Polynomial Invariant Generation for Non-deterministic Recursive Programs

Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, Ehsan Kafshdar Goharshady

Polynomial Invariant Generation for Non-deterministic Recursive Programs

KRISHNENDU CHATTERJEE, IST Austria (Institute of Science and Technology Austria), Austria
HONGFEI FU, Shanghai Jiao Tong University, China
AMIR KAFSHDAR GOHARSHADY, IST Austria (Institute of Science and Technology Austria), Austria
EHSAN KAFSHDAR GOHARSHADY, Ferdowsi University of Mashhad, Iran

We consider the classical problem of invariant generation for programs with polynomial assignments and focus on synthesizing invariants that are a conjunction of strict polynomial *inequalities*. We present a sound and semi-complete method based on positivstellensätze, i.e. theorems in semi-algebraic geometry that characterize positive polynomials over a semi-algebraic set. To the best of our knowledge, this is the first invariant generation method to provide completeness guarantees for invariants consisting of polynomial inequalities. Moreover, on the theoretical side, the worst-case complexity of our approach is subexponential, whereas the worst-case complexity of the previously-known complete method ([Colón et al. 2003], CAV 2003), which could only handle linear invariants, is exponential. On the practical side, we reduce the invariant generation problem to quadratic programming (QCLP), which is a classical optimization problem with many industrial solvers. Finally, we demonstrate the applicability of our approach by providing experimental results on several academic benchmarks.

Additional Key Words and Phrases: Invariant generation, Positivstellensätze, Polynomial programs

1 INTRODUCTION

Invariants. Given a program, an assertion at a program location that is always satisfied by the program variables when the location is reached is called an *invariant*. Invariants are essential for many different formal and quantitative analyses of various types of programs [Chatterjee et al. 2017a; Halbwachs et al. 1997; Henzinger and Ho 1994; Ngo et al. 2018]. Invariant generation is a classical problem in formal verification and programming languages, and has been studied for decades, e.g. for analysis of safety and liveness properties of programs [Cousot and Cousot 1977; Cousot and Halbwachs 1978; Manna and Pnueli 1995].

Inductive Invariants. An *inductive* assertion is an assertion that holds at a location for the first visit to the location and is preserved under every cyclic execution path from and to the location. Inductive assertions are guaranteed to be invariants, and the well-established method to prove that an assertion is an invariant is to find an inductive invariant that strengthens it [Colón et al. 2003; Manna and Pnueli 1995].

Previous Approaches. Given the importance of invariant generation, the problem has received significant attention over the decades. A primary technique is *abstract interpretation* [Cousot and Halbwachs 1978] that considers symbolic execution, and its termination is guaranteed by an imprecise exploration operator called *widening*. However, this method is not complete and exploration heuristics do not always guarantee convergence.

Linear vs Polynomial Invariants. For *linear* invariant generation over programs with *linear* updates, a sound and complete methodology was obtained by [Colón et al. 2003]. In this work, we consider programs with *polynomial* updates and the problem of generating *polynomial* invariants, i.e. invariants that are a conjunction of polynomial *inequalities* over program variables. Hence, our setting is more general than [Colón et al. 2003] in terms of the programs we analyze, and also the desired invariants. To the best of our knowledge, there is no previous approach in the literature that provides completeness guarantees for this problem. On the other hand, efficient

sound methodologies without completeness guarantees were proposed in [Farzan and Kincaid 2015; Kincaid et al. 2017, 2018].

Motivation for Polynomial Invariant Generation. Invariants are used as prerequisites in a wide variety of program analysis methods, and their accuracy can directly impact the effectiveness of those analyses. Given that polynomial invariants provide a higher degree of accuracy in comparison with linear invariants, using them improves existing solutions to many classical problems in programming languages, including the following:

- *Safety Verification.* Consider the fundamental problem of safety verification which is one of the most well-studied model checking problems: Given a program and a set of safety assertions that must hold at specific points of the program, prove that the assertions hold or report that they might be violated by the program. Many existing approaches for safety verification rely on invariants to prove the desired assertions (see [Albarghouthi et al. 2012; Alur et al. 2006; Manna and Pnueli 1995; Padon et al. 2016]). In these cases, weak invariants can lead to an increase in false positives, i.e. if the supplied invariants are inaccurate and grossly overestimate the program's behavior, then the verifier might falsely infer that a true assertion can be violated.
- *Termination Analysis.* A principal approach in proving termination of programs is to synthesize ranking functions, i.e. well-founded functions whose value decreases at every step of the program [Floyd 1993]. Virtually all synthesis algorithms for ranking functions depend on invariants, e.g. see [Bradley et al. 2005; Chen et al. 2007; Colón and Sipma 2001]. Having inaccurate invariants, such as using linear instead of polynomial invariants, can lead to a failure in the synthesis and hence inability to prove termination. The same point also applies to termination analysis of probabilistic programs [Chakarov and Sankaranarayanan 2013].
- *Inferring Complexity Bounds.* Another fundamental problem is to find automated algorithms that infer asymptotic complexity bounds on the runtime of (recursive) programs. Current algorithms for tackling this problem, such as [Chatterjee et al. 2017a], rely heavily on invariants and their accuracy. Inaccurate invariants can lead to an over-approximation of complexity or even failure of the algorithm to synthesize any complexity bound.

Note that the points mentioned above not only justify the use of polynomial invariants due to their higher accuracy, but also demonstrate the need for approaches with completeness guarantees. Lacking completeness guarantees is a gap in the state-of-the-art in polynomial invariant generation. The current work aims to address this gap.

Our Contribution. We consider two variants of the invariant generation problem. Informally, the *weak* variant asks for an optimal invariant w.r.t. a given objective function, while the *strong* variant asks for a representative set of all invariants. Our contributions are as follows:

(i) *Soundness and Semi-completeness.* We present a sound and semi-complete method to generate polynomial invariants for programs with polynomial updates and solve the strong invariant generation problem. Our completeness requires a compactness condition that is satisfied by all real-world programs (see Remark 5). We also show that, using the standard notions of pre and post-conditions, our method can be extended to handle recursion as well.

(ii) *Theoretical Contribution.* We show that the worst-case complexity of our procedure is *subexponential*. Note that, in comparison, complexity of the procedure in [Colón et al. 2003], which is sound and complete for *linear* invariants, is *exponential*, whereas we show how to generate *polynomial* invariants in subexponential time.

(iii) *Practical Contribution.* We reduce the weak invariant generation problem to solving a quadratically-constrained linear program (QCLP). Solving QCLPs is a hot topic and active area of research in optimization. Moreover, there are many industrial solvers for handling real-world instances

of quadratic programming and, using our algorithm, practical improvements to such solvers carry over to polynomial invariant generation.

Hence, our main contribution is theoretical, i.e. presenting the first sound and *semi-complete* method for generating polynomial invariants. Nevertheless, we also demonstrate the applicability of our approach by providing experimental results on several academic examples from [Rodríguez-Carbonell 2018] that require polynomial invariants. Unsurprisingly, we observe that our approach is slower than previous sound but incomplete methods, so there is a trade-off between completeness and efficiency. However, we expect practical improvements in solving QCLPs to narrow the efficiency gap in the future.

Techniques. In terms of techniques, while the approach of [Colón et al. 2003] uses Farkas' lemma and quantifier elimination to generate linear invariants, our technique is based on a positivstellensatz*. Moreover, our method replaces the quantifier elimination step with either (i) an algorithm of [Grigor'ev and Vorobjov 1988] for characterizing solutions of systems of polynomial inequalities or (ii) a reduction to QCLP.

1.1 Related works

Invariant Generation. Automated invariant generation is an important research topic that has received much attention in the past years, and various classes of approaches have been proposed, including recurrence analysis [Farzan and Kincaid 2015; Humenberger et al. 2017; Kincaid et al. 2017, 2018], abstract interpretation [Bagnara et al. 2005; Chakarov and Sankaranarayanan 2014; Cousot et al. 2005; Müller-Olm and Seidl 2004; Rodríguez-Carbonell and Kapur 2007], constraint solving [Chatterjee et al. 2017b; Chen et al. 2015; Colón et al. 2003; Cousot 2005; de Oliveira et al. 2016; Feng et al. 2017; Katoen et al. 2010; Lin et al. 2014; Rodríguez-Carbonell and Kapur 2004; Sankaranarayanan et al. 2004; Yang et al. 2010], inference [Dillig et al. 2013; Gulwani et al. 2009; Sharma and Aiken 2016], interpolation [McMillan 2008], symbolic execution [Csallner et al. 2008], dynamic analysis [Nguyen et al. 2012] and learning [Garg et al. 2016].

Summary. A succinct summary of the results of the literature w.r.t. types of assignments, type of generated invariants (polynomial, linear, etc), programming language features that can be handled (i.e. non-determinism, probability and recursion), soundness, completeness, and whether the approach can handle weak/strong invariant generation is presented in Table 1. Moreover, for approaches that are applicable to weak/strong invariant generation, the respective runtimes are also reported. Note that most of the previous approaches, which are included in Table 1 for the sake of completeness, are indeed incomparable with our approach, given that they handle different problems, e.g. different types of programs. As compared to previous works, we present the first sound and semi-complete methodology for polynomial invariant generation, and our complexity guarantee (subexponential) is better than the previous exponential-time complexity of the sound and complete methods for linear invariants.

Recurrence Analysis. While approaches based on recurrence analysis can derive exact polynomial invariants, they are applicable to a restricted class of programs where closed-form solutions exist. Our approach however does not require closed-form solutions and is applicable to all programs.

Abstract Interpretation. Abstract interpretation is the oldest and most classical approach to invariant generation [Cousot and Cousot 1977; Cousot and Halbwachs 1978]. However, unlike our approach, it cannot provide any guarantee of completeness.

*A positivstellensatz (German for "positive locus theorem", plural: positivstellensätze) is a theorem in real semi-algebraic geometry that characterizes positive polynomials over a semi-algebraic set.

Approach	Assignments and Guards	Invariants	Nondet	Rec	Prob	Sound	Complete	Weak	Strong
This Work	Polynomial	Polynomial	✓	✓	×	✓	✓♦	✓ QCLP	✓ Subexp
[Colón et al. 2003] CAV 2003	Linear[c]	Linear	✓	×	×	✓	✓	✓ Exp†	✓ Exp†
[Feng et al. 2017] ATVA 2017	Polynomial	Polynomial	×	×	✓	✓	✓[a]	✓ Poly	×
[Hrushovski et al. 2018] LICS 2018	Linear‡	Polynomial *Equalities*	✓	×	×	✓‡	✓‡	×	✓‡,[b]
[Kincaid et al. 2018] POPL 2018	Polynomial, Exponential, Logarithmic	Polynomial, Exponential, Logarithmic	✓	×	×	✓	×	×	×
[Rodríguez-Carbonell and Kapur 2004]* ISSAC 2004	Polynomial, Exponential	Polynomial *Equalities*	✓	×	×	✓	✓	✓[b]	✓[b]
[Sankaranarayanan et al. 2004] POPL 2004	Polynomial[c]	Polynomial *Equalities*	✓	×	×	✓	✓[b]	✓[b]	✓[b]
[Farzan and Kincaid 2015] FMCAD 2015	General[d]	General[d]	✓	×[e]	×	✓	×	×	×
[Kincaid et al. 2017] PLDI 2017	General	General	✓	✓	×	✓	×	×	×
[de Oliveira et al. 2016] ATVA 2016	Polynomial, *Without Conditional Branching*	Polynomial *Equalities*	✓	×	×	✓	✓	✓ Poly	✓ Poly
[Humenberger et al. 2017]* ISSAC 2017	Polynomial‡	Polynomial *Equalities*	✓	×	×	✓	✓‡	✓‡,[b]	✓‡,[b]

♦ The approach is semi-complete and assumes compactness (see Lemma 3.7 and Remark 5).
† Similar to our approach, [Colón et al. 2003] generates a system of quadratic inequalities, but then applies quantifier elimination, leading to exponential runtime. This can be reduced to match our runtime by employing the same techniques for solving the system.
‡ Treats branching conditions as non-determinism.
∗ Does not support nested loops.
[a] The approach is semi-complete.
[b] Uses Gröbner basis computations (super-exponential worst-case runtime in theory).
[c] Considers general transition systems instead of programs.
[d] Handles non-linearity using linearization heuristics.
[e] Can be extended to handle recursion (see [Kincaid et al. 2017]).

Table 1. A summary of invariant generation methods in the literature.

Constraint Solving. Our approach falls in this category. First, our approach can handle polynomial invariants, thus extending the approaches based on linear arithmetics, such as [Chatterjee et al. 2017b; Colón et al. 2003; de Oliveira et al. 2016; Katoen et al. 2010]. Second, our approach can handle invariants consisting of polynomial *inequalities*, whereas several previous approaches generate polynomial *equalities* [Rodríguez-Carbonell and Kapur 2004; Sankaranarayanan et al. 2004]. Third, our approach is semi-complete, thus it is more accurate than approaches with relaxations (e.g. [Cousot 2005; Lin et al. 2014]). Fourth, compared to previous complete approaches that solve formulas in the first-order theory of reals (e.g. [Chen et al. 2015; Yang et al. 2010]) to generate invariants for a more limited set of programs, our approach has lower computational complexity, i.e. our approach is subexponential, whereas quantifier elimination and solving first-order formulas take exponential time.

Comparison with [Feng et al. 2017]. Finally, we compare our approach with the most related work, i.e. [Feng et al. 2017]. A main difference is that our approach can find a representative set of all solutions, but [Feng et al. 2017] might miss some solutions, i.e. it only guarantees to find

at least one solution as long as the problem is feasible. In terms of techniques, [Feng et al. 2017] uses Stengle's positivstellensatz, while we use Putinar's positivstellensatz, Schweighofer's theorem and the algorithm of Grigor'ev and Vorobjov. Moreover, [Feng et al. 2017] considers the class of probabilistic programs without non-determinism and only focuses on single probabilistic while loops, while we consider programs in general form, with non-determinism and recursion, but without probabilistic behavior.

