint to γ if there is a pair of natural transformations

- $\eta : 1_{\mathbf{TF}_{\mathcal{S}}} \Rightarrow \gamma \circ \alpha$ (the *unit* of the adjunction)
- $\epsilon : \alpha \circ \gamma \Rightarrow 1_{\mathbf{M}_{\mathcal{S}}}$ (the *counit* of the adjunction)

such that (1) for all transition formulas F, we have $1_{\alpha(F)} = \epsilon_{\alpha(F)} \circ \alpha(\eta_F)$ and (2) for all abstract machines M, we have $1_{\gamma(M)} = \gamma(\epsilon_M) \circ \eta_{\gamma(M)}$. The best abstraction of a transition formula F can be conceived of as the pair $(\alpha(F), \eta_F)$ consisting of an abstract machine $\alpha(F)$ (*which* machine best captures the behavior of F) and a simulation $\eta_F : F \to \gamma(\alpha(F))$ (*how* the machine $\alpha(F)$ captures the behavior of F). The sense in which $(\alpha(F), \eta_F)$ is *best* abstraction is that for any other machine M and simulation S, there is a unique simulation $\overline{S} : \alpha(F) \to M$ such that $S = \gamma(\overline{S}) \circ \eta_F$; that is, the following diagram commutes:

$$\begin{tikzcd}
 & \gamma(M) \\
F \arrow[ur, "S"] \arrow[r, "\eta_F"'] & \gamma(\alpha(F)) \arrow[u, dashed, "\gamma(\overline{S})"']
\end{tikzcd}$$

As a consequence, we have that there is no other machine and simulation that yields a better approximation of the transitive closure of a formula F than (α, η_F). This is summarized in the following:

Proposition 1. *Let F be a transition formula, let M be a machine, and let $S : F \to \gamma(M)$ be a simulation. Then $\eta_F \triangleright c\ell(\alpha(F)) \models S \triangleright c\ell(M)$.*

Proof. Let $\overline{S}$ be the unique simulation $\overline{S} : \alpha(F) \to M$ such that $S = \gamma(\overline{S}) \circ \eta_F$. Since $c\ell$ is a functor, we have a simulation $c\ell(\overline{S}) : c\ell(\alpha(F)) \to c\ell(M)$. It follows that

$$c\ell(\alpha(F)) \models c\ell(\overline{S}) \triangleright c\ell(M) = \overline{S} \triangleright c\ell(M) \ .$$

Conjugating by η_F yields

$$\begin{aligned} \eta_F \triangleright c\ell(\alpha(F)) &\models \eta_F \triangleright \left(\overline{S} \triangleright c\ell(M)\right) \\ &\models \left(\eta_F \circ \overline{S}\right) \triangleright c\ell(M) \\ &= S \triangleright c\ell(M) \ . \end{aligned}$$

Summing up, we have the following recipe for summarizing loops:

Let $\mathcal{S}$ be a class of simulation relations, $\mathbf{TF}_{\mathcal{S}}$ be a category of transition formulas, $\mathbf{M}_{\mathcal{S}}$ be a category of abstract machines, $\alpha : \mathbf{TF}_{\mathcal{S}} \to \mathbf{M}_{\mathcal{S}}$, $\gamma : \mathbf{M}_{\mathcal{S}} \to \mathbf{TF}_{\mathcal{S}}$, and $c\ell : \mathbf{M}_{\mathcal{S}} \to \mathbf{TF}_{\mathcal{S}}$ be functors, and let $\eta : 1_{\mathbf{TF}_{\mathcal{S}}} \Rightarrow \gamma \circ \alpha$ be a natural transformation such that
1. $\mathcal{R}[\![c\ell(F)]\!] = \mathcal{R}[\![\gamma(F)]\!]^*$
2. α is left adjoint to γ, with unit η.

Then the function $F^{\circledast} \triangleq \eta_F \triangleright c\ell(\alpha(F))$ is an iteration operator.

If we derive an iteration operator $(-)^{\circledast}$ by following this recipe, then it is an easy consequence of Proposition 1 that $(-)^{\circledast}$ is monotone: if $F \models G$, then $F^{\circledast} \models G^{\circledast}$. This property makes it easier to reason about the behavior of program analyses.

3.3 Compositional recurrence analysis

We will now present compositional recurrence analysis as a sequence of examples of this recipe.

