Compositional Recurrence Analysis

Azadeh Farzan
University of Toronto
azadeh@cs.toronto.edu

Zachary Kincaid
University of Toronto
zkincaid@cs.toronto.edu

***Abstract*—This paper presents a new method for automatically generating numerical invariants for imperative programs. The procedure computes a transition formula which over-approximates the behaviour of a given input program. It is compositional in the sense that it operates by decomposing the program into parts, computing a transition formula for each part, and then composing them. Transition formulas for loops are computed by extracting recurrence relations from a transition formula for the loop body and then computing their closed forms. Experimentation demonstrates that this method is competitive with leading verification techniques based on abstraction refinement.**

I. Introduction

Compositional program analyses operate by decomposing a program into parts, computing an abstract meaning of each part, and then composing the meanings. Compositional analyses have a number of desirable properties, including scalability, parallelizability, and applicability to incomplete programs. However, compositionality comes at a price: since each program fragment is analyzed independently of its context, the analysis cannot benefit from contextual information. This paper presents a compositional analysis which, despite loss of contextual information, is capable of generating precise numerical invariants.

```
while (⋆):
  x := x + 1
  y := y − 2
```

Compositional recurrence analysis, the technique proposed in this paper, aims to compute a *transition formula* that over-approximates the behaviour of a given program. As with any invariant generation technique, the crucial question is how to approximate the behaviour of loops. The basic idea can be illustrated with the example loop to the right, which loops for a non-deterministic number of iterations, adding 1 to $\mathtt{x}$ and subtracting 2 from $\mathtt{y}$ at each iteration. The body of this loop can be described by a system of recurrence equations:

$$\mathtt{x}^{(k)} = \mathtt{x}^{(k-1)} + 1$$
$$\mathtt{y}^{(k)} = \mathtt{y}^{(k-1)} - 2$$

(where $\mathtt{x}^{(k)}$ represents the value of $\mathtt{x}$ on the k^{th} iteration of the loop). A transition formula for the loop can be computed by taking the closed form of this system. Using $\mathtt{x}$ and $\mathtt{y}$ to denote the values of those variables before executing the loop and $\mathtt{x'}$ and $\mathtt{y'}$ to their values after the loop, a transition formula that *precisely* describes this loop is:

$$\exists k \in \mathbb{N}. \mathtt{x'} = \mathtt{x} + k \wedge \mathtt{y'} = \mathtt{y} - 2k$$

The idea of using recurrences to abstract loops is classical, and there exist powerful techniques for solving very general classes of recurrence equations. However, two barriers stand in the way of applying recurrence analysis to real programs. First, loop bodies may have arbitrary control flow. Extracting recurrence equations from a straight-line loop body like the one above is straightforward, but what if the loop body contains branching, or nested loops, or even unstructured control flow? Second, the behaviour of a loop may not be describable as a system of recurrence equations (for example, consider a loop where $\mathtt{x}$ is non-deterministically incremented by either 1 or 2). How can recurrence analysis be used to over-approximate loops whose behaviour is not determined by system of recurrence equations?

Our approach exploits compositionality to overcome these barriers. The assumption of compositionality demands that a transition formula for a loop is computed from a transition formula for its body. This makes the control flow of the loop body irrelevant: whether it is a sequence of assignments or contains branching or nested loops – its transition formula is just a formula. While this circumvents the first barrier to practical recurrence analysis, it also presents a new challenge: how can we extract a system of recurrence equations from a formula representing a loop body? Our solution is to use a Satisfiablity Modulo Theories (SMT) solver to extract recurrence equations which are semantically implied by a loop body formula. In fact, our method goes beyond systems of recurrence equations over program variables: it extracts a system of recurrence *inequations* over *linear terms*. This allows compositional recurrence analysis to compute accurate over-approximations of loops which cannot be described as a system of recurrence equations, thereby overcoming the second barrier to practical recurrence analysis.

The rest of the paper is organized as follows. In the next section, we give a high-level overview of our algorithm. In Section III, we describe our strategy for over-approximating the behaviour of loop whose body is expressed as a linear arithmetic formula; Section IV describes a method for over-approximating a non-linear arithmetic formula by a linear arithmetic formula, so that our loop approximation procedure may be applied to any arithmetic formula. In Section V, we demonstrate experimentally that compositional recurrence analysis compares favourably with leading (non-compositional) verification techniques. Section VI compares with related work, and Section VII concludes.

II. Overview

We will adopt a simple intra-procedural program model in which a program is represented by a control flow automaton (CFA) where edges are labeled by program statements. Figure 1(b) depicts such a CFA for a program that computes the quotient and remainder of division of a variable x by a variable y. Naturally, compositional recurrence analysis can be extended to a program model with procedures by using the analysis to compute procedure summaries [31], but for the sake of simplicity we will not discuss this extension formally.

Compositional recurrence analysis (CRA) is presented in the algebraic abstract interpretation framework described in [10]. In [10], a program analysis is defined by an *interpretation*, which consists of a *semantic algebra* and a *semantic function*. A semantic algebra consists of a *universe* which defines the space of possible program meanings, along with *sequencing* ($\odot$), *choice* ($\oplus$), and *iteration* ($\circledast$) operators, which define how to compose program meanings. A semantic function is a mapping from control flow edges to elements of the universe which defines the meaning of each control flow edge.

Path expression algorithms [7], [29], [32] form the algorithmic foundation of the algebraic framework. A *path expression* is a regular expression over an alphabet of control flow edges which recognizes the set of paths through a control flow automaton. Intuitively, path expression algorithms operate by computing a path expression to a control point of interest and then interpreting that path expression in the semantic algebra defining the analysis. The exponential blow-up of computing a regular expression from a control flow automaton is avoided by sharing sub-expressions and evaluating the path expression bottom-up. Path expression algorithms can share work over multiple queries to avoid duplicate work if there are multiple points of interest (e.g., if there is more than one assertion to be verified).

