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.

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 \( x \) and subtracting 2 from \( y \) at each iteration. The body of this loop can be described by a system of recurrence equations:

\[
\begin{align*}
x^{(k)} &= x^{(k-1)} + 1 \\
y^{(k)} &= y^{(k-1)} - 2
\end{align*}
\]

(where \( x^{(k)} \) represents the value of \( x \) 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 \( x \) and \( y \) to denote the values of those variables before executing the loop and \( x' \) and \( y' \) to their values after the loop, a transition formula that precisely describes this loop is:

\[
\exists k \in \mathbb{N}. x' = x + k \land y' = y - 2k
\]

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 \( x \) 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 Satisfiability 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 (⊙), choice (⊕), and iteration (Ө) 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 of CRA by recursing on the regular expression, interpreting each edge using the semantic function in the semantic algebra of CRA by recursing on the regular expression, replacing each primed variable x′ with its interpretation in the semantic algebra of CRA and each regular expression operator by its counterpart in the semantic algebra of CRA. The result is a transition formula which approximates the executions which end at a regular expression.

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 of CRA by recursing on the regular expression, interpreting each edge using the semantic function in the semantic algebra of CRA by recursing on the regular expression, replacing each primed variable x′ with its interpretation in the semantic algebra of CRA and each regular expression operator by its counterpart in the semantic algebra of CRA. The result is a transition formula which approximates the executions which end at a regular expression.

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 of CRA by recursing on the regular expression, interpreting each edge using the semantic function in the semantic algebra of CRA by recursing on the regular expression, replacing each primed variable x′ with its interpretation in the semantic algebra of CRA and each regular expression operator by its counterpart in the semantic algebra of CRA. The result is a transition formula which approximates the executions which end at a regular expression.

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 of CRA by recursing on the regular expression, interpreting each edge using the semantic function in the semantic algebra of CRA by recursing on the regular expression, replacing each primed variable x′ with its interpretation in the semantic algebra of CRA and each regular expression operator by its counterpart in the semantic algebra of CRA. The result is a transition formula which approximates the executions which end at a regular expression.

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 of CRA by recursing on the regular expression, interpreting each edge using the semantic function in the semantic algebra of CRA by recursing on the regular expression, replacing each primed variable x′ with its interpretation in the semantic algebra of CRA and each regular expression operator by its counterpart in the semantic algebra of CRA. The result is a transition formula which approximates the executions which end at a regular expression.

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 of CRA by recursing on the regular expression, interpreting each edge using the semantic function in the semantic algebra of CRA by recursing on the regular expression, replacing each primed variable x′ with its interpretation in the semantic algebra of CRA and each regular expression operator by its counterpart in the semantic algebra of CRA. The result is a transition formula which approximates the executions which end at a regular expression.
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:

$$
\varphi_{\text{inner}} = \exists k. k \geq 0 \wedge r' = r - k \wedge t' = t - k \wedge \text{id}(\{q, x, y\})
$$

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

$$
\varphi_{\text{outer}} = [\langle v_2, v_3 \rangle \circ [\langle v_3, v_4 \rangle \circ \varphi_{\text{inner}} \circ [\langle v_4, v_5 \rangle \circ [\langle v_5, v_6 \rangle \circ [\langle v_6, v_4 \rangle] \circ [\langle v_7, v_2 \rangle]]]]
$$

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 $\varphi_{\text{body}}$ which approximates the behaviour of the body of a loop. The goal of the iteration operator is to compute a formula $\varphi^\bullet_{\text{body}}$ 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 $\varphi_{\text{body}}$ 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 $\varphi_{\text{body}}$ representing the body of a loop. We assume that $\varphi_{\text{body}}$ 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 $\varphi_{\text{body}}$ is satisfiable; if $\varphi_{\text{body}}$ is unsatisfiable, then we may take $\varphi_{\text{body}}^\bullet$ to be $\bigwedge_{x \in \text{Var}} x' = x$, 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 $\varphi_{\text{body}}$ 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 $\varphi_{\text{body}}$, we say 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 \( \varphi_{\text{body}} \) can be detected as follows. First, query an SMT solver for a model \( m \) of \( \varphi_{\text{body}} \). Then, for each variable \( x \), ask an SMT solver if \( \varphi_{\text{body}} \) 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 \( x \) satisfies a simple recurrence equation \( x' = x + 1 \), and that \( y \) satisfies a (non-simple) recurrence equation \( y' = y + x + 1 \). A closed form for \( y \)'s recurrence is \( y^{(k)} = y^{(0)} + \sum_{i=0}^{k-1} (x^{(i)} + 1) \).