2 PRELIMINARIES

In this section, we define the syntax and semantics of the programs we are considering in this work, and then formally define the concept of invariants and the problem of invariant generation. We fix two disjoint finite sets: the set $\mathbf{V}$ of program variables and the set $\mathbf{F}$ of function names. We define several basic notions and then present the syntax.

Valuations. A *valuation* over a set $W \subseteq \mathbf{V}$ of variables is a function $\nu : W \to \mathbb{R}$ that assigns a real value to each variable in W. We denote the set of all valuations on W by $\mathbb{R}^W$. We sometimes use a valuation ν over a set $W' \subset W$ of variables as a valuation over W. In such cases, we assume that $\nu(w) = 0$ for every $w \in W \setminus W'$. Given a valuation ν, a variable v and $x \in \mathbb{R}$, we write $\nu[v \leftarrow x]$ to denote a valuation ν' such that $\nu'(v) = x$ and ν' agrees with ν for every other variable.

Polynomial Arithmetic Expressions. A *polynomial arithmetic expression* $\mathfrak{e}$ over $\mathbf{V}$ is an expression built from the variables in $\mathbf{V}$, real constants, and the arithmetic operations of addition, subtraction and multiplication.

Propositional Polynomial Predicates. A *propositional polynomial predicate* is a propositional formula built from (i) *atomic assertions* of the form $\mathfrak{e}_1 \bowtie \mathfrak{e}_2$, where $\mathfrak{e}_1$ and $\mathfrak{e}_2$ are polynomial arithmetic expressions, and $\bowtie \in \{<, \leq, \geq, >\}$ and (ii) propositional connectives $\vee$, $\wedge$ and $\neg$. The satisfaction relation $\models$ between a valuation ν and a propositional polynomial predicate ϕ is defined in the natural way, i.e. by substituting the variables with their values in ν and evaluating the resulting boolean expression.

2.1 Syntax

In this work, we consider non-deterministic recursive programs with polynomial assignments and guards. Our syntax is formally defined by the grammar in Figure 1[†]. Below, we intuitively explain some aspects of the syntax:

$$
\begin{array}{rcl}
\langle \mathit{prog} \rangle & ::= & \langle \mathit{func} \rangle \mid \langle \mathit{func} \rangle\ \langle \mathit{prog} \rangle \\
\langle \mathit{func} \rangle & ::= & \langle \mathit{fname} \rangle\ \text{`('}\ \langle \mathit{varlist} \rangle\ \text{`)'}\ \text{`\{'}\ \langle \mathit{stmtlist} \rangle\ \text{`\}'} \\
\langle \mathit{varlist} \rangle & ::= & \langle \mathit{var} \rangle \mid \langle \mathit{var} \rangle\ \text{`,'}\ \langle \mathit{varlist} \rangle \\
\langle \mathit{stmtlist} \rangle & ::= & \langle \mathit{stmt} \rangle \mid \langle \mathit{stmt} \rangle\ \text{`;'}\ \langle \mathit{stmtlist} \rangle \\
\langle \mathit{stmt} \rangle & ::= & \text{`}\mathbf{skip}\text{'} \mid \langle \mathit{var} \rangle\ \text{`:='}\ \langle \mathit{expr} \rangle \\
 & & \mid \text{`}\mathbf{if}\text{'}\ \langle \mathit{bexpr} \rangle\ \text{`}\mathbf{then}\text{'}\ \langle \mathit{stmtlist} \rangle\ \text{`}\mathbf{else}\text{'}\ \langle \mathit{stmtlist} \rangle\ \text{`}\mathbf{fi}\text{'} \\
 & & \mid \text{`}\mathbf{if}\text{'}\ \text{`}\star\text{'}\ \text{`}\mathbf{then}\text{'}\ \langle \mathit{stmtlist} \rangle\ \text{`}\mathbf{else}\text{'}\ \langle \mathit{stmtlist} \rangle\ \text{`}\mathbf{fi}\text{'} \\
 & & \mid \text{`}\mathbf{while}\text{'}\ \langle \mathit{bexpr} \rangle\ \text{`}\mathbf{do}\text{'}\ \langle \mathit{stmtlist} \rangle\ \text{`}\mathbf{od}\text{'} \\
 & & \mid \langle \mathit{var} \rangle := \langle \mathit{fname} \rangle\ \text{`('}\ \langle \mathit{varlist} \rangle\ \text{`)'} \\
 & & \mid \text{`}\mathbf{return}\text{'}\ \langle \mathit{expr} \rangle
\end{array}
$$

Fig. 1. The syntax of non-deterministic recursive programs. See Appendix A for more details.

- *Variables and Function Names.* Expressions $\langle \mathit{var} \rangle$ (resp. $\langle \mathit{fname} \rangle$) range over the set $\mathbf{V}$ (resp. $\mathbf{F}$).

[†]See Appendix A for a more detailed grammar.

- *Arithmetic and Boolean Expressions.* Expressions ⟨*expr*⟩ range over all polynomial arithmetic expressions over program variables. Similarly, expressions ⟨*bexpr*⟩ range over propositional polynomial predicates.
- *Statements.* A statement can be in one of the following forms:
  - A special type of statement called '**skip**' which does not do anything,
  - An assignment statement (⟨*var*⟩ ':=' ⟨*expr*⟩),
  - A conditional branch ('**if**' ⟨*bexpr*⟩) in which the ⟨*bexpr*⟩ serves as the branching condition;
  - A non-deterministic branch ('**if** ⋆'),
  - A while-loop ('**while**' ⟨*bexpr*⟩) in which the ⟨*bexpr*⟩ serves as the loop guard;
  - A function call statement (⟨*var*⟩ := ⟨*fname*⟩ '(' ⟨*varlist*⟩ ')') which calls the function specified by ⟨*fname*⟩ using the parameters specified in the ⟨*varlist*⟩ and assigns the resulting returned value to the variable on its left hand side;
  - A return statement ('**return**' ⟨*expr*⟩) that ends the current function and returns the value of the expression ⟨*expr*⟩ and the control to the parent function or ends the program if there is no parent function.
- *Programs and Functions.* A program is simply a list of functions. Each function has a name, a set of parameters and a body. The function body is simply a sequence of statements. We assume that there is a distinguished function $f_{\texttt{main}}$ that serves as the starting point of the program.

Simple vs. Recursive Programs. We call a program *simple*, or *non-recursive*, if it contains only one function and no function call statements. Otherwise, we say that the program is *recursive*.

Program Counters (Labels). We assign a unique *program counter* to each statement of the program and the endpoint of every function. We also refer to program counters as *labels*. We use $\mathbf{L}$ to denote the set of all labels. We denote the first label in a function f by $\ell^f_{\texttt{in}}$ and the label of its endpoint by $\ell^f_{\texttt{out}}$.

Types of Labels. We partition the set $\mathbf{L}$ of labels as follows:

- $\mathbf{L}_a$: Labels of **a**ssignment, skip or return statements,
- $\mathbf{L}_b$: Labels of conditional **b**ranching (if) and while-loop statements,
- $\mathbf{L}_c$: Labels of function **c**all statements,
- $\mathbf{L}_d$: Labels of non-**d**eterministic branching statements,
- $\mathbf{L}_e$: Labels of the **e**ndpoints of functions.

Example 1. Consider the simple program in Figure 2. The numbers on the left are the labels and their subscripts denote their types. We will use this program as our running example. It contains a single function sum that takes a parameter n and then non-deterministically sums up some of the numbers between 1 and n and returns the summation. Our goal is to prove that the return value of sum is always less than $0.5 \cdot n^2 + 0.5 \cdot n + 1$.

■

2.2 Semantics

New Variables. In the sequel, we fix a non-deterministic recursive program P with variables $\mathbf{V}$, functions $\mathbf{F}$ and labels $\mathbf{L}$. For each function $f \in \mathbf{F}$, whose header is of the form $f(v_1, \ldots, v_n)$, we define $n+1$ new variables $\texttt{ret}^f, \bar{v}_1, \ldots, \bar{v}_n$. Intuitively, $\texttt{ret}^f$ models the return value of the function f and each variable $\bar{v}_i$ holds the value passed to the function f from its parent function for parameter v_i without allowing f to change it.

```
    sum(n) {
1_a: i := 1;
2_a: s := 0;
3_b: while i ≤ n do
4_d:       if ⋆ then
5_a:            s := s + i
           else
6_a:            skip
           fi;
7_a:       i := i + 1
     od;
8_a: return s
9_e: }
```

Fig. 2. A non-deterministic summation program

Notation. We define $\mathbf{V}^f_* := \{\mathtt{ret}^f, v_1, \ldots, v_n, \bar{v}_1, \ldots, \bar{v}_n\}$ and let $\mathbf{V}^f$ be the set containing all members of $\mathbf{V}^f_*$, as well as any variable that appears somewhere in the body of the function f. W.l.o.g. we assume that the $\mathbf{V}^f$'s are pairwise disjoint. Moreover, we write $\mathbb{R}^f$ as a shorthand for $\mathbb{R}^{\mathbf{V}^f}$. In other words, $\mathbb{R}^f$ is the set of all valuations over the variables that appear in f, including its header, its body and its new variables. Similarly, we define $\mathbf{L}^f$ as the set of labels that occur in f. We use the standard notion of control flow graphs as in [Allen 1970; Chatterjee et al. 2017a].

CFGs. A *Control Flow Graph (CFG)* is a triple $(\mathbf{F}, \mathbf{L}, \rightarrow)$ where:

- $\mathbf{F}$ is the set of functions;
- the labels $\mathbf{L}$ form the set of vertices, and
- $\rightarrow$ is a relation whose members are triples (ℓ, α, ℓ') in which the source label ℓ and the target label ℓ' are in the same $\mathbf{L}^f$, i.e. they correspond to labels in the same function, the source label is not the end of function label, i.e. $\ell \neq \ell^f_{\mathsf{out}}$, and α is one of the following: (i) an update function $\alpha : \mathbb{R}^f \rightarrow \mathbb{R}^f$ if $\ell \in \mathbf{L}_a$, or (ii) a propositional polynomial predicate over $\mathbf{V}^f$ if $\ell \in \mathbf{L}_b$, or (iii) $\bot$ if $\ell \in \mathbf{L}_c$, or (iv) $\star$ if $\ell \in \mathbf{L}_d$.

Intuitively, we say that a CFG $(\mathbf{F}, \mathbf{L}, \rightarrow)$ is the CFG of program P if (i) for each label $\ell \in \mathbf{L}$, the successors of ℓ in $\rightarrow$ are the labels that are in the same function as ℓ and can possibly be executed right after ℓ, and (ii) the α's correspond to the behavior of the program, e.g. if $(\ell, \alpha, \ell') \in \rightarrow$, ℓ is an if statement and ℓ' is the first statement in its '**else**' part, then α should be the negation of the if condition. See [Chatterjee et al. 2017a] for more details. Note that a return statement in a function f changes the value of the variable $\mathtt{ret}^f$ and is succeeded by the endpoint label ℓ^f_{out}.

Example 2. Figure 3 provides the CFG of the program in Example 1. ■

We use the classical semantics for non-deterministic recursive programs.

Stack Elements and Configurations. A *stack element* ξ is a tuple (f, ℓ, v) where $f \in \mathbf{F}$ is a function and $\ell \in \mathbf{L}^f$ and $v \in \mathbb{R}^f$ are respectively a label and a valuation in f. A configuration $\kappa = \langle \xi_i \rangle_{i=0}^n$ is a finite sequence of stack elements.

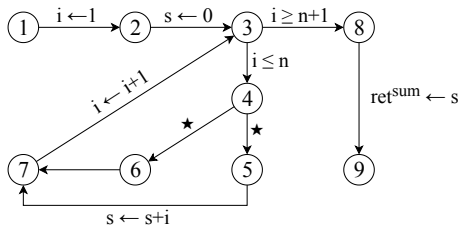

Fig. 3. CFG of the Program in Example 1 (Figure 2)

Notation. Given a configuration κ and a stack element ξ, we write $\kappa \cdot \xi$ to denote the configuration obtained by adding ξ to the end of κ. Also, we define κ^{-i} as the sequence obtained by removing the last i stack elements of κ.

A *run* is an infinite sequence of configurations that starts at the first label of $f_{\texttt{main}}$ and follows the requirements of the CFG. Intuitively, a run models the sequence of configurations that are met in an execution of the program. We now formalize this notion.

Runs. Given a program P and its CFG $(\mathbf{F}, \mathbf{L}, \rightarrow)$, a *run* is a sequence $\rho = \{\kappa_i\}_{i=0}^{\infty}$ of configurations such that:

- $\kappa_0 = \langle (f_{\texttt{main}}, \ell_{\texttt{in}}^{f_{\texttt{main}}}, \nu) \rangle$ for some valuation $\nu \in \mathbb{R}^{f_{\texttt{main}}}$. Intuitively, a run begins from the $f_{\texttt{main}}$ function.
- If $|\kappa_i| = 0$, then $|\kappa_{i+1}| = 0$, too. Informally, this case corresponds to when the program has already terminated.
- Let $\xi = (f, \ell, \nu)$ be the last stack element in κ_i. Then, κ_{i+1} should satisfy one of the following rules:
  (a) $\ell \in \mathbf{L}_a$ and $(\ell, \alpha, \ell') \in \rightarrow$ and $\kappa_{i+1} = \kappa_i^{-1} \cdot (f, \ell', \alpha(\nu))$.
  (b) $\ell \in \mathbf{L}_b$ and $(\ell, \phi, \ell') \in \rightarrow$ where ϕ is a predicate such that $\nu \models \phi$ and $\kappa_{i+1} = \kappa_i^{-1} \cdot (f, \ell', \nu)$.
  (c) $\ell \in \mathbf{L}_c$, the statement corresponding to ℓ is the function call $v_0 := f'(v_1, v_2, \ldots, v_n)$, the header of the function f' is $f'(v_1', v_2', \ldots, v_n')$, and $\kappa_{i+1} = \kappa_i \cdot (f', \ell_{\texttt{in}}^{f'}, \nu')$ where
  $$\nu'(x) = \begin{cases} \nu(v_i) & x \in \{v_i', \bar{v}_i'\} \\ 0 & \text{otherwise} \end{cases}.$$
  Intuitively, this corresponds to adding the new function to the stack.
  (d) $\ell \in \mathbf{L}_d$ and $(\ell, \star, \ell') \in \rightarrow$ and $\kappa_{i+1} = \kappa_i^{-1} \cdot (f, \ell', \nu)$.
  (e_1) $\ell \in \mathbf{L}_e$ and $|\kappa_i| = 1$ and $|\kappa_{i+1}| = 0$. Informally, this case corresponds to the termination of the program when the $f_{\texttt{main}}$ function returns and the stack becomes empty.
  (e_2) $\ell \in \mathbf{L}_e$, $|\kappa_i| > 1$, $\hat{\xi} = (\hat{f}, \hat{\ell}, \hat{\nu})$ is the stack element before ξ in κ_i, the label $\hat{\ell}$ corresponds to a function call of the form $v_0 := f(v_1, \ldots, v_n)$, $(\hat{\ell}, \perp, \hat{\ell}') \in \rightarrow$ and $\kappa_{i+1} = \kappa_i^{-2} \cdot (\hat{f}, \hat{\ell}', \hat{\nu}[v_0 \leftarrow \nu(\texttt{ret}^f)])$. Informally, this corresponds to returning control from the function f into its parent function $\hat{f}$.