More concretely, suppose that we wish to prove that the assertion on the edge from v_8 to v^{exit} always succeeds using CRA. First, we compute a path expression for vertex v_8 (Figure 1(c)). This regular expression recognizes the set of paths from v^{entry} to v_8. Second, we evaluate this path expression in the semantic algebra of CRA by recursing on the regular expression, interpreting each edge using the semantic function and each regular expression operator by its counterpart in the semantic algebra that defines CRA. The result is a transition formula which approximates the executions which end at v_8. Third, we ask an SMT solver whether the transition formula implies that the assertion holds in the post-state. If the implication holds, then we may conclude that the assertion is safe. If not, then the verification is inconclusive: either the assertion fails in some execution, or the assertion is safe but the transition formula computed by CRA is not strong enough to prove it (the analysis cannot distinguish between these cases).

Keeping this framework in mind, we proceed to describe the interpretation which defines compositional recurrence analysis.

CRA Universe. The semantic universe of CRA (i.e., the space of program meanings) is the set of arithmetic *transition formulas*. Letting Var denote the set of program variables and Var' the set of "primed" copies of program variables, a transition formula is an arithmetic formula with free variables in $\mathsf{Var} \cup \mathsf{Var}'$.

Transition formulas may contain existential quantifiers and non-linear arithmetic. For readability, we will often simplify formulas by eliminating quantifiers in the remainder of the paper. However, CRA does not require quantifier elimination (and indeed, there is no quantifier elimination procedure for the class of formulas we consider, since we allow non-linear integer arithmetic).

CRA Semantic Function. The semantic function $[\![\cdot]\!]$ maps each edge of a control flow automaton to its interpretation as a transition formula. For example (again considering Figure 1), we have

$$[\![\langle v^{\mathsf{entry}}, v_1\rangle]\!] \triangleq \boxed{r' = x \wedge id(\{q,t,x,y\})}$$
$$[\![\langle v_2, v_3\rangle]\!] \triangleq \boxed{r \geq y \wedge id(\{q,r,t,x,y\})}$$

where for $X \subseteq \mathsf{Var}$, we define $id(X) \triangleq \boxed{\bigwedge_{x\in X} x' = x}$; we use id to factor out equalities from the formulas and make them more legible. Boxes around formulas have no meaning, and are used to make it easier to distinguish between equalities in formulas and the meta-language.

CRA Operators. The sequencing and choice operators of CRA are defined as follows:

$$\varphi \odot \psi \triangleq \boxed{\exists x''.\varphi[x''/x'] \wedge \psi[x''/x]} \qquad \text{Sequencing}$$
$$\varphi \oplus \psi \triangleq \boxed{\varphi \vee \psi} \qquad \text{Choice}$$

(where $\varphi[x''/x']$ denotes the formula obtained from φ by replacing each primed variable x' by its double-primed counterpart x'', and $\psi[x''/x]$ similarly replaces unprimed variables in ψ with double-primed variables).

The semantic function, sequencing, and choice operators are sufficient to analyze loop-free code. For example, CRA computes a transition formula for the body of the inner loop of Figure 1 as follows:

$$[\![\langle v_4,v_5\rangle \cdot \langle v_5,v_6\rangle]\!] = [\![\langle v_4,v_5\rangle]\!] \odot [\![\langle v_5,v_6\rangle]\!]$$
$$\equiv \boxed{t \neq 0 \wedge r' = r-1 \wedge id(\{q,t,x,y\})}$$
$$[\![\langle v_4,v_5\rangle \cdot \langle v_5,v_6\rangle \cdot \langle v_6,v_4\rangle]\!] = [\![\langle v_4,v_5\rangle \cdot \langle v_5,v_6\rangle]\!] \odot [\![\langle v_6,v_4\rangle]\!]$$
$$\equiv \boxed{t \neq 0 \wedge r' = r-1 \wedge t' = t-1 \wedge id(\{q,x,y\})}$$

Having defined the semantic function, semantic universe, and sequencing and choice operators for CRA, it remains only to define its iteration operator. The formal definition of the iteration operator appears in the next section; in this section, we will illustrate the iteration operator on the running example.

Let $\varphi_{\mathsf{inner}} \triangleq [\![\langle v_4,v_5\rangle \cdot \langle v_5,v_6\rangle \cdot \langle v_6,v_4\rangle]\!]$ be the formula above, which represents the body of the inner loop. The meaning of the inner loop is computed by applying the iteration operator to φ_{inner}.

CRA's iteration operator begins by extracting the recurrence equations shown to the right (note

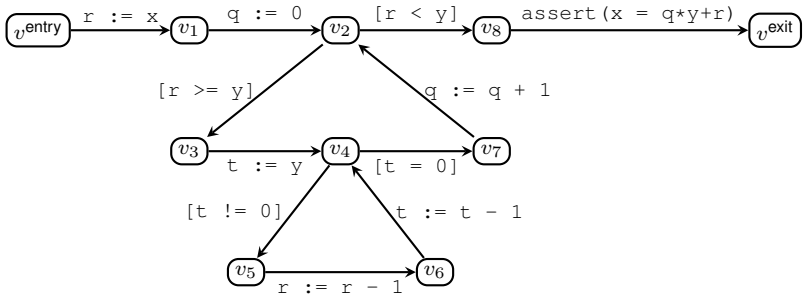
Recurrence	Closed form
$r' = r - 1$	$r^{(k)} = r^{(0)} - k$
$t' = t - 1$	$t^{(k)} = t^{(0)} - k$

```
r := x  // remainder
q := 0  // quotient
while(r >= y):
   // subtract y from r
   t := y
   while(t != 0)
      r := r - 1
      t := t - 1

   q := q + 1

assert(x = q*y + r)
```

(a) Program text



(b) Control Flow Automaton for (a)

$$\langle v^{\text{entry}}, v_1\rangle \cdot \langle v_1, v_2\rangle \cdot \Big(\langle v_2, v_3\rangle \cdot \langle v_3, v_4\rangle \cdot \overbrace{\big(\langle v_4, v_5\rangle \cdot \langle v_5, v_6\rangle \cdot \langle v_6, v_4\rangle\big)^*}^{\text{Inner loop}} \cdot \langle v_4, v_7\rangle \cdot \langle v_7, v_2\rangle\Big)^* \cdot \langle v_2, v_8\rangle$$

(c) Path expression to v_8

Fig. 1. An integer division program, computing a quotient and remainder. A statement $[\psi]$ denotes an *assumption* which blocks if ψ does not hold.