Since \( x \) is an induction variable, we have a closed form for \( x \) \( (x^{(i)} = x^{(0)} + i) \), which we may use to simplify \( y \)'s recurrence:

\[
y^{(k)} = y^{(0)} + \sum_{i=0}^{k-1} (x^{(0)} + i + 1)
= y^{(0)} + kx^{(0)} + k + \sum_{i=0}^{k-1} i
= y^{(0)} + kx^{(0)} + \frac{k(k + 1)}{2}.
\]

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 \( \varphi_{\text{body}} \) are defined recursively as:

- A simple recurrence equation which is satisfied by \( \varphi_{\text{body}} \) is a stratified recurrence equation of \( \varphi_{\text{body}} \) (and a simple induction variable is a stratified induction variable) at stratum 0.

- Let \( y \) denote a vector of the stratified induction variables of strata \( \leq N \). A recurrence of the form \( x' = x + cy + d \) (where \( c \) 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 \( \text{aff}(\varphi_{\text{body}}) \) of \( \varphi_{\text{body}} \). The affine hull \( \text{aff}(\varphi_{\text{body}}) \) of a formula \( \varphi_{\text{body}} \) is the smallest affine set which contains \( \varphi_{\text{body}} \), represented as (the set of solutions to) a system of equations \( Ax = b \),

**Algorithm 1: Affine hull.**

**Input:** Satisfiable formula \( \varphi_{\text{body}} \)

**Output:** Affine hull of \( \varphi_{\text{body}} \)

\[
\begin{align*}
H & \leftarrow \bot; \psi \leftarrow \varphi_{\text{body}}; \\
\text{while there exists a model } m \text{ of } \psi \text{ do} \\
H' & \leftarrow \{ x = [x]^m : x \in \text{Var} \cup \text{Var}' \}; \\
H & \leftarrow H \cup H'; \\
\psi & \leftarrow \psi \land \neg H; \\
\text{return } H.
\end{align*}
\]

where \( x = [x_1, \ldots, x_n, x'_1, \ldots, x'_n] \) is a vector of all variables in \( \text{Var} \cup \text{Var}' \). Logically, \( \text{aff}(\varphi_{\text{body}}) \) is a system of equations such that (1) \( \varphi_{\text{body}} \models \text{aff}(\varphi_{\text{body}}) \), and (2) every affine equation over \( \text{Var} \cup \text{Var}' \) which is implied by \( \varphi_{\text{body}} \) is also implied by \( \text{aff}(\varphi_{\text{body}}) \). The affine hull of a formula may be computed using Algorithm 1 (a specialization of the algorithm \( \alpha \) 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:

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

Our strategy for generating stratified recurrence equations from \( \text{aff}(\varphi_{\text{body}}) \) is based on the following lemma. Combined with Property (2) of \( \text{aff}(\varphi_{\text{body}}) \) above, this lemma implies that any affine equation implied by \( \varphi_{\text{body}} \) can be expressed as a linear combination of the equations in \( \text{aff}(\varphi_{\text{body}}) \). Thus, after computing the affine hull of \( \varphi_{\text{body}} \), 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]).** Let \( A \) be a matrix, \( b \) be a column vector, \( c \) be a row vector, and \( d \) be a constant. Assume that the system \( Ax = b \) has a solution. Then \( Ax = b \) implies \( cx = d \) if there is a row vector \( \lambda \) such that \( \lambda A = c \) and \( \lambda b = d \).

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

- \( \lambda A = c \) and \( \lambda b = d \) (i.e., \( cx = d \) is implied by \( \text{aff}(\varphi_{\text{body}}) \) and thus by \( \varphi_{\text{body}} \))
- \( c_i = 1 \) 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 \( N \):

\[
x' = c_1y_1 + \cdots + c_ny_n + b \quad \text{(all } y_1, ..., y_n \text{ are of strata } < N)\]

Then we may write

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

By our induction hypothesis, each \( y_i^{(i)} \) 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

\[
x^{(k)} = x^{(0)} + \sum_{i=0}^{k-1} p_0(i) + p_1(i)y_1^{(0)} + \cdots + p_n(i)y_n^{(0)}
\]