Return Assumption. We assume that every execution of a function ends with a **return** statement. If this is not the case, we can add "**return** 0" to suitable points of the program to obtain an equivalent program that satisfies this condition.

Semi-runs and Paths. A *semi-run* starting at a stack element $\xi = (f, \ell, \nu)$ is a sequence $\varrho = \langle \kappa_i \rangle_{i=0}^{\infty}$ that satisfies all the conditions of a run, except that it starts with $\kappa_0 = \langle \xi \rangle$. A *path* $\pi = \langle \kappa_i \rangle_{i=0}^{n}$ of length n is a finite prefix of a semi-run.

2.3 Invariants

We now present the notion of inductive invariants and extend it to recursive programs using the standard concepts of pre and post-conditions. Then, we formally define the strong and weak variants of the invariant synthesis problem.

Pre-conditions. A *pre-condition* is a function Pre mapping each label $\ell \in \mathbf{L}^f$ of a function f of the program to a conjunctive propositional formula $\mathsf{Pre}(\ell) := \bigwedge_{i=0}^{m} (\mathfrak{e}_i \geq 0)$, where each $\mathfrak{e}_i$ is an arbitrary arithmetic expression over the set $\mathbf{V}^f$ of variables[‡§].

Intuitively, a pre-condition specifies a set of requirements for the runs of the program, i.e. a run that does not satisfy the pre-condition is considered to be invalid (e.g. impossible to happen or leading to a runtime error).

Post-conditions. A *post-condition* is a function Post that maps each program function f of the form $f(v_1, \ldots, v_n)$ to a conjunctive propositional formula $\mathsf{Post}(f) := \bigwedge_{i=0}^{m} (\mathfrak{e}_i > 0)$ over $\{\mathtt{ret}^f, \bar{v}_1, \ldots, \bar{v}_n\}$.

Intuitively, a post-condition characterizes the return value $\mathtt{ret}^f$ of each function f based on the values of parameters passed to f when it was called.

Remark 1 (Strict and Non-strict Inequalities). *Note that in the definitions above the inequalities in post-conditions are strict, whereas pre-conditions contain non-strict inequalities. There is a technical reason behind this choice, having to do with Theorem 3.1. Basically, Putinar's positivstellensatz characterizes* strictly positive *polynomials over a* closed *semi-algebraic set. Therefore, this subtle difference in the definitions of pre and post-conditions is necessary for our completeness result (Lemma 3.7). However, the soundness of our approach does not depend on it.*

Valid Runs and Reachable Stack Elements. A run ρ is *valid* w.r.t. a pre-condition Pre, if for every stack element $\xi = (f, \ell, \nu)$ appearing in one of its configurations, we have $\nu \models \mathsf{Pre}(\ell)$. Valid semi-runs and paths are defined similarly. A stack element is called *reachable* if it appears in a valid run.

Model of Computation. As mentioned in the previous section, we consider programs in which variables can have arbitrary real values. However, some of our results only hold if the variable values are bounded. In such cases we explicitly mention that the result holds on "bounded reals". The formal interpretation of this point is that there exists a constant value $c \in \mathbb{R}^+$ such that for every label $\ell \in \mathbf{L}^f$ and every variable $v \in \mathbf{V}^f$, the pre-condition $\mathsf{Pre}(\ell)$ contains the inequalities $-c \leq v \leq c$. In other words, in the bounded reals model of computation, a variable overflows if its value becomes more than c (resp. underflows if its value becomes less than $-c$), and any run containing an overflow or underflow is considered invalid. As a direct consequence, for every reachable stack element $\xi = (f, \ell, \nu)$, we have $\|\nu\|_2 \leq c\sqrt{|\mathbf{V}^f|}$. Hence, when discussing bounded reals, we assume that every pre-condition contains the inequality $\left\|\mathbf{V}^f\right\|_2^2 \leq c^2 |\mathbf{V}^f|$ as well[¶].

Invariants. Given a program P and a pre-condition Pre, an *invariant* is a function Inv mapping each label $\ell \in \mathbf{L}^f$ of the program to a conjunctive propositional formula $\mathsf{Inv}(\ell) := \bigwedge_{i=0}^{m} (\mathfrak{e}_i > 0)$ over $\mathbf{V}^f$, such that for every reachable stack element (f, ℓ, ν), it holds that $\nu \models \mathsf{Inv}(\ell)$.

[‡]Classically, pre-conditions are only defined for the first labels of functions. In contrast, we allow pre-conditions for every label. Note that this setting is strictly more general, given that one can let $\mathsf{Pre}(\ell) = \mathbf{true}$ for every other kind of label.

[§]By definition of runs, the value of every variable $v \in \mathbf{V}^f \setminus \mathbf{V}^f_*$ is always 0 when the program reaches ℓ^f_{in}. Hence, w.l.o.g. we assume that $\mathsf{Pre}(\ell^f_{\mathrm{in}})$ contains the assertions $v \geq 0$ and $-v \geq 0$. Similarly, we assume that for every parameter v of f, we have the assertions $v - \bar{v} \geq 0$ and $\bar{v} - v \geq 0$ in $\mathsf{Pre}(\ell^f_{\mathrm{in}})$.

[¶]On the LHS, $\mathbf{V}^f$ is considered as a vector and $\left\|\mathbf{V}^f\right\|_2$ is its Euclidean norm. On the RHS, $\mathbf{V}^f$ is a set and $|\mathbf{V}^f|$ is its size. More concretely, if $\mathbf{V}^f = \{v_1, v_2, \ldots, v_n\}$, then the pre-condition contains the inequality $v_1^2 + v_2^2 + \ldots + v_n^2 \leq c^2 \cdot n$.

Positivity Witnesses. Let $\mathfrak{e}$ be an arithmetic expression over program variables and $\phi = \bigwedge_{i=0}^{m}(\mathfrak{e}_i \bowtie_i 0)$ for $\bowtie_i \in \{>, \geq\}$, such that for every valuation ν, we have $\nu \models \phi \Rightarrow \mathfrak{e}(\nu) > 0$. We say that a constant $\epsilon > 0$ is a *positivity witness* for $\mathfrak{e}$ w.r.t. ϕ if for every valuation ν, we have $\nu \models \phi \Rightarrow \mathfrak{e}(\nu) > \epsilon$. In the sequel, we limit our focus to inequalities that have positivity witnesses. Intuitively, this means that we consider invariants of the form $\bigwedge_{j=1}^{m}(\mathfrak{e}_j > 0)$ where the values of $\mathfrak{e}_j$'s in the runs of the program cannot get arbitrarily close to 0[‖].

Inductive Assertion Maps. Given a *non-recursive* program P and a pre-condition Pre, an *inductive assertion map* is a function Ind mapping each label $\ell \in \mathbf{L}^f$ of the program to a conjunctive propositional formula $\mathsf{Ind}(\ell) := \bigwedge_{i=0}^{m}(\mathfrak{e}_i > 0)$ over $\mathbf{V}^f$, such that the following two conditions hold:

- *Initiation.* For every stack element $\xi = (f_{\mathtt{main}}, \ell_{\mathtt{in}}^{f_{\mathtt{main}}}, \nu_0)$, we have $\nu_0 \models \mathsf{Pre}(\ell_{\mathtt{in}}^{f_{\mathtt{main}}}) \Rightarrow \nu_0 \models \mathsf{Ind}(\ell_{\mathtt{in}}^{f_{\mathtt{main}}})$. Intuitively, this means that $\mathsf{Ind}(\ell_{\mathtt{in}}^{f_{\mathtt{main}}})$ should be deducible from the pre-condition $\mathsf{Pre}(\ell_{\mathtt{in}}^{f_{\mathtt{main}}})$.
- *Consecution.* For every valid unit-length path $\pi = \langle (f_{\mathtt{main}}, \ell_0, \nu_0), (f_{\mathtt{main}}, \ell_1, \nu_1)\rangle$ that starts at ℓ_0 and ends at ℓ_1, we have $\nu_0 \models \mathsf{Ind}(\ell_0) \Rightarrow \nu_1 \models \mathsf{Ind}(\ell_1)$. Intuitively, this condition means that the inductive assertion map cannot be falsified by running a valid step of the execution of the program.

It is well-known that every inductive assertion map is also an invariant. Hence, inductive assertion maps are often called *inductive invariants*, too [Colón et al. 2003]. We present a short proof of this fact.

Lemma 2.1. *Given a non-recursive program P and a pre-condition* Pre*, every inductive assertion map* Ind *is an invariant.*

Proof. *Consider an arbitrary valid run $\rho = \langle \kappa_i \rangle_{i=0}^{\infty} = \langle \langle (f_{\mathtt{main}}, \ell_i, \nu_i)\rangle\rangle_{i=0}^{\infty}$ of P. Let $\pi = \langle \kappa_i \rangle_{i=0}^{n}$ be a prefix of ρ, which is a valid path of length n. We prove that $\nu_n \models \mathsf{Ind}(\ell_n)$. Our proof is by induction on n. For the base case of $n = 0$, we have $\ell_0 = \ell_{\mathtt{in}}^{f_{\mathtt{main}}}$. By validity of ρ, we have $\nu_0 \models \mathsf{Pre}(\ell_{\mathtt{in}}^{f_{\mathtt{main}}})$. Hence, by initiation, $\nu_0 \models \mathsf{Ind}(\ell_{\mathtt{in}}^{f_{\mathtt{main}}})$. For the induction step, assuming that $\nu_{n-1} \models \mathsf{Ind}(\ell_{n-1})$, we prove that $\nu_n \models \mathsf{Ind}(\ell_n)$. We apply the consecution property to the unit-length valid path $\langle (f_{\mathtt{main}}, \ell_{n-1}, \nu_{n-1}), (f_{\mathtt{main}}, \ell_n, \nu_n)\rangle$, which leads to $\nu_n \models \mathsf{Ind}(\ell_n)$.*

Hence, for every reachable stack element $(f_{\mathtt{main}}, \ell, \nu)$, we have $\nu \models \mathsf{Ind}(\ell)$, which means Ind *is an invariant.* □

Example 3. Consider the program in Figure 2. Assuming that we have the pre-condition $n \geq 0$ at label 1, it is easy to show that for any $\epsilon > 0$, $\mathsf{Ind}(\ell) := (n + \epsilon > 0 \wedge i + \epsilon > 0 \wedge s + \epsilon > 0)$ for all $\ell \in \{1, \ldots, 9\}$ is an inductive assertion map, i.e. it holds at the beginning of the program and no valid execution step falsifies it. Hence, it is also an invariant. Moreover, using this invariant, one can prove the post-condition $\mathtt{ret}^{\mathsf{sum}} + \epsilon > 0$. ■

We extend inductive invariants to recursive programs using abstract paths:

Abstract Paths. Given a pre-condition Pre and a post-condition Post, an *abstract path* starting at a stack element $\xi = (f, \ell_0, \nu_0)$ is a sequence $\varpi = \langle \kappa_i = \langle (f, \ell_i, \nu_i)\rangle\rangle_{i=0}^{n}$ such that for all $i < n$, κ_{i+1} satisfies either one of the conditions (a), (b) and (d) as in the definition of runs or the following modified (c) condition:

(c′) $\ell_i \in \mathbf{L}_c$, i.e. the statement corresponding to ℓ_i is a function call $v_0 := f'(v_1, \ldots, v_n)$ where f' is a function with the header $f'(v_1', \ldots, v_n')$ and $\nu_i \models \mathsf{Pre}(\ell_{\mathtt{in}}^{f'})[v_i' \leftarrow v_i, \bar{v}_i' \leftarrow v_i]$. Moreover,

[‖]Note that this is a very minor restriction, in the sense that if $\mathfrak{e} > 0$ is an invariant, then so is $\mathfrak{e} + \epsilon > 0$. Our algorithm is unable to find invariants $\mathfrak{e} > 0$ where $\mathfrak{e}$ can get arbitrarily close to 0 over all valid runs of the program. However, in such cases, our approach can synthesize $\mathfrak{e} + \epsilon > 0$ for *any* positive ϵ.

$(\ell_i, \bot, \ell_{i+1}) \in \longrightarrow$, the valuation ν_{i+1} agrees with ν_i over every variable, except possibly v_0, and $\nu_{i+1} \models \mathsf{Post}(f)[\bar{v}_i' \leftarrow v_i, \mathtt{ret}^f \leftarrow v_0]$. The latter is the result of replacing each occurrence of $\bar{v}_i'$ with its respective v_i and $\mathtt{ret}^f$ with v_0 in $\mathsf{Post}(f)$.

Informally, an abstract path is a path in which all function calls are abstracted, i.e. removed and replaced with a simple transition that respects the pre and post-condition of the called function. An abstract path always remains in the same function and hence each configuration in an abstract path consists of only one stack element. A valid abstract path is defined similarly to a valid path.