that this table omits "uninteresting" recurrences, such as $q' = q + 0$, which indicate that a variable does not change in a loop). It then computes closed forms for these recurrences, also shown to the right (where $x^{(k)}$ denotes the value that the variable x takes on the k^{th} iteration of the loop). These closed forms are used to abstract the loop as follows:

$$\varphi^{\circledast}_{\text{inner}} = \boxed{\exists k.k \geq 0 \wedge r' = r - k \wedge t' = t - k \wedge id(\{q, x, y\})}$$
$$\equiv \boxed{r' = r + t' - t \wedge t' \leq t \wedge id(\{q, x, y\})}$$

The path expression algorithm uses the formula $\varphi^{\circledast}_{\text{inner}}$ for the inner loop to compute a transition formula representing the body of the outer loop as follows:

$$\varphi_{\text{outer}} = [\![\langle v_2, v_3\rangle]\!] \odot [\![\langle v_3, v_4\rangle]\!] \odot \varphi^{\circledast}_{\text{inner}} \odot [\![\langle v_4, v_7\rangle]\!] \odot [\![\langle v_7, v_2\rangle]\!]$$
$$\equiv \boxed{q' = q+1 \wedge r' = r+t'-y \wedge t' = 0 \wedge r \geq y \wedge id(\{x, y\})}$$

Recurrence	Closed form
$q' = q + 1$	$q^{(k)} = q^{(0)} + k$
$r' = r - y$	$r^{(k)} = r^{(0)} - y^{(0)}k$

We then apply the iteration operator to φ_{outer} to compute a transition formula for the outer loop. The recurrences found for the outer loop and their closed forms are shown to the right (again, omitting "uninteresting" recurrences). Note that our algorithm extracts these recurrences from φ_{outer} using only semantic operations: the fact that φ_{outer} is an abstraction of a looping computation is completely transparent to the analysis. Using the closed forms of the recurrences to the right, we compute the following transition formula for the outer loop:

$$\varphi^{\circledast}_{\text{outer}} = \boxed{\exists k.k \geq 0 \wedge q' = q + k \wedge r' = r - ky \wedge id(\{x, y\})}$$
$$\equiv \boxed{q' \geq q \wedge r' = r - (q' - q)y \wedge id(\{x, y\})}$$

Finally, we compute a transition formula that approximates all executions which end at v_8 as follows:

$$\varphi_P = [\![\langle v^{\text{entry}}, v_1\rangle \cdot \langle v_1, v_2\rangle]\!] \odot \varphi^{\circledast}_{\text{outer}} \odot [\![\langle v_2, v_8\rangle]\!]$$
$$\equiv \boxed{q' \geq 0 \wedge r' = x - q'y \wedge r \leq y \wedge id(\{x, y\})}$$

This formula is strong enough to imply that $x' = q'y' + r'$. Thus CRA verifies that the assertion holds at v_8.

III. CRA Iteration Operator

In this section, we describe the iteration operator of compositional recurrence analysis. Suppose that we have a formula φ_{body} which approximates the behaviour of the body of a loop. The goal of the iteration operator is to compute a formula $\varphi^{\circledast}_{\text{body}}$ which represents the effect of zero or more executions of the loop body. CRA's iteration operator works by extracting recurrence relations from the formula φ_{body} and then computing closed forms for these relations. We present the iteration operator in three stages, based on the types of recurrence relations being considered: *simple recurrence equations*, *stratified recurrence equations*, and *linear recurrence (in)equations*. Simple and stratified recurrences are classical types of recurrence equations. Linear recurrence (in)equations generalize classical recurrence equations by considering inequalities over linear terms (rather than equalities over variables).

In the remainder of this section, fix a formula φ_{body} representing the body of a loop. We assume that φ_{body} is expressed in linear (rational and integer) arithmetic (our strategy for linearizing non-linear arithmetic is described in Section IV). Additionally, we will assume that φ_{body} is satisfiable; if φ_{body} is unsatisfiable, then we may take $\varphi^{\circledast}_{\text{body}}$ to be $\boxed{\bigwedge_{x \in \text{Var}} x' = x}$, which represents zero iterations of the loop.

A. Simple recurrence equations

We start by defining simple recurrences and induction variables.

Definition 1. *A* simple recurrence *for a formula φ_{body} is an equation of the form $x' = x + c$ (for a constant c) such that $\varphi_{\text{body}} \models x' = x + c$. If $x' = x + c$ is a simple recurrence for φ_{body}, we say that x* satisfies *the recurrence $x' = x + c$, and if there is some c such that x satisfies the recurrence $x' = x + c$,*

we say that x is an induction variable.

The set of all simple recurrences which are satisfied by a transition formula φ_{body} can be detected as follows. First, query an SMT solver for a model m of φ_{body}. Then, for each variable x, ask an SMT solver if φ_{body} implies the equation $x' = x + [\![x' - x]\!]^m$ (where $[\![x' - x]\!]^m$ denotes the interpretation of the term $x' - x$ in the model m). This implication holds iff x is an induction variable.

If x is an induction variable that satisfies the recurrence $x' = x + c$, then the closed form for x is $x^{(k)} = x^{(0)} + kc$ (writing $x^{(k)}$ for the value that x obtains on the k^{th} iteration of the loop).

B. Stratified recurrences equations

Consider the loop shown to the right. One can see that $\mathtt{x}$ satisfies a simple recurrence equation $\mathtt{x}' = \mathtt{x}+1$, and that $\mathtt{y}$ satisfies a (non-simple) recurrence equation $\mathtt{y}' = \mathtt{y} + \mathtt{x} + 1$. A closed form for $\mathtt{y}$'s recurrence is $\mathtt{y}^{(k)} = \mathtt{y}^{(0)} + \sum_{i=0}^{k-1}(\mathtt{x}^{(i)}+1)$. Since $\mathtt{x}$ is an induction variable, we have a closed form for $\mathtt{x}$ ($\mathtt{x}^{(i)} = \mathtt{x}^{(0)}+i$), which we may use to simplify $\mathtt{y}$'s recurrence:

```
while (x ≤ 10):
  x := x + 1
  y := y + x
  z := 2 * x
```

$$\begin{aligned}\mathtt{y}^{(k)} &= \mathtt{y}^{(0)} + \sum_{i=0}^{k-1}(\mathtt{x}^{(0)} + i + 1)\\ &= \mathtt{y}^{(0)} + k\mathtt{x}^{(0)} + k + \sum_{i=0}^{k-1} i\\ &= \mathtt{y}^{(0)} + k\mathtt{x}^{(0)} + \frac{k(k+1)}{2}.\end{aligned}$$

