Non-linear Reasoning for Invariant Synthesis

ZACHARY KINCAID, Princeton University, USA
JOHN CYPHERT and JASON BRECK, University of Wisconsin, USA
THOMAS REPS, University of Wisconsin, USA and GrammaTech, Inc., USA

Automatic generation of non-linear loop invariants is a long-standing challenge in program analysis, with many applications. For instance, reasoning about exponentials provides a way to find invariants of digital-filter programs, and reasoning about polynomials and/or logarithms is needed for establishing invariants that describe the time or memory usage of many well-known algorithms. An appealing approach to this challenge is to exploit the powerful recurrence-solving techniques that have been developed in the field of computer algebra, which can compute exact characterizations of non-linear repetitive behavior. However, there is a gap between the capabilities of recurrence solvers and the needs of program analysis: (1) loop bodies are not merely systems of recurrence relations—they may contain conditional branches, nested loops, non-deterministic assignments, etc., and (2) a client program analyzer must be able to reason about the closed-form solutions produced by a recurrence solver (e.g., to prove assertions).

This paper presents a method for generating non-linear invariants of general loops based on analyzing recurrence relations. The key components are an abstract domain for reasoning about non-linear arithmetic, a semantics-based method for extracting recurrence relations from loop bodies, and a recurrence solver that avoids closed forms that involve complex or irrational numbers. Our technique has been implemented in a program analyzer that can analyze general loops and mutually recursive procedures. Our experiments show that our technique shows promise for non-linear assertion-checking and resource-bound generation.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Automated static analysis**;

Additional Key Words and Phrases: Invariant generation, Recurrence relation, Operational calculus

ACM Reference Format:
Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. 2018. Non-linear Reasoning for Invariant Synthesis . *Proc. ACM Program. Lang.* 2, POPL, Article 54 (January 2018), 33 pages. https://doi.org/10.1145/3158142

1 INTRODUCTION

Recurrence equations have a long history, and variety of techniques for solving them are known. A natural question is to ask how these techniques can be put to work for invariant generation.

One line of work in this direction focuses on computing very accurate information about a syntactically restricted class of loops. For example, Rodríguez-Carbonell and Kapur [2004] and Kovács [2008] use recurrence solving to compute *all* invariant polynomial equations of two (different) classes of loops. Neither technique can compute any invariants for loops that have (for example) nondeterministic assignments or nested loops.

Authors' addresses: Zachary Kincaid, zkincaid@cs.princeton.edu, Princeton University, USA; John Cyphert, jcyphert@wisc.edu; Jason Breck, jbreck@wisc.edu, University of Wisconsin, USA; Thomas Reps, reps@cs.wisc.edu, University of Wisconsin, USA, GrammaTech, Inc. USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
2475-1421/2018/1-ART54
https://doi.org/10.1145/3158142

Compositional recurrence analysis (CRA) [Farzan and Kincaid 2015] is another line of work, which focuses on over-approximate analysis of general loops rather than precise analysis of syntactically restricted loops. Recent work [Kincaid et al. 2017] extends the generality of CRA even further, showing how the approach can be applied to recursive procedures as well as loops. The key idea that makes CRA so broadly applicable is that it represents loop behavior using logical formulas, and uses semantics-based techniques to find implied recurrence relations. However, CRA's ability to reason about non-linear behavior is limited by the fact that it uses SMT solving, linear algebra, and polyhedral techniques to extract recurrences from loop bodies, and polynomial summation to solve them. In particular, CRA is only capable of extracting recurrence relations that can be expressed in linear integer arithmetic and that have polynomial closed forms—effectively exploiting only a fraction of what recurrence solvers (e.g., the ones used in [Kovács 2008; Rodríguez-Carbonell and Kapur 2004]) are capable of.

In this paper, we present extensions to the numerical-reasoning techniques underlying CRA, and demonstrate that these extensions enable CRA to establish many non-linear numerical invariants. The contributions of the paper are three-fold:

- We present the *wedge* abstract domain, a numerical abstract domain capable of reasoning about non-linear arithmetic. Just as convex polyhedra represent properties in the conjunctive fragment of linear arithmetic, wedges represent properties in the conjunctive fragment of non-linear arithmetic (including polynomials, exponentials, and logarithms). The deductive power of wedges is due to polyhedral and Gröbner-basis techniques, congruence closure, and simple inference rules for non-linear functions. The key operation supported by the domain is *symbolic abstraction* [Reps et al. 2004; Thakur and Reps 2012], which, given an arbitrary non-linear formula φ, computes a wedge that over-approximates φ. (See §4.)
- We present a semantics-based algorithm for extracting recurrence relations that are entailed by a loop-body formula. The algorithm is based on first over-approximating the loop body by a wedge, and then using techniques from linear algebra to extract recurrences from the wedge. The algorithm can extract recurrences involving non-linear arithmetic and inter-dependent program variables; the class of recurrences that can be extracted by this algorithm corresponds to *C-finite sequences* [Kauers and Paule 2011, §4.2]. (See §5.)
- We present an algorithm, OCRS, that is able to solve these recurrences, and find closed-form solutions that include polynomials, exponentials, and logarithms. OCRS is based on an automated and enhanced form of the discrete *operational calculus* of Berg [1967]. Classically, the closed forms of C-finite sequences involve algebraic irrational or algebraic complex numbers,[1] but OCRS avoids non-rational numbers by using what we call *implicitly interpreted functions*. Each implicitly interpreted function is associated with a term in the logic of OCRS that exactly characterizes the function, but outside of the recurrence solver (and in particular, within the wedge domain) an implicitly interpreted function is treated as an uninterpreted function symbol. (See §6.)

Our approach builds upon the recent work of Kincaid et al. [2017], which extended CRA so that it can analyze recursive programs using essentially the same approach that it uses to handle loops.

Organization. §2 illustrates the main features of our method via a series of examples. §3 presents relevant background material. §4 presents the wedge abstract domain. §5 describes the method used in our system for extracting a recurrence relation from a wedge. §6 presents OCRS. §7 presents experimental results. §8 discusses related work.

[1] An algebraic number is a complex number that is a root of a non-zero univariate polynomial with rational coefficients. Henceforth, we shorten "algebraic irrational" and "algebraic complex" to "irrational" and "complex," respectively.

```
int ticks = 0;
for(int a = 0; a < N; a++)
   for(int b = 0; b < N; b++)
      for(int c = 0; c < N; c++)
         ticks++;
```

(a)

```
int pos = 1, steps = 0, found = 0;
while(pos < array_size) {
   steps += 1;
   if (array[pos] == val) {
      found = 1;
      break;
   }
   if (array[pos] < val) {
      pos = 2 * pos;
   } else {
      pos = 2 * pos + 1;
   }
}
```

(b)

```
int temp, x = -1, y = 0;
for(int n = 0; n < 1; n++) {
   temp = x; x = y; y = -temp;
}
if (y == 1) goto errorlabel;
```

(c)

```
int fib(int n, int high) {
    int f1 = 1, f2 = 0, temp = 0;
    if (high) {
       while(n > 0) {
          f1 = f1 + f2;
          f2 = f1 - f2;
          n--;
       }
    } else {
       while(n > 0) {
          temp = f2;
          f2 = f1;
          f1 = f2 + temp;
          n--;
       }
    }
    return f1;
}
void main() {
    int n1, n2, obs1, obs2, high1, high2;
    assume(n1 == n2);
    // Note: high1 might not equal high2
    obs1 = fib(n1, high1);
    obs2 = fib(n2, high2);
    assert(obs1 == obs2);
}
```

(d)

Fig. 1. Four examples: (a) cubic-time loop nest; (b) binary search; (c) rotation in the x, y plane; (d) absence of information flow.

2 OVERVIEW AND PROBLEM STATEMENT

Our system follows in the tradition of CRA [Farzan and Kincaid 2015] and ICRA [Kincaid et al. 2017] in how it combines symbolic analysis and abstract interpretation:

- It uses an abstract domain of transition formulas, and thus models non-looping program behavior precisely.
- It conservatively explores all behaviors of a loop by over-approximating the transitive closure of the loop body: a formula for the loop body is converted into a system of recurrences, which are then solved to create a transition formula that summarizes the overall action of the loop.

For brevity, we refer to the actions performed to analyze a loop as the *star operator*.

This section illustrates at a high level the main features of the star operator, focusing on how the system reasons about non-linear relationships, such as polynomials, exponentials, and logarithms. We present two examples of running-time analysis, along with two other examples, one analyzing a rotation, and one demonstrating information-flow analysis.

Example 2.1. Consider the loop nest shown in Fig. 1(a), which runs in cubic time. CRA analyzes the loop nest "bottom-up", starting from the most deeply nested loop and moving outwards, applying the star operator at each nesting level. Nested loops lead to a design constraint in the program analyzer: the star operator must be able to reason about its own output.

Nesting Level	Recurrence	Solution
3	$\mathtt{c}^{[k+1]} = \mathtt{c}^{[k]} + 1 \wedge \mathtt{ticks}^{[k+1]} = \mathtt{ticks}^{[k]} + 1$	$\mathtt{c}^{[k]} = \mathtt{c}^{[0]} + k \wedge \mathtt{ticks}^{[k]} = \mathtt{ticks}^{[0]} + k$
2	$\mathtt{b}^{[k+1]} = \mathtt{b}^{[k]} + 1 \wedge \mathtt{ticks}^{[k+1]} = \mathtt{ticks}^{[k]} + \mathtt{N}$	$\mathtt{b}^{[k]} = \mathtt{b}^{[0]} + k \wedge \mathtt{ticks}^{[k]} = \mathtt{ticks}^{[0]} + \mathtt{N} * k$
1	$\mathtt{a}^{[k+1]} = \mathtt{a}^{[k]} + 1 \wedge \mathtt{ticks}^{[k+1]} = \mathtt{ticks}^{[k]} + \mathtt{N} * \mathtt{N}$	$\mathtt{a}^{[k]} = \mathtt{a}^{[0]} + k \wedge \mathtt{ticks}^{[k]} = \mathtt{ticks}^{[0]} + \mathtt{N} * \mathtt{N} * k$

Fig. 2. Recurrences created for the nested loops in Fig. 1(a).

Fig. 2 shows the recurrences created for the program in Fig. 1(a). We use, e.g., $\mathtt{c}^{[k]}$ to denote the value of variable c at the beginning of iteration k of the innermost loop (at nesting-level 3). As shown in Fig. 2, the body of the innermost loop can be described by a linear recurrence, and the solution is also linear. Using the solution to that recurrence, along with the fact that the inner loop iterates N times, we obtain a linear recurrence for the loop at nesting-level 2. However, the solution to this recurrence is non-linear: it involves the term $\mathtt{N} * k$. Consequently, the body of the outer loop (at nesting-level 1) has the following non-linear transition formula:

$$\exists k.\mathtt{ticks}' = \mathtt{ticks} + \mathtt{N} * k \wedge k \leq \mathtt{N} \wedge k \geq \mathtt{N} \wedge \mathtt{a}' = \mathtt{a} + 1,$$

which cannot be analyzed directly using many of the classical tools of program analysis, such as Satisfiability Modulo Theories (SMT) solvers and the abstract domain of polyhedra. Thus, although each iteration of the outermost loop increases the value of ticks by $\mathtt{N} * \mathtt{N}$, the recurrence-extraction technique of Farzan and Kincaid [2015] is unable to find the relevant recurrence. This paper presents an extension of CRA that is able to handle polynomial recurrences such as this one, and is therefore able to prove that the above triply nested loop increases the value of ticks by exactly $\mathtt{N} * \mathtt{N} * \mathtt{N}$.

To obtain tight bounds on the running time of other programs, logarithms are required. Moreover, loop bodies can contain branching code—and thus different paths through the loop body may be taken on different iterations.

Example 2.2. Consider the loop shown in Fig. 1(b), which performs a binary search over a complete binary tree that is stored in an array. Our analysis establishes that the running time of this loop is logarithmic in array_size by first constructing a recurrence saying that, on each iteration, steps is incremented by 1 and pos (at least) doubles:

$$\mathtt{steps}^{[k+1]} = \mathtt{steps}^{[k]} + 1 \wedge \mathtt{pos}^{[k+1]} \geq 2\mathtt{pos}^{[k]}.$$

Extraction is a nontrivial operation because the inner loop contains branching code. Moreover, the exact behavior of the variable pos cannot be described by a recurrence equation. Instead, we must *approximate* the behavior of pos by a recurrence *in*equation.

Next, the recurrence solver finds the solution to this recurrence:

$$\mathtt{steps}^{[k]} = \mathtt{steps}^{[0]} + k \wedge \mathtt{pos}^{[k]} \geq 2^k\mathtt{pos}^{[0]},$$

and this result, in combination with the initial conditions of the loop, establishes that whenever control reaches the head of the loop, we have $\mathtt{pos} \geq 2^{\mathtt{steps}}$. The solver also establishes that, upon exit from the loop, $\mathtt{pos} < 2(\mathtt{array_size})$, because otherwise the loop would have exited earlier. We conclude that $\mathtt{steps} < 1 + log_2(\mathtt{array_size})$, if $\mathtt{array_size} > 0$, and $\mathtt{steps} = 0$ otherwise.

This example shows that the analysis of a logarithmic-time algorithm sometimes requires finding exponential solutions to recurrences.

Some programs have non-linear relationships that cannot be expressed using polynomials, exponentials, and logarithms of integers.

Example 2.3. Consider the loop shown in Fig. 1(c). One iteration of the loop performs a 90-degree clockwise rotation of the point (x, y) about the origin. Because of the mutual dependence between x and y, we extract a matrix recurrence for this loop. Below, we show the matrix recurrence (on the

left) and the solution to that recurrence that would be obtained by a typical off-the-shelf recurrence solver (on the right, with i being the imaginary unit).

$$\begin{bmatrix} \mathsf{x}^{[k+1]} \\ \mathsf{y}^{[k+1]} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} \mathsf{x}^{[k]} \\ \mathsf{y}^{[k]} \end{bmatrix} \qquad \begin{bmatrix} \mathsf{x}^{[k]} \\ \mathsf{y}^{[k]} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}((-i)^k + i^k) & \frac{i}{2}((-i)^k - i^k) \\ -\frac{i}{2}((-i)^k - i^k) & \frac{1}{2}((-i)^k + i^k) \end{bmatrix} \begin{bmatrix} \mathsf{x}^{[0]} \\ \mathsf{y}^{[0]} \end{bmatrix}$$

Complex exponentiation concisely represents the rotation performed by this loop, but the use of complex numbers creates a problem for the later steps of our analysis, as demonstrated below.

Suppose that we want to know whether `errorlabel` is reachable. We answer that question by using an SMT solver to check the satisfiability of the transition formula from program entry to `errorlabel`. Upon entry to the loop, `n` is initialized to 0 and the loop condition is "`n` < 1", so the loop will always execute exactly once. Thus, the value of `y` at the end of the loop is $\mathsf{y}^{[1]} = -\frac{i}{2}((-i)^1 - i^1)\mathsf{x}^{[0]} + \frac{1}{2}((-i)^1 - i^1)\mathsf{y}^{[0]} = -i * i$, and the relevant part of the formula for `y` is:

$$\mathsf{y}' = 1 \wedge \mathsf{y}' = -i * i. \tag{1}$$

If i is interpreted as the imaginary unit, then Eqn. (1) is satisfiable, which gives us the answer we expect: `errorlabel` is reachable. However, we would like to be able to use an off-the-shelf SMT solver that does not support complex numbers. Unfortunately, it is not sound to interpret Eqn. (1) in real arithmetic with i as a symbolic constant. In real arithmetic, we know that for all a, $\mathsf{a} * \mathsf{a} \geq 0$, and thus $\mathsf{y}' = -i * i \leq 0$, which is inconsistent with $\mathsf{y}' = 1$; thus, Eqn. (1) is unsatisfiable, which falsely suggests that `errorlabel` is unreachable.

To avoid problems of this kind, we developed a recurrence solver (described in §6) that is able to solve recurrences like the one in this example, and communicate the solutions to the rest of our analyzer without the use of complex numbers. As a principled way to handle the issue of communication, the paper introduces *implicitly interpreted functions* (IIFs) (§6.4). An IIF is a representation of the solution to a recurrence that would otherwise need to be expressed using complex or irrational numbers. Inside the recurrence solver, each IIF is associated with a term in the logic of the recurrence solver, which represents an *exact representation* of the function (and the recurrence solver is able to manipulate this exact representation). Outside the recurrence solver—e.g., in the wedge domain—an IIF is treated as an uninterpreted function.

The following example illustrates that IIFs retain enough information to prove some sophisticated program properties.

Example 2.4. Consider the program shown in Fig. 1(d), which illustrates how our analyzer is able to establish the absence of information flow (i.e., "non-interference"), using the self-composition technique of Barthe et al. [2004]. For procedure `fib`, we will assume that variable `n` is a low-security input, `high` is a high-security input, and the return value of `fib` is a low-security output. The information-flow property that we wish to establish is that `fib`'s return value is unaffected by the value passed in for `high`.

Secure information flow is not a safety property:[2] a safety property can be refuted by observing a finite trace of a program, whereas to refute secure information flow, one has to observe two finite traces. Barthe et al. show that secure information flow for a program P can be encoded as a safety property of a more complicated program that consists of P followed by P'—a second copy of P with its variables suitably renamed. By this means, secure information flow can be reduced to an assertion-checking problem in the program $P; P'$. They call this technique *self composition.*

In the program above, the self-composition technique is captured by procedure `main`, which uses two calls to `fib` rather than two copies of `fib`. In `main`, the statement `assume(n1 == n2)` ensures that the low-security inputs to the two calls on `fib` are arbitrary, but *equal.* The absence of any constraint on `high1` and `high2` means that they may have *different values* in the two calls to `fib`.

[2]Technically, we are referring to the so-called "termination-insensitive" secure-information-flow problem.