Example 4. Assume the pre and post-conditions of Example 3 and consider a program P whose main function $f_{\mathtt{main}}$ calls the function sum of Figure 2. Suppose sum is called at label ℓ, i.e. the statement at ℓ is $y := \mathsf{sum}(x)$, and $(\ell, \bot, \ell') \in \longrightarrow$. In a normal run of P, when the program reaches ℓ, rule (c) is applied (cf. definition of runs) and control moves to sum. In contrast, in an abstract path starting at ℓ, rule (c') is applied and control directly moves to ℓ', provided that no variable other than y gets its value changed and that the pre-condition and post-condition are satisfied. For example, the following sequences are abstract paths:

$$\langle\langle(f_{\mathtt{main}}, \ell, x = 3, y = 0)\rangle, \langle(f_{\mathtt{main}}, \ell', x = 3, y = \epsilon)\rangle\rangle$$
$$\langle\langle(f_{\mathtt{main}}, \ell, x = 3, y = 1)\rangle, \langle(f_{\mathtt{main}}, \ell', x = 3, y = 99.9)\rangle\rangle$$

Note that the latter configuration cannot happen in any valid run, but it does not violate the conditions of an abstract path. This is because the post-condition in this example is very weak and hence abstract paths grossly overestimate valid paths. As we will see, our algorithms synthesize stronger post-conditions as part of the invariant generation process. Finally, in contrast to the sequences above, the following are *not* abstract paths:

$\langle\langle(f_{\mathtt{main}}, \ell, x = -1, y = 1)\rangle, \langle(f_{\mathtt{main}}, \ell', x = -1, y = 10)\rangle\rangle$	Violates $\mathsf{Pre}(\ell^{\mathsf{sum}}_{\mathsf{in}})[n \leftarrow x, \bar{n} \leftarrow x]$
$\langle\langle(f_{\mathtt{main}}, \ell, x = 1, y = 1)\rangle, \langle(f_{\mathtt{main}}, \ell', x = 1, y = -1)\rangle\rangle$	Violates $\mathsf{Post}(\mathsf{sum})[\bar{n} \leftarrow x, \mathtt{ret}^{\mathsf{sum}} \leftarrow y]$
$\langle\langle(f_{\mathtt{main}}, \ell, x = 3, y = 3)\rangle, \langle(f_{\mathtt{main}}, \ell', x = 2, y = 4)\rangle\rangle$	Changes the value of x

■

Recursive Inductive Invariants. Given a recursive program P and a pre-condition Pre, a *recursive inductive invariant* is a pair (Post, Ind) where Post is a post-condition and Ind is a function that maps every label $\ell \in \mathbf{L}^f$ of the program to a conjunctive propositional formula $\mathsf{Ind}(\ell) := \bigwedge_{i=0}^{m} (\mathrm{e}_i > 0)$, such that the following requirements are satisfied:

- *Initiation.* For every stack element $\xi = (f, \ell^f_{\mathsf{in}}, v_0)$ at the starting label of any function f, we have $v_0 \models \mathsf{Pre}(\ell^f_{\mathsf{in}}) \Rightarrow v_0 \models \mathsf{Ind}(\ell^f_{\mathsf{in}})$.
- *Consecution.* For every valid unit-length *abstract* path $\pi = \langle(f, \ell_0, v_0), (f, \ell_1, v_1)\rangle$ that starts at $\ell_0 \in \mathbf{L}^f$ and ends at $\ell_1 \in \mathbf{L}^f$, we have $v_0 \models \mathsf{Ind}(\ell_0) \Rightarrow v_1 \models \mathsf{Ind}(\ell_1)$.
- *Post-condition Consecution.* For every valid unit-length *abstract* path $\pi = \langle(f, \ell_0, v_0), (f, \ell^f_{\mathsf{out}}, v_1)\rangle$ that starts at $\ell_0 \in \mathbf{L}^f$ and ends at the endpoint label ℓ^f_{out}, we have $v_0 \models \mathsf{Ind}(\ell_0) \Rightarrow v_1 \models \mathsf{Post}(f)$.

Following an argument similar to the case of inductive invariants, we prove that if (Post, Ind) is a recursive inductive invariant, then Ind is an invariant.

Lemma 2.2. *Given a recursive program P and a pre-condition* Pre*, if* (Post, Ind) *is a recursive inductive invariant, then the function* Ind *is an invariant.*

Proof. *Consider an arbitrary valid run $\rho = \langle\kappa_i\rangle_{i=0}^{\infty}$ of P. Let $\pi = \langle\kappa_i\rangle_{i=0}^{n}$ be a prefix of ρ, which is a valid path of length n and $\xi = (f, \ell, v)$ the last stack element of κ_n. We prove that $v \models \mathsf{Ind}(\ell)$. Our proof is by induction on n.*

For the base case of $n = 0$, we have $\ell = \ell_{\mathsf{in}}^{f_{\mathsf{main}}}$. By validity of ρ, we have $v \models \mathsf{Pre}(\ell_{\mathsf{in}}^{f_{\mathsf{main}}})$. Hence, by initiation, $v \models \mathsf{Ind}(\ell_{\mathsf{in}}^{f_{\mathsf{main}}})$. For the inductive step, we let $\xi' = (f', \ell', v')$ be the last stack element in κ_{n-1}. We prove that $v \models \mathsf{Ind}(\ell)$. We consider the following cases:

- *If $f' = f$, then $\langle (f', \ell', v'), (f, \ell, v) \rangle$ is a valid abstract path of length 1. Hence, by consecution, we have $v \models \mathsf{Ind}(\ell)$.*
- *If f' is the parent function of f, i.e. $\ell = \ell_{\mathsf{in}}^{f}$ and ℓ' is a function-call statement calling f, then by validity of ρ, we have $v \models \mathsf{Pre}(\ell)$ and by initiation, we infer $v \models \mathsf{Ind}(\ell)$.*
- *If f is the parent function of f', i.e. $\ell' = \ell_{\mathsf{out}}^{f'}$, then let $\hat{\xi} = (f, \hat{\ell}, \hat{v})$ be the last visited stack element in f before f' was called. It is easy to verify that $\hat{\ell}$ is a function-call statement calling f' and $(\hat{\ell}, \bot, \ell) \in \longrightarrow$. By post-condition consecution, $v' \models \mathsf{Post}(f')$, hence $\langle (f, \hat{\ell}, \hat{v}), (f, \ell, v) \rangle$ is a valid abstract path of length 1. By the induction hypothesis, we have $\hat{v} \models \mathsf{Ind}(\hat{\ell})$, hence, by consecution, we deduce $v \models \mathsf{Ind}(\ell)$.*

Hence, for every reachable stack element $\xi = (f, \ell, v)$, we have $v \models \mathsf{Ind}(\ell)$ which means Ind *is an invariant.* □

We define our synthesis problem in terms of (recursive) inductive invariants, because the classical method for finding or verifying invariants is to consider inductive invariants that strengthen them [Colón et al. 2003; Manna and Pnueli 1995].

The Invariant Synthesis Problem. Given a program P, together with a pre-condition Pre, the invariant synthesis problem asks for (recursive) inductive invariants of a given form and size (e.g. linear or polynomial of a given degree). The problem can be divided into two variants:

- The *Strong Invariant Synthesis Problem* asks for a characterization or a representative set of all possible invariants.
- The *Weak Invariant Synthesis Problem* provides an objective function over the invariants (e.g. a function over the coefficients of polynomial invariants) and asks for an invariant that maximizes the objective function.

Polynomial Invariants. In the sequel, we consider the synthesis problems for *polynomial* invariants and pre and post-conditions, i.e. we assume that all arithmetic expressions used in the atomic assertions are polynomials.

3 POLYNOMIAL INVARIANTS FOR NON-RECURSIVE PROGRAMS

In this section, we consider the problem of synthesizing polynomial inductive invariants of a given degree for an input *non-recursive* program P and pre-condition Pre. We first provide a sound and semi-complete reduction from inductive invariants to solutions of a system of quadratic equalities. Our main mathematical tool is a theorem in real semi-algebraic geometry called Putinar's positivstellensatz [Putinar 1993].

We show that the Strong Invariant Synthesis problem can be solved in subexponential time. On the other hand, we show that the Weak Invariant Synthesis problem can be reduced to Quadratically-Constrained Linear Programming (QCLP). We begin with presenting the required mathematical tools and then proceed to strong and weak invariant generation.

3.1 Mathematical Tools and Lemmas

We start with a positivstellensatz due to Putinar. This theorem is our main tool in reducing invariant generation to quadratic systems.

Theorem 3.1 (Putinar's Positivstellensatz [Putinar 1993]). *Let X be a finite set of variables and $g, g_1, \ldots, g_m \in \mathbb{R}[X]$ be polynomials over X with real coefficients. We define $\Pi := \{x \in$*

$\mathbb{R}^X \mid \forall i\ g_i(x) \geq 0\}$ *as the set of all points in which every* g_i *is non-negative. If (i) there exists some* g_k *such that the set* $\{x \in \mathbb{R}^X \mid g_k(x) \geq 0\}$ *is compact, and (ii)* $g(x) > 0$ *for all* $x \in \Pi$*, then*

$$g = h_0 + \sum_{i=1}^{m} h_i \cdot g_i \tag{1}$$

where each polynomial h_i *is the sum of squares of some polynomials in* $\mathbb{R}[X]$*, i.e.* $h_i = \sum_{j=0}^{n} f_j^2$ *for some* f_j*'s in* $\mathbb{R}[X]$*.*

Corollary 3.2. *Let* $X, g, g_1, \ldots, g_m$ *and* Π *be as in the theorem above. Then* $g(x) > 0$ *for all* $x \in \Pi$ if and only if*:*

$$g = \epsilon + h_0 + \sum_{i=1}^{m} h_i \cdot g_i \tag{2}$$

where $\epsilon > 0$ *is a real number and each polynomial* h_i *is the sum of squares of some polynomials in* $\mathbb{R}[X]$*.*

Proof. *It is obvious that if (2) holds, then* $g(x) > 0$ *for all* $x \in \Pi$*. We prove the other side. Let* $g(x) > 0$ *for all* $x \in \Pi$*. Given that* Π *is compact and* g *continuous,* $g(\Pi)$ *must also be compact and hence closed. Therefore,* $\delta := \inf_{x \in \Pi} g(x) > 0$*. Let* $\epsilon = \delta/2$*, then* $g(x) - \epsilon > 0$ *for all* $x \in \Pi$*. Applying Putinar's Positivstellensatz, i.e. equation (1), to* $g - \epsilon$ *leads to the desired result.* □

Hence, Putinar's positivstellensatz provides a characterization of all polynomials g that are positive over the closed set Π. Intuitively, given a set of atomic non-negativity assumptions $g_i(x) \geq 0$, in order to find all polynomials g that are positive under these assumptions, we only need to look into polynomials of form (2). Moreover, the real number ϵ in (2) serves as a positivity witness for g.

Remark 2. *As an alternative to Putinar's positivstellensatz, one can use the following theorem, due to Schweighofer, that provides necessary and sufficient conditions for a polynomial* g *to be positive over a semi-algebraic set described by linear and polynomial inequalities. Hence, it can replace Corollary 3.2 and be directly applied in our algorithm.*

Theorem 3.3 (Schweighofer [Maréchal et al. 2016; Schweighofer 2002]). *Let* $P = \{C_1 \geq 0, \ldots, C_p \geq 0\}$ *be a polytope where each* C_i *is an affine polynomial over* $x = (x_1, \ldots, x_n)$*. Let* $g_{p+1}, \ldots, g_q$ *and* g *be polynomials. Then* $g(x) > 0$ *on* $P \cap \{g_{p+1} \geq 0, \ldots, g_q \geq 0\}$ if and only if

$$g = \lambda_0 + \sum_{I \in \mathbb{N}^q} \lambda_I \cdot S^I \qquad \lambda_0 \in \mathbb{R}^{*+}, \lambda_I \in \mathbb{R}^+,$$

where $S^{(i_1,\ldots,i_q)} = C_1^{i_1} \cdot \ldots \cdot C_p^{i_p} \cdot g_{p+1}^{i_{p+1}} \cdot \ldots \cdot g_q^{i_q}$*.*

In our algorithm in the next section, we have to reduce the problem of checking whether a polynomial $h \in \mathbb{R}[V]$ is a sum-of-squares to solving a quadratic system. We now present this reduction in detail. Our reduction is based on the following two well-known theorems:

Theorem 3.4 (See [Horn and Johnson 1990], Corollary 7.2.9). *A polynomial* $h \in \mathbb{R}[V]$ *of even degree* d *is a sum-of-squares* if and only if *there exists a* k*-dimensional symmetric positive semi-definite matrix* Q *such that* $h = y^T Q y$*, where* k *is the number of monomials of degree no greater than* $d/2$ *and* y *is a column vector consisting of every such monomial.*

Theorem 3.5 ([Golub and Van Loan 1996; Higham 2009]). *A symmetric square matrix* Q *is positive semi-definite* if and only if *it has a Cholesky decomposition of the form* $Q = LL^T$ *where* L *is a lower-triangular matrix with non-negative diagonal entries.*

Given the two theorems above, our reduction uses the following procedure for generating quadratic equations that are equivalent to the assertion that h is a sum-of-squares:

The Reduction. The algorithm generates the set $\mathbf{M}_{\lfloor d/2 \rfloor}$ of monomials of degree at most $\lfloor d/2 \rfloor$ over V. It then orders these monomials arbitrarily into a vector y and symbolically computes the equality

$$h = y^T L L^T y \tag{3}$$

where L is a lower-triangular matrix whose every non-zero entry is a new variable in the system. We call these variables l-variables. For every $l_{i,i}$, i.e. every l-variable that appears on the diagonal of L, the algorithm adds the constraint $l_{i,i} \geq 0$ to the quadratic system. Then, it translates Equation (3) into quadratic equations over the coefficients of h[**] and l-variables by equating the coefficients of corresponding terms on the two sides of (3). The resulting system encodes the property that h is a sum-of-squares.