Stratified recurrence equations generalize this idea: starting from simple recurrence equations, solve more and more complex recurrences using the closed forms for simpler ones. Stratified recurrence equations are formalized as follows:

Definition 2. *The stratified recurrence equations (and stratified induction variables) of a formula φ_{body} are defined recursively as:*

- *A simple recurrence equation which is satisfied by φ_{body} is a stratified recurrence equation of φ_{body} (and a simple induction variable is a stratified induction variable) at stratum 0.*
- *Let $\boldsymbol{y}$ denote a vector of the stratified induction variables of strata $\leq N$. A recurrence of the form $x' = x + \boldsymbol{cy} + d$ (where $\boldsymbol{c}$ is a rational vector and d is a rational) is a stratified recurrence at stratum $N+1$ (and if x satisfies such a recurrence, it is a stratified induction variable at stratum $N+1$).*

We now present a method of generating all stratified induction variables from loop body formula. In order to reduce the number of SMT queries made, our method begins by constructing an intermediate object from the loop body formula, from which recurrence equations may be easily extracted. This intermediate object is the affine hull *aff*$(\varphi_{\mathsf{body}})$ of φ_{body}. The *affine hull aff*$(\varphi_{\mathsf{body}})$ of a formula φ_{body} is the smallest affine set which contains φ_{body}, represented as (the set of solutions to) a system of equations $A\boldsymbol{x} = \boldsymbol{b}$,

Algorithm 1: Affine hull.

Input : Satisfiable formula φ_{body}
Output: Affine hull of φ_{body}
$H \leftarrow \bot$; $\psi \leftarrow \varphi_{\mathsf{body}}$;
while *there exists a model m of ψ* **do**
  $H' \leftarrow \bigwedge\{x = [\![x]\!]^m : x \in \mathsf{Var} \cup \mathsf{Var}'\}$;
  $H \leftarrow H \sqcup^{=} H'$; /*Affine equality join*/
  $\psi \leftarrow \psi \wedge \neg H$;
end
return H

where $\boldsymbol{x} = \left[x_1, \cdots, x_n, x'_1, \cdots, x'_n\right]$ is a vector of all variables in $\mathsf{Var} \cup \mathsf{Var}'$. Logically, *aff*$(\varphi_{\mathsf{body}})$ is a system of equations such that (1) $\varphi_{\mathsf{body}} \models$ *aff*$(\varphi_{\mathsf{body}})$, and (2) every affine equation over $\mathsf{Var} \cup \mathsf{Var}'$ which is implied by φ_{body} is also implied by *aff*$(\varphi_{\mathsf{body}})$. The affine hull of a formula may be computed using Algorithm 1 (a specialization of the algorithm $\widehat{\alpha}$ in [26] to the abstract domain of affine equalities). For example, one representation of the affine hull of the example loop given at the beginning of this section is:

$$\begin{bmatrix}-1 & 0 & 0 & 1 & 0 & 0\\ 1 & 1 & 0 & 0 & -1 & 0\\ 0 & 0 & 0 & 2 & 0 & -1\end{bmatrix}\begin{bmatrix}\mathtt{x}\\ \mathtt{y}\\ \mathtt{z}\\ \mathtt{x}'\\ \mathtt{y}'\\ \mathtt{z}'\end{bmatrix} = \begin{bmatrix}1\\ -1\\ 0\end{bmatrix}$$

Our strategy for generating stratified recurrence equations from *aff*$(\varphi_{\mathsf{body}})$ is based on the following lemma. Combined with property (2) of *aff*$(\varphi_{\mathsf{body}})$ above, this lemma implies that any affine equation implied by φ_{body} can be expressed as a linear combination of the equations in *aff*$(\varphi_{\mathsf{body}})$. Thus, after computing the affine hull of φ_{body}, determining whether a given variable satisfies a stratified recurrence is simply a matter of solving a system of linear equations (e.g., using Gaussian elimination).

Lemma 3 ([30], Corollary 3.1d)**.** *Let A be a matrix, $\boldsymbol{b}$ be a column vector, $\boldsymbol{c}$ be a row vector, and d be a constant. Assume that the system $A\boldsymbol{x} = b$ has a solution. Then $A\boldsymbol{x} = \boldsymbol{b}$ implies $\boldsymbol{cx} = d$ iff there is a row vector $\boldsymbol{\lambda}$ such that $\boldsymbol{\lambda}A = \boldsymbol{c}$ and $\boldsymbol{\lambda b} = d$.*

An algorithm for finding all stratified induction variables of φ_{body} is as follows. Let us write *aff*$(\varphi_{\mathsf{body}})$ as $A\boldsymbol{x} = \boldsymbol{b}$. The algorithm operates by induction on strata. In the base case, we compute all simple induction variables using the method of the previous section. For the induction step, we suppose that we have detected all induction variables of strata $< N$. Then for each variable x_i which is *not* an induction variable of stratum $< N$, we ask if there exists $\boldsymbol{\lambda}$, $\boldsymbol{c}$, and d such that:

- $\boldsymbol{\lambda}A = \boldsymbol{c}$ and $\boldsymbol{\lambda b} = d$ (i.e, $\boldsymbol{cx} = d$ is implied by *aff*$(\varphi_{\mathsf{body}})$ and thus by φ_{body}).
- $c_i = 1$ and $c_{i+n} = -1$ (the coefficients of x_i and x'_i are 1 and -1, respectively).
- For all j such that $j \neq i + n$ and $n \leq j \leq 2n$, we have $c_j = 0$ (except for x'_i, all coefficients of primed variables are 0).
- For all $j \neq i$ such that x_j is *not* an induction variable of stratum $< N$, we have $c_j = 0$ (except for x_i and

induction variables of strata $< N$, all coefficients for unprimed variables are 0).

This system of linear equations has a solution if and only if x_i is an induction variable of stratum N. The algorithm terminates when it has computed a recurrence equation for every variable, or when it fails to detect any induction variables at some stratum.