Example 3.5 The main content of [17, §III, A, B, C] is an algorithm for finding an affine transformation in **1-LT** that simulates a transition formula. We can give a more fine-grained description of the algorithm by describing the sense in which it is best.

The input to the algorithm is a linear arithmetic transition formula F. The algorithm operates in two steps. The first is to compute a best abstraction of F as an affine transformation in **1-LT** with respect to simulations of the form $\boldsymbol{x}' = S\boldsymbol{x}$ where each row of S is a standard basis vector. The fact that the rows of S are required to be standard basis vectors is due to the fact that they correspond to variables that satisfy a recurrence relation. In the light of the perspective of abstract machines, we see an opportunity for improving the analysis by allowing S to be an arbitrary linear transformation—computing best abstractions in this setting is an easy extension of the existing algorithm, and yields a strictly more precise analysis.

Having fixed a **1-LT** affine transformation $f(\boldsymbol{x}) = A\boldsymbol{x}+\boldsymbol{b}$ (of some dimension, say n) and a simulation S with $S : F \Vdash f$, the second step of the algorithm is to compute a best abstraction of F as a **1-LT** affine transformation of the form

$$g(\boldsymbol{x}) = \begin{bmatrix} A & 0 \\ B & I \end{bmatrix} \boldsymbol{x} + \begin{bmatrix} \boldsymbol{b} \\ \boldsymbol{c} \end{bmatrix}$$

with respect to linear simulations of the form $\boldsymbol{x}'_1 = S\boldsymbol{x} \wedge \boldsymbol{x}'_2 \geq S'\boldsymbol{x}$, where $\boldsymbol{x}'_1$ denotes the vector of variables $x'_1 ... x'_n$ and $\boldsymbol{x}'_2$ denotes the vector of variables $x'_{n+1} ... x'_m$. That is, we allow the affine transformation f to be extended with additional dimensions ($n + 1$ through m), which act as upper bounds on linear terms over the variables of F (as illustrated in Example 3). This allows CRA to infer invariant polynomial inequations as well as equations. The fact that the lower right corner of the transformation matrix g is restricted to be the identity is a non-trivial restriction: new abstraction techniques are required to lift the assumption while retaining the property of being a best abstraction. $\lrcorner$

Example 3.6 Affine transformations can be used to capture the relationship between the pre-state and post-state of a loop body, but information about the guard of the loop (i.e., relationships between pre-state variables) is lost. This information can be recovered via a *pre-state formula*, which is a transition formula in which only the pre-state variables $x_1, x_2, ...$ appear [17, §III, D]. The concretization of a pre-state formula G is G itself, the closure is defined by

$$c\ell(G) = \exists k \in \mathbb{N}.(k = 0 \wedge \bigwedge_{x \in X} x' = x) \vee (k > 0 \wedge G) \ .$$

The abstraction function is $\alpha(F) = \exists \boldsymbol{x}'.F$, which is *best* with respect to identity simulations. Post-state formulas can be defined dually.

Given a formula F, we can combine the pre-state, post-state, and affine transformation abstractions of F, by separately computing the closure of each abstraction and then conjoining the results. Better still, we can take a kind of reduced product [14] of the abstractions by synchronizing on the existentially quantified iteration variable k. This combination yields the compositional recurrence analysis described in [17]. $\lrcorner$

Note that although compositional recurrence analysis computes (best) abstractions of *linear* formulas, the closure operator produces *polynomial* formulas. Before applying the abstraction functions, we first linearize the loop body formula [17, §IV]. The abstraction function of CRA is not best for polynomial arithmetic transition formulas (and no non-trivial abstraction function can be, since integer polynomial arithmetic is undecidable), but the fact that the abstraction function is best for linear arithmetic suggests that information is lost *only* because of incomplete reasoning about non-linear arithmetic.

Example 3.7 Affine maps capture non-linear behavior where non-linearity is a function of time, but some systems exhibit non-linear behavior even in a single step. *Solvable polynomial maps* are a class of abstract machines that can capture some such behavior while still being relatively easy to reason about.