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

Recurrence equations (such as the simple and stratified variables) 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 \( x \) nor \( y \) satisfy a recurrence equation in the loop to the right. However, they do satisfy recurrence inequations: \( x - 1 \leq x', x' \leq x, y - 1 \leq y', \text{ and } y' \leq y \). These inequations can be closed to yield \( x^{(0)} - k \leq x^{(k)} \) and \( x^{(k)} \leq x^{(0)}, y^{(0)} - k \leq y^{(k)}, \text{ and } y^{(k)} \leq y^{(0)} \). 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 \( \varphi \) is an (in)equation which is implied by \( \varphi \) and which is of the form

\[
ca' \bowtie cx + by + d
\]

where \( \bowtie \in \{<, \leq, =\} \), \( x \) is any vector of variables, \( y \) is a vector of stratified induction variables in \( \varphi_\text{body} \), \( c, b \) 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 \((x' + y') = (x + y) + 1\) is satisfied by the body of the loop above, which can be closed to yield \((x^{(k)} + y^{(k)}) = (x^{(0)} + y^{(0)}) + k\).

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

\[
\delta(\varphi_\text{body}) \equiv \exists \text{Var}^d \cup \text{Var}^d \cdot \varphi_\text{body} \land \{\delta_x = x' - x : x \in \text{Var}^d\}
\]

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

\[
0 \leq \delta_x \land \delta_x \leq 1 \land 0 \leq \delta_y \land \delta_y \leq 1 \land \delta_x + \delta_y = 1
\]

Algorithm 2: Convex hull.

Input: Formula of the form \( \exists X.\psi \), where \( \psi \) is satisfiable and quantifier-free

Output: Convex hull of \( \exists X.\psi \)

\[
P \leftarrow \bot; \quad \text{while there exists a model } m \text{ of } \psi \text{ do}
\]

1. Let \( Q \) be a cube of the DNF of \( \psi \) s.t. \( m \models Q \);
2. \( P \leftarrow \text{project}(Q, X); \quad /*\text{Polyhedral projection}*/
3. \( P \leftarrow P \cup Q; \quad /*\text{Polyhedral join}*/
4. \( \psi \leftarrow \psi \land \neg P; \)

end

return \( P \)

Note that the only variables which appear in the (in)equations in \( \text{hull}(\delta(\varphi_\text{body})) \) are (stratified) induction variables and difference variables. Thus, any (in)equation in \( \text{hull}(\delta(\varphi_\text{body})) \) may be written as \( c\delta \bowtie by + d \) (where \( \delta \) is
the vector of difference variables, \( y \) is the vector of stratified induction variables, \( c \) and \( b \) are constant vectors, and \( d \) is a constant). Recalling the definition of the difference variables, such an inequation may be rewritten as \( c(x' - x) \ni by + d \) and thus as \( cax + by + d \), which matches the definition of linear recurrence (in)equations given in Definition 4. The closed form of this recurrence inequation is
\[
ca^{(k)} \ni ca^{(0)} + \sum_{i=0}^{k-1} by^{(i)} + d
\]
where the closed form of the summation \( \sum_{i=0}^{k-1} by^{(i)} + d \) 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 \( x^{(k)} \leq 10 \) for each \( k < n \). Further, consider that the variable \( z \) is a function of the simple induction variable \( x \), and so \( z^{(k)} \) can be described precisely in terms of the pre-state variables (even though it does not itself satisfy any recurrence):
\[
z^{(k)} = \begin{cases} z^{(0)} & \text{if } k = 0 \\ 2(x^{(0)} + k + 1) & \text{otherwise.} \end{cases}
\]
The question is: how can this type of information be recovered from a loop body formula?

We define the guard of a transition formula \( \varphi_{body} \) as follows:
\[
\text{guard}(\varphi_{body}) = (\exists \text{Var}', \varphi_{body}) \land (\exists \text{Var}, \varphi_{body})
\]
If \( \varphi_{body} \) is a loop body formula, then \( \text{guard}(\varphi_{body}) \) is a formula which over-approximates the effect of executing at least one execution of the loop. Intuitively, \( (\exists \text{Var}, \varphi_{body}) \) is a precondition that must hold before every iteration of the loop and \( (\exists \text{Var}', \varphi_{body}) \) 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:
\[
\varphi_{body} = x \leq 10 \land x' = x + 1 \land y' = y + x' \land z' = 2x'
\]
Following the definition of guard, we have:
\[
\text{guard}(\varphi_{body}) = (\exists x', y, z. \varphi_{body}) \land (\exists x', y', z'. \varphi_{body})
\]
\[
\equiv (x' \leq 11 \land z' = 2x') \land (x \leq 10).
\]
Thus, \( \text{guard}(\varphi_{body}) \) recovers the desired information about \( x \) and \( z \).