The goal is to prove the assertion on the last line of `main`, which claims that the low-security outputs `obs1` and `obs2` are equal, and hence unaffected by any difference in the values of high-security inputs `high1` and `high2`.

To convince yourself that the assertion does indeed hold, observe that the two while loops in `fib` both compute the $\mathtt{n}^{th}$ Fibonacci number, albeit in slightly different ways. Thus, because the values of `n1` and `n2` are equal, `obs1` and `obs2` will be equal—even though `high1` and `high2` might differ, which would cause different while loops to be executed during the two calls on `fib`.

The challenge in analyzing this example is that the return value of `fib` is a complicated function of its inputs. Classically, this function would be represented using irrational numbers: the return value of `fib(n, high)` is $\frac{1}{\sqrt{5}}((\frac{1+\sqrt{5}}{2})^{\mathtt{n}} - (\frac{1+\sqrt{5}}{2})^{\mathtt{n}})$. As in the previous example (Ex. 2.3), we avoid manipulation of non-rational numbers using IIFs.

In the program shown above, the analyzer summarizes the behaviors of the two while loops in `fib` using two IIFs. The two IIFs are associated with equal terms in the logic of the recurrence solver. Consequently, the wedge domain is able to prove the assertion because it sees it as a comparison of two identical uninterpreted functions.

Problem Statement. The foregoing examples illustrate several challenges that our work addresses:

(1) The star operator takes as input a logical representation of the body of loop. The loop body formula typically contains disjunctions, existentially quantified variables, and inequations, and so cannot be directly interpreted as a recurrence relation. Therefore, we need an automatic method for extracting recurrences that lie within the input language of our recurrence solver. In general, these recurrences may include non-linear relationships.

(2) Once we have a recurrence relation, we need to solve it. In general, an extracted recurrence may contain polynomial and exponential terms, and program variables may mutually depend on one another. We need to be able to compute a closed form that can be represented in the base logic of the program analyzer (see §6.3).

The analysis method presented in the paper addresses these challenges by the following means:

- Challenge (1) is addressed via a new abstract domain, called the *wedge domain* (§4), which encodes non-linear relationships by extending the abstract domain of polyhedra with new dimensions that represent non-linear terms. §5 presents an algorithm that uses the wedge abstract domain to extract recurrence relations from a loop body formula.
- Challenge (2) is addressed via a new recurrence solver that we developed, based on the discrete *operational calculus* of Berg [1967]. (See §6.)

The balancing act in our work is that there is a mutual dependence between the wedge domain and the recurrence solver: each places constraints on the other. In addressing challenges (1) and (2), we faced a situation that often comes up in system building, namely, it is not always possible to use an existing component as just a black box. Although methods for solving recurrences are known (see §8 for references and a discussion), the closed forms computed by these methods may involve irrational and complex numbers, which are not permitted in the base logic of our program analyzer. The desire to have IIFs for reasoning *outside* the recurrence solver required us to have some control over what goes on *inside* the recurrence solver, to identify—and return as part of the answer—terms that define IIFs.

3 BACKGROUND

3.1 Linear Real Arithmetic

Fix a set of variables $z_1, \ldots, z_n$. A **linear term** is an expression of the form $a_1 z_1 + \cdots + a_n z_n + b$, where each $a_i \in \mathbb{Q}$ and $b \in \mathbb{Q}$. We will typically write such a term as $\mathbf{a}\mathbf{z} + b$, where $\mathbf{a} = [a_1, \ldots, a_n]$ and

$\mathbf{z} = [z_1, ..., z_n]^t$ are vectors containing the term's rational coefficients and variables, respectively. A **linear inequality** is of the form $\mathbf{a}\mathbf{z} \geq b$. A system of linear inequalities $\mathbf{a}_1\mathbf{z} \geq b_1 \wedge \cdots \wedge \mathbf{a}_m\mathbf{z} \geq b_m$ can be written succinctly as $A\mathbf{z} \geq \mathbf{b}$, where A is a matrix whose rows are $\mathbf{a}_1, ..., \mathbf{a}_m$, and $\mathbf{b} = [b_1, ..., b_m]^t$. The set of points $\mathbf{z}$ such that $A\mathbf{z} \geq \mathbf{b}$ is a **(convex) polyhedron**.

3.2 (Non)linear Real Arithmetic with Uninterpreted Functions

Let $\Sigma = \langle F, ar \rangle$ be a functional first-order vocabulary, consisting of a set of function symbols F and a function $ar : F \to \mathbb{N}$ associating each function symbol with an arity. We define the syntax of a logical language that extends linear real arithmetic with the vocabulary Σ as follows:

$$s, t \in \mathit{Term}(\Sigma) ::= a \in \mathbb{Q} \mid v \in \mathrm{Var} \mid a \times t \mid s + t \mid \mathsf{f}(t_1, ..., t_{ar(\mathsf{f})})$$
$$\phi, \psi \in \mathit{Formula}(\Sigma) ::= t \geq 0 \mid t > 0 \mid \phi \wedge \psi \mid \phi \vee \psi \mid \exists v.\phi$$

A term of the form a, $s + t$, or $a \times t$ is called *linear* (with the rest called *non-linear*). We assume that Σ includes a vocabulary Σ_N of designated function symbols that represent multiplication, exponentiation, logarithm, modulus, and floor, which we will write using standard notation:

$$\mathit{Term}(\Sigma_N) ::= ... \mid s \times t \mid s^t \mid \log_s(t) \mid s \bmod t \mid \lfloor t \rfloor$$

We will make use of two different semantics for this logic. A **linear structure** $\mathfrak{L}$ consists of an interpretation of each function symbol $\mathsf{f} \in \Sigma$ as a function $\mathsf{f}^{\mathfrak{L}} : \mathbb{R}^{ar(f)} \to \mathbb{R}$; the semantics of numerals, addition, and numeral multiplication are the usual ones. We write $\mathfrak{L} \models_L \phi$ to denote that $\mathfrak{L}$ is a linear structure that satisfies the sentence ϕ; write $\phi \models_L \psi$ if every linear structure that satisfies the sentence ϕ also satisfies the sentence ψ; and write $\phi \equiv_L \psi$ if $\phi \models_L \psi$ and $\psi \models_L \phi$. A **non-linear structure** $\mathfrak{N}$ is a linear structure that interprets the designated function symbols in Σ_N with their usual interpretation on their domain of definition (i.e., $\times^{\mathfrak{N}}$ is multiplication, $\log_2^{\mathfrak{N}}$ is base-2 logarithm for positive arguments and arbitrary for non-positive arguments, etc). We use $\models_N$ and $\equiv_N$ to denote the non-linear analogues of $\models_L$ and $\equiv_L$. Obviously, $\phi \models_L \psi$ implies that $\phi \models_N \psi$, but the reverse does not hold. Note that membership of a pair of formulas in the relation $\models_L$ is decidable, but $\models_N$ is not.

We are particularly interested in formulas that represent transition relations of some fragment of a program. Formally, we suppose that Σ is a vocabulary that contains a designated set $\mathrm{PVar} = \{\mathsf{x}_1, ..., \mathsf{x}_n\}$ of constant symbols and also a disjoint set $\mathrm{PVar}' = \{\mathsf{x}'_1, ..., \mathsf{x}'_n\}$ of primed copies, representing the values of program variables before and after a transition. A formula ϕ's **concretization** $\gamma_N(\phi)$ is the transition relation defined by its *non-linear* models:

$$\gamma_N(\phi) \stackrel{\text{def}}{=} \{\langle \mathbf{a}, \mathbf{a}' \rangle : \exists \mathfrak{N}.\mathfrak{N} \models_N \phi, a_1 = \mathsf{x}_1^{\mathfrak{N}}, ..., a'_n = \mathsf{x}_n'^{\mathfrak{N}}\}.$$

3.3 Polynomials, Ideals, and Gröbner Bases

We use $\mathbb{Q}[z_1, ..., z_n]$ to denote the ring of polynomials with rational coefficients over the variables $z_1, ..., z_n$. An **ideal** is a set $I \subseteq \mathbb{Q}[z_1, ..., z_n]$ of polynomials over the variables $z_1, ..., z_n$ that contains 0, is closed under addition, and such that for any $p \in I$ and any $q \in \mathbb{Q}[z_1, ..., z_n]$, we have $pq \in I$. Intuitively, one may think of an ideal I as a set of polynomial equations $\{p = 0 : p \in I\}$—the closure conditions for ideals can be read as simple inference rules: $0 = 0$, if $p = 0$ and $q = 0$ then $p + q = 0$, and if $p = 0$ then $pq = 0$ for any q. Any set of polynomials $P \subseteq \mathbb{Q}[z_1, ..., z_n]$ *generates* an ideal $\langle P \rangle$, which is the smallest ideal containing P:

$$\langle P \rangle \stackrel{\text{def}}{=} \left\{ \sum_{i=0}^{k} q_i p_i : k \geq 0, q_i \in \mathbb{Q}[z_1, ..., z_n], p_i \in P \right\}$$

The set of polynomials P is called a **basis** for $\langle P \rangle$.

A Gröbner basis for an ideal is a particular kind of basis that has many computational applications. For a good introduction to Gröbner basis theory, see [Cox et al. 2015]. In the remainder of this section we give a short overview of the properties of Gröbner bases that are needed for the rest of this paper.

Fix a set of variables $z_1, \ldots, z_n$. A **monomial** is a product of variables $m = z_1^{d_1} \cdots z_n^{d_n}$. The sum $d_1 + \cdots + d_n$ is called the **total degree** of m. A **monomial ordering** $\preceq$ is a total ordering on monomials such that for any $m \preceq n$ and any monomial v, we have $mv \preceq nv$ (some important examples to follow). The **leading term** of a polynomial $a_1 m_1 + \cdots + a_n m_k$ is the greatest monomial among $m_1, \ldots, m_k$ (with respect to a given monomial ordering).

A monomial ordering gives us a way of orienting a polynomial equation $p = 0$ into a rewrite rule $m \rightarrow q$, where m is the leading term of p and $p = am - aq$ for some nonzero $a \in \mathbb{Q}$. Fixing a monomial ordering, we may think of a set of polynomials as a rewrite system. A **Gröbner basis G** is a set of polynomials that results in a *confluent* (and terminating) rewrite system—every polynomial can be rewritten to a unique normal form. There is a procedure (Buchberger's algorithm [Buchberger 1976], among others) that computes a Gröbner basis for any ideal under any monomial ordering. Given a Gröbner basis $\mathbf{G}$ for an ideal I we use $red_{\mathbf{G}} : \mathbb{Q}[z_1, \ldots, z_n] \rightarrow \mathbb{Q}[z_1, \ldots, z_n]$ to denote the function that maps polynomials to their normal forms. Two important properties of the reduction function are:

(1) $p \in I$ if and only if $red_{\mathbf{G}}(p) = 0$, and
(2) The leading term of $red_{\mathbf{G}}(p)$ is $\preceq$ the leading term of p.

One monomial ordering of interest is the *degrevlex* ("degree reverse lexicographic") ordering, which compares monomials first by total degree and then by reverse lexicographic order (the reverse lexicographic part is not essential for our applications—any total order that extends degree ordering will do). Property 2 above implies that if $\mathbf{G}$ is a Gröbner basis with respect to degrevlex order, then the degree of $red_{\mathbf{G}}(p)$ never exceeds the degree of p.

Gröbner bases can also be used to project variables out of polynomial systems (similarly to Gaussian elimination for linear systems). Fix a set of variables $X \subseteq \{z_1, \ldots, z_n\}$ that we wish to eliminate. Any monomial can be written as a product $m = m_X m_{\overline{X}}$, where m_X is a monomial over X and $m_{\overline{X}}$ is a monomial over the remaining variables. We define the *elimination order* w.r.t X as follows: $m \preceq_X n$ if $m_X \preceq n_X$ or if $m_X = n_X$ and $m_{\overline{X}} \preceq n_{\overline{X}}$, where $\preceq$ denotes degrevlex order. Property 2 above implies that if $\mathbf{G}$ is a Gröbner basis for an ideal I under the elimination order w.r.t. X, then if p is free of X variables then so is $red_{\mathbf{G}}(p)$. Considering $red_{\mathbf{G}}(p)$ as the canonical representative of the equivalence class of polynomials that reduce to $red_{\mathbf{G}}(p)$, the latter property means that if any polynomial in the equivalence class is free of X variables (albeit there may be no such polynomial), then so is the representative.

3.4 Reasoning About Non-Linear Arithmetic

In this paper, we use a variety of techniques to reason about non-linear arithmetic. The following is a road map that outlines the techniques used in each section. Within the wedge domain (§4), we use Gröbner bases and congruence closure to reason about non-linear equations, and inference rules to reason about non-linear inequations. In §5, we use Gröbner bases to extract non-linear recurrence relations from the body of a loop (represented by a wedge). In §6, we present a recurrence solver that can solve recurrences involving polynomials and exponentials, and which uses implicitly interpreted functions (IIFs) to represent closed forms for recurrences that would otherwise require exponentiation of irrational or complex numbers.

As demonstrated in Ex. 2.2, logarithmic relationships are obtained as derived information from exponential relationships (cf. Fig. 3).

4 THE WEDGE ABSTRACT DOMAIN

The wedge abstract domain is a numerical abstract domain that can express properties involving linear and non-linear arithmetic as well as uninterpreted function symbols. Fixing a vocabulary Σ, a Σ-wedge is a simply a conjunction of ground atomic Σ-formulas:

$$w \in \mathit{Wedge}(\Sigma) ::= t \geq 0 \mid t > 0 \mid w_1 \wedge w_2 \quad \text{where } t \in \mathit{Term}(\Sigma) \text{ and has no free variables (§3.2)}$$

Note that $t = 0$ can be expressed as $t \geq 0 \wedge -t \geq 0$. Semantically, we are typically interested in viewing a wedge as a transition relation—that is, the natural concretization function is γ_N. The concretization function γ_N induces an approximation pre-order on wedges: $w \sqsubseteq_N w'$ iff $\gamma_N(w) \subseteq \gamma_N(w')$. However, this pre-order is not effective, necessitating the use of a linear variation: $w \sqsubseteq_L w'$ iff $w \models_L w'$. The wedge domain's ability to reason about non-linear operations stems from *strengthening* operations (§4.1) that saturate a wedge with properties that hold in all of its non-linear models.

Operations for manipulating wedges are based on viewing a wedge as a set of points in a real space with one coordinate corresponding to each term. This view is formalized in the following.

Definition 4.1. A Σ-**coordinate system** is a list of Σ-terms $\tau = [t_1, ..., t_n]$ such that:

(1) Each t_i is a term of the form $\mathsf{f}(t_1, ..., t_n)$ for some $\mathsf{f} \in \Sigma$ with no free variables. (Note that this definition includes terms of the form x for a constant symbol $\mathsf{x} \in \Sigma$). Such terms are called *application terms.*
(2) There are no duplicates.
(3) For any i, for any application sub-term s of t_i, we must have $s = t_j$ for some $j < i$.

We say that τ **admits** a term t if all of t's application sub-terms belong to τ (including t itself, if t is an application term). We say that τ admits a wedge if it admits all the terms of that wedge. We say that τ is a **minimal** coordinate system for a wedge w if τ admits w and no proper sub-list of τ admits w.

Given a coordinate system $\tau = [t_1, ..., t_n]$, we define a *linearization* function lin_τ that expresses admissible terms and wedges in linear arithmetic over the set of variables $\{z_1, ..., z_n\}$, where variable z_i denotes the ith coordinate of $\mathbb{R}^n$ (representing the term t_i).

$$\begin{aligned} \mathit{lin}_\tau(a) &\stackrel{\text{def}}{=} a, \text{ if } a \in \mathbb{Q} \\ \mathit{lin}_\tau(t_i) &\stackrel{\text{def}}{=} z_i \\ \mathit{lin}_\tau(u_1 + u_2) &\stackrel{\text{def}}{=} \mathit{lin}_\tau(u_1) + \mathit{lin}_\tau(u_2) \\ \mathit{lin}_\tau(k \times u) &\stackrel{\text{def}}{=} k \times \mathit{lin}_\tau(u), \text{ if } k \in \mathbb{Q} \end{aligned} \qquad \begin{aligned} \mathit{lin}_\tau(u \geq 0) &\stackrel{\text{def}}{=} \mathit{lin}_\tau(u) \geq 0 \\ \mathit{lin}_\tau(u > 0) &\stackrel{\text{def}}{=} \mathit{lin}_\tau(u) > 0 \\ \mathit{lin}_\tau(w_1 \wedge w_2) &\stackrel{\text{def}}{=} \mathit{lin}_\tau(w_1) \wedge \mathit{lin}_\tau(w_2) \end{aligned}$$

Note that for a wedge w, $\mathit{lin}_\tau(w)$ defines a convex polyhedron, which we call the **underlying polyhedron** of w.

We also define a reverse operation, interp_τ, which maps a linear term over variables $\{z_1, ..., z_n\}$ into a term:

$$\mathit{interp}_\tau(a_1 z_1 + \cdots + a_n z_n + b) \stackrel{\text{def}}{=} a_1 t_1 + \cdots + a_n t_n + b.$$

For any term t and any coordinate system τ that admits t, true $\models_L \mathit{interp}_\tau(\mathit{lin}_\tau(t)) = t$ holds.

Example 4.2. A minimal coordinate system that admits the wedge $w \stackrel{\text{def}}{=} 2\mathsf{x}^2 - \mathsf{y} = 0 \wedge \mathsf{y} - 3\mathsf{x} + \mathsf{y} \geq 0$ is $\tau = [\mathsf{x}, \mathsf{y}, \mathsf{x}^2]$. The linearization of w is the polyhedron $\mathit{lin}_\tau(w) = -z_2 + 2z_3 = 0 \wedge -3z_1 + 2z_2 \geq 0$.