Next, we give a procedure computing closed forms of stratified induction variables. Again, this procedure operates by induction on strata. For the base case, we compute closed forms for simple induction variables, as in the previous section. For the induction step, we make use of the induction hypothesis that *the closed form for a stratified induction variable of stratum N is of the form*

$$x^{(k)} = p_0(k) + p_1(k)y_1^{(0)} + \cdots + p_n(k)y_n^{(0)}$$

where each y_i is a stratified induction variable of stratum $< N$ and each $p_i(k) \in \mathbb{Q}[k]$ is a polynomial of one variable with rational coefficients.

Suppose that we have a recurrence equation at stratum N: $x' = x + c_1y_1 + \cdots + c_ny_n + b$ (all $y_1, ..., y_n$ are of strata $< N$). Then we may write

$$x^{(k)} = x^{(0)} + \sum_{i=0}^{k-1} \left(c_1y_1^{(i)} + \cdots + c_ny_n^{(i)} + b\right).$$

By our induction hypothesis, each $y_j^{(i)}$ can be written as a linear term with coefficients from $\mathbb{Q}[k]$. It follows that there exists $p_0, ..., p_n \in \mathbb{Q}[k]$ so that

$$c_1y_1^{(i)} + \cdots + c_ny_n^{(i)} + b = p_0(i) + p_1(i)y_1^{(0)} + \cdots + p_n(i)y_n^{(0)}$$

Thus we have

$$\begin{aligned} x^{(k)} &= x^{(0)} + \sum_{i=0}^{k-1} p_0(i) + p_1(i)y_1^{(0)} + \cdots + p_n(i)y_n^{(0)} \\ &= x^{(0)} + \sum_{i=0}^{k-1} p_0(i) + y_1^{(0)} \sum_{i=0}^{k-1} p_1(i) + \cdots + y_n^{(0)} \sum_{i=0}^{k-1} p_n(i) \end{aligned}$$

The closed form of a summation of a polynomial of degree m is a polynomial of degree $m+1$. Polynomial curve fitting is an elementary algorithm which can be used to compute the closed form for the summation: compute the first $m+1$ terms of the summation and then solve the corresponding linear system of equations for the coefficients of the polynomial.

C. Linear recurrence (in)equations

```
while (x ≥ 0 ∧ y ≥ 0):
  if (⋆): x := x - 1
  else: y := y - 1
```

Recurrence equations (such as the simple and stratified varieties) yield precise descriptions of the dynamics of *some* variables, but what about variables which do not satisfy *any* recurrence equation? For example, consider that neither $\mathtt{x}$ nor $\mathtt{y}$ satisfy a recurrence equation in the loop to the right. However, they *do* satisfy recurrence *inequations*: $\mathtt{x} - 1 \leq \mathtt{x}'$, $\mathtt{x}' \leq \mathtt{x}$, $\mathtt{y} - 1 \leq \mathtt{y}'$, and $\mathtt{y}' \leq \mathtt{y}$. These inequations can be closed to yield $\mathtt{x}^{(0)} - k \leq \mathtt{x}^{(k)}$ and $\mathtt{x}^{(k)} \leq \mathtt{x}^{(0)}$, $y^{(0)} - k \leq y^{(k)}$, and $y^{(k)} \leq y^{(0)}$. We will now describe a method for extracting and solving linear recurrence (in)equations, which allows CRA to compute accurate approximations for loops that cannot be completely described by a system of recurrence equations.

Definition 4. *A* linear recurrence (in)equation *of a formula φ is an (in)equation which is implied by φ and which is of the form*

$$\boldsymbol{cx}' \bowtie \boldsymbol{cx} + \boldsymbol{by} + d$$

where $\bowtie \in \{<, \leq, =\}$, $\boldsymbol{x}$ is any vector of variables, $\boldsymbol{y}$ is a vector of stratified induction variables in φ_{body}, $\boldsymbol{c}$, $\boldsymbol{b}$ are constant vectors, and d is a constant.

Linear recurrence (in)equations generalize recurrence equations in two ways: first, they allow for *inequalities* rather than equalities. Second, they allow recurrences for *linear terms*, rather than just variables. For example, the linear recurrence equation $(\mathtt{x}' + \mathtt{y}') = (\mathtt{x} + \mathtt{y}) + 1$ is satisfied by the body of the loop above, which can be closed to yield $(\mathtt{x}^{(k)} + \mathtt{y}^{(k)}) = (\mathtt{x}^{(0)} + \mathtt{y}^{(0)}) + k$.

We now describe a method for detecting and solving linear recurrence (in)equations. Let $\mathsf{Var}^\sharp$ denote the set of variables which are *not* stratified induction variables of φ_{body}. Introduce a set of *difference variables* δ_x, one for each variable x in $\mathsf{Var}^\sharp$ (stratified induction variables are precisely described by recurrence equations, so they need not be approximated). Construct the strongest formula $\delta(\varphi_{\mathsf{body}})$ which is implied by φ_{body} (conjoined with definitional equalities for each difference variable) and which the only free variables are the difference variables and the stratified induction variables of φ_{body} as:

$$\delta(\varphi_{\mathsf{body}}) \triangleq \exists \mathsf{Var}' \cup \mathsf{Var}^\sharp . \varphi_{\mathsf{body}} \wedge \bigwedge \{\delta_x = x' - x : x \in \mathsf{Var}^\sharp\}$$

Next, use Algorithm 2 to compute the convex hull of $\delta(\varphi_{\mathsf{body}})$. Geometrically, the convex hull *hull*(ψ) of a formula ψ is the smallest convex polyhedron which contains ψ. Logically, it is a set of (in)equations such that (1) every (in)equation in *hull*(ψ) is implied by ψ, and (2) any affine (in)equation (over the free variables of ψ) which is implied by ψ is also implied by *hull*(ψ). For example, *hull*$(\delta(\varphi_{\mathsf{body}}))$ for the loop above is:

$$0 \leq \delta_x \wedge \delta_x \leq 1 \wedge 0 \leq \delta_y \wedge \delta_y \leq 1 \wedge \delta_x + \delta_y = 1$$

Algorithm 2: Convex hull.