Definition 4 ([35]). *A function $f : \mathbb{Q}^n \to \mathbb{Q}^n$ is a **solvable polynomial map** if there exists a partition of $\{1, ..., n\}$, $\boldsymbol{x}_1 \cup \cdots \cup \boldsymbol{x}_m$ with $\boldsymbol{x}_i \cap \boldsymbol{y}_j = \emptyset$ for $i \neq j$ such that for all $1 \leq i \leq m$ we have*

$$f_{\boldsymbol{y}_i}(\boldsymbol{x}) = A_i \boldsymbol{y}_i^t + p_i(\boldsymbol{y}_1, ..., \boldsymbol{y}_{i-1})$$

where $f_{\boldsymbol{x}_i}(\boldsymbol{x})$ denotes $f(\boldsymbol{x})$ projected onto the coordinates, $\boldsymbol{x}_i$, $A_i \in \mathbb{Q}^{|\boldsymbol{x}_i| \times |\boldsymbol{x}_i|}$ and $p_i \in \mathbb{Q}[\boldsymbol{x}_1, ..., \boldsymbol{x}_{i-1}]$.

The concretization of a solvable polynomial is $\gamma(f) \triangleq \boldsymbol{x}' = f(\boldsymbol{x})$. The closure $c\ell(f)$ of f is defined to be the reachability relation of f—[26] gives an algorithm for computing a closed form representation of the reachability relation of a solvable polynomial map in a logic involving polynomials, exponenentials, and also *operators* in the Berg's operational calculus [5], which can be treated as uninterpreted function symbols by an SMT solver.

The abstraction algorithm presented in [26] begins by computing a conjunction of polynomial equations and inequalities that are entailed by the formula. Since the logic is undecidable, we can make no guarantees about the quality of this approximation (however, it is best in the sense that, if the formula is expressed in linear arithmetic, then we compute the convex hull of the formula). A simulating solvable polynomial map is then extracted from this system of equations and inequalities in two steps, just as in [17]. The first step computes the best abstraction of a transition formula as a solvable polynomial map with respect to simulations of the form $\boldsymbol{x}' = S\boldsymbol{x}$. The abstraction is best under the assumption that the input transition formula is of the form $\bigwedge_{i=1}^{n} p_i(\boldsymbol{x}, \boldsymbol{x}') = 0$, where each p_i is a polynomial and such that

1. For every polynomial $p(\boldsymbol{x}, \boldsymbol{x}')$ such that $F \models p(\boldsymbol{x}, \boldsymbol{x}') = 0$, we have p in the ideal generated by $\{p_1, ..., p_n\}$.
2. F is total—for every $\mathbf{u}$ there exists some $\boldsymbol{v}$ such that $F(\boldsymbol{u}, \boldsymbol{v})$ holds.

Similarly to [17], we then extend this solvable polynomial map with additional dimensions to capture inequalities. However, the algorithm for computing inequalities makes use of polyhedral widening, so it need not be a best abstraction. ⌟

4 Control flow and Recursive Procedures

This section explains how the style of analysis in §2 can be extended to a more realistic program model that has unstructured control flow and recursive procedures. The foundation is the algebraic view of program analysis pioneered by Tarjan, who developed an efficient algorithm for computing solutions to intraprocedural program analysis problems [39, 40]. The extension to the interprocedural setting is based on [25], which exploits abstract machines to compute approximations of recursive procedures.

We begin by formulating a new program model on top of the simple programming language defined in §2. Let Proc denote a finite set of procedure names. Define a syntactic category of *instructions*:

$$x \in \mathsf{Var} \qquad e \in \mathsf{Expr} \qquad c \in \mathsf{Cond} \qquad p \in \mathsf{Proc}$$
$$\mathsf{Instr} ::= x \texttt{ := } t \mid \texttt{assume}(c) \mid \texttt{assert}(c) \mid \texttt{call } p$$

A *control flow graph* $G = (V, \Delta, en, ex)$ consists of a finite set of nodes V, a finite set of instruction-labeled edges $\Delta \subseteq V \times \mathsf{Instr} \times V$, a distinguished entry vertex en, and a distinguished exit vertex ex. A *program* $P = \{G_p\}_{p \in \mathsf{Proc}}$ consists of a collection of control flow graphs indexed by procedure names.