4.1 Strengthening

Let w be a wedge. A **strengthening** of w is any wedge w' such that $w \equiv_N w'$ and $w' \models_L w$. For example, a strengthening of the wedge $\mathsf{x} = \mathsf{y}+1 \wedge \mathsf{y}^2 \geq \mathsf{x}^2$ is $\mathsf{x} = \mathsf{y}+1 \wedge \mathsf{x}^2 \geq 0 \wedge \mathsf{x}^2 = \mathsf{y}^2+2\mathsf{y}+1 \wedge \mathsf{y} \geq 0$. In the terminology of abstract interpretation, strengthening a wedge amounts to applying a sequence

of reductive operators, each of which over-approximates the lower closure operator induced by the Galois insertion of the powerset lattice of non-linear models into the powerset lattice of linear models [Granger 1992]. This section presents techniques for strengthening wedges, presented in two parts: first, inferring implied equalities; second, inferring implied inequalities.

4.1.1 Equalities. Let $\tau = [t_1, ..., t_n]$ be a coordinate system and let w be an admissible wedge of τ. Let P denote the underlying polyhedron of w, and let $\mathit{aff}(P)$ denote a basis for the vector space

$$\left\{ \begin{bmatrix} b_0 & b_1 & \dots & b_n \end{bmatrix} : \forall \begin{bmatrix} a_1 & \dots & a_n \end{bmatrix} \in P . b_1 a_1 + \cdots + b_n a_n + b_0 = 0 \right\} .$$

$\mathit{aff}(P)$ is a representation of the set of affine equations that are satisfied by all points in P; it can be computed easily from a constraint representation of P. Define $\mathbb{I}(w, \tau)$ to be the ideal generated by $\mathit{aff}(P)$ and the definitional equalities encoding multiplication and reciprocal coordinates:

$$\mathbb{I}(w, \tau) \stackrel{\text{def}}{=} \left\langle \begin{array}{l} \{b_1 z_1 + \ldots + b_n z_n + b_0 : \begin{bmatrix} b_0 & b_1 & \dots & b_n \end{bmatrix} \in \mathit{aff}(P)\} \\ \cup \{z_i - \mathit{lin}_\tau(s)\mathit{lin}_\tau(s') : t_i = s \times s'\} \\ \cup \{\mathit{lin}_\tau(s) z_i - 1 : t_i = s^{-1} \land w \models_L \neg(s = 0)\} \end{array} \right\rangle$$

For any ideal $I \subseteq \mathbb{Q}[z_1, ..., z_n]$ and admissible terms s and t, we write $I \models s = t$ iff $(\mathit{lin}_\tau(s) - \mathit{lin}_\tau(t)) \in I$. We say that I is *congruence closed* if for all terms $t_i = \mathsf{f}(s_1, ..., s_m)$ and $t_j = \mathsf{f}(s'_1, ..., s'_m)$ of τ such that for all i we have $I \models s_1 = s'_1, ..., I \models s_m = s'_m$ then we have $I \models \mathsf{f}(s_1, ..., s_m) = \mathsf{f}(s'_1, ..., s'_m)$. The congruence closure of an ideal I is the smallest congruence-closed ideal that contains I. We say that a wedge w is *equationally saturated in* τ if for every degree-1 polynomial $a_1 z_1 + \ldots + a_n z_n + b$ in the congruence closure of the ideal $\mathbb{I}(w, \tau)$ vanishes on the underlying polyhedron of w. The equational saturation of w is the unique (up to $\equiv_L$) equationally saturated wedge w' such that $w \equiv_N w'$ and for all equationally saturated w'' such that $w \equiv_N w''$ we have $w'' \models_L w'$. A procedure for computing the equational saturation of a wedge is given as Algorithm 1.

Example 4.3. Consider the wedge $w = (\mathsf{x} \times 2^{\mathsf{x}} = \mathsf{v} \land \mathsf{v} \leq (2^{\mathsf{y}} + 1) \times \mathsf{y} \land \mathsf{x} = \mathsf{y} \land \mathsf{x} \leq 0)$ under the coordinate system $\tau = [\mathsf{v}, \mathsf{x}, \mathsf{y}, 2^{\mathsf{x}}, \mathsf{x} \times 2^{\mathsf{x}}, 2^{\mathsf{y}}, (2^{\mathsf{y}} + 1) \times \mathsf{y}]$. The association between terms and coordinates (defined by τ) is as follows:

$z_1 : \mathsf{v}$ $\quad z_2 : \mathsf{x}$ $\quad z_3 : \mathsf{y}$ $\quad z_4 : 2^{\mathsf{x}}$ $\quad z_5 : \mathsf{x} \times 2^{\mathsf{x}}$ $\quad z_6 : 2^{\mathsf{y}}$ $\quad z_7 : (2^{\mathsf{y}} + 1) \times \mathsf{y}$

The underlying polyhedron P of w is $z_5 = z_1 \land z_1 \leq z_7 \land z_2 = z_3 \land z_2 \leq 0$, and the ideal $\mathbb{I}(w, \tau)$ is:

$$\mathbb{I}(w, \tau) \stackrel{\text{def}}{=} \langle \underbrace{z_5 - z_1, z_2 - z_3}_{\mathit{aff}(P)}, \underbrace{z_5 - z_2 z_4, z_7 - (z_6 + 1) z_3}_{\text{Definitional equalities}} \rangle .$$

We will now illustrate Algorithm 1 on the wedge w. First we compute a (degrevlex) Gröbner basis for $\mathbb{I}(w, \tau)$: $\mathbf{G} = \{z_5 - z_1, z_3 - z_2, z_2 z_4 - z_1, z_2 z_6 - z_7 + z_2, z_1 z_6 - z_4 z_7 + z_2 z_4\}$ and set the active coordinates to be $\{4, 6\}$, the ones corresponding to the terms 2^{x} and 2^{y}.

First iteration of the outer loop: When processing coordinate 4 in the inner loop, we add the mapping $CC[2^{z_2}] := z_4$ to CC, indicating that the representative for the term 2^{z_2} is z_4. We then process coordinate 6, and find that the term $(\mathit{red}_{\mathbf{G}}(2))^{\mathit{red}_{\mathbf{G}}(z_3)} = 2^{z_2}$ already has a representative (z_4), and add $z_6 - z_4$ to B (i.e., we conclude from the fact that $\mathsf{x} = \mathsf{y}$ that $2^{\mathsf{x}} = 2^{\mathsf{y}}$). We add the equation $2^{\mathsf{y}} - 2^{\mathsf{x}} = 0$ to w and compute a Gröbner basis for the ideal generated by $\mathbf{G}$ and B: $\mathbf{G} = \{z_5 - z_1, z_3 - z_2, z_6 - z_4, z_2 z_4 - z_1, z_7 - z_2 - z_1\}$. Since B contains a non-trivial polynomial, namely $z_6 - z_4$, we continue to a second iteration.

Second iteration of the outer loop: Since the two application terms 2^{x} and 2^{y} were identified in the previous iteration, the inner loop does not produce any new equations. However, the Gröbner basis $\mathbf{G}$ changed in the previous iteration (in particular, it now contains $z_7 - z_2 - z_1$). Since $\mathit{red}_{\mathbf{G}}(z_7) = z_2 + z_1$, we add the equation $(2^{\mathsf{y}} + 1) \times \mathsf{y} = \mathsf{v} + \mathsf{x}$ to w (line 17). Since w contains the inequation $\mathsf{v} \leq (2^{\mathsf{y}} + 1) \times \mathsf{y}$ and (now) the equation $(2^{\mathsf{y}} + 1) \times \mathsf{y} = \mathsf{v} + \mathsf{x}$, we have $\mathsf{v} \leq \mathsf{v} + \mathsf{x}$ and thus $0 \leq \mathsf{x}$; since w also contains $\mathsf{x} \leq 0$, we have $\mathsf{x} = 0$. Thus, the underlying polyhedron of w is $z_5 = z_1 \land z_1 \leq z_7 \land z_2 = z_3 \land z_2 = 0$,

```
     Input   : Coordinate system τ = [t_1, ..., t_n] and a wedge w
     Output : Equational saturation of w in τ
 1   G ← Gröbner basis (degrevlex) for 𝕀(w, τ);
 2   active ← {i : t_i is an application};
 3   repeat
 4   |   B ← {0};
     |   /* Compute congruence closure. CC maps terms to representative coordinates   */
 5   |   CC ← empty map;
 6   |   foreach i in active do
 7   |   |   Let t_i = f(s_1, ..., s_m);
 8   |   |   Let r_j ← red_G(lin_τ(s_i)) for all j;
 9   |   |   if CC[f(r_1, ..., r_n)] is defined then
10   |   |   |   z_r ← CC[f(r_1, ..., r_n)];
11   |   |   |   B ← B ∪ {red_G(z_i − z_r)};
     |   |   |   /* i is represented by z_r - it need not be processed again              */
12   |   |   |   Remove i from active;
13   |   |   else
14   |   |   |   CC[f(r_1, ..., r_n)] := z_i;
15   |   |   end
16   |   end
     |   /* Add implied linear equations to w. Note that since G is computed w.r.t
     |      degrevlex, the degree of red_G(z_i) must be ≤ 1 (i.e., it's linear)          */
17   |   w ← w ∧ ( ∧_{i=1}^{n} interp_τ(z_i = red_G(z_i)) ) ∧ ( ∧_{b∈B} interp_τ(b = 0) );
18   |   B ← B ∪ {red_G(b) : b ∈ basis for 𝕀(w, τ)};
19   |   G ← Gröbner basis (degrevlex) for ⟨G ∪ B⟩;
20   until B = {0};
21   return w
```

Algorithm 1: Equational saturation

and z_2 belongs to $\mathbb{I}(w, \tau)$. We add z_2 to B and compute a Gröbner basis for the ideal generated by $\mathbf{G}$ and B: $\mathbf{G} = \{z_1, z_2, z_3, z_7, z_6 - z_4\}$. Since B contains a non-trivial polynomial (z_2), we continue to a third iteration.

Third iteration of the outer loop: Again the inner loop does not produce any new equations. On line 17 we derive the equations $\mathsf{v} = 0$, $\mathsf{x} = 0$, $\mathsf{y} = 0$, $\mathsf{x} \times 2^{\mathsf{x}} = 0$, $(2^{\mathsf{y}} + 1) \times \mathsf{y} = 0$, due to the fact that $\mathbf{G}$ reduces z_1, z_2, z_3, z_5, and z_7 to 0. Since $\mathbf{G}$ reduces every basis polynomial in $\mathbb{I}(w, \tau)$ to 0, the loop exits and the algorithm returns the following (equationally saturated) wedge:

$$\mathsf{v} = \mathsf{x} = \mathsf{y} = \mathsf{x} \times 2^{\mathsf{x}} = (2^{\mathsf{y}} + 1) \times \mathsf{y} = 0 \wedge 2^{\mathsf{x}} = 2^{\mathsf{y}} .$$

4.1.2 Inequalities. Let $\tau = [t_1, ..., t_n]$ be a coordinate system, let w be an equationally saturated admissible wedge of τ, and let $\mathbf{G}$ be the Gröbner basis of $\mathbb{I}(w, \tau)$ (w.r.t degrevlex order). Strengthening operations for inferring inequalities that hold in all non-linear models of w are presented as a set of inference rules in Fig. 3. The hypotheses and consequences are expressed as polynomial inequalities in the coordinates $z_1, ..., z_n$. A rule may only be applied if hypotheses and consequences are linear and admissible. Each appearance of $lin_\tau(t)$ is implicitly guarded by the assumption that τ admits t (so for example, the Floor rule is only applied to floor terms that appear in τ).

The Interval rule assumes that each function f is associated with a monotone function f^{ivl} that approximates f on intervals (e.g., the interval approximation of multiplication is $[a, b] \times^{ivl} [c, d] \stackrel{\text{def}}{=}$

INTERVAL
$$\frac{lin_\tau(s_1) \in [a_1, b_1] \quad \dots \quad lin_\tau(s_m) \in [a_m, b_m]}{lin_\tau(\mathtt{f}(s_1, ..., s_m)) \in \mathtt{f}^{ivl}([a_1, b_1], ..., [a_n, b_n])}$$

PRODUCT
$$\frac{lin_\tau(s) \geq 0 \qquad lin_\tau(t) \geq 0}{red_G(lin_\tau(s) lin_\tau(t)) \geq 0}$$

SQUARE
$$\frac{}{red_G(z_i z_i) \geq 0}$$

POWLOGLOWER
$$\frac{lin_\tau(ab^t) \geq lin_\tau(c) \quad lin_\tau(c) > 0 \qquad lin_\tau(b) > 1}{lin_\tau(\log_b(a)) + lin_\tau(t) \geq lin_\tau(\log_b(c))}$$

MOD
$$\frac{t \geq 0}{0 \leq lin_\tau(mod(s, t)) \leq lin_\tau(t) - 1}$$

POWLOGUPPER
$$\frac{lin_\tau(c) \geq lin_\tau(ab^t) \quad lin_\tau(a) > 0 \qquad lin_\tau(b) > 1}{lin_\tau(\log_b(c)) \geq lin_\tau(\log_b(a)) + lin_\tau(t)}$$

FLOOR
$$\frac{}{lin_\tau(s) - 1 < lin_\tau(\lfloor s \rfloor) \leq lin_\tau(s)}$$

Fig. 3. Inference rules for implied inequalities

Input : Σ-wedge w and sub-vocabulary $\Sigma' \subseteq \Sigma$
Output: Σ'-wedge w' with $w \models_N w'$

1 $\tau = [t_1, ..., t_n] \leftarrow$ minimal coord. system admitting w;
2 $safe \leftarrow$ empty map;
3 **for** $i = 1$ *to* n **do**
4   **if** $t_i \in Term(\Sigma')$ **then**
5         $safe[i] \leftarrow t_i$;
6   **end**
7 **end**
8 **repeat**
9     $\{i_1, ..., i_k\} \leftarrow \text{dom}(safe)$;
10     $unsafe \leftarrow \{z_i : i \in \{1, ..., n\} \setminus \text{dom}(safe)\}$;
11   G $\leftarrow$ Gröbner basis (elim. order $\preceq_{unsafe}$) for $\mathbb{I}(w, \tau)$;
12   **for** $i \in unsafe$ **do**
13   Let $t_i = \mathtt{f}(s_1, ..., s_m)$;
14   Let $r_j \leftarrow red_G(lin_\tau(s_j))$ for all j;
15   **if** $f \in \Sigma'$ *and* $r_j \in \mathbb{Q}[z_{i_1}, ..., z_{i_k}]$ *for all* j **then**
16             $s'_j \leftarrow r_j$ with each z_{i_ℓ} replaced by $safe[i_\ell]$;
17             $safe[i] \leftarrow \mathtt{f}(s'_1, ..., s'_m)$
18   **end**
19   **end**
20 **until** $dom(safe) \cap unsafe = \emptyset$;
`// Polyhedral projection`
21 $P \leftarrow project(lin_\tau(w), \{z_i : i \in \text{dom}(safe)\})$;
22 Let $A\mathbf{z} \geq \mathbf{b}$ be a constraint representation of P;
23 $w' \leftarrow A\mathbf{z} \geq \mathbf{b}$ with each z_i replaced by $safe[i]$;
24 **return** w

Algorithm 2: Projection

$[\min\{ac, ad, bc, bd\}, \max\{ac, ad, bc, bd\}])$. The interval constraints in the hypothesis of the rule are computed using linear programming.

Unlike the equational inference rules, which we can apply until the wedge is saturated, the inequality inference rules can be applied indefinitely without ever converging. To enforce termination, we use a simple heuristic: we apply each inference rule in sequence (first we apply the INTERVAL rule for every term, then the PRODUCT rule for every pair of inequalities, ...), and do not use the consequence of one application of a rule as a hypothesis for the next (e.g., if an inequality was derived via the PRODUCT rule, it may not later be used as the hypothesis of another application of the PRODUCT rule). As a consequence, our algorithm for strengthening a wedge is not complete with respect to the inference rules (i.e., there are inequations that can be deduced by the inference rules that will not be be deduced by a single run of the strengthening algorithm).

4.2 Basic Operations on Wedges

We now define some basic operations for manipulating wedges: pseudo-join, meet, widening, and projection.

Let w_1 and w_2 be wedges. The **pseudo-join** of w_1 and w_2 is a wedge that is an upper bound of w_1 and w_2 with respect to the ordering $\models_N$ (it is not, however, a *least* upper bound with respect to $\models_N$, which is not computable). We compute the pseudo-join $w_1 \widetilde{\sqcup} w_2$ by first finding a minimal

coordinate system τ that admits both w_1 and w_2, strengthening both within τ to get wedges w'_1 and w'_2, and setting

$$w_1 \mathbin{\widetilde{\sqcup}} w_2 \overset{\text{def}}{=} interp_\tau(lin_\tau(w'_1) \sqcup_L lin_\tau(w'_2))$$

where $\sqcup_L$ denotes polyhedral join. Observe that due to the strengthening operation, $w_1 \mathbin{\widetilde{\sqcup}} w_2$ is not necessarily an upper bound with respect to $\models_L$.

Example 4.4. Consider the wedges $y \geq 1 \land y^2 \geq x$ and $0 \geq x$. A coordinate system admitting both wedges is $[x, y, y^2]$. Strengthening $y \geq 1 \land y^2 \geq x$ yields $y \geq 1 \land y^2 \geq x \land y^2 \geq y$, and strengthening $0 \geq x$ yields $0 \geq x \land y^2 \geq 0$. The pseudo-join is $y^2 \geq x \land y^2 \geq 0$. Notice that $0 \geq x \not\models_L y^2 \geq x \land y^2 \geq 0$.

Unlike the least upper bound, greatest lower bounds of wedges are exact and easily computed. The **meet** $w_1 \sqcap w_2$ is simply the conjunction: $w_1 \sqcap w_2 \overset{\text{def}}{=} w_1 \land w_2$.