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 3 to compute the convex hull of \( \text{guard}(\varphi_{body}) \), or to use optimization modulo theories [18] to compute intervals for each pre- and post-state variable in \( \varphi_{body} \).

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 \( CR(\varphi_{body}) \) denote the set of closed linear recurrence (in)equations (including simple and stratified recurrence equations) satisfied by \( \varphi_{body} \). Each such (in)equation is of the form \( ca^{(k)} \ni t \), where the free variables of \( t \) are drawn from \( \{x^{(0)} : x \in \text{Var}\} \) and a distinguished variable \( k \notin \text{Var} \) indicating the loop iteration. Letting \( t[x^{(0)} \mapsto x] \) denote the term \( t \) with every variable of the form \( x^{(0)} \) is replaced by the corresponding variable \( x \), we define \( \varphi_{\text{body}}^+ \) to be the following formula:
\[
\exists k. k \geq 1 \land (\{ca^{(k)} \ni t[x^{(0)} \mapsto x] \} : ca^{(k')} \ni t \in CR(\varphi_{body}))
\]
Finally, the iteration operator of CRA is defined as:
\[
\varphi_{\text{body}}^* \equiv (\varphi_{\text{body}}^+ \land \text{guard}(\varphi_{body})) \lor \bigwedge_{x \in \text{Var}} x' = x.
\]

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 \( \varphi_{outer} \) 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 \( \varphi \), computes a formula \( \text{lin}(\varphi) \) which over-approximates \( \varphi \) (i.e., \( \varphi \models \text{lin}(\varphi) \)), 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):
\[
\psi \equiv 1 \leq w = x < y < 5 \land wy \leq z \leq xy
\]
Our algorithm begins by normalizing \( \psi \), separating it into a linear part and a set of non-linear equations (introducing Skolem constants as necessary). For example, the result of normalizing \( \psi \) is:
\[
(1 \leq w = x < y < 5 \land \gamma_0 \leq z \leq \gamma_1) \land (\gamma_0 = wy \land \gamma_1 = xy)
\]
The left conjunct is a linear over-approximation of \( \psi \), but it is very imprecise: semantically equal (but syntactically distinct) non-linear terms become semantically unequal 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 \( \psi \), 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 \( \psi \). 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 \cite{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: $2 \leq \gamma_1 \leq 12 \land y \leq \gamma_1 \leq 3y \land 2x \leq \gamma_1 \leq 4x$.

We take $\ln(\psi)$ to be the initial coarse linear approximation of $\psi$ conjointed 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.

\section{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 \cite{24} front-end). We use Z3 \cite{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 \cite{4}. The quantifier elimination algorithm from \cite{22} is used to compute loop guards. The tool and benchmarks are available at \url{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 \cite{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 \ref{fig:example}. The results for the 81 safe, integer-only benchmarks from these suites are shown in Table \ref{tab:results}. 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.

\section{VI. Related Work}

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

\textbf{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 \cite{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 \cite{27} and Kovács \cite{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 \cite{27, 14}, but admit a simpler algorithm for computing closed forms. Kroening et al. \cite{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 inequations for while loops with affine bodies \cite{2}. As with the method for computing linear recurrence inequations presented in Section \ref{sec:related-work-1}, 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.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Test Suite & \# Tests & CRA+Oct & CRA & CPAchecker & Seahorn \\
\hline
SVComp-15 & 74 & 65 & 60 & 37 & 65 \\
Non-linear & 7 & 6 & 6 & 1 & 3 \\
Total & 81 & 71 (88\%) & 65 (80\%) & 38 (47\%) & 68 (85\%) \\
\hline
\end{tabular}
\caption{Experimental results.}
\end{table}
Acceleration. Acceleration is a technique closely related to recurrence analysis that was pioneered in infinite-state model checking \cite{25}, \cite{12}, \cite{31}, and which has recently found use in program analysis \cite{12}, \cite{17}, \cite{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 of iterations of the loop (although there is recent work on over-approximate post-abstract acceleration of iterations of the loop). 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 interprocedural analyses based on summarization \cite{31} and elimination-style dataflow analyses (a good overview of which can be found in \cite{28}). The following surveys recent work on compositional analysis for numerical invariants.

Kroening et al. \cite{16} and Biabas et al. \cite{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 \cite{23}, an algorithm for detecting polynomial equalities between program variables \cite{8}, a disjunctive polyhedra analysis which uses widening to compute loop summaries \cite{25}, and a method for automatically synthesizing transfer functions for template abstract domains using quantifier elimination \cite{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 \cite{17} was inspired by Miné’s procedure for approximating non-linear abstract transformers \cite{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