Example 5. Let $V = \{a, b\}$ be the set of variables and $h \in \mathbb{R}[V]$ a quadratic polynomial, i.e. $h(a, b) = t_1 + t_2 \cdot a + t_3 \cdot b + t_4 \cdot a^2 + t_5 \cdot a \cdot b + t_6 \cdot b^2$. We aim to encode the property that h is a sum-of-squares as a system of quadratic equalities and inequalities. To do so, we first generate all monomials of degree at most $\lfloor d/2 \rfloor = 1$, which are 1, a and b. Hence, we let $y = \begin{bmatrix} 1 & a & b \end{bmatrix}^T$. We then generate a lower-triangular matrix L whose every non-zero entry is a new variable:

$$L = \begin{bmatrix} l_1 & 0 & 0 \\ l_2 & l_3 & 0 \\ l_4 & l_5 & l_6 \end{bmatrix}.$$

We also add the inequalities $l_1 \geq 0, l_3 \geq 0$ and $l_6 \geq 0$ to our system. Now, we write the equation $h = y^T L L^T y$ and compute it symbolically:

$$h = \begin{bmatrix} 1 & a & b \end{bmatrix} \begin{bmatrix} l_1 & 0 & 0 \\ l_2 & l_3 & 0 \\ l_4 & l_5 & l_6 \end{bmatrix} \begin{bmatrix} l_1 & l_2 & l_4 \\ 0 & l_3 & l_5 \\ 0 & 0 & l_6 \end{bmatrix} \begin{bmatrix} 1 \\ a \\ b \end{bmatrix},$$

which leads to:

$$\begin{aligned} & t_1 + t_2 \cdot a + t_3 \cdot b + t_4 \cdot a^2 + t_5 \cdot a \cdot b + t_6 \cdot b^2 \\ = \; & l_1^2 + 2 \cdot l_1 \cdot l_2 \cdot a + 2 \cdot l_1 \cdot l_4 \cdot b + (l_2^2 + l_3^2) \cdot a^2 + (2 \cdot l_2 \cdot l_4 + 2 \cdot l_3 \cdot l_5) \cdot a \cdot b + (l_4^2 + l_5^2 + l_6^2) \cdot b^2. \end{aligned}$$

Note that both sides of the equation above are polynomials over $\{a, b\}$, hence they are equal iff their corresponding coefficients are equal. So, we get the following quadratic equalities over the t-variables and l-variables: $t_1 = l_1^2, t_2 = 2 \cdot l_1 \cdot l_2, \ldots, t_6 = l_4^2 + l_5^2 + l_6^2$. This concludes the construction of our quadratic system. ■

We now have all the necessary ingredients to present our algorithms for the Invariant Synthesis problems.

3.2 Overview of the Approach

In this section, we provide an overview of our algorithms. The next sections go through all the details. Our algorithms for Strong and Weak Invariant Synthesis are very similar. They each consist of four main steps and differ only in the last step. The steps are as follows:

Step 1) First, the algorithm creates a template for the inductive invariant at each label. More specifically, it creates polynomial templates of the desired size and degree, but with *unknown coefficients*. The goal is to synthesize values for these unknown coefficients so that the template becomes a valid inductive invariant.

[**] These coefficients are called t-variables in our algorithm.

Step 2) The algorithm generates a set of constraints that should be satisfied by the template so as to ensure that it becomes an inductive invariant. These constraints encode the initiation and consecution requirements as in the definition of inductive invariants. Moreover, they have a very specific form: each constraint consists of polynomials $g_1, \ldots, g_m$ and g and encodes the requirement that for every valuation ν, if we have $g_1(\nu) \geq 0, g_2(\nu) \geq 0, \ldots, g_m(\nu) \geq 0$, then we must also have $g(\nu) > 0$.

Step 3) Exploiting the structure of the constraints generated in the previous step, the algorithm applies Putinar's positivstellensatz to translate the constraints into quadratic equalities over the unknown coefficients.

Step 4) The algorithm uses an external solver for handling the system of quadratic equalities generated in the previous step. In case of Strong Invariant Synthesis, the external solver would use the algorithm of [Grigor'ev and Vorobjov 1988] to provide a representative set of all invariants. In contrast, for Weak Invariant Synthesis, the external solver is an optimization suite for quadratic programming (QCLP).

3.3 Strong Invariant Synthesis and Theoretical Guarantees

We now provide a formal description of the input to our algorithm for Strong Invariant Synthesis and then present details of every step.

The StrongInvSynth Algorithm. We present an algorithm StrongInvSynth that gets the following items as its input:

- A non-recursive non-deterministic program P, with polynomial assignments and guards, which is generated by the grammar in Figure 1,
- A polynomial pre-condition Pre,
- Three positive integers d, n and Υ, where d is the degree of polynomials in the desired inductive invariants, n is the desired size of the invariant generated at each label, i.e. number of atomic assertions, and Υ is a technical parameter, used to ensure semi-completeness, which will be discussed later;

and produces a representative set of all inductive invariants Ind of the program P, such that for all $\ell \in \mathbf{L}$, the set $\mathsf{Ind}(\ell)$ consists of n atomic assertions of degree at most d. Our algorithm consists of the following four steps:

Step 1) Setting up templates. Let $\mathbf{V}^f = \{v_1, v_2, \ldots, v_t\}$ and define $\mathbf{M}_d^f = \{m_1, m_2, \ldots, m_r\}$ as the set of all monomials of degree at most d over $\mathbf{V}^f$, i.e. $\mathbf{M}_d^f := \{\prod_{i=1}^t v_i^{\alpha_i} \mid \forall i\ \alpha_i \in \mathbb{N}_0 \ \wedge\ \sum_{i=1}^t \alpha_i \leq d\}$. At each label $\ell \in \mathbf{L}^f$ of the program P, the algorithm generates a template $\eta(\ell) := \bigwedge_{i=1}^n \varphi_{\ell,i}$ where each $\varphi_{\ell,i}$ is of the form $\varphi_{\ell,i} := \left(\sum_{j=1}^r s_{\ell,i,j} \cdot m_j > 0\right)$. Here, the $s_{\ell,i,j}$'s are new unknown variables. For brevity, we call them s-variables. Intuitively, our goal is to synthesize values for s-variables such that η becomes an inductive invariant.

Example 6. Consider the summation program in Figure 2. We have $\mathbf{V}^{\mathsf{sum}} = \{n, \bar{n}, i, s, \mathtt{ret}^{\mathsf{sum}}\}$. For brevity we define $r := \mathtt{ret}^{\mathsf{sum}}$. Suppose that we want to synthesize a single quadratic assertion as the invariant at each label. In Step 1, the algorithm creates the following template for each label $\ell \in \{1, 2, \ldots, 9\}$:

$$\begin{aligned}\eta(\ell) :=\ & s_{\ell,1,1} + s_{\ell,1,2} \cdot n + s_{\ell,1,3} \cdot \bar{n} + s_{\ell,1,4} \cdot i + s_{\ell,1,5} \cdot s + s_{\ell,1,6} \cdot r + \\ & s_{\ell,1,7} \cdot n^2 + s_{\ell,1,8} \cdot n \cdot \bar{n} + s_{\ell,1,9} \cdot n \cdot i + s_{\ell,1,10} \cdot n \cdot s + s_{\ell,1,11} \cdot n \cdot r + \\ & s_{\ell,1,12} \cdot \bar{n}^2 + s_{\ell,1,13} \cdot \bar{n} \cdot i + s_{\ell,1,14} \cdot \bar{n} \cdot s + s_{\ell,1,15} \cdot \bar{n} \cdot r + s_{\ell,1,16} \cdot i^2 + \\ & s_{\ell,1,17} \cdot i \cdot s + s_{\ell,1,18} \cdot i \cdot r + s_{\ell,1,19} \cdot s^2 + s_{\ell,1,20} \cdot s \cdot r + s_{\ell,1,21} \cdot r^2 > 0.\end{aligned}$$

■

Step 2) Setting up constraint pairs. For each transition $e = (\ell, \alpha, \ell')$ of the CFG of P, the algorithm constructs a set Λ_e of *constraint pairs* of the form $\lambda = (\Gamma, g)$ where $\Gamma = \bigwedge_{i=1}^{m} (g_i \geq 0)$ and $g, g_1, \ldots, g_m$ are polynomials with unknown coefficients (based on the s-variables). Intuitively, a condition pair (Γ, g) encodes the following condition:

$$\forall \nu \in \mathbb{R}^f \ \nu \models \Gamma \Rightarrow g(\nu) > 0 \ \equiv \ \forall \nu \in \mathbb{R}^{\mathbf{V}^f} \ (\forall g_i \in \Gamma \ g_i(\nu) \geq 0) \Rightarrow g(\nu) > 0.$$

The construction is as follows (note that all computations are done symbolically):

- If $\ell \in \mathbf{L}_a$, for every polynomial g for which $g > 0$ appears in $\eta(\ell')$, the algorithm adds the condition pair $(\mathsf{Pre}(\ell) \wedge \eta(\ell) \wedge (\mathsf{Pre}(\ell') \circ \alpha), g \circ \alpha)$ to Λ_e. Note that in this case α is an update function that assigns a polynomial to every variable and hence the constraint pair can be computed symbolically.
- If $\ell \in \mathbf{L}_b$, then α is a propositional predicate. The algorithm first writes α in disjunctive normal form as $\alpha = \alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_a$. Each α_i is a conjunction of atomic assertions. For every α_i and every g such that $g > 0$ appears in $\eta(\ell')$, the algorithm adds the condition pair $(\mathsf{Pre}(\ell) \wedge \eta(\ell) \wedge \mathsf{Pre}(\ell') \wedge \alpha_i, g)$ to Λ_e.
- If $\ell \in \mathbf{L}_d$, for every g for which $g > 0$ appears in $\eta(\ell')$, the algorithm adds the condition pair $(\mathsf{Pre}(\ell) \wedge \eta(\ell) \wedge \mathsf{Pre}(\ell'), g)$ to Λ_e.

Finally, the algorithm constructs the following set Λ_{in}:

- For every polynomial g for which $g > 0$ appears in $\eta(\ell^f_{\text{in}})$, the algorithm constructs the constraint pair $(\mathsf{Pre}(\ell^f_{\text{in}}), g)$ and adds it to Λ_{in}.

Example 7. In the summation program of Figure 2, suppose that $\mathsf{Pre}(1) := (n \geq 0) \wedge (i \geq 0) \wedge (-i \geq 0) \wedge (s \geq 0) \wedge (-s \geq 0) \wedge (\mathtt{ret}^{\mathtt{sum}} \geq 0) \wedge (-\mathtt{ret}^{\mathtt{sum}} \geq 0) \wedge (n - \bar{n} \geq 0) \wedge (\bar{n} - n \geq 0)$ and $\mathsf{Pre}(\ell) := (1 \geq 0) \equiv \textbf{true}$ for every $\ell \neq 1$. Note that $(n \geq 0)$ is the only non-trivial assertion and all the other assertions are true by definition, given that 1 is the first statement in sum. We provide some examples of constraint pairs generated in Step 2 of the algorithm:

- $1 \in \mathbf{L}_a$ and $e_1 = (1, [i \leftarrow 1], 2) \in \longrightarrow$ (see the CFG in Figure 3). Hence, we have the following constraint pair:

  $$(\mathsf{Pre}(1) \wedge \eta(1) \wedge \mathsf{Pre}(2)[i \leftarrow 1], \eta(2)[i \leftarrow 1])$$

  which is symbolically computed as:

  $$\begin{pmatrix} (n \geq 0) \wedge (i \geq 0) \wedge (-i \geq 0) \wedge \\ (s \geq 0) \wedge (-s \geq 0) \wedge (r \geq 0) \wedge \\ (-r \geq 0) \wedge (n - \bar{n} \geq 0) \wedge (\bar{n} - n \geq 0) \wedge \\ (s_{1,1,1} + s_{1,1,2} \cdot n + \ldots + s_{1,1,21} \cdot r^2 \geq 0) \end{pmatrix}, \begin{pmatrix} (s_{2,1,1} + s_{2,1,2} \cdot n + \ldots + s_{2,1,4} + \ldots + \\ s_{2,1,9} \cdot n + \ldots + s_{2,1,13} \cdot \bar{n} + s_{2,1,16} + \\ s_{2,1,17} \cdot s + s_{2,1,18} \cdot r + \ldots + \\ s_{2,1,21} \cdot r^2) \end{pmatrix}$$

  and added to Λ_{e_1}. Note that $\mathsf{Pre}(2)$ is **true** and can be ignored.
- $3 \in \mathbf{L}_b$ and $e_2 = (3, (n - i \geq 0), 4) \in \longrightarrow$, so the constraint pair $(\mathsf{Pre}(3) \wedge \eta(3) \wedge \mathsf{Pre}(4) \wedge (n - i \geq 0), \eta(4)) \equiv (\eta(3) \wedge (n - i \geq 0), \eta(4))$ is symbolically computed (as above) and added to Λ_{e_2}.
- $1 = \ell^{\mathsf{sum}}_{\text{in}}$, so the constraint pair $(\mathsf{Pre}(1), \eta(1))$ is symbolically computed and added to Λ_{in}.

■

Step 3) Translating constraint pairs to quadratic equalities. Let $\Lambda := \bigcup_{e \in \longrightarrow} \Lambda_e \cup \Lambda_{\text{in}}$ be the set of all constraint pairs constructed in the previous step. For each $\lambda = \left(\bigwedge_{i=1}^{m} (g_i \geq 0), g\right) \in \Lambda$, the algorithm takes the following actions:

(i) Let $V = \{v_1, \ldots, v_{t'}\}$ be the set of all program variables that appear in g or the g_i's. The algorithm computes the set $\mathbf{M}_\Upsilon = \{m'_1, m'_2, \ldots, m'_{r'}\}$ of all monomials of degree at most Υ over V. Note that Υ is a technical parameter that was supplied as part of the input.