The link from the effective denotational semantics of §2 to this program model is through the medium of **path expressions**: regular expressions that represent paths through a program. For our purposes, we may define a path expression to be a regular expression over the alphabet of instructions:

$$E \in \mathsf{PathExp} ::= \ instr \in \mathsf{Instr} \mid E_1 + E_2 \mid E_1E_2 \mid E^* \mid 0 \mid 1$$

Suppose that we fix an iteration operator $(-)^{\circledast} : \mathbf{TF} \to \mathbf{TF}$ that over-approximates the transitive closure of a transition formula. Then given a path expression E and a *summary map* $S : \mathsf{Proc} \to \mathbf{TF}$ that maps each procedure to a transition formula, we can define a transition formula $\mathbf{TF}[\![E]\!](S)$ that over-approximates the paths in the path expression:

$$\begin{aligned}
\mathbf{TF}[\![\texttt{x := } e]\!](S) &\triangleq \texttt{x}' = e \wedge \bigwedge_{\texttt{y} \neq \texttt{x} \in \mathsf{Var}} \texttt{y}' = \texttt{y} \\
\mathbf{TF}[\![\texttt{assume}(c)]\!](S) &\triangleq c \wedge \bigwedge_{\texttt{x} \in \mathsf{Var}} \texttt{x}' = \texttt{x} \\
\mathbf{TF}[\![\texttt{assert}(c)]\!](S) &\triangleq \mathbf{TF}[\![\texttt{assume}(c)]\!](S) \\
\mathbf{TF}[\![\texttt{call } p]\!](S) &\triangleq S(p) \\
\mathbf{TF}[\![E_1 + E_2]\!](S) &= \mathbf{TF}[\![E_1]\!](S) \vee \mathbf{TF}[\![E_2]\!](S) \\
\mathbf{TF}[\![E_1E_2]\!](S) &= \mathbf{TF}[\![E_1]\!](S) \odot \mathbf{TF}[\![E_2]\!](S) \\
\mathbf{TF}[\![E^*]\!](S) &= \mathbf{TF}[\![E]\!](S)^{\circledast} \\
\mathbf{TF}[\![1]\!](S) &= \mathbf{TF}[\![\texttt{assume}(0 = 0)]\!](S) \\
\mathbf{TF}[\![0]\!](S) &= \mathit{false}
\end{aligned}$$

Tarjan gave an efficient algorithm for the *single-source path expression problem*: given a control flow graph $G_p = (V_p, \Delta_p, en_p, ex_p)$, compute for each vertex $v \in V$ a path expression $\mathbf{P}_{G_p}[en_p, v]$ representing the set of all paths from en_p to v in G_p. A summary for the procedure p may be computed by evaluating $\mathbf{TF}[\![\mathbf{P}_{G_p}[en_p, ex_p]]\!](S)$, or we prove that an assertion $(u, \texttt{assert}(c), v)$ never fails by checking that $\mathbf{TF}[\![\mathbf{P}_{G_p}[en_p, u]]\!](S) \wedge \neg(c[X \mapsto X'])$ is unsatisfiable. The naïve definition of $\mathbf{TF}[\![-]\!]$ given above may use exponentially many transition formula operations due to repeated sub-path expressions. By using memoization or path compression, only linearly many operations are needed [39].

4.1 Interprocedural analysis

Tarjan's algorithm assumes that we know how to compute a transition formula for each instruction in the programming language. For a language with procedure calls, this means that we require as input a summary map $S : \mathsf{Proc} \to \mathbf{TF}$. For programs without recursive procedures, we can use Tarjan's algorithm to compute the summary map: first place the procedures in reverse topological order $p_1, ..., p_n$ (so that if p_i calls p_j then $j < i$), and then compute

$$S_0 = \lambda p.\mathit{false}$$
$$S_i = S_{i-1}\{p_i \mapsto \mathbf{TF}[\![\mathbf{P}_{G_{p_i}}[en_{p_i}, ex_{p_i}]]\!](S_{i-1})\}$$

The summary map S_n maps each procedure to a transition formula that over-approximates its behavior.