Widening. In contrast to conventional methods, the program analysis proposed in this paper makes no use of widening in the analysis of loops: a loop is analyzed by extracting recurrences and computing closed forms. However, the interprocedural framework for compositional recurrence analysis does make use of widening when analyzing functions with non-linear recursion (more than one recursive call on some path through the function) [Kincaid et al. 2017]. Thus for completeness, we will define the widening operator for wedges.

Like the pseudo-join operation, the widening operation of wedges relies on the operations of their underlying polyhedra. Unlike pseudo-join, the dimension of the coordinate system must *decrease* rather than *increase*. Let τ_1 and τ_2 be minimal coordinate systems admitting w_1 and w_2, respectively. Compute a coordinate system τ that contains only the terms that are common to both τ_1 and τ_2—w.l.o.g. assume that it is a prefix of both τ_1 and τ_2. We define $w_1 \triangledown w_2$ as

$$w_1 \triangledown w_2 \overset{\text{def}}{=} interp_\tau(project_L(lin_{\tau_1}(w_1), |\tau|) \triangledown_L project_L(lin_{\tau_2}(w_2), |\tau|))$$

where $project_L(P, n)$ denotes the projection of a polyhedron onto its first n dimensions and $\triangledown_L$ denotes polyhedral widening.

Projection. Let Σ' be a sub-vocabulary of Σ. The result of *projecting* a Σ-wedge w onto Σ' is a Σ'-wedge w' such that $w \models_N w'$. While sometimes we wish to eliminate function symbols of arity ≥ 1, most commonly we are interested in the case where Σ and Σ' differ by a set X of constant symbols, in which case projecting onto Σ' results in an over-approximation of the formula $(\exists X.w)$.

A projection algorithm is given as Algorithm 2. It operates as follows. Let w be a Σ-wedge and let $\tau = [t_1, ..., t_n]$ be a minimal coordinate assignment admitting w. If t_i is a Σ'-term, we mark i as *safe* and associate i with t_i; otherwise, it is *unsafe*. We then compute the Gröbner basis for the ideal $\mathbb{I}(w, \tau)$ w.r.t. an an elimination ordering for $\{z_i : t_i \text{ is unsafe}\}$. We then traverse the *unsafe* terms $t_i = \mathsf{f}(s_1, ..., s_m)$: if every coordinate in $red_G(lin_\tau(s_j))$ is safe for all s_j, we mark i safe and associate it with an equivalent Σ'-term. We iterate this process until no more coordinates are marked safe. We then use polyhedral projection to eliminate the unsafe dimensions, and then replace the safe dimensions with their associated Σ'-terms to obtain a Σ' wedge w'.

4.3 Symbolic Abstraction

Symbolic abstraction is the key operation supported by the wedge domain, and the foundation of our method for extracting recurrence relations from transition formulas (see §5). Given a formula ϕ, symbolic abstraction computes a *wedge* w that over-approximates it (that is, $\phi \models_N w$) and is *as precise as possible* (noting that the latter part of the specification is informal, since the *most* precise wedge is not necessarily computable or even well defined). Phrased differently, symbolic abstraction computes a system of equations and inequations that are implied by a given formula.

Let ϕ be a formula that we would like to over-approximate by a wedge. If ϕ happens to be a purely conjunctive formula then it is *already* a wedge, and we may simply strengthen (§4.1) and return it. In general, ϕ may contain disjunctions and existential quantifiers; the necessary ingredients for treating both were given in the previous section. In principle, one may compute a wedge w that over-approximates ϕ as follows (1) Skolemize ϕ and rewrite in disjunctive normal form (DNF) $\phi \equiv w_1 \vee \ldots \vee w_n$, (2) for each disjunct w_i, project out Skolem constants (noting that each disjunct is a wedge), resulting in a wedge w'_i, and (3) take the pseudo-join of all resulting wedges $w \stackrel{\text{def}}{=} w'_1 \,\widetilde{\sqcup} \cdots \widetilde{\sqcup}\, w'_n$. While this is a relatively straightforward procedure, it is worth noting that the rest of the operations of the wedge domain were designed specifically to support it. A faster algorithm that computes the same wedge but can often avoid the worst-case behavior of DNF conversion is given as Algorithm 3.

The symbolic-abstraction procedure assumes that the wedge projection and pseudo-join operations emit justifying axioms. More precisely, *project*(w, Σ') returns a pair $\langle w', A\rangle$ consisting of a Σ'-wedge and a formula A such that true $\models_N A$ and $A \wedge w \models_L w'$. Similarly, $w_1 \,\widetilde{\sqcup}\, w_2$ returns a pair $\langle w, A\rangle$ consisting of a wedge w and a formula A such that $(w_1 \vee w_2) \wedge A \models_L w$. Computing this extra information is straightforward: For the inferred inequalities, we track the inference rules that are applied during the course of the operation. For the inferred equalities, we must keep track of every non-trivial equation that is added to the Gröbner basis ("asserted equalities"), and whenever a non-trivial equation is added to the polyhedron P ("derived equality"), we emit the implication that the asserted equalities imply the derived equality.

Input : Formula ϕ over vocabulary Σ
Output: Wedge w over vocabulary Σ such that $\phi \models_N w$
1. $w \leftarrow$ false;
2. $F \leftarrow$ Skolemize ϕ;
3. $R \leftarrow F$;
4. **while** *there exists a model* $\mathfrak{L} \models_L R$ **do**
5. Let w_I be an implicant of F s.t. $\mathfrak{L} \models_L w_I$;
   `// Wedge projection: remove Skolem constants`
6. $\langle w'_I, A_\exists\rangle \leftarrow$ *project*(w_I, Σ);
   `// Wedge pseudo-join`
7. $\langle w, A_\sqcup\rangle \leftarrow w \,\widetilde{\sqcup}\, w'_I$;
8. $R \leftarrow R \wedge A_\exists \wedge A_\sqcup \wedge \neg w$;
9. **end**
10. **return** w

Algorithm 3: Symbolic abstraction.

5 EXTRACTING RECURRENCES

The fundamental question of interest in this paper is how to compute an over-approximation of the transitive closure of a transition formula. As in CRA [Farzan and Kincaid 2015], our approach (1) computes a system of recurrence relations that are semantically entailed by the input transition formula, and (2) finds a closed-form representation of those recurrences. In this section, we give a method for extracting recurrences from a transition formula that is based on the wedge domain (§4). A description of how this method goes beyond that of Farzan and Kincaid [2015] is given in §8.

In §6, we describe a method for solving recurrence equations of the form

$$\begin{bmatrix} y_1^{[n]} \\ \vdots \\ y_k^{[n]} \end{bmatrix} = A \begin{bmatrix} y_1^{[n-1]} \\ \vdots \\ y_k^{[n-1]} \end{bmatrix} + \begin{bmatrix} r_1 \\ \vdots \\ r_k \end{bmatrix}, \tag{2}$$

(or more succinctly, $\mathbf{y}^{[n]} = A\mathbf{y}^{[n-1]} + \mathbf{r}(n)$), where $A \in \mathbb{Q}^{k\times k}$ and each r_i is a term of the form:

$$r \in \mathit{RecTerm}(K, F, n) ::= a \times r \mid \mathsf{x} \times r \mid r_1 + r_2 \mid \mathsf{f}(n) \mid u \qquad a \in \mathbb{Q}, \mathsf{x} \in K, \mathsf{f} \in F$$

$$u \in \mathit{MultRecTerm}(K, n) ::= a \mid \mathsf{x} \mid n \mid a^n \mid u_1 + u_2 \mid u_1 \times u_2 \qquad a \in \mathbb{Q}, \mathsf{x} \in K$$

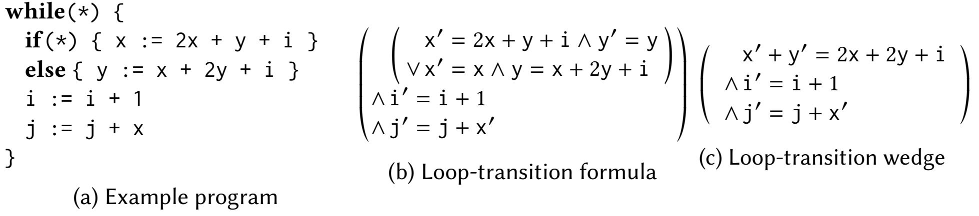

Fig. 4. A loop, along with its transition formula and transition wedge

where K is a set of symbolic constants (initial values of program variables) and F is a set of implicitly interpreted functions (IIFs). The closed forms computed by the recurrence solver have the same form as *RecTerm* above.

In general, the behavior of a loop cannot be expressed as such a system of recurrence equations: loops may have non-deterministic assignments, conditional branches, nested loops, etc. The question is how we can exploit a solver for recurrences of this form, even though the loop itself does not have such a description. Our solution involves extracting multiple systems of recurrences for the same loop, where each system approximates some aspect of the dynamics of the loop. In the following, we present three different mechanisms for extracting recurrences from a transition formula, each building on the last.

The recurrence-extraction mechanisms all assume a wedge representation of the loop body, which eliminates the need to deal with disjunction and existentially quantified variables (both of which are present in transition formulas). Thus, *the first step we take in computing recurrences entailed by a transition formula ϕ_{body} is to find an over-approximating wedge w_{body} using the symbolic-abstraction procedure from §4.3* (Algorithm 3).

Before proceeding to the recurrence-extraction algorithms, we will formalize the problem set-up that is common to all. We fix ϕ_{body} to be a loop-body transition formula, and w_{body} to be a wedge that over-approximates it. Suppose that PVar is a set of constant symbols that represent program variables, and PVar′ is a disjoint set of "primed copies" of program variables. Suppose Σ is a vocabulary containing PVar, PVar′, and Σ_N. Fix a Σ-wedge w_{body} that over-approximates the transition relation of a loop body. Let $\tau = [t_1, \ldots, t_n]$ be a coordinate system admitting w_{body}. Without loss of generality, suppose that the terms $t_1, \ldots, t_m$ correspond to the program variables in PVar and the terms $t_{m+1}, \ldots, t_{2m}$ correspond to the program variables in PVar′. We use $\mathbf{x} \stackrel{\text{def}}{=} [z_1, \ldots, z_m]$ to denote the vector of variables corresponding to PVar and $\mathbf{x}' \stackrel{\text{def}}{=} [z_{m+1}, \ldots, z_{2m}]$ to denote the vector of variables corresponding to PVar′.

5.1 Affine Recurrences

An *affine recurrence* is a recurrence for a linear combination of program variables. Our goal is to compute matrices $A \in \mathbb{Q}^{k \times m}$ and $B \in \mathbb{Q}^{k \times k}$ and a vector $\mathbf{c} \in \mathbb{Q}^k$ such that $w_{\text{body}} \models_L A\mathbf{x}' = BA\mathbf{x} + \mathbf{c}$. The relation $A\mathbf{x}' = BA\mathbf{x} + \mathbf{c}$ defines a set of affine recurrences for the linear combinations in $A\mathbf{x}$. Moreover, we want the affine space consisting of all valuations of $\mathbf{x}$ and $\mathbf{x}'$ that satisfy $A\mathbf{x}' = BA\mathbf{x} + \mathbf{c}$ to be *least* (in the sense that if $A', B', \mathbf{c}'$ satisfy $w_{\text{body}} \models_L A'\mathbf{x}' = B'A'\mathbf{x} + \mathbf{c}'$, then we have $A\mathbf{x}' = BA\mathbf{x} + \mathbf{c} \models_L A'\mathbf{x}' = B'A'\mathbf{x} + \mathbf{c}'$). Suppose that $\mathbf{cl}$ is a closed-form solution to the system of recurrence equations $\mathbf{y}_n = B\mathbf{y}_{n-1} + \mathbf{c}$ (which matches the form in Eqn. (2)), in the sense that $\mathbf{y}^{[n]} = B\mathbf{y}^{[n-1]} + \mathbf{c} \iff \mathbf{y}^{[n]} = \mathbf{cl}(\mathbf{y}^{[0]}, n)$. Then the transitive closure of ϕ_{body} must entail $\exists n \geq 0. A\mathbf{x}' = \mathbf{cl}(A\mathbf{x}, n)$.

As a motivating example, consider the loop from Fig. 4(a) and the associated transition formula and wedge. Observe that (1) the loop is non-deterministic, and there is no exact representation of

the closed form of the variables x or y, and (2) despite the fact that there is an apparently recursive assignment to the variable j, there is no exact closed form for j because it depends on variable x, for which there is no closed form. For this example, the best[3] affine recurrence is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \mathtt{i}' \\ \mathtt{j}' \\ \mathtt{x}' \\ \mathtt{y}' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \mathtt{i} \\ \mathtt{j} \\ \mathtt{x} \\ \mathtt{y} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{3}$$

(or more simply, $\mathtt{i}' = \mathtt{i} + 1$ and $(\mathtt{x}' + \mathtt{y}') = 2(\mathtt{x} + \mathtt{y}) + \mathtt{i}$). Note how the second row of matrix A in Eqn. (3) creates the linear combinations $\mathtt{x}' + \mathtt{y}'$ and $\mathtt{x} + \mathtt{y}$ in the latter recurrence equation.

The best affine recurrence entailed by w can be computed via successive approximation. We begin by explaining the procedure in the abstract, and return to the concrete example below. The first approximation $A_1\mathbf{x}' = M_1\mathbf{x} + \mathbf{c}_1$ is merely the constraint representation of the affine hull of the underlying polyhedron of w, projected onto the first $2m$ dimensions (corresponding to the $\mathbf{x}$ and $\mathbf{x}'$ symbols). If M_1 factors as BA_1, then we are done: the best affine recurrence entailed by w is $A_1\mathbf{x}' = BA_1\mathbf{x} + \mathbf{c}_1$. If M_1 does not factor as BA_1, then the equation $A_1\mathbf{x}' = M_1\mathbf{x} + \mathbf{c}_1$ cannot be interpreted as a recurrence. (The rowspace of A_1 must contain the rowspace of M_1—intuitively, the terms that appear on the right-hand side of a recurrence equation must be linear combinations of terms that appear on the left.) Thus, we compute a second approximation $A_2\mathbf{x}' = M_2\mathbf{x} + \mathbf{c}_2$. For any approximation i, the $(i+1)^{\text{th}}$ approximation is computed from the ith by finding a matrix T_i whose rows form a basis for the vector space $\{\mathbf{v} : \exists \mathbf{u}.\mathbf{u}A_i = \mathbf{v}B_i\}$ and taking $A_{i+1} = T_iA_i$, $M_{i+1} = T_iM_i$, and $c_{i+1} = T_ic_i$. Eventually this process must reach a fixpoint, because each step decreases the dimension of the rowspace of A_i by at least 1.

Returning to our running example Fig. 4, the first approximation is essentially just the wedge Fig. 4(c) itself:

$$\underbrace{\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 \end{bmatrix}}_{A_1} \begin{bmatrix} \mathtt{i}' \\ \mathtt{j}' \\ \mathtt{x}' \\ \mathtt{y}' \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 2 & 2 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}}_{M_1} \begin{bmatrix} \mathtt{i} \\ \mathtt{j} \\ \mathtt{x} \\ \mathtt{y} \end{bmatrix} + \underbrace{\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}}_{c_1}$$

Observe that the rowspace of A_1 does not contain $\begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}$, the third row of M_1 (i.e., j appears on the right-hand side of an equation but we cannot isolate j′ on the left). We find a matrix T_1 whose rows generate the space $\{\mathbf{v} : \exists \mathbf{u}.\mathbf{u}A_1 = \mathbf{v}B_1\}$ pictured below to the left (intuitively, T_1 is a linear transformation that projects out j), and multiply both sides of the equation to get the equation pictured below to the right:

$$\underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}}_{T_1} \qquad \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}}_{A_2} \begin{bmatrix} \mathtt{i}' \\ \mathtt{j}' \\ \mathtt{x}' \\ \mathtt{y}' \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 2 & 2 \end{bmatrix}}_{M_2} \begin{bmatrix} \mathtt{i} \\ \mathtt{j} \\ \mathtt{x} \\ \mathtt{y} \end{bmatrix} + \underbrace{\begin{bmatrix} 1 \\ 0 \end{bmatrix}}_{c_2}$$

The rowspace of A_2 *does* contain the rowspace of M_2, so are done. We factor M_2 as BA_2 and obtain the affine recurrence shown in Eqn. (3).