Input : Formula of the form $\exists X.\psi$, where ψ is satisfiable and quantifier-free
Output: Convex hull of $\exists X.\psi$

```
P ← ⊥;
while there exists a model m of ψ do
    Let Q be a cube of the DNF of ψ s.t. m ⊨ Q;
    Q ← project(Q, X);   /*Polyhedral projection*/
    P ← P ⊔ Q;           /*Polyhedral join*/
    ψ ← ψ ∧ ¬P;
end
return P
```

Note that the only variables which appear in the (in)equations in *hull*$(\delta(\varphi_{\mathsf{body}}))$ are (stratified) induction variables and difference variables. Thus, any (in)equation in *hull*$(\delta(\varphi_{\mathsf{body}}))$ may be written as $\boldsymbol{c\delta} \bowtie \boldsymbol{by} + d$ (where $\boldsymbol{\delta}$ is

the vector of difference variables, $\boldsymbol{y}$ is the vector of stratified induction variables, $\boldsymbol{c}$ and $\boldsymbol{b}$ are constant vectors, and d is a constant). Recalling the definition of the difference variables, such an inequation may be rewritten as $\boldsymbol{c}(\boldsymbol{x}' - \boldsymbol{x}) \bowtie \boldsymbol{by} + d$ and thus as $\boldsymbol{cx}' \bowtie \boldsymbol{cx} + \boldsymbol{by} + d$, which matches the definition of linear recurrence (in)equations given in Definition 4. The closed form of this recurrence inequation is

$$\boldsymbol{cx}^{(k)} \bowtie \boldsymbol{cx}^{(0)} + \sum_{i=0}^{k-1} \boldsymbol{by}^{(i)} + d$$

where the closed form of the summation $\sum_{i=0}^{k-1} \boldsymbol{by}^{(i)} + d$ is computed as in Section III-B.

D. Loop guards

Typically, there is crucial information about the execution of a loop that cannot be captured by recurrence relations. For example, consider the loop in Section III-B. Supposing that the loop executes n times, it must be the case that $\mathtt{x}^{(k)} \leq 10$ for each $k < n$. Further, consider that the variable $\mathtt{z}$ is a function of the simple induction variable $\mathtt{x}$, and so $\mathtt{z}^{(k)}$ can be described precisely in terms of the pre-state variables (even though it does not itself satisfy any recurrence):

$$\mathtt{z}^{(k)} = \begin{cases} \mathtt{z}^{(0)} & \text{if } k = 0 \\ 2(\mathtt{x}^{(0)} + k + 1) & \text{otherwise.} \end{cases}$$

The question is: how can this type of information be recovered from a loop body formula?

We define the *guard* of a transition formula φ_{body} as follows:

$$\mathit{guard}(\varphi_{\mathsf{body}}) \triangleq \boxed{(\exists \mathsf{Var}.\varphi_{\mathsf{body}}) \wedge (\exists \mathsf{Var}'.\varphi_{\mathsf{body}})}$$

If φ_{body} is a loop body formula, then $\mathit{guard}(\varphi_{\mathsf{body}})$ is a formula which over-approximates the effect of executing *at least one* execution of the loop. Intuitively, $(\exists \mathsf{Var}'.\varphi_{\mathsf{body}})$ is a precondition that must hold before every iteration of the loop and $(\exists \mathsf{Var}.\varphi_{\mathsf{body}})$ is a post-condition of the loop that must hold after each iteration.

Consider again the example loop in Section III-B. The loop body formula is as follows:

$$\varphi_{\mathsf{body}} = \boxed{\mathtt{x} \leq 10 \wedge \mathtt{x}' = \mathtt{x} + 1 \wedge \mathtt{y}' = \mathtt{y} + \mathtt{x}' \wedge \mathtt{z}' = 2\mathtt{x}'}$$

Following the definition of *guard*, we have:

$$\begin{aligned} \mathit{guard}(\varphi_{\mathsf{body}}) &\triangleq \boxed{(\exists \mathtt{x}, \mathtt{y}, \mathtt{z}.\varphi_{\mathsf{body}}) \wedge (\exists \mathtt{x}', \mathtt{y}', \mathtt{z}'.\varphi_{\mathsf{body}})} \\ &\equiv \boxed{(\mathtt{x}' \leq 11 \wedge \mathtt{z}' = 2\mathtt{x}') \wedge (\mathtt{x} \leq 10)}\,. \end{aligned}$$

Thus, $\mathit{guard}(\varphi_{\mathsf{body}})$ recovers the desired information about $\mathtt{x}$ and $\mathtt{z}$.

Since loop body formulas may be large, in practice it may be advantageous to simplify the guard formula by eliminating the quantifiers (as we did above). A second option, which is more efficient but less precise, is to over-approximate quantifier elimination. Two possibilities are to use Algorithm 2 to compute the convex hull of $\mathit{guard}(\varphi_{\mathsf{body}})$, or to use optimization modulo theories [18] to compute intervals for each pre- and post-state variable in φ_{body}.

E. Bringing it all together

We close this section by describing how the pieces defined in this section fit into the iteration operator of compositional recurrence analysis. Let $\mathit{CR}(\varphi_{\mathsf{body}})$ denote the set of closed linear recurrence (in)equations (including simple and stratified recurrence equations) satisfied by φ_{body}. Each such (in)equation is of the form $\boldsymbol{cx}^{(k)} \bowtie t$, where the free variables of t are drawn from $\{x^{(0)} : x \in \mathsf{Var}\}$ and a distinguished variable $k \notin \mathsf{Var}$ indicating the loop iteration. Letting $t[\boldsymbol{x}^{(0)} \mapsto \boldsymbol{x}]$ denote the term t with every variable of the form $x^{(0)}$ is replaced by the corresponding variable x, we define $\varphi^{+}_{\mathsf{body}}$ to be the following formula:

$$\boxed{\exists k.k \geq 1 \wedge \textstyle\bigwedge \{\boldsymbol{cx'} \bowtie t[\boldsymbol{x}^{(0)} \mapsto \boldsymbol{x}] : \boldsymbol{cx}^{(k)} \bowtie t \in \mathit{CR}(\varphi_{\mathsf{body}})\}}.$$

Finally, the iteration operator of CRA is defined as:

$$\varphi^{\circledast}_{\mathsf{body}} \triangleq \boxed{(\varphi^{+}_{\mathsf{body}} \wedge \mathit{guard}(\varphi_{\mathsf{body}})) \vee \textstyle\bigwedge_{x \in \mathsf{Var}} x' = x}.$$

IV. Linearization

The iteration operator presented in the previous section operates under the assumption that loop body formulas are expressed in linear arithmetic. However, a program may contain non-linear instructions, and even if it does not, CRA's iteration operator may introduce non-linearity (consider Example 1, where the transition formula for the outer loop $\varphi^{\circledast}_{\mathsf{outer}}$ contains the non-linear proposition $r' = x - q'y$). A solution to this problem is to *linearize* non-linear formulas before passing them to the iteration operator.