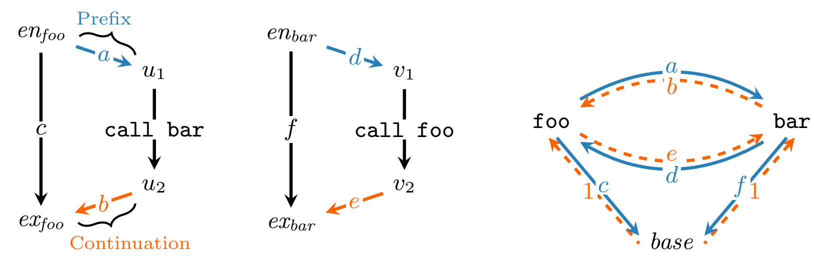
For programs with *recursive* procedures, however, this process does not work. For recursive procedures we can always fall back on iterative techniques for resolving fixed point equations [13] (as we did in [17]), but this is not a very satisfying solution: we have a methodology for designing powerful invariant generators for loops (§3), and we would like to be able use this same methodology to analyze recursion.

The first important development in this direction was the work of Reps et al. [34], which showed that Tarjan's algorithm could be used to compute summaries for programs with *linear* recursion (i.e., in each path through each procedure, there is at most call instruction). The intuition behind their approach is illustrated in Figure 1. Any path that contains a single function call, say a(`call bar`)b, can be thought of as a pair consisting of a *prefix* a—a path from entry to the call, and a *continuation* κ—a path from the call to exit. Call the pair consisting of a and b a *tensored path*, and write it as $a \otimes b$. We can construct a *call graph CG* where the vertices are the procedure `foo` and `bar` and there is an edge from `foo` to `bar` labeled with the tensored path $a \otimes b$ (corresponding to the path in `foo` that calls `bar`) and similarly an edge from `bar` to `foo` labeled $d \otimes e$. We also add a *base* vertex to the graph, and draw an edge from each procedure to *base* representing the path on which there is no recursive call. Tarjan's algorithm can be used to compute for each procedure a regular expression over an alphabet of tensored paths that represents the tensored paths from that procedure to *exit*.

$$\mathbf{P}_{CG}[\texttt{foo}, \mathit{base}] = ((a \otimes b)(d \otimes e))^* (c \otimes 1) + ((a \otimes b)(d \otimes e))^* (a \otimes b)(f \otimes 1)$$
$$\mathbf{P}_{CG}[\texttt{bar}, \mathit{base}] = ((a \otimes b)(d \otimes e))^* (c \otimes 1) + ((a \otimes b)(d \otimes e))^* (a \otimes b)(f \otimes 1)$$

These regular expressions represent the language of interprocedural paths through their respective procedures, where each tensored path (for instance, the path $(a \otimes b)(d \otimes e)(a \otimes b)(d \otimes e)(c \otimes 1)$ which belongs to $\mathbf{P}_{CG}[\texttt{foo}, \mathit{base}]$), can be understood as an interprocedural path by reading the *prefix* of each tensor left-to-right followed by the *continuation* of each tensor right-to-left (that is, the path *adadcebeb*).

We can use transition formulas to represent the behavior of a program along a path; we can also use them to represent the behavior of a program along a *tensored* path, by using twice as many variables: one set of variables for the prefix, and one set for the continuation. That is:



(a) Procedure `foo` (b) Procedure `bar` (c) Call graph CG

Fig. 1: A schematic recursive program with two procedures `foo` and `bar`, along with its call graph labeled with tensored paths. Instructions labeling non-call edges are abstracted away by letters.

Definition 5. *Given two n-transition formulas F and G, their* ***tensor product*** *$F \otimes G$ is defined to be the $(2n)$-transition formula*

$$F \otimes G \triangleq F \wedge (G[x_i \mapsto x'_{n+i}, x_i \mapsto x_{n+i}]_{i=1}^{n})$$

Observe that we have $(F_1 \otimes G_1) \odot (F_2 \otimes G_2) \equiv (F_1 \odot F_2) \otimes (G_2 \odot G_1)$, so that composition of transition relations respects the left-to-right prefix, right-to-left continuation interpretation of tensored paths.

Summaries for the procedures `foo` and `bar` may thus be obtained by recursion on the regular expression of tensored paths, using the tensor product of transition formulas to interpret tensored paths, and finally converting the tensored transition formula back into a transition formula using the following *detensor* operator, which connects the prefix and continuation into a transition formula representing an ordinary path:

$$D(T) \triangleq \left(\exists \boldsymbol{x}'. \left(F \wedge \bigwedge_{i=1}^{n} x'_i = x'_{i+n}\right)\right)[x_{n+i} \mapsto x'_i]_{i=1}^{n} \ .$$

That is, we have

$$S(\texttt{foo}) \triangleq D(\mathbf{TF}[\![\mathbf{P}_{CG}[\texttt{foo}, \mathit{base}]]\!])$$
$$S(\texttt{bar}) \triangleq D(\mathbf{TF}[\![\mathbf{P}_{CG}[\texttt{bar}, \mathit{base}]]\!])$$

(omitting the S argument to $\mathbf{TF}[\![-]\!]$ since call graph path expressions are free of `call`s).