5.2 Stratified Recurrences

Consider the simple loop `while(*){ y += x*x; x++; }`. The affine recurrence extracted by the algorithm in the preceding section is $\mathtt{x}' = \mathtt{x} + 1$: no recurrence is extracted for y because its recurrence has a non-linear dependence on x. However, because there is no circular dependence of

[3]"Best" in the sense that it generates the least affine space. The affine space is unique, but its representation as a system of equations is not. However, the representation is irrelevant for our purposes.

x on y, we can arrange the recurrences into *strata* and leverage the closed form of x to compute a closed form for y:

Stratum	Recurrence	Closed form
0	$x' = x + 1$	$x^{[n]} = x^{[0]} + n$
1	$y' = y + x \times x$	$y^{[n]} = y^{[0]} + \sum_{i=0}^{n-1} x^{[i]} = y^{[0]} + \sum_{i=0}^{n-1} (x^{[0]} + i)^2$ $= y^{[0]} + n/6 - n^2/2 + n^3/3 - nx^{[0]} + n^2 x^{[0]} + n(x^{[0]})^2$

More generally, we can compute a sequence of *stratified recurrences* such that each recurrence in the sequence may have non-linear dependencies on previous terms in the sequence:

$$\begin{aligned} w_{\text{body}} &\models_L A_0\mathbf{x}' = B_0A_0\mathbf{x} + c \\ w_{\text{body}} &\models_L A_1\mathbf{x}' = B_1A_1\mathbf{x} + \mathbf{p}_1(A_0\mathbf{x}) \\ &\vdots \\ w_{\text{body}} &\models_L A_i\mathbf{x}' = B_iA_i\mathbf{x} + \mathbf{p}_i(A_0\mathbf{x}, \dots, A_{i-1}\mathbf{x}) \end{aligned}$$

where $A_0\mathbf{x}' = B_0A_0\mathbf{x} + c$ is an affine recurrence and each $\mathbf{p}_i(A_0\mathbf{x}, \dots, A_{i-1}\mathbf{x})$ is a polynomial over the terms in $A_0\mathbf{x}, \dots, A_{i-1}\mathbf{x}$. For each stratum j, the closed form $\mathbf{cl}_j$ is computed as the solution to a single matrix recurrence (of the form in Eqn. (2)) by substituting closed forms of previous strata into the next:

$$\begin{aligned} \mathbf{y}_0^{[n]} = B_0\mathbf{y}_0^{[n-1]} + \mathbf{c} &\iff \mathbf{y}_0^{[n]} = \mathbf{cl}_0(\mathbf{y}_0^{[0]}, n) \\ \mathbf{y}_1^{[n]} = B_1\mathbf{y}_1^{[n-1]} + \mathbf{p}_1(\mathbf{cl}_0(\mathbf{y}_0^{[n]}, n)) &\iff \mathbf{y}_1^{[n]} = \mathbf{cl}_1(\mathbf{y}_0^{[0]}, \mathbf{y}_1^{[0]}, n) \\ &\vdots \\ \mathbf{y}_i^{[n]} = B_i\mathbf{y}_i^{[n-1]} + \mathbf{p}_i(\mathbf{cl}_0(\mathbf{y}_0^{[n]}, n), \dots, \mathbf{cl}_{i-1}(\mathbf{y}_0^{[n]}, \dots, \mathbf{y}_{i-1}^{[n]}, n)) &\iff \mathbf{y}_i^{[n]} = \mathbf{cl}_i(\mathbf{y}_0^{[0]}, \dots, \mathbf{y}_{i-1}^{[0]}, n) \end{aligned}$$

Observing that substitution of *MultRecTerms* into a polynomial yields a *MultRecTerm*, this procedure works as long as no $\mathbf{cl}_i(\mathbf{y}_0^{[0]}, \dots, \mathbf{y}_{i-1}^{[0]}, n)$ contains an IIF.

The procedure for extracting stratified recurrences similarly proceeds iteratively on strata, beginning at stratum 0 (the best affine recurrence satisfied by the input wedge, as computed in §5.1), and continuing until reaching a stratum where no new recurrences are found. For each stratum i, we compute the recurrence equation

$$A_i\mathbf{x}' = B_iA_i\mathbf{x} + \mathbf{p}_i(A_0\mathbf{x}, \dots, A_{i-1}\mathbf{x})$$

as follows. Let $T = \{t_1, \dots, t_k\}$ be a set consisting of all linear terms of the form $\mathbf{ax}$, where $\mathbf{a}$ is a row of one of the matrices $A_0, \dots, A_{i-1}$; T corresponds to the set of linear terms for which there is a recurrence equation in some strata $< i$ (and thus, we know a closed form for each t_j). For each term $t_j \in T$ create a fresh variable r_j and define I_i to be the ideal generated by $\mathbb{I}(w_{\text{body}}, \tau)$ along with the (linear) polynomials $\{r_j - t_j : 1 \le j \le k\}$. Compute a Gröbner basis $\mathbf{G}_i$ for I_i w.r.t. an elimination order for $\{z_{2m+1}, \dots, z_n\}$, with the r variables ordered before the z variables. Recall that the variables $z_{2m+1}, \dots, z_n$ correspond to non-linear terms over program variables, so reduction by $\mathbf{G}_i$ projects out non-linear terms over the program variables, but in doing so may introduce polynomials of the new r variables (which is desirable, since those are the polynomials for which we can compute closed forms). Compute an initial system of equations

$$A_{i,1}\mathbf{x}' = M_{i,1}\mathbf{x} + \mathbf{p}_{i,1}$$

where $A_{i,1}$ and $M_{i,1}$ are rational matrices, and $\mathbf{p}$ is a vector of polynomials in $\mathbb{Q}[t_1, \dots, t_k]$ by taking the constraint representation of the affine hull of the underlying polyhedron of w, reducing each equation by $\mathbf{G}_i$, collecting the subset of equations that have the appropriate form, and finally replacing each (z and r) variable with its corresponding (x, x', and t_i) term. Then perform the same fixpoint algorithm from §5.1 to reduce this system of equations to a system of recurrence equations of the form $A_i\mathbf{x}' = B_iA_i\mathbf{x} + \mathbf{p}_i$, and proceed to the next stratum $i + 1$.

5.3 Recurrence Inequations

In §5.1 and §5.2, we described methods for describing the exact behavior of linear terms over program variables using recurrence equations. In this sub-section, our goal shifts to finding *inequations* that bound linear terms. Our goal is to compute a system of recurrence inequations

$$w_{\text{body}} \models_L A\mathbf{x}' \leq BA\mathbf{x} + \mathbf{p}(A_0\mathbf{x}, ..., A_k\mathbf{x})$$

where $A \in \mathbb{Q}^{k \times m}$ is rational matrix, $B \in \mathbb{Q}^{k \times k}$ is a non-negative matrix (that is, a matrix with no negative entries), and $\mathbf{p}(A_0\mathbf{x}, ..., A_k\mathbf{x})$ is a polynomial over $A_0\mathbf{x}, ..., A_k\mathbf{x}$, where $A_0, ..., A_k$ are as in §5.2.

The procedure for extracting recurrence inequations operates similarly to the one for affine recurrences, except that rather than finding some i such that A_i and M_i generate the same row space, we require that A_i and M_i generate the same *cone*. Given a matrix A with rows $\mathbf{a}_1, ..., \mathbf{a}_k$, the **(convex) cone** generated by the rows of A is as follows:

$$cone(A) \stackrel{\text{def}}{=} \{\lambda_1\mathbf{a}_1 + \cdots + \lambda_k\mathbf{a}_k : \lambda_1 \geq 0, ..., \lambda_k \geq 0\}$$

The requirement that A_i and M_i generate the same cone guarantees that there exists a factorization $M_i = BA_i$ where B is non-negative. A second complication is that the iterative algorithm may fail to terminate because the lattice of polyhedra (in contrast to the lattice of vector spaces) has infinite ascending chains. We use polyhedral widening to guarantee termination of the successive-approximation algorithm.

6 OPERATIONAL CALCULUS RECURRENCE SOLVING

6.1 Introduction

Operational calculus [J.-G.-M. 1983] is a technique for transforming a problem in analysis into an algebraic problem. Differencing and summation correspond to algebraic operators in a certain underlying algebra. Most commonly, operational-calculus techniques are used to solve differential equations. However, using the operational-calculus algebra of Berg [1967, Ch. II] it is possible to solve recurrence relations using operational calculus. The analysis problem of solving a recurrence equation is transformed into an equation or equation system, which is then solved by algebraic manipulations. A solution to the original problem can be read out of a solution to the algebraic problem.

6.1.1 The Ring of Functions. In this section, we review the univariate operational calculus of Berg [1967, Ch. II]. The ring of functions $\mathcal{R}$ is defined as $\mathcal{R} \stackrel{\text{def}}{=} (R, +, \cdot, \underline{0}, \underline{1})$, where $R = \mathbb{N} \to \mathbb{Q}$.[4]

It is often helpful to consider a ring element $r \in R$ as an infinite sequence $\langle r^{[0]}, r^{[1]}, r^{[2]}, \ldots \rangle$. For $r \in R$, we use both $r(i)$ and $r^{[i]}$ to denote the ith element of r. (We use $r(i)$ when we wish to emphasize that r is the function $\lambda i.r(i)$, and $r^{[i]}$ when we wish to emphasize that r is the sequence $\langle r^{[i]} \rangle$.)

Ring addition is pointwise addition: for all $a, b \in R$, $a + b \stackrel{\text{def}}{=} \lambda i.a(i) + b(i)$, so the additive identity element, $\underline{0}$, is $\lambda i.0 = \langle 0, 0, 0, \ldots \rangle$. Ring multiplication is the following convolution-product difference:

Definition 6.1. For all $a, b \in R$,

$$a \cdot b \stackrel{\text{def}}{=} \lambda n. \sum_{\nu=0}^{n} a^{[\nu]} b^{[n-\nu]} - \sum_{\nu=0}^{n-1} a^{[\nu]} b^{[n-1-\nu]}. \tag{4}$$

[4]Berg uses $\mathbb{N} \to \mathbb{C}$, where $\mathbb{C}$ is the set of complex numbers. However, our goal is to compute closed forms for recurrences that can be expressed in the term language of the logic defined in §3.2, which requires using $\mathbb{Q}$.

We will often omit multiplication operators, and write $a \cdot b$ as ab. The multiplicative identity element, $\underline{1}$, is $\lambda i.1 = \langle 1, 1, 1, \ldots \rangle$. It is easy to show that multiplication is commutative and that multiplication distributes over addition [Berg 1967, §7].

The ***constant*** elements of R are those of the form $\underline{c} \stackrel{\text{def}}{=} \lambda n.c = \langle c, c, c, \ldots \rangle$. When either argument of a multiplication is a constant, ring multiplication acts like scalar multiplication: for all $b \in R$, $\underline{c} \cdot b = \lambda n.cb^{[n]}$. When both arguments are scalars, ring multiplication is isomorphic to ordinary multiplication: $\underline{c} \cdot \underline{d} = \underline{cd}$.

It is sometimes useful to write Eqn. (4) as

$$a \cdot b \stackrel{\text{def}}{=} \lambda n. \sum_{\nu=0}^{n-1} a^{[\nu]}(b^{[n-\nu]} - b^{[n-1-\nu]}) + a^{[n]}b^{[0]}. \tag{5}$$

Id denotes the identity function $\lambda n.n = \langle 0, 1, 2, \ldots \rangle$. Using Eqn. (5), note that for all $a \in R$,

$$\text{Id} \cdot a = a \cdot \text{Id} = \lambda n. \sum_{\nu=0}^{n-1} a^{[\nu]}((n - \nu) - (n - 1 - \nu)) + a^{[n]}0 = \lambda n. \sum_{\nu=0}^{n-1} a^{[\nu]}.$$

In other words, Id acts as a summation operator, producing the sequence of (exclusive) partial sums of a. For instance, $\text{Id} \cdot \text{Id} = \langle 0, 0, 1, 3, 6, 10, 15, \ldots \rangle = \lambda n.\binom{n}{2}$.[5]

Let $v \stackrel{\text{def}}{=} \langle 0, 1, 1, 1, \ldots \rangle$. Let $c^{[n]}$ denote an arbitrary infinite sequence $\langle c^{[0]}, c^{[1]}, \ldots \rangle$. Then

$$v \cdot c^{[n]} = \lambda n. \begin{cases} 0 & \text{if } n = 0 \\ c^{[n-1]} & \text{if } n \geq 1 \end{cases}$$

and hence ring multiplication by v performs a right shift (and ring multiplication by v^m performs a right shift of $c^{[n]}$ by m places.).

6.1.2 The Field of Operators. Ring $\mathcal{R}$ has no divisors of $\underline{0}$, and hence can be extended to a field $\mathcal{F} \stackrel{\text{def}}{=} (F, +, \cdot, -, ^{-1}, \underline{0}, \underline{1})$ An element of F is called an *operator*. Some operators are elements of R; however, there are interesting operators that are not members of R. For instance, the equation

$$qv = \underline{1} \tag{6}$$

has no solution in $\mathcal{R}$. However, because $v \neq \underline{0}$, there is an operator $q \in F$ that solves Eqn. (6). Intuitively, q should be a left-shift operator.

As Berg points out [Berg 1967, §8], an operator can be interpreted as a sequence with a finite number of non-zero elements at negative-index positions (whereas a ring element is a sequence with *no* non-zero elements at negative-index positions). Therefore, it is easy to show that the appropriate representation for q, the operator that solves Eqn. (6), is

		...	−2	−1	0	1	2	3	...
q	=	...	0	1	1	1	1	1	...

Let $y^{[n]}$ denote an arbitrary infinite sequence $\langle y^{[0]}, y^{[1]}, \ldots \rangle$, and let $y^{[n+1]}$ denote $\langle y^{[1]}, y^{[2]}, \ldots \rangle$. The following general formula expresses $y^{[n+1]}$ in terms of $y^{[n]}$, $y^{[0]}$, and q. It works by multiplying $y^{[n]}$ by q to shift the sequence to the left by one position, and subtracting away the value that would otherwise appear at index position −1, giving us $y^{[n+1]}$:

$$y^{[n+1]} \;=\; q \cdot y^{[n]} - (q - \underline{1})\underline{y^{[0]}}. \tag{7}$$

[5] As is standard, for $n \geq k$, the ***binomial coefficient*** $\binom{n}{k}$ denotes $\frac{n!}{k!(n-k)!}$. We define $\binom{0}{0} = 1$ and, for all $k > n$, $\binom{n}{k} = 0$.

We can depict some of the infinite sequences that appear in Eqn. (7) as follows:

		...	−2	−1	0	1	2	3	...
$y^{[n]}$	=	...	0	0	$y^{[0]}$	$y^{[1]}$	$y^{[2]}$	$y^{[3]}$	...
$y^{[n+1]}$	=	...	0	0	$y^{[1]}$	$y^{[2]}$	$y^{[3]}$	$y^{[4]}$	...
$q \cdot y^{[0]}$	=	...	0	$y^{[0]}$	$y^{[1]}$	$y^{[2]}$	$y^{[3]}$	$y^{[4]}$	...
$(q - \underline{1})\underline{y^{[0]}}$	=	...	0	$y^{[0]}$	0	0	0	0	...

Henceforth, sequences like the ones above will be written as follows:

$$y^{[n]} = \langle y^{[0]}, y^{[1]}, y^{[2]}, y^{[3]}, \ldots \rangle \qquad q \cdot y^{[n]} = \langle y^{[0]} \parallel y^{[1]}, y^{[2]}, y^{[3]}, y^{[4]}, \ldots \rangle$$
$$y^{[n+1]} = \langle y^{[1]}, y^{[2]}, y^{[3]}, y^{[4]}, \ldots \rangle \qquad (q - \underline{1})\underline{y^{[0]}} = \langle y^{[0]} \parallel 0, 0, 0, 0, \ldots \rangle$$

where $\parallel$ is used to separate the negative-index positions (if any) from the non-negative positions. The values at all negative-index positions that are not shown are assumed to be 0.

Using this notation, the operators q and $(q - \underline{1})$ have the sequences shown below:

$$q = \langle 1 \parallel 1, 1, 1, 1, \ldots \rangle \qquad (q - \underline{1}) = \langle 1 \parallel 0, 0, 0, 0, \ldots \rangle$$

Eqn. (7) can be generalized to create $y^{[n+m]} = \langle y^{[m]}, y^{[m+1]}, y^{[m+2]}, y^{[m+3]}, \ldots \rangle$ from $y^{[n]}$ as follows:

$$y^{[n+m]} = q^m \cdot y^{[n]} - \sum_{\mu=0}^{m-1} q^\mu \cdot (q - \underline{1})\underline{y^{[m-1-\mu]}}. \tag{8}$$

In this case, it is necessary to knock out the m values $y^{[0]}, y^{[1]}, \ldots, y^{[m-1]}$ from positions $-m, -m+1, \ldots, -1$, respectively. Eqn. (8) can be understood by recognizing that

$$(q - \underline{1})\underline{y^{[m-1-\mu]}} = \langle y^{[m-1-\mu]} \parallel 0, 0, 0, 0, \ldots \rangle,$$

which is then shifted μ positions to the left by q^μ.

6.1.3 Translation Rules. In some cases, functions that are expressed in terms of operators can also be expressed purely in terms of ordinary algebra. For example, consider the functional sequence $\lambda n.\alpha^n = \langle \alpha^0, \alpha^1, \alpha^2, \ldots \rangle$ for $n \in \mathbb{N}$ and $\alpha \in \mathbb{Q}$. By Eqn. (7) and $\alpha^0 = 1$ we have:

$$\alpha^{n+1} = q\alpha^n - (q - \underline{1})\underline{\alpha^0}$$
$$q\alpha^n - \alpha^{n+1} = (q - \underline{1})$$
$$\alpha^n(q - \underline{\alpha}) = (q - \underline{1})$$
$$\alpha^n = \frac{q - \underline{1}}{q - \underline{\alpha}}$$

Thus we have a way of translating a recognizable functional sequence in standard algebra to the operational-calculus algebra. We will write this translation as:

$$\mathcal{T}_n\left(\frac{q - \underline{1}}{q - \underline{\alpha}}\right) = \alpha^n$$

Fig. 5 shows the above rule along with other useful translation rules.

6.2 Solving First-Order Univariate Recurrences

The properties of the algebra described in §6.1 give us a nice algebraic way of solving recurrences of the following form:

$$y^{[n+1]} = \alpha \cdot y^{[n]} + \sum_i f_i(n) \qquad \text{where} \qquad f_i(n) = poly_i(n) * \beta_i^n \quad \text{and} \quad poly_i(n) \in \mathbb{Q}[n] \tag{17}$$

Algorithm 4 presents a method for solving such univariate recurrences.

$$\mathcal{T}_n\left(\underline{c}\right) = c \quad (9)$$

$$\mathcal{T}_n\left(\sum_{k\in K} g_k\right) = \sum_{k\in K}\mathcal{T}_n\left(g_k\right) \quad (10)$$

$$\mathcal{T}_n\left(\underline{c}f\right) = c\mathcal{T}_n\left(f\right) \quad (11)$$

$$\mathcal{T}_n\left(\frac{\underline{1}}{(q-\underline{1})^i}\right) = \binom{n}{i} \quad (12)$$

$$\mathcal{T}_n\left(\frac{q-\underline{1}}{q-\underline{k}}\right) = k^n \quad (13)$$

$$\mathcal{T}_n\left(\frac{q-\underline{1}}{(q-\underline{k})^{c+1}}\right) = \binom{n}{c}k^{n-c} \quad (14)$$

$$\mathcal{T}_n\left(\frac{\underline{1}}{q-\underline{k}}\right) = \frac{k^n-1}{k-1} \quad (15)$$

$$\mathcal{T}_n\left(\frac{\underline{1}}{(q-\underline{k})^c}\right) = \frac{\binom{n}{c-1} - \mathcal{T}_n\left(\frac{\underline{1}}{(q-\underline{k})^{c-1}}\right)}{k-1} \quad (16)$$

Fig. 5. Translation function $\mathcal{T}_n : F \to R$. In Eqn. (10) $K \subseteq \mathbb{N}$ is some finite index set.