Linearization is an operation that, given an (arbitrary) arithmetic formula φ, computes a formula $\mathit{lin}(\varphi)$ which over-approximates φ (i.e., $\varphi \models \mathit{lin}(\varphi)$), but which is expressed in linear arithmetic. There is generally no best approximation of a non-linear formula as a linear formula, so our method is (necessarily) heuristic.

We explain our linearization algorithm informally using an example. Consider the following non-linear formula (where w, x, y, z are integers):

$$\psi \triangleq 1 \leq w = x < y < 5 \wedge wy \leq z \leq xy$$

Our algorithm begins by normalizing ψ, separating it into a linear part and a set of non-linear equations (introducing Skolem constants as necessary). For example, the result of normalizing ψ is:

$$\big(1 \leq w = x < y < 5 \wedge \gamma_0 \leq z \leq \gamma_1\big) \wedge \big(\gamma_0 = wy \wedge \gamma_1 = xy\big)$$

The left conjunct is a linear over-approximation of ψ, but it is very imprecise: semantically equal (but syntactically distinct) non-linear terms become semantically *un*equal in the over-approximation, and all information about the magnitude of non-linear terms is lost. To increase precision of this approximation, we use two strengthening steps.

1) Replace the non-linear operations with uninterpreted function symbols and compute the affine hull of the resulting formula to infer affine equalities between Skolem constants (representing non-linear terms). For our example ψ, this step discovers the equality $\gamma_0 = \gamma_1$.
2) Compute concrete and symbolic intervals for non-linear terms. Consider $\gamma_1 = xy$ from our example ψ. First

compute concrete ($x \in [1,3]$ and $y \in [2,4]$) and symbolic ($x \in [x,x]$ and $y \in [y,y]$) intervals for the operands x and y, using symbolic optimization [18] to compute the concrete intervals. Obtain a concrete interval for xy ($xy \in [2,12]$) by multiplying the concrete intervals of its operands. Obtain symbolic intervals for xy ($xy \in [y,3y]$ and $xy \in [2x,4x]$) by multiplying the concrete interval for x by the symbolic interval for y and vice-versa. As a result of interval computation, we discover: $2 \leq \gamma_1 \leq 12 \wedge y \leq \gamma_1 \leq 3y \wedge 2x \leq \gamma_1 \leq 4x$.

We take $\mathit{lin}(\psi)$ to be the initial coarse linear approximation of ψ conjoined with the facts discovered by the two strengthening steps.

We expect linearization to have applications outside of the context in which we presented it, particularly in program analysis, where over-approximation can be tolerated but non-linear terms cannot. Finding improved linearization heuristics is an interesting direction of future work.

V. Experiments

In this section, we present an experimental evaluation of CRA. We aim to support our hypothesis that, despite the fact that CRA may not take advantage of contextual information, it is competitive with leading verification techniques based on abstraction refinement.

We implemented CRA in a tool that analyzes C code (using the CIL [24] front-end). We use Z3 [9] to resolve SMT queries that result from applying the iteration operator and checking assertion violations. Polyhedra operations are passed to the New Polka library implemented in Apron [4]. The quantifier elimination algorithm from [22] is used to compute loop guards. The tool and benchmarks are available at http://www.cs.toronto.edu/~zkincaid/cra.

We tested two different configurations of CRA: one which is fully compositional (CRA) and does not take advantage of contextual information, and one (CRA+Oct) which uses an intra-procedural octagon analysis [19] to gain *some* contextual information, but which is otherwise compositional. We compare CRA's performance against the state-of-the-art invariant generation and verification tools CPAchecker (overall winner of the 2015 Software Verification Competition) and SeaHorn (winner of the loops category among tools which are sound for verification).

To evaluate the precision of CRA we used it to verify the correctness of a suite of 119 small loop benchmarks of varying difficulty. Our benchmark suite was drawn from the *loops* category of the 2015 Software Verification Competition (SVComp-15), as well as a set of *non-linear* benchmarks (Non-linear), such as the one in Figure 1. The results for the 81 safe, integer-only benchmarks from these suites are shown in Table I. The suite also contains 38 *unsafe* benchmarks: CRA and CRA+Oct have no false negatives on these benchmarks; CPAchecker has 3 and SeaHorn has 2.

CRA can prove safety for 27 more programs than CPAchecker and 3 fewer than SeaHorn, thus demonstrating that CRA is capable of generating competitively precise invariants. This holds despite the fact that CRA is a compositional analysis which does not use contextual information or employ abstraction refinement. The performance of CRA+Oct (compared to CRA) indicates that CRA can be combined with other invariant generation techniques to increase precision.

Test Suite	#Tests	CRA+Oct	CRA	CPAchecker	SeaHorn
SVComp-15	74	65	60	37	65
Non-linear	7	6	5	1	3
Total	81	71 (88%)	65 (80%)	38 (47%)	68 (85%)
Running time across all test suites					
Mean		1.9s	1.8s	42.4s	37.7s
Median		0.6s	0.6s	1.6s	0.2s

TABLE I
Experimental results.

VI. Related work

In this section, we compare compositional recurrence analysis to a sampling of related work on recurrence analysis and compositional invariant generation.

Recurrence analysis. The idea of using closed forms of recurrence relations to approximate loops has appeared in a number of other papers. Generally speaking, CRA differs from previous work in two essential ways: first, CRA uses an SMT solver to extract *semantic* recurrences, rather than *syntactic* recurrences. Second, CRA goes beyond exact recurrences (equations over variables) to *approximate recurrences* (inequations over linear terms).