Unfortunately, this idea does not extend to non-linear recursive procedures, so in the general case we must fall back on iterative methods for solving semantic equations. Naïve application of the iterative method requires designing an equivalence relation and widening operator for transition formulas. However, this is at odds with our goal of generating invariants in expressive logics, for which such operations are not readily available.

[25] gives an alternate approach, which again exploits abstract machines. The idea is that we can use widening and equivalence operators at the level of *abstract machines* rather than transition formulas. Abstract machines have simpler structure than general transition formulas and are more amenable to this kind of operation.

Example 4.8 In [17], the iteration operator extracts an affine transformation f and a linear simulation S. $S \triangleright \gamma(f)$ is a formula of a particular kind: a convex polyhedron. Widening operators for convex polyhedra are well known [15]. $\lrcorner$

Example 4.9 In [26], the iteration operator extracts a solvable polynomial map f and a linear simulation S. The formula $S \triangleright \gamma(f)$ is a conjunction of polynomial equations and inequations. Such formulas can be represented precisely by the *wedge* abstract domain, presented (along with its widening operator) in [26]. $\lrcorner$

The idea behind [25] is simple: each time we apply the iteration operator $\circledast$ to a transition formula F, we will compute an abstract machine that simulates F. Rather than using widening to ensure the convergence of the sequence of procedures summaries for each procedure, we use widening to ensure the convergence of the sequence of abstract machines for each loop. Soundness and termination of this approach relies on the property that every recursive call is contained inside some loop. Obviously, this need not be the case for the original program, but [25] gives an alternative algorithm to Tarjan's path expression algorithm that can be used to obtain *tensored* path expressions for each procedure that do satisfy this property.

5 Related work

5.1 Abstract machines with closure

This section surveys a selection of work that, seen through the lens of §3, computes closure operators for some class of abstract machines.

Linear machines (Discrete) linear dynamical systems are a well-studied class of machines, in which the state space is a vector space and the state evolves by applying a linear transformation—i.e., the transition formula of a linear dynamical system is of the form $\mathbf{x}' = A\mathbf{x}$. A formula representing the reachability relation of such a machine can be computed via symbolic matrix exponentiation: $c\ell(A) = \exists k \in \mathbb{N}.\mathbf{x}' = A^k\mathbf{x}$. A symbolic representation A^k can be expressed in terms of exponential-polynomials, where the base of each exponential term is drawn from the eigenvalues of A. However, since the eigenvalues of A may be complex, it is desirable to consider simpler closed forms.

The question of when the reachability relation of an affine dynamical system can be expressed in Presburger arithmetic was answered by Boigelot [8]. Boigelot gave a procedure for computing A^k under the assumption that the multiplicative monoid generated by A, $\{A^i : i \in \mathbb{N}\}$, is finite. Boigelot gives necessary and sufficient conditions for an iterated affine map to be definable in Presburger arithmetic, and also Presburger arithmetic extended with a single function V_r mapping each integer z to the greatest power of r that divides z. Boigelot also considers the case that the linear map is equipped with a polyhedral guard (which can restrict the number of times the linear map is iterated), in which case his conditions are necessary but not sufficient. Finkel and Leroux [18] extends further to guards defined in Presburger arithmetic.

Jeannet et al. developed a technique for *over-approximating* the behavior of linear dynamical systems, which is based on approximating the exponential of the real Jordan form of the transition matrix by an abstract domain of template polyhedron matrices [24].

An *affine program* consists of a finite graph where each edge is labeled by an affine transformation. A special case of interest for our purposes is with only one vertex: such an affine program corresponds to a transition formula of the form

$$\boldsymbol{x}' = A_1\boldsymbol{x} + \boldsymbol{b}_1 \vee \cdots \vee \boldsymbol{x}' = A_m\boldsymbol{x} + \boldsymbol{b}_m \tag{3}$$

Haase and Halfon gave a polytime procedure for computing a Presburger formula defining the reachability relation of affine programs for which each transition matrix is diagonal and has either 0 or 1 on the diagonal (i.e., an integer vector addition system with states and resets) [20]. Müller-Olm and Seidl give a procedure for computing the smallest affine space that contains the reachability relation of affine programs [32]. Hrushovski et al. [21] gives a procedure to compute the smallest algebraic variety that contains the reachability relation.