Input : A recurrence inequation r, a variable to solve for y, and an index variable n
Output: The closed form solution for y

1. $R \leftarrow$ Simplify r ;
2. $OpR \leftarrow$ Translate R into $\mathcal{F}$ using Eqn. (8) and $\mathcal{T}_n^{-1}$;
3. $OpR \leftarrow$ Solve for y in OpR ;
4. $OpR \leftarrow$ Perform partial-fraction decomposition on OpR ;
5. $S \leftarrow \mathcal{T}_n\left(OpR\right)$;
6. $S \leftarrow$ Simplify S ;
7. **return** S

Algorithm 4: Univariate Operational Calculus Recurrence Solving

6.2.1 Explanation of Operational Calculus Recurrence Solving. Between each step of the algorithm we simplify expressions using a simplification algorithm, given in Cohen [2003], which performs standard algebraic simplifications, such as collecting like terms and combining exponents. Simplification helps to maintain a more canonical form for expression matching in later steps.

Consider how Step 2 of Algorithm 4, translates a recurrence of the form of Eqn. (17). We can use Eqn. (7) to write $y^{[n+1]}$ as $qy^{[n]} - (q-\underline{1})\underline{y^{[0]}}$. For the right-hand side of Eqn. (17), $\alpha \cdot y^{[n]}$ remains the same, except that the constant α becomes the constant sequence $\underline{\alpha}$. What remains to describe is how we obtain $\mathcal{T}_n^{-1}\left(\sum_i f_i(n)\right)$.

As a quick detour, note that for all the rules in Fig. 5, polynomials are represented using a binomial operator. Thus we must have an appropriate way to represent polynomials using binomials. This transformation can be performed via the following relation, where $\left\{ d \atop j \right\}$ denotes a Stirling number of the second kind:

$$n^d = \sum_{j=0}^{d} j! \left\{ d \atop j \right\} \binom{n}{j} \quad (18)$$

To perform the translation $\mathcal{T}_n^{-1}\left(\sum_i f_i(n)\right)$, first note that by Eqn. (10), we can pull the summation out. We now seek a characterization of $\mathcal{T}_n^{-1}\left(poly_i(n) * \beta_i^n\right)$. First, consider rewriting $poly_i(n)$ using Eqn. (18), where $d = \deg(poly_i(n))$.

$$f_i(n) = \beta_i^n * poly_i(n) = \beta_i^n * \sum_{j=0}^{d} v_{i,j}\binom{n}{j} = \sum_{j=0}^{d} \beta_i^j * v_{i,j}\binom{n}{j}\beta_i^{n-j}$$

We can use the general transformation

$$\mathcal{T}_n^{-1}\left(\binom{n}{c}k^{n-c}\right) = \frac{q-\underline{1}}{(q-\underline{k})^{c+1}}$$

to see that every $f_i(n)$ can be translated into the following form in the operational-calculus algebra:

$$\mathcal{T}_n^{-1}(f_i(n)) = \sum_{j=0}^{d} \frac{\underline{\gamma_{i,j}}(q-\underline{1})}{(q-\underline{\beta_i})^{c_j}}$$

where $\gamma_{i,j}$ is an appropriate constant. Thus, after Step 2 Algorithm 4 produces an equation of the form

$$q \cdot y^{[n]} - (q-\underline{1})\underline{y^{[0]}} = \underline{\alpha} \cdot y^{[n]} + \sum_i \sum_{j=0}^{d} \frac{\underline{\gamma_{i,j}}(q-\underline{1})}{(q-\underline{\beta_i})^{c_{i,j}}}$$

The algorithm continues, solving the above equation for y_n in Step 3.

$$y^{[n]} = \frac{q-\underline{1}}{q-\underline{\alpha}}\underline{y^{[0]}} + \sum_l \frac{\underline{\gamma_l}(q-\underline{1})}{(q-\underline{\alpha})(q-\underline{\beta_l})^{c_l}}$$

Note the double summation has been replaced by a single summation with a new index, l.

Thus, after Step 3, we need to transform all the summands to a form that has a translation rule in Fig. 5. We can accomplish this goal by performing partial-fraction decomposition on all the summands (Step 4).

$$\frac{q-\underline{1}}{q-\underline{\alpha}}\underline{y^{[0]}} + \sum_k \frac{\underline{a_k}}{(q-\underline{b_k})^{c_k}}$$

where a_k and b_k are constants resulting from partial-fraction decomposition, and every c_k is a non-negative integer. After Step 4, the right-hand side of the equation will be a sum of terms, each of which has a translation rule in Fig. 5 that can be applied to translate the term back into standard algebra (Step 5). Finally, in Step 6, the algorithm removes binomials and simplifies the result.

The preceding four paragraphs have shown that Algorithm 4 obtains a closed-form solution to any recurrence of the form given in Eqn. (17). The steps used in Algorithm 4 never get stuck; i.e., operations such as partial-fraction decomposition in Step 4 can always be performed. In other words, we have proven the following theorem:

THEOREM 6.2. *Algorithm 4 is complete for the class of recurrences given in Eqn. (17).*

Example 6.3. Consider the following example, which generates an exponential closed form.

$$y^{[n+1]} = 2y^{[n]} + 3^n$$

Step 2 generates the following equation in $\mathcal{F}$:

$$q \cdot y^{[n]} - (q-\underline{1})\underline{y^{[0]}} = \mathcal{T}_n^{-1}\left(2y^{[n]} + 3^n\right)$$

$$q \cdot y^{[n]} - (q-\underline{1})\underline{y^{[0]}} = \underline{2}y^{[n]} + \frac{q-\underline{1}}{q-\underline{3}}$$

By Eqn. (11), Eqn. (9), Eqn. (10), Eqn. (13), and Eqn. (7).

After solving for $y^{[n]}$, Step 3 produces

$$y^{[n]} = \frac{q-\underline{1}}{q-\underline{2}}\underline{y^{[0]}} + \frac{q-\underline{1}}{(q-\underline{3})(q-\underline{2})}$$

Step 4 performs partial-fraction decomposition on the right-hand side of the equation where appropriate, obtaining

$$y^{[n]} = \frac{q-\underline{1}}{q-\underline{2}}\underline{y^{[0]}} + \frac{\underline{2}}{q-\underline{3}} - \frac{\underline{1}}{q-\underline{2}}$$

Finally, Step 5 translates the resultant expression back to standard algebra and the algorithm performs a final simplification.

$$y^{[n]} = \mathcal{T}_n\left(\frac{q-\underline{1}}{q-\underline{2}}\underline{y^{[0]}} + \frac{\underline{2}}{q-\underline{3}} - \frac{\underline{1}}{q-\underline{2}}\right)$$

$$= 2^n y^{[0]} + 2\frac{3^n - 1}{3 - 1} - \frac{2^n - 1}{2 - 1}$$
$$= 2^n(y^{[0]} - 1) + 3^n$$

6.3 Solving First-Order Matrix Recurrences

We now give details about solving what we call *matrix recurrences.* Consider a vector of recurrence variables indexed by $n+1$, $\mathbf{x}^{[n+1]} = \left[x_1^{[n+1]}, \; x_2^{[n+1]}, \; \ldots, \; x_k^{[n+1]}\right]^t$; their copies, indexed by n, $\mathbf{x}^{[n]} = \left[x_1^{[n]}, \; x_2^{[n]}, \; \ldots, \; x_k^{[n]}\right]^t$; a matrix $\mathbf{A}$ with entries in $\mathbb{Q}$; and a vector $\mathbf{b} = \left[f_1(n), \; f_2(n), \; \ldots, \; f_k(n)\right]^t$ in which each $f_i(n)$ is a *RecTerm* as given in §5. We seek a closed-form solution for $\mathbf{x}^{[n]}$ in the recurrence

$$\mathbf{x}^{[n+1]} = \mathbf{A}\mathbf{x}^{[n]} + \mathbf{b} \tag{19}$$

Let us consider how to extend the method from §6.1 and §6.2 to solve such an equation.

Left-Hand Side of Eqn. (19). We seek a rule for the vector $\mathbf{x}^{[n+1]}$ that is analogous to Eqn. (7). If we consider applying Eqn. (7) to each of the entries of $\mathbf{x}^{[n+1]}$, we have the following:

$$\mathbf{x}^{[n+1]} = \begin{bmatrix} q \cdot x_1^{[n]} - (q - \underline{1})\underline{x_1^{[0]}} \\ \vdots \\ q \cdot x_k^{[n]} - (q - \underline{1})\underline{x_k^{[0]}} \end{bmatrix} = \begin{bmatrix} q \cdot x_1^{[n]} \\ \vdots \\ q \cdot x_k^{[n]} \end{bmatrix} - \begin{bmatrix} (q - \underline{1})\underline{x_1^{[0]}} \\ \vdots \\ (q - \underline{1})\underline{x_k^{[0]}} \end{bmatrix} = \begin{bmatrix} q & \underline{0} & \cdots & \underline{0} \\ \underline{0} & q & \cdots & \underline{0} \\ \vdots & \vdots & \ddots & \underline{0} \\ \underline{0} & \underline{0} & \cdots & q \end{bmatrix} \begin{bmatrix} x_1^{[n]} \\ \vdots \\ x_k^{[n]} \end{bmatrix} - \begin{bmatrix} \underline{x_1^{[0]}}(q - \underline{1}) \\ \vdots \\ \underline{x_k^{[0]}}(q - \underline{1}) \end{bmatrix}$$
$$= q\mathbf{I}\mathbf{x}^{[n]} - \mathbf{x}^{[0]}(q - \underline{1}) \tag{20}$$

$\mathbf{I}$ denotes the identity matrix of dimension $k \times k$, and $\mathbf{x}^{[0]}$ denotes the vector of symbols $\left[\underline{x_1^{[0]}}, \; \ldots, \; \underline{x_k^{[0]}}\right]^t$.

Right-Hand Side of Eqn. (19). We now consider rewriting the right-hand side of Eqn. (19). In other words, we seek to know $\mathcal{T}_n^{-1}\left(\mathbf{A}\mathbf{x}^{[n]} + \mathbf{b}\right)$, where we interpret $\mathcal{T}_n^{-1}$ as being applied to each row of the vector $\mathbf{A}\mathbf{x}^{[n]} + \mathbf{b}$. By Eqns. (10) and (11), we have

$$\mathcal{T}_n^{-1}\left(\mathbf{A}\mathbf{x}^{[n]} + \mathbf{b}\right) = \underline{\mathbf{A}}\mathbf{x}^{[n]} + \mathcal{T}_n^{-1}(\mathbf{b}). \tag{21}$$

By Eqns. (20) and (21), we can rewrite the right-hand side of Eqn. (19) as follows:

$$\begin{aligned} q\mathbf{I}\mathbf{x}^{[n]} - (q - \underline{1})\mathbf{x}^{[0]} &= \underline{\mathbf{A}}\mathbf{x}^{[n]} + \mathcal{T}_n^{-1}(\mathbf{b}) \\ q\mathbf{I}\mathbf{x}^{[n]} - \underline{\mathbf{A}}\mathbf{x}^{[n]} &= \mathbf{x}^{[0]}(q - \underline{1}) + \mathcal{T}_n^{-1}(\mathbf{b}) \\ (q\mathbf{I} - \underline{\mathbf{A}})\mathbf{x}^{[n]} &= \mathbf{x}^{[0]}(q - \underline{1}) + \mathcal{T}_n^{-1}(\mathbf{b}) \\ \mathbf{x}^{[n]} &= (q\mathbf{I} - \underline{\mathbf{A}})^{-1}(\mathbf{x}^{[0]}(q - \underline{1}) + \mathcal{T}_n^{-1}(\mathbf{b})) \end{aligned} \tag{22}$$

Eqn. (22) gives us a closed form for $\mathbf{x}^{[n]}$ as a vector of elements in $\mathcal{F}$. To obtain the corresponding solution in standard algebra, we must apply $\mathcal{T}_n$ to each entry in the resulting vector.

Consequently, a general solution to a matrix recurrence of the form given in Eqn. (19) can be expressed as follows:

$$\mathbf{x}^{[n]} = \mathcal{T}_n\left((q\mathbf{I} - \underline{\mathbf{A}})^{-1}(\mathbf{x}^{[0]}(q - \underline{1}) + \mathcal{T}_n^{-1}(\mathbf{b}))\right) \tag{23}$$

Algorithm 5 gives a procedure that implements Eqn. (23), where some additional details have been given in Steps 5, 6, and 7. Unlike the univariate case, Algorithm 5 may not always return a polynomial or exponential closed form for the class of matrix recurrences defined by Eqn. (19). In particular, note that all translation rules in Fig. 5 involve rational functions in which the denominator is a function with only linear terms. In other words, for there always to be an appropriate translation

Input : A vector of recurrence variables indexed by $n+1$, $\mathbf{x}^{[n+1]}$, a matrix $\mathbf{A}$, a vector of recurrence variables indexed by n, $\mathbf{x}^{[n]}$, and an additive vector $\mathbf{b}$

Output: The closed form solution for $\mathbf{x}^{[n]}$, in $\mathbf{x}^{[n+1]} = \mathbf{A}\mathbf{x}^{[n]} + \mathbf{b}$

1. $\mathbf{b} \leftarrow$ Simplify $\mathbf{b}$;
2. $\mathbf{Opb} \leftarrow \mathbf{x}^{[0]}(q - \underline{1}) + \mathcal{T}_n^{-1}(\mathbf{b})$;
3. $\mathbf{D} \leftarrow$ Symbolically calculate $(q\mathbf{I} - \underline{\mathbf{A}})^{-1}$;
4. $\mathbf{V} \leftarrow \mathbf{D} * \mathbf{Opb}$;
5. $\mathbf{V} \leftarrow$ Convert every entry in $\mathbf{V}$ to the form $\frac{p}{q}$, where p and q are polynomials in $\mathbb{Q}[q]$;
6. $\mathbf{V} \leftarrow$ for every $\frac{p_i}{q_i} \in \mathbf{V}$, factor q_i into its irreducible factors in $\mathbb{Q}[q]$;
7. $\mathbf{V} \leftarrow$ for every $v_i \in \mathbf{V}$, perform partial-fraction decomposition on v_i ;
8. $\mathbf{S} \leftarrow$ for every $v_i \in \mathbf{V}$, replace v_i with $\mathcal{T}_n(v_i)$; if no translation applies to a term τ, replace τ with $IIF(\tau)$;
9. $\mathbf{S} \leftarrow$ for every $s_i \in \mathbf{S}$, simplify s_i ;
10. **return** S

Algorithm 5: Matrix Operational Calculus Recurrence Solving

rule in Fig. 5, we must be able to find all roots of the polynomials in the denominator in Step 6. This requirement presents several challenges. First, it has high implementation complexity: it demands an implementation of algebraic numbers, including, in general, algebraic numbers that arise as roots of polynomials of degree ≥ 5, which may not be expressible in terms of radicals. Second, one or more root may be complex, which means that even if there is a translation into standard algebra, the translation uses complex exponentiation, which (as shown in Ex. 2.3) is inadmissible in non-linear *real* arithmetic. Thus, to mitigate these difficulties, Step 6 finds all *rational* roots for the polynomial.

However, the restricted scope of Step 6 means that at Step 8 Algorithm 5 can have rational terms that have no corresponding translations in Fig. 5. When that happens, Algorithm 5 creates an *implicitly interpreted function* (IIF). An IIF is essentially an uninterpreted function, but has an association with its defining term—the rational term from Step 6 for which Fig. 5 had no rule.

IIFs occur exactly when the input matrix $\mathbf{A}$ contains at least one irrational or complex eigenvalue. IIFs are beneficial because they give the wedge domain an exact and canonical characterization of the recurrence solution, albeit one with one or more "uninterpreted functions." However, because the system retains the exact characterization of each IIF in the recurrence solver's logic, an IIF has an explicit meaning inside the recurrence solver. Consequently, when presented with an IIF, the recurrence solver can manipulate and operate over the defining term as appropriate. For an example of the creation of an IIF, see Ex. 6.5; for a demonstration of the utility of IIFs, see Ex. 2.4.