Ammarguellat and Harrison give a method for detecting induction variables which is compositional in the sense that it uses closed forms for inner loops in order to recognize nested recurrences [1]. Maps from variables to symbolic terms (effectively a symbolic constant propagation domain) are used as an abstract domain (in contrast to CRA's use of arbitrary arithmetic formulas). Rodríguez-Carbonell and Kapur [27] and Kovács [14] developed techniques for discovering invariant polynomial equations based on solving recurrence relations. The classes of simple and stratified recurrence equations are subsumed by the ones considered in [27], [14], but admit a simpler algorithm for computing closed forms. Kroening et al. [15] present a technique for computing *under*-approximations of loops which uses polynomial curve-fitting to directly compute closed forms for recurrences rather than extracting recurrences and then solving them in a separate step.

Ancourt et al. give a technique for computing recurrence *in*equations for **while** loops with affine bodies [2]. As with the method for computing linear recurrence inequations presented in Section III-C, their method is based on difference variables and polyhedral projection. CRA generalizes this work by (1) extending it to arbitrary control flow and non-linear arithmetic, (2) integrating recurrence inequations with stratified induction variables, thereby allowing enabling the computation of invariant polynomial inequations. Ancourt et al. discuss a method for computing invariant polynomial inequations which is based on higher-order differences rather than stratified recurrence inequations.

Acceleration. *Acceleration* is a technique closely related to recurrence analysis that was pioneered in infinite-state model checking [6], [11], [3], and which has recently found use in program analysis [12], [17], [13]. Given a set of reachable states and an affine transformation describing the body of a loop, acceleration computes an *exact* post-image which describes the set of reachable states after executing any number of iterations of the loop (although there is recent work on *abstract acceleration* that computes over-approximate post-images [12], [13]). In contrast, CRA is *approximate* rather than exact, and computes transition formulas rather than post-images. A result of these two features is that CRA can be applied to arbitrary loops, while acceleration is classically limited to simple loops where the body consists of a sequence of assignment statements.

Compositional program analysis. Compositional program analysis has a long history. Particular examples are inter-procedural analyses based on summarization [31] and elimination-style dataflow analyses (a good overview of which can be found in [28]). The following surveys recent work on compositional analysis for numerical invariants.

Kroening et al. [16] and Biallas et al. [5] present compositional analysis techniques based on predicate abstraction. In addition to predicate abstraction, there are a few papers which use numerical abstract domains for compositional analysis. These include an algorithm for detecting affine equalities between program variables [23], an algorithm for detecting polynomial equalities between program variables [8], a disjunctive polyhedra analysis which uses widening to compute loop summaries [25], and a method for automatically synthesizing transfer functions for template abstract domains using quantifier elimination [21]. Our abstract domain is the set of arbitrary arithmetic formula, which is more expressive than these domains, but which (as usual) incurs a potential price in performance. It would be interesting to apply abstractions to our formulas to improve the performance of our analysis.

Linearization. The linearization algorithm in Section IV was inspired by Miné's procedure for approximating non-linear abstract transformers [20]. Miné's procedure abstracts non-linear terms by linear terms with interval coefficients using the abstract value in the pre-state to derive intervals for variables. Our algorithm abstracts non-linear terms by sets of symbolic and concrete intervals, and applies to the more general setting of approximating arbitrary formulas.

VII. Conclusion

This paper describes compositional recurrence analysis, a fully compositional algorithm for generating numerical invariants of imperative programs. There are two main points to take away. The first is that it is possible to design a fully compositional analysis (which makes no use of contextual information) that is competitive with state-of-the-art verification techniques based on abstraction refinement. Second, recurrence-based program analysis may be extended to programs with arbitrary control flow by exploiting compositionality and SMT solving.

References

[1] Z. Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *PLDI*, pages 283–295, 1990.

[2] C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electron. Notes Theor. Comput. Sci.*, 267(1):3–16, Oct. 2010.

[3] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *ATVA*, pages 474–488. 2005.

[4] J. Bertrand and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.

[5] S. Biallas, J. Brauer, A. King, and S. Kowalewski. Loop leaping with closures. In *SAS*, pages 214–230, 2012.

[6] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *CAV*, pages 55–67. 1994.

[7] K. Chatterjee, R. Ibsen-Jensen, A. Pavlogiannis, and P. Goyal. Faster algorithms for algebraic path properties in recursive state machines with constant treewidth. In *POPL*, pages 97–109, 2015.

[8] M. A. Colón. Approximating the algebraic relational semantics of imperative programs. In *SAS*, pages 296–311. 2004.

[9] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[10] A. Farzan and Z. Kincaid. An algebraic framework for compositional program analysis. *CoRR*, abs/1310.3481, 2013.

[11] A. Finkel and J. Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In *FST TCS*, pages 145–156, 2002.

[12] L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *SAS*, pages 144–160. 2006.

[13] B. Jeannet, P. Schrammel, and S. Sankaranarayanan. Abstract acceleration of general linear loops. In *POPL*, pages 529–540, 2014.

[14] L. Kovács. Reasoning algebraically about P-solvable loops. In *TACAS*, pages 249–264. 2008.

[15] D. Kroening, M. Lewis, and G. Weissenbacher. Under-approximating loops in C programs for fast counterexample detection. In *CAV*, pages 381–396. 2013.

[16] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. Wintersteiger. Loop summarization using abstract transformers. In *ATVA*, pages 111–125. 2008.

[17] J. Leroux and G. Sutre. Accelerated data-flow analysis. In *SAS*, pages 184–199, 2007.

[18] Y. Li, A. Albarghouthi, Z. Kincaid, A. Gurfinkel, and M. Chechik. Symbolic optimization with SMT solvers. In *POPL*, pages 607–618, 2014.

[19] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[20] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI*, pages 348–363, 2006.

[21] D. Monniaux. Automatic modular abstractions for linear constraints. In *POPL*, pages 140–151, 2009.

[22] D. Monniaux. Quantifier elimination by lazy model enumeration. In *CAV*, pages 585–599, 2010.

[23] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, pages 330–341, 2004.

[24] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.

[25] C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *ASIAN*, pages 331–345, 2007.

[26] T. W. Reps, S. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.

[27] E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *ISSAC*, pages 266–273, 2004.

[28] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Comput. Surv.*, 18(3):277–316, Sept. 1986.

[29] B. Scholz and J. Blieberger. A new elimination-based data flow analysis framework using annotated decomposition trees. In *CC*, pages 202–217, 2007.

[30] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[31] M. Sharir and A. Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, 1981.

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