(ii) It symbolically computes an equation of the form (2), i.e.

$$g = \epsilon + h_0 + \sum_{i=1}^{m} h_i \cdot g_i \qquad (\dagger)$$

where ϵ is a new unknown and positive real variable and each polynomial h_i is of the form $\sum_{j=1}^{r} t_{i,j} \cdot m'_j$. Here, the $t_{i,j}$'s are also new unknown variables. Intuitively, we aim to synthesize values for both t-variables and s-variables in order to ensure the polynomial equality (†). Note that both sides of (†) are polynomials in $\mathbb{R}[V]$ whose coefficients are quadratic expressions over the newly-introduced s-, t- and ϵ-variables.

(iii) The algorithm equates the coefficients of corresponding monomials in the left and right hand sides of (†), hence obtaining a set of quadratic equalities over the new variables.

(iv) The algorithm computes a set of quadratic equalities which are equivalent to the assertion that the h_i's can be written as sums of squares. See Section 3.1 for more details.

The algorithm conjunctively compiles all the generated quadratic equalities into a single system.

Remark 3. *Based on above, the technical parameter Υ is the maximum degree of the sum-of-squares polynomials h_i in (†). More specifically, in Step 3, we are applying a special case of Putinar's positivstellensatz, in which the sum-of-square polynomials can have a degree of at most Υ.*

Example 8. Consider the first constraint pair generated in Example 7. The algorithm writes (†), i.e. $g = \epsilon + h_0 + \sum_{i=1}^{10} h_i \cdot g_i$ where $g = s_{2,1,1} + \ldots + s_{2,1,21} \cdot r^2$ (the polynomial in the second component of the constraint pair), $g_1 = n$, $g_2 = i$, $g_3 = -i$, …, $g_{10} = s_{1,1,1} + \ldots + s_{1,1,21}$ (the polynomials in the first component of the constraint pair) and each h_i is a newly generated polynomial containing all possible monomials of degree at most Υ, e.g. if $\Upsilon = 2$, we have $h_i = t_{i,1} + t_{i,2} \cdot n + \ldots + t_{i,21} \cdot r^2$, where each $t_{i,j}$ is a new unknown variable. It then equates the coefficients of corresponding monomials on the two sides of (†). For example, consider the monomial r^2. Its coefficient in the LHS of (†) is $s_{2,1,21}$. In the RHS of (†), there are a variety of ways to obtain r^2, hence its coefficient is the sum of the following:

- $t_{0,21}$, i.e. the coefficient of r^2 in h_0,
- $t_{6,6}$, i.e. the coefficient of r^2 in $h_6 \cdot g_6 = h_6 \cdot r$,
- $-t_{7,6}$, i.e. the coefficient of r^2 in $h_7 \cdot g_7 = h_7 \cdot (-r)$,
- $t_{10,1} \cdot s_{1,1,21} + t_{10,6} \cdot s_{1,1,6} + t_{10,21} \cdot s_{1,1,1}$, i.e. the coefficient of r^2 in $h_{10} \cdot g_{10}$.

Hence, the algorithm generates the quadratic equality $s_{2,1,21} = t_{0,21} + t_{6,6} - t_{7,6} + t_{10,1} \cdot s_{1,1,21} + t_{10,6} \cdot s_{1,1,6} + t_{10,21} \cdot s_{1,1,1}$ over the $s-$ and $t-$variables. The algorithm computes similar equalities for every other monomial. ■

Step 4) Finding representative solutions. The previous step has created a system of quadratic equalities over s-variables and other new variables. In this step, the algorithm finds a representative set Σ of solutions to this system by calling an external solver. Then, for each solution $\sigma \in \Sigma$, it plugs the values synthesized for the s-variables into the template η to obtain an inductive invariant $\eta_\sigma := \eta[s_{\ell,i,j} \leftarrow \sigma(s_{\ell,i,j})]$. The algorithm outputs $I = \{\eta_\sigma \mid \sigma \in \Sigma\}$.

Remark 4 (Representative Solutions). *In real algebraic geometry, a standard notion for a representative set of solutions to a polynomial system of equalities is to include one solution from each connected component of the set of solutions [Basu et al. 2007]. The classical algorithm for this problem is called* cylindrical algebraic decomposition *and has a doubly-exponential runtime [Basu et al. 2007; Sturmfels 2002]. However, if the coefficients are limited to rational numbers instead of real numbers, then a subexponentail algorithm is provided in [Grigor'ev and Vorobjov 1988][††]. Hence, Step 4 of* StrongInvSynth *has subexponential runtime in theory.*

[††]No tight runtime analysis is available for this algorithm, but [Grigor'ev and Vorobjov 1988] proves that its runtime is subexponential.

Lemma 3.6 (Soundness). *Every output of* StrongInvSynth *is an inductive invariant. More generally, for every solution $\sigma \in \Sigma$ obtained in Step 4, the function $\eta_\sigma := \eta[s_{\ell,i,j} \leftarrow \sigma(s_{\ell,i,j})]$ is an inductive invariant.*

Proof. *The valuation σ satisfies the system of quadratic equalities obtained in Step 3. Hence, for every constraint pair $(\Gamma, g) \in \Lambda$, $g[s_{\ell,i,j} \leftarrow \sigma(s_{\ell,i,j})]$ can be written in the form (†). Hence, we have $\sigma \models (\Gamma, g)$. By definition of Step 2, this is equivalent to η_σ having the initiation and consecution properties and hence being an inductive invariant.* □

We now prove our completeness result. Our approach is semi-complete for bounded reals in the sense of [Chatterjee et al. 2016]. Concretely, this means that if we assume the bounded reals model of computation (see Section 2.3), then any valid inductive invariant can be found by our approach so long as the technical parameter Υ is large enough. Recall that Υ is an upperbound on the degree of the sum-of-square polynomials (see Remark 3).

Lemma 3.7 (Semi-completeness with Compactness). *If the pre-condition* Pre *satisfies the compactness condition of Theorem 3.1 (Putinar's Positivstellensatz), i.e. if in every label ℓ,* Pre(ℓ) *contains an atomic proposition of the form $g \geq 0$ such that the set $\{\nu \in \mathbb{R}^{\mathbf{V}^f} \mid g(\nu) \geq 0\}$ is compact, then for every inductive invariant* Ind *that has the form of the template η, there exists a natural number Υ_{Ind}, such that for every technical parameter $\Upsilon \geq \Upsilon_{\mathsf{Ind}}$, the invariant* Ind *corresponds to a solution of the system of quadratic equalities obtained in Step 3 of* StrongInvSynth.

Proof. *Let* Ind *be an inductive invariant in the form of the template η. We denote the value of $s_{\ell,i,j}$ in* Ind *by $\sigma(s_{\ell,i,j})$. Given that* Ind *satisfies initiation and consecution, the valuation σ satisfies every constraint pair (Γ, g) generated in Step 2. By assumption, each such Γ contains an assertion $g_i \geq 0$ such that $\{x \in \mathbb{R}^{\mathbf{V}^f} \mid g_i(x) \geq 0\}$ is compact. Hence, by Corollary 3.2, g can be written in the form (†)[‡‡]. So, for large enough Υ, there exists a solution to the system of quadratic equalities that maps each $s_{\ell,i,j}$ to $\sigma(s_{\ell,i,j})$.* □

Remark 5 (Bounded Reals, Compactness and Real-world programs). *Note that in the bounded reals model of computation, every pre-condition enforces that the value of every variable is between $-c$ and c and also contains the polynomial inequality $\left\|\mathbf{V}^f\right\|_2^2 \leq c^2 \cdot |\mathbf{V}^f|$ (see Section 2.3). The set of valuations that satisfy the latter polynomial are points in $\mathbb{R}^f$ whose distance from the origin is at most a fixed amount $c\sqrt{|\mathbf{V}^f|}$. Hence, this set is closed and bounded and therefore compact, and satisfies the requirement of Putinar's positivstellensatz. So, our approach is semi-complete for bounded reals.*

It is worth mentioning that almost all real-world programs have bounded variables. For example, programs that use floating-point variables can at most store a finite number of different values in each variable, hence their variables are always bounded[§§]. Also, note that while the completeness result is dependent on bounded variables, our soundness result holds for general unbounded real variables.

Remark 6 (Non-strict inequalities). *Although we considered invariants consisting of inequalities with positivity witnesses, i.e. invariants of the form $\bigwedge(g(x) > 0)$, our algorithm can easily be extended to generate invariants with non-strict inequalities, i.e. invariants of the form $\bigwedge(g(x) \geq 0)$. To do so, it suffices to replace Equation (†) in Step 3 of the algorithm with Equation (1), i.e. remove the ϵ-variables (positivity witnesses). This results in a sound, but not complete, method for generating non-strict polynomial invariants.*

[‡‡]Note that Putinar's positivstellensatz requires the compactness condition and so does Corollary 3.2.

[§§]In order to obtain a more realistic model of floating-point variables, one should also introduce constraints that ensure a variable cannot hold an arbitrarily small non-zero value. However, these details are beyond the scope of the current work.

Remark 7 (Complexity). *It is straightforward to verify that Steps 1–3 of* StrongInvSynth *have polynomial runtime. Hence, our algorithm provides a polynomial reduction from the Strong Invariant Synthesis problem to the problem of finding representative solutions of a system of quadratic equalities. As mentioned earlier, this problem is solvable in subexponential time (see [Grigor'ev and Vorobjov 1988]). Hence, the total runtime of our approach is subexponential, too.*

Given the discussion above, we have the following theorem:

Theorem 3.8 (Strong Invariant Synthesis). *Given a non-recursive program P and a pre-condition* Pre *that satisfies the compactness condition, the* StrongInvSynth *algorithm solves the Strong Invariant Synthesis problem in subexponential time. This solution is sound and semi-complete.*

Remark 8 (Practical Inefficiency). *Despite its subexponential runtime, the algorithm of [Grigor'ev and Vorobjov 1988] has a poor performance in practice [Hong 1991]. Hence, Theorem 3.8 can only be considered as a theoretical contribution and is not applicable to real-world programs.*

3.4 Weak Invariant Synthesis and Practical Method for Invariant Generation

Due to the practical inefficiency mentioned in Remark 8, in this section we focus on using a very similar approach to reduce the Weak Invariant Synthesis problem to QCLP. Given that there are many industrial solvers capable of handling large real-world instances of QCLP, this reduction will provide a practical sound and semi-complete method for polynomial invariant generation. We now provide an algorithm for the Weak Invariant Synthesis problem. This algorithm is very similar to StrongInvSynth, so we only describe the differences.

The WeakInvSynth Algorithm. Our algorithm WeakInvSynth takes the same set of inputs as StrongInvSynth, as well as an objective function obj over the resulting inductive invariants. We assume that obj is a linear or quadratic polynomial over the s-variables in the template. Intuitively, obj serves as a measure of desirability of a synthesized invariant and the goal is to find the most desirable invariant.

The first three steps of the algorithm are the same as StrongInvSynth. The only difference is in Step 4, where WeakInvSynth needs to find only one solution for the computed system of quadratic equalities, i.e. the solution that maximizes obj. Hence, Step 4 is changed as follows:

Step 4) Finding the optimal solution. Step 3 has generated a system of quadratic equalities. In this step, the algorithm uses a QCLP-solver to find a solution σ of this system that maximizes the objective function obj. It then outputs the inductive invariant $\eta_\sigma := \eta[s_{\ell,i,j} \leftarrow \sigma(s_{\ell,i,j})]$.

Example 9. In Example 1, we mentioned that our goal is to prove that the return value of sum is less than $0.5 \cdot n^2 + 0.5 \cdot n + 1$, i.e. we want to obtain

$$0.5 \cdot \bar{n}^2 + 0.5 \cdot \bar{n} + 1 - r > 0 \ (*)$$

at the endpoint label 9 of sum. To do so, our algorithm calls a QCLP-solver over the system of quadratic equalities obtained in Example 8, with the objective of minimizing the Euclidean distance between the coefficients synthesized for $\eta(9)$ and those of $(*)$. The QCLP-solver obtains a solution σ (i.e. a valuation to the new unknown $s-$, $t-$ and $\epsilon-$variables), such that $\eta(9)[s_{9,i,j} :\leftarrow \sigma(s_{9,i,j})] = 0.5 \cdot \bar{n}^2 + 0.5 \cdot \bar{n} + 1 - r > 0$, hence proving the desired invariant. The complete solution is provided in Appendix B.1. ■

Remark 9 (Form of the Objective Function). *At first sight, the objective functions considered above might seem rather bizarre, given that they are functions of the unknown s-variables, i.e. the coefficients of the invariant which should be synthesized by the algorithm. We remark two points:*

- *In our view, this is the most useful formulation. Note that in many cases, such as the example above and our experimental results in Section 5, the goal of a verification process is to prove that a certain desired invariant* $\mathsf{Inv}(\ell)$ *holds at a specific point* ℓ *of the program. This goal can be specified as an objective function over the s-variables. However, it does not simplify the invariant generation problem, because although* $\mathsf{Inv}(\ell)$ *is given, in order to prove that it is really an invariant, the algorithm has to generate an inductive invariant at every other point of the program, too.*
- *Our approach does not depend on the form of objective functions, hence the objective can be any other linear or quadratic function (possibly depending on other variables) and our results will remain intact. It can even be a non-quadratic function, in which case the reduction would be to general quadratically-constrained optimization.*

Remark 10 (QCLP). *QCLP is one of the most well-studied optimization problems [Chen et al. 2017; Linderoth 2005]. It is NP-hard in theory, but in practice, there are many efficient solvers (e.g. [Andersen and Andersen 2018; IBM 2019; Rothberg et al. 2018]) for handling its real-world instances. These solvers are scalable and have been successfully applied to important real-world verification problems, such as solving large POMDPs [Amato et al. 2007].*

Note that Lemmas 3.6 and 3.7 (Soundness and Completeness) carry over to this case without any modification, so we have the following theorem:

Theorem 3.9 (Weak Invariant Synthesis). *Given a non-recursive program P, a pre-condition* Pre *that satisfies the compactness condition and a linear/quadratic objective function obj, the* WeakInvSynth *algorithm reduces the Weak Invariant Synthesis problem to QCLP in* polynomial time. *This reduction is sound and semi-complete.*

4 POLYNOMIAL INVARIANTS FOR RECURSIVE PROGRAMS

In this section, we extend our algorithms to handle recursion. Recall that the only differences between recursive and non-recursive inductive invariants are (i) the presence of function-call statements in recursive programs, (ii) the presence of post-conditions, and (iii) the post-condition consecution requirement. Note that we expect an invariant generation algorithm for recursive programs to also synthesize a post-condition for every function. We now describe how to change the algorithms in the previous section to handle these points and illustrate the changes on an example program.

```
  rsum(n) {
1:  if n ≤ 0 then
2:      return n
    else
3:      m := n − 1;
4:      s := rsum(m);
5:      if ⋆ then
6:          s := s + n
        else
7:          skip
        fi;
8:      return s
    fi
9:  }
```

Fig. 4. A recursive variant of the non-deterministic summation program

Example 10. Consider the program in Figure 4, which is a recursive variant of the non-deterministic summation program of Figure 2. We use this program to illustrate our approach for handling recursion. ■

The RecStrongInvSynth and RecWeakInvSynth Algorithms. Our algorithm for Strong (resp. Weak) Invariant Synthesis over a recursive program P is called RecStrongInvSynth (resp. RecWeakInvSynth). It takes the same input as StrongInvSynth (resp. WeakInvSynth), except that the input program P can now be recursive. It performs the same steps as in its non-recursive counterpart, except that the following additional actions are taken in Steps 1 and 2[¶¶]:

Step 1.a) Setting up a template for the post-condition. Let $\hat{\mathbf{M}}_d^f = \{\hat{m}_1, \hat{m}_2, \ldots, \hat{m}_{\hat{r}}\}$ be the set of all monomials of degree at most d over $\{\mathtt{ret}^f, \bar{v}_1, \ldots, \bar{v}_n\}$. The algorithm generates an additional template $\mu(f) := \bigwedge_{i=1}^{n} \varphi_{f,i}$ where each $\varphi_{f,i}$ is of the form $\varphi_{f,i} := \left(\sum_{j=1}^{\hat{r}} s_{f,i,j} \cdot \hat{m}_j > 0\right)$ where the $s_{f,i,j}$'s are additional new s-variables. Intuitively, our goal is to synthesize the right value for s-variables such that (μ, η) becomes a recursive inductive invariant. As a consequence, μ will be a post-condition and η a valid invariant.

Example 11. Consider the program in Figure 4 and assume that each desired invariant/post-condition consists of a single quadratic inequality. The algorithm generates a template $\mu(\mathsf{rsum})$ for the post-condition of rsum. By definition, such a post-condition can only depend on $\bar{n}$, i.e. the value passed for the parameter n when rsum is called, and the return value $r := \mathtt{ret}^{\mathsf{rsum}}$. Hence, the algorithm generates the following template:

$$\mu(\mathsf{rsum}) := s_{\mathsf{rsum},1,1} + s_{\mathsf{rsum},1,2} \cdot \bar{n} + s_{\mathsf{rsum},1,3} \cdot r + s_{\mathsf{rsum},1,4} \cdot \bar{n}^2 + s_{\mathsf{rsum},1,5} \cdot \bar{n} \cdot r + s_{\mathsf{rsum},1,6} \cdot r^2 > 0$$

■

Step 2.a) Setting up constraint pairs at function-call statements. For every transition $e = (\ell, \bot, \ell')$ where ℓ is a function-call statement of the form $v_0 := f'(v_1, \ldots, v_n)$ calling a function

[¶¶]More specifically, the order of steps in the recursive variants of our algorithms is as follows: 1, 1.a, 2, 2.a, 2.b, 3, 4.

with header $f'(v'_1, \ldots, v'_n)$, and every polynomial g for which $g > 0$ appears in $\eta(\ell')$, the algorithm defines a new program variable v^*_0 and adds the following constraint pair to Λ_e:

$$\left(\begin{array}{c} \mathsf{Pre}(\ell) \wedge \eta(\ell) \wedge \mathsf{Pre}(\ell^{f'}_{\mathsf{in}})[v'_i \leftarrow v_i, \bar{v}'_i \leftarrow v_i] \wedge \\ \mu(f')[\mathtt{ret}^{f'} \leftarrow v^*_0, \bar{v}'_i \leftarrow v_i] \wedge \mathsf{Pre}(\ell')[v_0 \leftarrow v^*_0] \end{array} \quad , \quad g[v_0 \leftarrow v^*_0] \right),$$

in which $\phi[x \leftarrow y]$ is the result of replacing every occurrence of x in ϕ with a y. Intuitively, v^*_0 models the value of v_0 after the function call (equivalently the return value of f')[***]. The constraint pair above encodes the consecution requirement at function-call labels, i.e. it simply requires every valid *abstract* path that satisfies the invariant at ℓ to satisfy it at ℓ', too. Note that a valid abstract path must satisfy the post-condition and all the pre-conditions.

Example 12. Consider the transition $e = (4, \bot, 5)$ in Figure 4. The algorithm computes the following constraint and adds it to Λ_e:

$$\left(\begin{array}{c} \mathsf{Pre}(4) \wedge \eta(4) \wedge \mathsf{Pre}(1)[n \leftarrow m, \bar{n} \leftarrow m] \wedge \\ \mu(\mathsf{rsum})[\mathtt{ret}^{\mathsf{rsum}} \leftarrow s^*, \bar{n} \leftarrow m] \wedge \mathsf{Pre}(5)[s \leftarrow s^*] \end{array} \quad , \quad \eta(5)[s \leftarrow s^*] \right).$$

We now explain this constraint in detail. The purpose of this constraint is to enforce the consecution property in the transition e from label 4 to label 5. Recall that the consecution property requires that for every *valid* unit-length *abstract* path $\pi = \langle(\mathsf{rsum}, 4, \nu_4), (\mathsf{rsum}, 5, \nu_5)\rangle$, we have $\nu_4 \models \eta(4) \implies \nu_5 \models \eta(5)$. Since the variable s is updated in line 4, we use s to denote its value before execution of the recursive call and s^* to model its value after the function call. Hence, $\nu_5 \models \eta(5)$ can be simply rewritten as $\eta(5)[s \leftarrow s^*]$ (the second component of the above constraint). On the other hand, the first component of the constraint should encode the properties that (a) $\nu_4 \models \eta(4)$ and (b) π is a valid abstract path. The property (a) is ensured by including $\eta(4)$ in the first component of the constraint. Similarly, (b) is encoded as follows:

- $\mathsf{Pre}(4)$ encodes the requirement $\nu_4 \models \mathsf{Pre}(4)$.
- $\mathsf{Pre}(1)[n \leftarrow m, \bar{n} \leftarrow m]$ encodes the requirement that the function rsum can be called using the parameter m, i.e. that m satisfies the pre-condition of $1 = \ell^{\mathsf{rsum}}_{\mathsf{in}}$.
- $\mu(\mathsf{rsum})[\mathtt{ret}^{\mathsf{rsum}} \leftarrow s^*, \bar{n} \leftarrow m]$ checks that the call to rsum is abstracted correctly, i.e. that the value s^* returned by rsum respects the post-condition $\mu(\mathsf{rsum})$. Note that in this case we know that rsum was called with parameter m, hence the replacement $\bar{n} \leftarrow m$.
- Finally, $\mathsf{Pre}(5)[s \leftarrow s^*]$ encodes the requirement that the program should be able to continue its execution from point 5 with the new value of s, or equivalently that $\nu_5 \models \mathsf{Pre}(5)$.

■

Step 2.b) Setting up constraint pairs for post-condition consecution. For each transition $e = (\ell, \alpha, \ell')$ where ℓ is a **return** statement and $\ell' = \ell^f_{\mathsf{out}}$ for some program function f, the algorithm generates the following constraint pairs:

- For every polynomial g such that $g > 0$ appears in $\mu(f)$, the algorithm adds the condition pair $(\mathsf{Pre}(\ell) \wedge \eta(\ell) \wedge (\mathsf{Pre}(\ell') \circ \alpha), g \circ \alpha)$ to Λ_e.

Informally, these constraints encode the post-condition consecution requirement.

Example 13. Consider the transition $e = (2, \mathtt{ret}^{\mathsf{rsum}} \leftarrow n, 9)$ in the program of Figure 4. The algorithm generates the following constraint and adds it to Λ_e :

$$\left(\mathsf{Pre}(2) \wedge \eta(2) \wedge \mathsf{Pre}(9)[\mathtt{ret}^{\mathsf{rsum}} \leftarrow n] \quad , \quad \mu(\mathsf{rsum})[\mathtt{ret}^{\mathsf{rsum}} \leftarrow n]\right) =$$

[***]Note that v_0 is the only variable in f whose value might change after the call to f'. Hence, we need to distinguish between the initial value of v_0 and its value after the execution of f', which is denoted by v^*_0. However, such distinction is unnecessary for other variables.

$$\left(\mathsf{Pre}(2) \wedge \eta(2) \wedge \mathsf{Pre}(9)[\mathtt{ret}^{\mathsf{rsum}} \leftarrow n] \quad , \quad \begin{array}{c} s_{\mathsf{rsum},1,1} + s_{\mathsf{rsum},1,2} \cdot \bar{n} + s_{\mathsf{rsum},1,3} \cdot n + s_{\mathsf{rsum},1,4} \cdot \bar{n}^2 \\ + s_{\mathsf{rsum},1,5} \cdot \bar{n} \cdot n + s_{\mathsf{rsum},1,6} \cdot n^2 > 0 \end{array}\right).$$

Intuitively, the constraint above enforces the post-condition consecution requirement, i.e. that in every valid execution step going from line 2 to line 9, the post-condition $\mu(\mathsf{rsum})$ holds. The exact same process applies to the transition $(8, \mathtt{ret}^{\mathsf{rsum}} \leftarrow s, 9)$ and produces the following constraint:

$$\big(\mathsf{Pre}(8) \wedge \eta(8) \wedge \mathsf{Pre}(9)[\mathtt{ret}^{\mathsf{rsum}} \leftarrow s] \quad , \quad \mu(\mathsf{rsum})[\mathtt{ret}^{\mathsf{rsum}} \leftarrow s]\big) =$$

$$\left(\mathsf{Pre}(8) \wedge \eta(8) \wedge \mathsf{Pre}(9)[\mathtt{ret}^{\mathsf{rsum}} \leftarrow s] \quad , \quad \begin{array}{c} s_{\mathsf{rsum},1,1} + s_{\mathsf{rsum},1,2} \cdot \bar{n} + s_{\mathsf{rsum},1,3} \cdot s + s_{\mathsf{rsum},1,4} \cdot \bar{n}^2 \\ + s_{\mathsf{rsum},1,5} \cdot \bar{n} \cdot s + s_{\mathsf{rsum},1,6} \cdot s^2 > 0 \end{array}\right).$$

■

The soundness, completeness and complexity arguments carry over from the non-recursive case. Hence, we have the following theorems:

Theorem 4.1 (Recursive Strong Invariant Synthesis). *Given a recursive program P and a pre-condition* Pre *that satisfies the compactness condition, the* RecStrongInvSynth *algorithm solves the Strong Invariant Synthesis problem in subexponential time. This solution is sound and semi-complete.*

Theorem 4.2 (Recursive Weak Invariant Synthesis). *Given a recursive program P, a pre-condition* Pre *that satisfies the compactness condition and a linear/quadratic objective function obj, the* RecWeakInvSynth *algorithm reduces the Weak Invariant Synthesis Problem to QCLP/QCQP in* polynomial time. *This reduction is sound and semi-complete.*

5 EXPERIMENTAL RESULTS

Implementation. We implemented our algorithms for weak invariant generation, i.e. WeakInvSynth and RecWeakInvSynth, in Java and used the LOQO optimizer [Vanderbei 2006] for solving the QCLPs. All results were obtained on an Intel Core i5-7200U (2.5 GHz) machine with 6 GB of RAM, running Ubuntu 18.04.

Non-recursive Results. We used the benchmarks in [Rodríguez-Carbonell 2018], which contain programs, pre-conditions, and the desired partial invariants (invariants at a few points of the program) that are needed for their verification. We ignored benchmarks that contained non-polynomial assignments/pre-conditions. The results are summarized in Table 2. Note that our algorithm is not complete for *non-strict* invariants (Remark 6), but it could successfully generate all the desired invariants for these benchmarks.

Benchmark	n	d	|V|	|S|	Runtime
cohendiv	1	1	6	622	15.236s
divbin	1	1	5	738	5.399s
hard	1	2	6	8324	27.952s
mannadiv	1	2	5	2561	18.222s
wensely	1	2	7	9422	20.051s
sqrt	1	2	4	2030	5.808s
dijkstra	1	2	5	5072	12.776s
z3sqrt	1	2	6	4692	12.944s
freire1	1	2	3	1210	26.474s
freire2	1	2	4	1016	10.670s
euclidex1	1	2	11	11191	1m37.493s
euclidex2	1	2	8	11156	39.323s
euclidex3	1	2	13	36228	3m23.110s
lcm1	1	2	6	6589	17.851s
lcm2	1	2	6	6176	18.714s
prodbin	1	2	5	5038	12.125s
prod4br	1	2	6	10522	43.205s
cohencu	1	2	5	3424	11.778s
petter	1	2	3	1080	20.390s

Table 2. Experimental results over the benchmarks of [Rodríguez-Carbonell 2018]. Each benchmark contained a single function. |V| is the number of program variables and |S| is the size of the quadratic system, i.e. number of quadratic equalities and inequalities, generated by our approach.