6.4 Matrix Examples

Example 6.4. Consider the following matrix recurrence:

$$\begin{bmatrix} x^{[k+1]} \\ y^{[k+1]} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -2 & 4 \end{bmatrix} \begin{bmatrix} x^{[k]} \\ y^{[k]} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

For this example, we have

$$\mathbf{b} = \begin{bmatrix} 0 & 1 \end{bmatrix}^t \qquad \mathbf{x}^{[0]} = \begin{bmatrix} \underline{x^{[0]}} & \underline{y^{[0]}} \end{bmatrix}^t \qquad \mathcal{T}_n^{-1}(\mathbf{b}) = \begin{bmatrix} \underline{0} & \underline{1} \end{bmatrix}^t$$

The additive vector, $\mathbf{b}$ in this recurrence is already simplified, and the translation, $\mathcal{T}_n^{-1}(\mathbf{b})$, is given above. Thus, performing Steps 1 and 2 leaves us with the vector

$$\begin{bmatrix} \underline{x^{[0]}}(q - \underline{1}) & \underline{y^{[0]}}(q - \underline{1}) + \underline{1} \end{bmatrix}^t$$

Step 3 symbolically computes

$$\begin{bmatrix} q-\underline{1} & \underline{-1} \\ \underline{2} & (q-\underline{4}) \end{bmatrix}^{-1} = \begin{bmatrix} \frac{q-\underline{4}}{q^2-\underline{5}q+\underline{6}} & \frac{\underline{1}}{q^2-\underline{5}q+\underline{6}} \\ \frac{\underline{-2}}{q^2-\underline{5}q+\underline{6}} & \frac{q-\underline{1}}{q^2-\underline{5}q+\underline{6}} \end{bmatrix}$$

After Steps 4 and 5, $\mathbf{V}$ equals the following vector:

$$\begin{bmatrix} \frac{(q^2-\underline{5}q+\underline{4})\underline{x^{[0]}}+(q-\underline{1})\underline{y^{[0]}}+\underline{1}}{q^2-\underline{5}q+\underline{6}} & \frac{(q^2-\underline{2}q+\underline{1})\underline{y^{[0]}}-(q-\underline{1})\underline{2x^{[0]}}+q-\underline{1}}{q^2-\underline{5}q+\underline{6}} \end{bmatrix}^t$$

The term $q^2 - \underline{5}q + \underline{6}$ can be factored into $(q - \underline{3})(q - \underline{2})$, so Steps 6 and 7 produce

$$\begin{bmatrix} \frac{\underline{1-2x^{[0]}+2y^{[0]}}}{q-\underline{3}} + \frac{\underline{-1+2x^{[0]}-y^{[0]}}}{q-\underline{2}} + \underline{x^{[0]}} & \frac{\underline{2-4x^{[0]}+4y^{[0]}}}{q-\underline{3}} + \frac{\underline{-1+2x^{[0]}-y^{[0]}}}{q-\underline{2}} + \underline{y^{[0]}} \end{bmatrix}^t$$

Finally, Steps 8 and 9 apply $\mathcal{T}_k$ to each entry of the above vector to obtain closed-form expressions for both $x^{[k]}$ and $y^{[k]}$.

$$x^{[k]} = \tfrac{1}{2} + (2x^{[0]} - y^{[0]} - 1)2^k + (-x^{[0]} + y^{[0]} + \tfrac{1}{2})3^k \qquad y^{[k]} = (2x^{[0]} - y^{[0]} - 1)2^k + (-2x^{[0]} + 2y^{[0]} + 1)3^k$$

Example 6.5. Consider the following matrix recurrence:

$$\begin{bmatrix} y^{[k+1]} \\ z^{[k+1]} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ -9 & 1 \end{bmatrix} \begin{bmatrix} y^{[k]} \\ z^{[k]} \end{bmatrix}$$

To simplify the example, suppose we know that the values of $y^{[0]}$ and $z^{[0]}$ are 1 and 0, respectively. If this information is used in Steps 1–5 of Algorithm 5, we are left with the following equation:

$$\begin{bmatrix} y^{[k]} & z^{[k]} \end{bmatrix}^t = \begin{bmatrix} \frac{(q-\underline{1})^2}{q^2-\underline{2}q+\underline{10}} & \frac{\underline{-9}(q-\underline{1})}{q^2-\underline{2}q+\underline{10}} \end{bmatrix}$$

Unfortunately, $q^2 - 2q + 10$ is irreducible in $\mathbb{Q}[q]$, and so we are not able to provide a solution for the recurrence only using rational coefficients. Step 7 (partial-fraction decomposition) yields

$$\begin{bmatrix} y^{[k]} & z^{[k]} \end{bmatrix}^t = \begin{bmatrix} \underline{1} - \frac{\underline{9}}{q^2-\underline{2}q+\underline{10}} & \frac{\underline{-9}q}{q^2-\underline{2}q+\underline{10}} + \frac{\underline{9}}{q^2-\underline{2}q+\underline{10}} \end{bmatrix}.$$

We can pull out the constant coefficients in the above closed form to yield the following closed-form expressions for $y^{[k]}$ and $z^{[k]}$.

$$\begin{bmatrix} y^{[k]} & z^{[k]} \end{bmatrix}^t = \begin{bmatrix} 1 - 9\mathcal{T}_k\left(\frac{\underline{1}}{q^2-\underline{2}q+\underline{10}}\right) & -9\mathcal{T}_k\left(\frac{q}{q^2-\underline{2}q+\underline{10}}\right) + 9\mathcal{T}_k\left(\frac{\underline{1}}{q^2-\underline{2}q+\underline{10}}\right) \end{bmatrix}. \tag{24}$$

There is no direct translation for the remaining terms. Thus, at this point Algorithm 5 creates IIFs for each term involving $\mathcal{T}_k(\cdot)$ to use in the solutions returned to the wedge domain. In the example above, the returned values are $1 - 9\mathit{IIF}[\frac{\underline{1}}{q^2-\underline{2}q+\underline{10}}](k)$ for $y^{[k]}$ and $-9\mathit{IIF}[\frac{q}{q^2-\underline{2}q+\underline{10}}](k) + 9\mathit{IIF}[\frac{\underline{1}}{q^2-\underline{2}q+\underline{10}}](k)$ for $z^{[k]}$.

7 EXPERIMENTS

Our techniques are implemented in a tool called ICRA, which uses Z3's UFLRA solver [de Moura and Bjørner 2008], and Apron's NewPolka polyhedron domain [Jeannet and Miné 2009]. Our experiments were designed to answer the following questions:

(1) How often is ICRA able to prove assertions for programs that have non-linear invariants?
(2) How often is ICRA able to generate upper bounds on the resource usage (e.g., running time) of programs?

Programs with Non-Linear Invariants. To address the question of ICRA's capabilities for proving assertions in programs that have non-linear invariants, we used a suite of 77 small programs with true assertions. These programs can be divided into three categories:

- 46 programs are from a suite used to test the invariant-generating tool HOLA [Dillig et al. 2013].

Table 1. Shows the results of the experiments to check non-linear assertions. Column 2 shows the total number of assertions in each benchmark suite. The two columns under each tool show the running time (in seconds) and the number of proved assertions. In each row the greatest number of assertions proved and the smallest running time are shown in boldface.

Benchmark	Total	ICRA		UAut.		CPA		SEA	
Suite	#A	Time	#A	Time	#A	Time	#A	Time	#A
HOLA	46	**123.5**	33	1571.9	20	2004.1	11	259.5	**38**
functional	21	**77.9**	**11**	732.8	0	1155.7	0	722.3	2
relational	10	**8.1**	**10**	473	0	603.0	0	121.8	4
Total	77	**209.5**	**54**	2777.7	20	3762.8	11	1103.6	44

- 21 programs have assertions that concern functional-correctness properties. These programs all have some non-linear structure. We created these benchmarks to test ICRA's non-linear capabilities.
- 10 programs are self-composed programs in the style described in §2. The Fibonacci program in §2 is an example of a program in this category. The other nine programs were obtained from [Antonopoulos et al. 2017; Barthe et al. 2011; Terauchi and Aiken 2005].

We compared ICRA on these 77 programs against three state-of-the-art software model checkers: Ultimate Automizer [Heizmann et al. 2013] from SV-COMP16 and CPAchecker [Beyer and Keremoglu 2011] from SV-COMP17, both based on predicate abstraction; and SeaHorn [Gurfinkel et al. 2015] version 0.1.0, a Horn-clause solver based on IC3. Timings (with a timeout limit of 60 seconds) were taken on a virtual machine (using Oracle VirtualBox), with a guest OS of Ubuntu 14.04, host of Microsoft Windows 7 Enterprise, and a 3.2 GHz quad-core Intel Core i5-4570 host CPU.

Tab. 1 shows that ICRA performs significantly better than the other tools on these 77 programs. In every category, ICRA was able to prove more assertions than any other tool, except for SeaHorn on the HOLA programs. Also, in terms of time ICRA was significantly faster than the other tools in all categories. This result is mostly due to ICRA having fewer timeouts compared with the other tools. ICRA had only 2 timeouts, where as SeaHorn had the next fewest with 17.

Resource-Bound Generation. To address the question of ICRA's capabilities for generating upper bounds on resource usage (in this case, upper bounds on running time), we ran experiments on 8 small benchmark programs. Five programs were taken from Brockschmidt et al. [2014], two examples are from §2 (Ex. 2.1 and Ex. 2.2), and one additional program is described below.

Example 7.1. The following program illustrates the relevance of non-linear invariant generation to resource-bound analysis. To upper-bound this program's running time, an analyzer must determine an upper bound on c that holds after the first loop:

```
void main(int a) {
    int b = 0, c = 1; assume(a >= 1 && c>= 1);
    while(a > b) { b ++; c += a; if (nondet()) { return; } }
    while(c >= 0) { c --; if (nondet()) { return; } } }
```

We compared ICRA on these 8 programs against two other resource-bound analysis tools, CoFloCo [Flores-Montoya and Hähnle 2014] and PUBS [Albert et al. 2013]. Timings (with a timeout of 60 seconds) were taken on a virtual machine (using Oracle VirtualBox), with a guest OS of Ubuntu 14.04, host OS of Fedora Core 6, and a 3.2 GHz quad-core Intel Core i5-4570 host CPU.

The number of upper bounds on running time generated by ICRA was 7; by CoFloCo, 7; and by PUBS, 4. The total time taken by ICRA was 11.0 seconds; by CoFloCo, 2.5 seconds, and by PUBS, 0.5 seconds. ICRA generated bounds on three examples where PUBS did not generate a bound, and on one example where CoFloCo did not generate a bound. From these experiments, we conclude that ICRA shows promise as a resource-bound-analysis tool.

8 RELATED WORK

Non-Linear Abstract Domains. There is a significant amount of prior work for reasoning about uninterpreted function symbols [Chang and Leino 2005; Gange et al. 2016; Gulwani et al. 2004] and non-linear arithmetic [Bagnara et al. 2005b; Colón 2004; Gulavani and Gulwani 2008; Müller-Olm and Seidl 2004]. A common theme in this work, including our own, is to introduce new dimensions in some relational abstract domain (such as polyhedra [Cousot and Halbwachs 1978] or octagons [Miné 2001]) to represent the values of terms that cannot be expressed in the domain *per se*.

Various techniques can be used to propagate information about the *alien* terms (borrowing the terminology of Chang and Leino [2005]) to the base domain. For example, Chang and Leino's congruence-closure abstract domain extends a base domain with terms built from arbitrary function symbols, and uses congruence-closure techniques to infer equations between terms [Chang and Leino 2005]. Colón [2004] extends the domain of affine relations with dimensions corresponding to all monomials less than some fixed degree, and infers new affine relations by closing under degree-bounded products; Bagnara et al. [2005b] generalize this idea, extending polyhedra to reason about polynomial inequalities. Gulavani and Gulwani [2008] present a framework for extending an abstract domain with alien functions using user-provided axioms for inference. As a specific instance, they extend polyhedra with terms involving multiplication, log, square root, and exponentiation, and give a set of axioms for reasoning about them. The wedge domain uses many of these ideas: the equational-saturation procedure for the wedge domain (Alg. 1) infers equalities using congruence closure (like Chang and Leino [2005]) and Gröbner-basis techniques (which generalize [Colón 2004]); the wedge domain deduces inequalities by applying a set of inference rules (like Gulavani and Gulwani [2008]); the Product rule generalizes the degree-bounded products of Bagnara et al. [2005b].

In abstract-interpretation terms, such techniques are using semantic reductions [Cousot and Cousot 1979] to improve the representation of an abstract-domain element. It is often useful to apply semantic reductions repeatedly, as a "best-effort" method to improve the representation, à la Granger's technique of local decreasing iteration [Granger 1992].

The design goals of wedges differ considerably from those of previous work, however. In particular, previous work has focused on abstract domains for *iterative* abstract interpreters, which compute loop invariants by evaluating the program under an abstract semantics until convergence on a property that abstracts all reachable states. Most such domains require fixing *a priori* the dimensions of the relational abstract domain (e.g., to all monomials of degree ≤ 2) to enforce convergence. In contrast, wedges are designed for *recurrence analysis*, wherein the fundamental problem of interest is symbolic abstraction (§4.3) (a problem not addressed by previous work). The association between non-linear terms and dimensions is more fluid in wedges, because there is no iterative process that we need to force to converge. Moreover, since abstract-domain operations are relatively infrequent in our analysis (in comparison to iterative program analysis), wedges are able to make use of computationally intensive operations, such as Gröbner-basis computation.

Other Methods for Non-Linear Invariant Generation. A variety of other methods have been developed for generating non-linear invariants [de Oliveira et al. 2016; Sankaranarayanan et al. 2004; Srikanth et al. 2017]. Srikanth et al. [2017] recently developed a method for determining if a program satisfies a given resource bound, which can be expressed using polynomials, exponentials, and logarithms. The approach uses an interpolating theorem prover for non-linear arithmetic. The method is property-directed, and thus fundamentally non-compositional.

Other Program Analyzers that Perform Recurrence Analysis. There are a number of program analyzers that find loop invariants by solving recurrence relations [Albert et al. 2008; Ammarguellat and Harrison 1990; Ancourt et al. 2010; Bagnara et al. 2005a; Blanc et al. 2010; Boigelot and Wolper

1994; Bozga et al. 2010; de Oliveira et al. 2016; Farzan and Kincaid 2015; Finkel and Leroux 2002; Humenberger et al. 2017; Jeannet et al. 2014; Kincaid et al. 2017; Kovács 2008; Rodríguez-Carbonell and Kapur 2004].

This paper builds upon previous work on compositional recurrence analysis (CRA) [Farzan and Kincaid 2015] and its interprocedural extension ICRA [Kincaid et al. 2017]. CRA uses recurrence analysis to *approximate* the transitive closure of *arbitrary loops* (rather than compute it exactly for a limited class of loops). Inspired by the methods used in so-called "Newtonian program analysis" [Esparza et al. 2010; Reps et al. 2016], the ICRA analysis framework of Kincaid et al. [2017] handles loops and recursion in a uniform way. In essence, their technique reduces arbitrary recursive programs (including ones with non-linear mutual recursion) to a sequence of ever-more-precise tail-recursive problems over a more complicated abstract domain. The latter are then analyzed using only slight generalizations of the CRA technique for extracting and solving recurrence relations for loops [Farzan and Kincaid 2015].

In this paper, we present techniques that improve (I)CRA's ability to generate non-linear invariants. In particular, [Farzan and Kincaid 2015]'s formulation of CRA is capable of extracting linear recurrence relations that have polynomial closed forms. It has limited support for reasoning about non-linear arithmetic: it uses a linearization technique that over-approximates a non-linear formula by a linear one. For example, consider the formula

$$\mathsf{x}' = \mathsf{x} + 1 \land \mathsf{y}' = \mathsf{y} + \mathsf{x} \times \mathsf{x} \land 0 \leq \mathsf{x} \leq 5\ .$$

Farzan and Kincaid's algorithm would compute the following linearization:

$$\mathsf{x}' = \mathsf{x} + 1 \land \mathsf{y} \leq \mathsf{y}' \leq \mathsf{y} + 25,$$

and thus lose the precise recurrence for y: $\mathsf{y}' = \mathsf{y} + \mathsf{x} \times \mathsf{x}$. In contrast, the method we present in this paper handles recurrence relations that involve polynomials and exponentials—and whose closed-form solutions include polynomials and exponentials (and therefore logarithmic relationships are implicit, as well). Consequently, this paper's techniques are different from those used by Farzan and Kincaid, for both extracting and solving recurrences. The wedge domain retains non-linear information rather than abstracting it away, thus allowing us to extract the recurrence $\mathsf{y}' = \mathsf{y} + \mathsf{x} \times \mathsf{x}$ for the example above.

Another distinction that can be drawn between this paper and Farzan and Kincaid's algorithm is that we extract and solve *matrix* recurrences. Matrix recurrences generalize the stratified recurrences of Farzan and Kincaid, which (like our stratified recurrences) orders recurrences into layers to aid solving. Unlike our notion of stratified recurrence (which is essentially a mechanism to cope with non-linear arithmetic), Farzan and Kincaid consider only linear arithmetic—all Farzan/Kincaid-style stratified recurrences appear in our matrix recurrences at stratum 0.

In both this paper and Farzan and Kincaid [2015], the key operation used for extracting recurrences is symbolic abstraction [Reps et al. 2004; Thakur and Reps 2012]. For more about symbolic abstraction in program analysis, see Reps and Thakur [2016]; Thakur [2014].

Compared to other program analyzers that are based on the principle of solving recurrence equations, the advantage of the technique described in this paper is that it can analyze general loops, including loops that contain (i) branching, (ii) nested loops, and (iii) non-deterministic assignments.

An important distinction between the recurrence solver presented in §6 and others that appear in the literature is that our solver will produce closed forms in terms of IIFs even if the closed form cannot be represented (e.g.) as an elementary function. Thus, for example, while the solver from Rodríguez-Carbonell and Kapur [2004] accepts the same class of stratified recurrences as our algorithm (§5.2), it only succeeds in computing closed forms in the case that all eigenvalues of every transformation matrix are real.