Ultimately periodic relations The transitive closure of difference-bound relations [12, 11] and octagon relations [9] has been shown to be definable in Presburger arithmetic, and computable in polytime [27]. The theory of ultimately periodic relations unifies work on linear systems and difference-bound/octagon relations [10].

Polynomial machines A **solvable polynomial machine** is a dynamical system with a transition formula of the form

$$\boldsymbol{x}' = p_1(\boldsymbol{x}) \vee \cdots \vee \boldsymbol{x}' = p_m(\boldsymbol{x}) \tag{4}$$

where each p_i is a solvable polynomial map. Rodríguez-Carbonell and Kapur [35] showed how to compute an algebraic variety that contains the reachability relation of a solvable polynomial machine with a real spectrum. Kovács improves upon this result, giving an algorithm for computing the smallest algebraic variety that contains the reachability relation of a solvable polynomial machine (without spectral assumptions), and further extends the technique to a broader class of machines with non-polynomial assignments [28]. The class of machines is extended even further in subsequent work by Humenberger et al. [22, 23].

5.2 Symbolic abstraction and abstract machines

Approximating programs by finite state machines using predicate abstraction is a classical technique in software model checking [19, 4]. Kroening et al. [29] and

Biallas et al. [6] present techniques for approximating the transitive closure of loops using predicate abstraction.

Sinn et al. have considered the problem of computing approximations of programs using vector addition systems [37] and difference-bound constraints [38] in the context of resource bound analysis. The technique is based on guessing a set of norms (integer-valued functions of the program state), which amounts to finding a linear simulation.

Recurrence analysis Recurrence analysis is a family of program analysis techniques initiated by Wegbreit, which approximate the behavior of loops by extracting recurrence relations from the program and computing their closed forms [43]. It is closely related to the approach presented in this paper, with recurrence relations serving an analogous role to abstract machines. Recurrence analysis is a particularly prevalent technique in resource bound analysis, where the ability to compute non-linear expressions representing resource usage (e.g., time complexity) is crucial [16, 3, 1, 7].

Symbolic Abstraction There has been a body of work on computing best approximations of a logical formulas within abstract domains. For a thorough overview of symbolic abstraction in program analysis, see [41, 33]. Here we highlight a few instances in which symbolic abstraction yields a complete instance of the recipe from § 3:

- *Difference bound / octagonal relations*: the best abstraction of a transition formula as a difference bound or octagonal relation with respect to identity simulations can be computed using optimization modulo theories [36, 30]. Transitive closure can be computed using the methods of [12, 11, 9, 27].
- *Lossy sums*: the best abstraction of a transition formula in the form $\mathbf{x}' \leq \mathbf{x} + \mathbf{b}$ with respect to linear simulations can be computed using symbolic abstraction in the domain of convex polyhedra [17, 42], and the method of Ancourt et al. for finding linear recurrence inequations from polyhedra [2].

6 Conclusion

Abstract machines give a mechanism for developing compositional program analyses that generate precise numerical invariants. There are two categories of work that are directly related to advancing this paradigm:

- Inventing new classes of abstract machines that admit effective closure operators, and which model interesting phenomena in dynamical systems.
- Developing techniques for computing best abstractions of transition formulas by abstract machines. E.g., there are a number of models (some of which referenced in §5) for which the best abstraction problem has not yet been investigated.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: SAS. pp. 221–237 (2008)
2. Ancourt, C., Coelho, F., Irigoin, F.: A modular static analysis approach to affine loop invariants detection. Electron. Notes Theor. Comput. Sci. **267**(1), 3–16 (Oct 2010)
3. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. CoRR **abs/cs/0512056** (2005)
4. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. In: PLDI. pp. 203–213 (2001)
5. Berg, L.: Introduction to the Operational Calculus. North-Holland Publishing Co., Amsterdam (1967)
6. Biallas, S., Brauer, J., King, A., Kowalewski, S.: Loop leaping with closures. In: SAS. pp. 214–230 (2012)
7. Blanc, R., Henzinger, T.A., Hottelier, T., Kovács, L.: ABC: Algebraic bound computation for loops. In: Int. Conf. on Logic for Programming, Art. Intell., and Reasoning. pp. 103–118 (2010)
8. Boigelot, B.: On iterating linear transformations over recognizable sets of integers. Theor. Comp. Sci. **309**(1), 413–468 (2003)
9. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: TACAS. pp. 337–351 (2009)
10. Bozga, M., Iosif, R., Konečný, F.: Fast acceleration of ultimately periodic relations. In: CAV. pp. 227–242 (2010)
11. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. In: Automata, Languages and Programming. pp. 577–588 (2006)
12. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: CAV. pp. 268–279 (1998)
13. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
14. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282 (1979)
15. Cousot, P., Halbwachs, N.: Automatic discovery of linear constraints among variables of a program. In: POPL (1978)
16. Debray, S.K., Lin, N., Hermenegildo, M.V.: Task granularity analysis in logic programs. In: PLDI. pp. 174–188 (1990)
17. Farzan, A., Kincaid, Z.: Compositional recurrence analysis. In: FMCAD (2015)
18. Finkel, A., Leroux, J.: How to compose Presburger-accelerations: Applications to broadcast protocols. In: FST TCS. pp. 145–156 (2002)
19. Graf, S., Saidi, H.: Construction of abstract state graphs with pvs. In: CAV. pp. 72–83 (1997)
20. Haase, C., Halfon, S.: Integer vector addition systems with states. In: Reachability Problems. pp. 112–124 (2014)
21. Hrushovski, E., Ouaknine, J., Pouly, A., Worrell, J.: Polynomial invariants for affine programs. In: Logic in Computer Science. pp. 530–539 (2018)
22. Humenberger, A., Jaroschek, M., Kovács, L.: Automated generation of non-linear loop invariants utilizing hypergeometric sequences. In: ISSAC (2017)
23. Humenberger, A., Jaroschek, M., Kovács, L.: Invariant generation for multi-path loops with polynomial assignments. In: VMCAI. pp. 226–246 (2018)

24. Jeannet, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. In: POPL. pp. 529–540 (2014)
25. Kincaid, Z., Breck, J., Forouhi Boroujeni, A., Reps, T.: Compositional recurrence analysis revisited. In: PLDI (2017)
26. Kincaid, Z., Cyphert, J., Breck, J., Reps, T.: Non-linear reasoning for invariant synthesis. PACMPL **2(POPL)**, 54:1–54:33 (2018)
27. Konečný, F.: Ptime computation of transitive closures of octagonal relations. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 645–661 (2016)
28. Kovács, L.: Reasoning algebraically about P-solvable loops. In: TACAS (2008)
29. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.: Loop summarization using abstract transformers. In: ATVA. pp. 111–125 (2008)
30. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with smt solvers. In: POPL. pp. 607–618 (2014)
31. Milner, R.: Communication and Concurrency. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
32. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL (2004)
33. Reps, T., Thakur, A.: Automating abstract interpretation. In: VMCAI (2016)
34. Reps, T., Turetsky, E., Prabhu, P.: Newtonian program analysis via tensor product. In: POPL (2016)
35. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial loop invariants: Algebraic foundations. In: ISSAC. pp. 266–273 (2004)
36. Sebastiani, R., Tomasi, S.: Optimization in smt with $\mathcal{LA}(\mathbb{Q})$ cost functions. In: IJCAR. pp. 484–498 (2012)
37. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: CAV. pp. 745–761 (2014)
38. Sinn, M., Zuleger, F., Veith, H.: Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In: FMCAD. pp. 144–151 (2015)
39. Tarjan, R.E.: Fast algorithms for solving path problems. J. ACM **28**(3), 594–614 (Jul 1981)
40. Tarjan, R.E.: A unified approach to path problems. J. ACM **28**(3), 577–593 (Jul 1981)
41. Thakur, A.: Symbolic Abstraction: Algorithms and Applications. Ph.D. thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI (Aug 2014), tech. Rep. 1812
42. Thakur, A., Reps, T.: A method for symbolic computation of abstract operations. In: CAV (2012)
43. Wegbreit, B.: Mechanical program analysis. Commun. ACM **18**(9), 528–539 (Sep 1975)