There exist algorithms for solving even broader classes of recurrences, such as the work of Humenberger et al. [2017]. For example, their technique can find closed forms that include factorials, whereas ours cannot. The methods of Rodríguez-Carbonell and Kapur [2004] and Humenberger et al. [2017] both search for polynomial equalities, whereas our technique can find invariants that are not polynomial equalities; also, neither Rodríguez-Carbonell and Kapur [2004] nor Humenberger et al. [2017] handle nested loops, whereas our technique does. Below, we show two examples of single-path loops that can be handled by our implementation (ICRA), but not by Aligator, which is the implementation of the techniques from Humenberger et al. [2017].

```
int x = 1, y = 1; for(n = 0; n < N; n++) { x *= 2; y *= 3; }
int a = 5, b = 3, c = 1; for(n = 0; n < N; n++) { a *= a; b += a; c -= a; }
```

In the first loop, no polynomial equality holds between x and y, so Aligator finds no invariant; ICRA is able to prove that $\mathsf{x} \leq \mathsf{y}$. In the second loop, ICRA cannot find closed-form solutions for a, b, or c, but it is able to prove the invariant $\mathsf{b} + \mathsf{c} = 4$ by finding a closed form for the sum ($\mathsf{b}' + \mathsf{c}' = \mathsf{b} + \mathsf{c}$); Aligator finds no invariant (it reports that the loop is "not P-solvable"). This example illustrates an advantage of ICRA: when it encounters a recurrence that it cannot solve, it may still produce a non-trivial overapproximation, unlike Aligator, which finds no invariants when it cannot solve the recurrences.

Recurrence Relations that Define C-Finite Sequences. The sequences defined by stratified recurrence relations fall into the class of *C-finite sequences*—sequences that satisfy linear recurrence relations with constant coefficients [Kauers and Paule 2011, Chapter 4].

Definition 8.1. ([Kovács 2008, Def. 3.2][Kauers and Paule 2011, §4.2]) A *C-finite recurrence* is a homogeneous linear recurrence equation with constant coefficients

$$f^{[n+r]} = a_{r-1} f^{[n+r-1]} + \ldots + a_1 f^{[n+1]} + a_0 f^{[n]},$$

where $n \in \mathbb{N}$, and $a_0, \ldots, a_{r-1}$ are constants with $a_0 \neq 0$. The recurrence is of order $r \in \mathbb{N}$.

A sequence $(f^{[n]})_{n=0}^{\infty}$ is a *C-finite sequence* if it satisfies a C-finite recurrence.

It is natural to ask whether stratified recurrence relations and C-finite recurrences are "equivalent."

- ⇒ The sequences defined by stratified recurrence relations are C-finite sequences. The essential argument appears in Kovács [2008]: a system of recurrences $\mathbf{x}^{[n]} = A\mathbf{x}^{[n-1]} + f(n)$ (with $f(n)$ a sum of polynomials multiplied by exponentials) can be transformed into a set of inhomogeneous C-finite recurrences, $x_i^{[n]} = a_{i,1} x_i^{[n]} + \cdots a_{i,k} x_i^{[n-k]} + g(n)$ (where $g(n)$ has the same form as $f(n)$), each of which can in turn can be transformed into a homogeneous C-finite recurrence.
- ⇐ The question is whether every C-finite sequence S is definable by a stratified recurrence relation. Suppose that S satisfies a C-finite recurrence R of order r. Then one can construct a program P (with a loop over loop-counter n) that computes S: the program will have one variable that holds the nth element of S, and $r - 1$ variables to hold onto the "lagging" values from prior iterations. All assignments would have linear expressions on the right-hand side. From P, one obtains a matrix recurrence M.

  In terms of the components of ICRA, the recurrence-extraction method from §5 would extract from P the matrix recurrence M (at stratum 0). OCRS would solve M, and would return an answer, which would contain one or more IIFs if the (standard) closed-form solution involves non-rationals (irrational numbers or complex numbers).

In this sense, §6 does not represent a method that can solve recurrences beyond what existing solvers can handle: there are other algorithms that might be applied to "solve" the recurrence relations that ICRA is capable of extracting.

The above equivalence argument must be tempered by the fact that not every program Q with a loop containing a variable that generates a C-finite sequence will be in the "normal form" enjoyed by P in case $\Leftarrow$. For such a Q, ICRA would extract a system of stratified matrix recurrences (involving multiple strata). If one knew the C-finite recurrence R_Q for Q, one could construct another program Q' in the normal form from case $\Leftarrow$; for Q', ICRA would extract a single matrix recurrence at stratum-0. (ICRA does not attempt to perform such a source-to-source transformation.)

Moreover, the notion of "solving" is predicated on the definition of "closed form," where closed forms need to be consumable by the client of the solver. In our work, the ability to process the output of the recurrence *solver* (§6) as the input to the recurrence *extractor* (§5) is essential for handling nested loops, as illustrated by Ex. 2.1. Although it is possible for the ICRA recurrence extractor to extract recurrences that the solver cannot solve, in practice, the solver presented in §6 is well-matched to the class of recurrences that are extracted by the algorithm given in §5.

Closed forms produced by the method given in §6 differ from those used in previous work (even if the class of sequences does not). Classical algorithms admit algebraic numbers, including complex numbers, in closed-form solutions. However, in analyzing an outer loop, the wedge domain needs to make use of answers obtained from the analysis of an inner loop. Because the wedge domain uses theorem proving modulo the theory of linear *real* arithmetic as a subroutine in the recurrence-extraction algorithm (§4 and §5), it would obviously not be able to make use of answers that involve complex numbers. Similarly, it would not be able to make use of answers that involve irrational numbers because the theory of linear real arithmetic admits only rational coefficients. Consequently, the solver described in §6 disallows irrational numbers and complex numbers from appearing in solutions, but allows there to be IIFs.

Finally, there is a syntactic difference between stratified recurrence relations and C-finite recurrences. The syntactic form of a system of stratified recurrence relations indicates explicitly the order in which individual recurrence equations should be solved. For instance, consider the program shown below on the left, from which ICRA would extract the loop-body transition formula shown on the right.

```
while(*) {  z = z + x*y;
            tmp = y;
            y = 2*x;
            x = x + tmp + 1; }
```

$$\varphi \stackrel{\text{def}}{=} \mathsf{x}' = \mathsf{x} + \mathsf{y} + 1 \land \mathsf{y}' = 2 \times \mathsf{x} \land \mathsf{z}' = \mathsf{z} + \mathsf{x} \times \mathsf{y}$$

The recurrence-extraction technique from §5 would identify the following system of stratified recurrence relations:

$$\text{Stratum 0:}\quad \begin{array}{rcl} \mathsf{x}^{[n]} &=& \mathsf{x}^{[n-1]} + \mathsf{y}^{[n-1]} + 1 \\ \mathsf{y}^{[n]} &=& 2\mathsf{x}^{[n-1]} \end{array} \qquad \text{Stratum 1:}\ \mathsf{z}^{[n]} = \mathsf{z}^{[n-1]} + \mathsf{x}^{[n-1]}\mathsf{y}^{[n-1]} \tag{25}$$

The recurrence equations at stratum 0 are solved first; their solutions are used to solve the recurrence equation at stratum 1.

The x, y, and z sequences that satisfy Eqn. (25) are all C-finite sequences. For instance, by writing

$$\begin{array}{rcrcrcrcl} & & \mathsf{x}^{[n]} & - & \mathsf{x}^{[n-1]} & - & 2\mathsf{x}^{[n-2]} & - 1 & = 0 \\ \mathsf{x}^{[n+1]} & - & \mathsf{x}^{[n]} & - & 2\mathsf{x}^{[n-1]} & & & - 1 & = 0 \end{array}$$

and subtracting the first line from the second, we see that the x sequence must satisfy

$$\mathsf{x}^{[n+1]} - 2\mathsf{x}^{[n]} - \mathsf{x}^{[n-1]} + 2\mathsf{x}^{[n-2]} = 0, \tag{26}$$

which is equivalent to the 3rd-order C-finite recurrence

$$\mathsf{x}^{[n+3]} = 2\mathsf{x}^{[n+2]} + \mathsf{x}^{[n+1]} - 2\mathsf{x}^{[n]}.$$

The recurrence equation for z, at stratum 1, is syntactically not a C-finite recurrence because it contains the *non-linear* term $\mathsf{x}^{[n-1]}\mathsf{y}^{[n-1]}$. [6]

The syntactic distinction between stratified recurrence relations and C-finite recurrences is important from the viewpoint of recurrence extraction. In particular, the loop-body formula φ does not entail Eqn. (26): $\mathsf{x}^{[n+1]}$ and $\mathsf{x}^{[n-2]}$ do not belong to the vocabulary of φ (which refers to only two copies of x—one in the pre-state and one in the post-state).

Generating Functions. Operational calculus is a transform method: a recurrence relation is converted into an algebraic problem; the algebraic problem is solved by algebraic manipulations; a solution in ordinary algebra is read out of the solution to the algebraic problem. Generating functions [Wilf 1994] are another transform method that can be used to solve recurrence relations [Flajolet and Sedgewick 2009]. With generating functions, a sequence $a = \langle a_0, a_1, a_2, \ldots \rangle$ is represented by the sum (or power series) $\sum_{n=0}^{\infty} a_n x^n$. Multiplication of two sequences is performed as the following convolution: $\sum_{n=0}^{\infty} a_n x^n * \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} \left(\sum_{i=0}^{n} a_i b_{n-i}\right) x^n$. Thus, the multiplication operator for generating functions is a convolution, whereas the multiplication in operational calculus is a convolution difference (cf. Eqn. (4)).

Operational calculus and generating functions create different terms for subsequent algebraic manipulation. Simpler terms might lead to a faster and simpler solver. However, as shown by the following examples, neither technique can be said to have an advantage in this respect:

Operational calculus creates simpler terms: In operational calculus, the constant sequence $\lambda n.c$ is represented by $\underline{c} = \langle c, c, \ldots \rangle$, whereas the generating function is $\frac{c}{1-x}$. The respective values for the sequence $\lambda n.n^2$ are $\frac{2}{(q-1)^2} + \frac{1}{q-1}$ and $\frac{x(x+1)}{(1-x)^3}$.

Generating functions create simpler terms: The generating function for the geometric sequence $\lambda n.\alpha^n$ is $\frac{1}{1-\alpha x}$, whereas the operational-calculus value is $\frac{q-\underline{1}}{q-\underline{\alpha}}$.

Operational calculus and generating functions create terms of similar complexity: The left-shift operators are q and $\frac{1}{x}$, respectively; the right-shift operators are v and x, respectively.

Operational Calculus. We are not aware of other work that solves recurrences automatically using operational calculus; Berg [1967] discusses recurrence solving in the style of a mathematics textbook, but provides no explicit algorithm. §6.2 describes our mechanization of Berg's approach. §6.3 concerns an extension to operational calculus not found in Berg (or elsewhere, as far as we know).

ACKNOWLEDGMENTS

We thank the anonymous reviewers and shepherd A. Rybalchenko for their comments on the submission. This work was supported in part by a gift from Rajiv and Ritu Batra; by AFRL under DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from WARF. Opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

REFERENCES

E. Albert, P. Arenas, S. Genaim, and G. Puebla. 2008. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *SAS*. 221–237.

E. Albert, S. Genaim, and A.N. Masud. 2013. On the inference of resource usage upper, lower bounds. *Trans. on Computational Logic* 14, 3 (2013).

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

[6] A C-finite *recurrence* can contain only linear terms; however, a C-finite *sequence* may satisfy a recurrence relation that contains a non-linear term.

C. Ancourt, F. Coelho, and F. Irigoin. 2010. A Modular Static Analysis Approach to Affine Loop Invariants Detection. *Electron. Notes Theor. Comput. Sci.* 267, 1 (Oct. 2010), 3–16.

T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei. 2017. Decomposition Instead of Self-composition for Proving the Absence of Timing Channels. In *PLDI*.

R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. 2005a. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. *CoRR* abs/cs/0512056 (2005).

R. Bagnara, E. Rodríguez-Carbonell, and E. Zaffanella. 2005b. Generation of Basic Semi-Algebraic Invariants Using Convex Polyhedra. In *SAS*.

G. Barthe, J. Crespo, and C. Kunz. 2011. Relational Verification Using Product Programs. In *Proceedings of the 17th International Conference on Formal Methods (FM)*.

G. Barthe, P.R. D'Argenio, and T. Rezk. 2004. Secure Information Flow by Self-Composition. In *Comp. Sec. Found. Workshop*.

L. Berg. 1967. *Introduction to the Operational Calculus*. North-Holland Publishing Co., Amsterdam.

D. Beyer and M.E. Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *CAV*.

R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. 2010. ABC: Algebraic Bound Computation for Loops. In *Int. Conf. on Logic for Programming, Art. Intell., and Reasoning*. 103–118.

B. Boigelot and P. Wolper. 1994. Symbolic verification with periodic sets. In *CAV*. 55–67.

M. Bozga, R. Iosif, and F. Konečný. 2010. Fast Acceleration of Ultimately Periodic Relations. In *CAV*. 227–242.

M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. 2014. Alternating runtime and size complexity analysis of integer programs. In *TACAS*.

B. Buchberger. 1976. A Theoretical Basis for the Reduction of Polynomials to Canonical Forms. *SIGSAM Bull.* 10, 3 (Aug. 1976), 19–29.

B.-Y.E. Chang and K.R.M Leino. 2005. Abstract Interpretation with Alien Expressions and Heap Structures. In *VMCAI*.

J. S. Cohen. 2003. *Computer Algebra and Symbolic Computation: Mathematical Methods*. A K Peters/CRC Press.

M. A. Colón. 2004. Approximating the Algebraic Relational Semantics of Imperative Programs. In *SAS*. 296–311.

P. Cousot and R. Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *POPL*. 269–282.

P. Cousot and N. Halbwachs. 1978. Automatic Discovery of Linear Constraints Among Variables of a Program. In *POPL*.

D. A. Cox, J. Little, and D. O'Shea. 2015. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra* (4th ed.). Springer Publishing Company, Incorporated.

L. de Moura and N. Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*.

S. de Oliveira, S. Bensalem, and V. Prevosto. 2016. Polynomial Invariants by Linear Algebra. In *ATVA*.

I. Dillig, T. Dillig, B. Li, and K. McMillan. 2013. Inductive Invariant Generation via Abductive Inference. In *OOPSLA*.

J. Esparza, S. Kiefer, and M. Luttenberger. 2010. Newtonian Program Analysis. *J. ACM* 57, 6 (2010).

A. Farzan and Z. Kincaid. 2015. Compositional Recurrence Analysis. In *FMCAD*.

A. Finkel and J. Leroux. 2002. How to Compose Presburger-Accelerations: Applications to Broadcast Protocols. In *FST TCS*. 145–156.

P. Flajolet and R. Sedgewick. 2009. *Analytic Combinatorics*. Cambridge University Press. pdfs.semanticscholar.org/d347/dbb4b2eea7fca0183b55112b9cc07faa51ff.pdf.

A. Flores-Montoya and R. Hähnle. 2014. Resource analysis of complex programs with cost equations. In *APLAS*.

G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. 2016. An Abstract Domain of Uninterpreted Functions. In *VMCAI*.

P. Granger. 1992. Improving the Results of Static Analyses Programs by Local Decreasing Iteration. In *FSTTCS*.

B.S. Gulavani and S. Gulwani. 2008. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *CAV*.

S. Gulwani, A. Tiwari, and G.C. Necula. 2004. Join Algorithms for the Theory of Uninterpreted Functions. In *FSTTCS*.

A. Gurfinkel, T. Kahsai, A. Komuravelli, and J.A. Navas. 2015. The SeaHorn Verification Framework. In *CAV*.

M. Heizmann, J. Christ, D. Dietsch, E. Ermis, J. Hoenicke, M. Lindenmann, A. Nutz, C. Schilling, and A. Podelski. 2013. Ultimate Automizer with SMTInterpol (Competition Contribution). In *TACAS*.

A. Humenberger, M. Jaroschek, and L. Kovacs. 2017. Automated Generation of Non-Linear Loop Invariants Utilizing Hypergeometric Sequences. In *ISSAC*.

J.-G.-M. 1983. Hypernumbers. I. Algebra. *Studia Math.* 77 (1983), 3–16. matwbn.icm.edu.pl/ksiazki/sm/sm77/sm7712.pdf Originally published in Polish in 1944.

B. Jeannet and A. Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*.

B. Jeannet, P. Schrammel, and S. Sankaranarayanan. 2014. Abstract Acceleration of General Linear Loops. In *POPL*. 529–540.

M. Kauers and P. Paule. 2011. *The Concrete Tetrahedron*. SpringerWienNewYork.

Z. Kincaid, J. Breck, A. Forouhi Boroujeni, and T. Reps. 2017. Compositional Recurrence Analysis Revisited. In *PLDI*.

L. Kovács. 2008. Reasoning Algebraically About P-Solvable Loops. In *TACAS*.

A. Miné. 2001. The Octagon Abstract Domain. In *Working Conf. on Rev. Eng.* 310–322.

M. Müller-Olm and H. Seidl. 2004. Precise Interprocedural Analysis through Linear Algebra. In *POPL*.

T. Reps, M. Sagiv, and G. Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *VMCAI*. 252–266.

T. Reps and A. Thakur. 2016. Automating Abstract Interpretation. In *VMCAI*.

T. Reps, E. Turetsky, and P. Prabhu. 2016. Newtonian Program Analysis via Tensor Product. In *POPL*.

E. Rodríguez-Carbonell and D. Kapur. 2004. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *ISSAC*. 266–273.

S. Sankaranarayanan, H.B. Sipma, and Z. Manna. 2004. Non-Linear Loop Invariant Generation using Gröbner Bases. In *POPL*.

A. Srikanth, B. Sahin, and W.R. Harris. 2017. Complexity Verification Using Guided Theorem Enumeration. In *POPL*.

T. Terauchi and A. Aiken. 2005. Secure Information Flow As a Safety Problem. In *Static Analysis Symp.*

A. Thakur. 2014. *Symbolic Abstraction: Algorithms and Applications*. Ph.D. Dissertation. Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI. Tech. Rep. 1812.

A. Thakur and T. Reps. 2012. A Method for Symbolic Computation of Abstract Operations. In *CAV*.

H. Wilf. 1994. *Generatingfunctionology, 2nd. Ed.* Academic Press. www.math.upenn.edu/ wilf/gfologyLinked2.pdf.