Recursive Results. Recursive results are shown in Table 3. Our recursive benchmarks can be divided in two categories:

- *Reinforcement Learning for Cyber-physical Systems.* We ran our approach on three programs from [Zhu et al. 2019] which are used for safety verification of reinforcement learning applications in cyber-physical systems such as Segway transporters. In these examples, the desired partial invariants are linear. However, the programs themselves contain polynomial assignments and conditions of degree 4. For this reason, the given partial invariants cannot be extended to a linear inductive invariant. Therefore, approaches for linear invariant generation cannot handle these programs.
- *Classical Examples.* Additionally, we considered the recursive summation program of Figure 4, as well as its extensions to sums of squares and cubes, in order to show that our algorithm is able to synthesize invariants of higher degrees. We also considered a program that recursively computes the largest power of 2 that is no more than a given bound x, showing that our algorithm can handle recursive invariants with more than one assertion at each label. Finally, we generated invariants for an implementation of the Merge Sort algorithm that returns the number of inversions in a given sequence [Cormen et al. 2009][†††]. See Appendix B.2 for details.

[†††]We replaced comparisons over sequence elements with non-determinism and obtained invariants that proved the return value, i.e. number of inversions in a sequence of length k, is at most $\binom{k}{2}$.

	Benchmark	n	d	|V|	|S|	Runtime
Reinforcement Learning [Zhu et al. 2019]	inverted-pendulum	1	3	7	9951	8m16.093s
	strict-inverted-pendulum	4	2	7	14390	9m47.783s
	oscillator	1	2	7	3552	39.749s
Classical Examples (Appendix B.2)	recursive-sum	1	2	3	1700	10.919s
	recursive-square-sum	1	3	3	1121	17.438s
	recursive-cube-sum	1	4	3	15840	3m41.211s
	pw2	2	1	3	430	5.438s
	merge-sort	1	2	13	33002	1m18.093s

Table 3. Experimental results over recursive programs. |V| is the number of program variables and |S| is the size of the quadratic system, i.e. number of quadratic equalities and inequalities, generated by our approach.

Remark 11. *Note that our runtimes are typically under a minute, while the maximum runtime is close to 10 minutes. This shows that our approach is applicable in practice and that the reduction to QCLP does not suffer from the same impracticalities as the algorithm of [Grigor'ev and Vorobjov 1988], which, according to [Hong 1991], would take years to solve problems of this size. Also note that previous approaches with completeness guarantees, such as [Colón et al. 2003], are not applicable to our benchmarks due to the presence of non-linear invariants/assignments.*

This being said, our approach is still much slower than previous sound approaches that do not provide any completeness guarantee, e.g. [Farzan and Kincaid 2015]. Hence, there is currently a trade-off between exactness (completeness guarantees) and efficiency. While the theoretical semi-completeness guarantee is a key novelty of our approach, we expect that further advancements in quadratic programming, which is an active area of research in optimization, would narrow the runtime gap.

6 CONCLUSION

We presented a sound and semi-complete method to generate polynomial invariants for programs with polynomial updates. On the theoretical side, we showed that our approach has subexponential complexity. On the practical side, we demonstrated how to generate polynomial invariants using QCLP. While there are many sound and efficient methods for polynomial invariant generation, they lack guarantees of completeness. The key novelty of our method is to ensure semi-completeness, albeit at the cost of efficiency. Using further optimization techniques in conjunction with our method, or finding further practical improvements to quadratic constraint solving that can be utilized for improving the efficiency of polynomial invariant generation are directions of future work.

REFERENCES

Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. 2012. Ufo: A framework for abstraction-and interpolation-based software verification. In *CAV*. Springer, 672–678.

Frances E Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*, Vol. 5. ACM, 1–19.

Rajeev Alur, Thao Dang, and Franjo Ivančić. 2006. Predicate abstraction for reachability analysis of hybrid systems. *ACM transactions on embedded computing systems (TECS)* 5, 1 (2006), 152–199.

Christopher Amato, Daniel S. Bernstein, and Shlomo Zilberstein. 2007. Solving POMDPs Using Quadratically Constrained Linear Programs. In *IJCAI*. 2418–2424.

Erling D. Andersen and Knud D. Andersen. 2018. MOSEK Optimization Suite. (2018). https://www.mosek.com/

Roberto Bagnara, Enric Rodríguez-Carbonell, and Enea Zaffanella. 2005. Generation of Basic Semi-algebraic Invariants Using Convex Polyhedra. In *SAS*. 19–34.

Saugata Basu, Richard Pollack, and Marie-Françoise Coste-Roy. 2007. *Algorithms in real algebraic geometry*. Vol. 10. Springer Science & Business Media.

Aaron R Bradley, Zohar Manna, and Henny B Sipma. 2005. Linear ranking with reachability. In *CAV*. Springer, 491–504.

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic program analysis with martingales. In *CAV*. Springer, 511–526.

Aleksandar Chakarov and Sriram Sankaranarayanan. 2014. Expectation Invariants for Probabilistic Program Loops as Fixed Points. In *SAS*. 85–100.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *CAV*. 3–22.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2017a. Non-polynomial Worst-Case Analysis of Recursive Programs. In *CAV*. 41–63.

Krishnendu Chatterjee, Petr Novotný, and Dorde Zikelic. 2017b. Stochastic invariants for probabilistic termination. In *POPL*. 145–160.

Chen Chen, Alper Atamtürk, and Shmuel S Oren. 2017. A spatial branch-and-cut method for nonconvex QCQP with bounded complex variables. *Mathematical Programming* 165, 2 (2017), 549–577.

Yinghua Chen, Bican Xia, Lu Yang, Naijun Zhan, and Chaochen Zhou. 2007. Discovering non-linear ranking functions by solving semi-algebraic systems. In *ICTAC*. Springer, 34–49.

Yu-Fang Chen, Chih-Duo Hong, Bow-Yaw Wang, and Lijun Zhang. 2015. Counterexample-Guided Polynomial Loop Invariant Generation by Lagrange Interpolation. In *CAV*. 658–674.

Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *CAV*. 420–432.

Michael A Colón and Henny B Sipma. 2001. Synthesis of linear ranking functions. In *TACAS*. Springer, 67–81.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

Patrick Cousot. 2005. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *VMCAI*. 1–24.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*. ACM, 238–252.

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *ESOP*. 21–30.

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *POPL*. ACM, 84–96.

Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: dynamic symbolic execution for invariant inference. In *ICSE*. 281–290.

Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. 2016. Polynomial Invariants by Linear Algebra. In *ATVA*. 479–494.

Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. 2013. Inductive invariant generation via abductive inference. In *OOPSLA*. 443–456.

Azadeh Farzan and Zachary Kincaid. 2015. Compositional Recurrence Analysis. In *FMCAD*. 57–64.

Yijun Feng, Lijun Zhang, David N. Jansen, Naijun Zhan, and Bican Xia. 2017. Finding Polynomial Loop Invariants for Probabilistic Programs. In *ATVA*. 400–416.

Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification*. Springer, 65–81.

Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. In *POPL*. 499–512.

Gene H Golub and Charles F Van Loan. 1996. *Matrix computations*. Johns Hopkins Universtiy Press.

Dima Grigor'ev and Nicolai Vorobjov. 1988. Solving systems of polynomial inequalities in subexponential time. *Journal of Symbolic Computation* 5, 1/2 (1988), 37–64.

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2009. Constraint-Based Invariant Inference over Predicate Abstraction. In *VMCAI*. 120–135.

Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. 1997. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design* 11, 2 (1997), 157–185.

Thomas Henzinger and Pei-Hsin Ho. 1994. Model checking strategies for linear hybrid systems. (1994).

Nicholas J Higham. 2009. Cholesky factorization. *Wiley Interdisciplinary Reviews: Computational Statistics* 1, 2 (2009).

Hoon Hong. 1991. Comparison of several decision algorithms for the existential theory of the reals. (1991).

Roger A Horn and Charles R Johnson. 1990. *Matrix analysis*. Cambridge university press.

Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. 2018. Polynomial Invariants for Affine Programs. In *LICS*. 530–539.

Andreas Humenberger, Maximilian Jaroschek, and Laura Kovács. 2017. Automated Generation of Non-Linear Loop Invariants Utilizing Hypergeometric Sequences. In *ISSAC*. 221–228.

IBM. 2019. CPLEX Optimizer: High-performance mathematical programming solver for linear programming, mixed-integer programming and quadratic programming. (2019). https://www.ibm.com/analytics/cplex-optimizer

Joost-Pieter Katoen, Annabelle McIver, Larissa Meinicke, and Carroll C. Morgan. 2010. Linear-Invariant Generation for Probabilistic Programs: - Automated Support for Proof-Based Methods. In *SAS*. 390–406.

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas W. Reps. 2017. Compositional recurrence analysis revisited. In *PLDI*. 248–262.

Zachary Kincaid, John Cyphert, Jason Breck, and Thomas W. Reps. 2018. Non-linear reasoning for invariant synthesis. *PACMPL* 2, POPL (2018), 54:1–54:33.

Wang Lin, Min Wu, Zhengfeng Yang, and Zhenbing Zeng. 2014. Proving total correctness and generating preconditions for loop programs via symbolic-numeric computation methods. *Frontiers of Computer Science* 8, 2 (2014), 192–202.

Jeff Linderoth. 2005. A simplicial branch-and-bound algorithm for solving quadratically constrained quadratic programs. *Mathematical Programming* 103, 2 (2005), 251–282.

Zohar Manna and Amir Pnueli. 1995. *Temporal verification of reactive systems: Safety*. Springer.

Alexandre Maréchal, Alexis Fouilhé, Tim King, David Monniaux, and Michaël Périn. 2016. Polyhedral approximation of multivariate polynomials using Handelman's theorem. In *VMCAI*. Springer, 166–184.

Kenneth L. McMillan. 2008. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *TACAS*. 413–427.

Markus Müller-Olm and Helmut Seidl. 2004. Computing polynomial program invariants. *Inform. Process. Lett.* 91, 5 (2004).

Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. In *PLDI*. ACM, 496–512.

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*. 683–693.

Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. *PLDI* (2016), 614–630.

Mihai Putinar. 1993. Positive polynomials on compact semi-algebraic sets. *Indiana University Mathematics Journal* 42, 3 (1993), 969–984.

Enric Rodríguez-Carbonell. 2018. Some programs that need polynomial invariants in order to be verified. (2018). http://www.cs.upc.edu/~erodri/webpage/polynomial_invariants/list.html

Enric Rodríguez-Carbonell and Deepak Kapur. 2004. Automatic generation of polynomial loop invariants: Algebraic foundations. In *ISSAC*. ACM, 266–273.

Enric Rodríguez-Carbonell and Deepak Kapur. 2007. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Science of Computer Programming* 64, 1 (2007), 54–75.

Edward Rothberg et al. 2018. *Gurobi Optimizer Reference Manual*. Technical Report. Gurobi Optimization, LLC. http://www.gurobi.com/documentation/8.1/refman/index.html

Sriram Sankaranarayanan, Henny Sipma, and Zohar Manna. 2004. Non-linear loop invariant generation using Gröbner bases. In *POPL*. 318–329.

Markus Schweighofer. 2002. An algorithmic approach to Schmüdgen's Positivstellensatz. *Journal of Pure and Applied Algebra* 166, 3 (2002), 307–319.

Rahul Sharma and Alex Aiken. 2016. From invariant checking to invariant inference using randomized search. *Formal Methods in System Design* 48, 3 (2016), 235–256.

Bernd Sturmfels. 2002. *Solving systems of polynomial equations*. American Mathematical Society.

Robert J. Vanderbei. 2006. *LOQO User's Manual - Version 4.05*. Technical Report. Princeton University.

Lu Yang, Chaochen Zhou, Naijun Zhan, and Bican Xia. 2010. Recent advances in program verification through computer algebra. *Frontiers of Computer Science in China* 4, 1 (2010), 1–16.

He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. 2019. An Inductive Synthesis Framework for Verifiable Reinforcement Learning. In *PLDI*.

A DETAILED SYNTAX

Detailed Grammar. Figure 5 provides a more detailed grammar specifying the syntax of non-deterministic recursive programs with polynomial assignments and guards.

$$
\begin{array}{rcl}
\langle prog \rangle & ::= & \langle func \rangle \\
 & & |\langle func \rangle\ \langle prog \rangle \\
\langle func \rangle & ::= & \langle fname \rangle\ \text{`('}\ \langle varlist \rangle\ \text{`)'}\ \text{`\{'}\ \langle stmtlist \rangle\ \text{`\}'} \\
\langle varlist \rangle & ::= & \langle var \rangle \\
 & & |\langle var \rangle\ \text{`,'}\ \langle varlist \rangle \\
\langle stmtlist \rangle & ::= & \langle stmt \rangle \\
 & & |\langle stmt \rangle\ \text{`;'}\ \langle stmtlist \rangle \\
\langle stmt \rangle & ::= & \text{`}\mathbf{skip}\text{'} \\
 & & |\langle var \rangle\ \text{`:='}\ \langle expr \rangle \\
 & & |\text{`}\mathbf{if}\text{'}\ \langle bexpr \rangle\ \text{`}\mathbf{then}\text{'}\ \langle stmtlist \rangle\ \text{`}\mathbf{else}\text{'}\ \langle stmtlist \rangle\ \text{`}\mathbf{fi}\text{'} \\
 & & |\text{`}\mathbf{if}\text{'}\ \text{`}\star\text{'}\ \text{`}\mathbf{then}\text{'}\ \langle stmtlist \rangle\ \text{`}\mathbf{else}\text{'}\ \langle stmtlist \rangle\ \text{`}\mathbf{fi}\text{'} \\
 & & |\text{`}\mathbf{while}\text{'}\ \langle bexpr \rangle\ \text{`}\mathbf{do}\text{'}\ \langle stmtlist \rangle\ \text{`}\mathbf{od}\text{'} \\
 & & |\langle var \rangle := \langle fname \rangle\ \text{`('}\ \langle varlist \rangle\ \text{`)'} \\
 & & |\text{`}\mathbf{return}\text{'}\ \langle expr \rangle \\
\langle bexpr \rangle & ::= & \langle literal \rangle \\
 & & |\text{`}\neg\text{'}\ \langle bexpr \rangle \\
 & & |\langle bexpr \rangle\ \text{`}\vee\text{'}\ \langle bexpr \rangle \\
 & & |\langle bexpr \rangle\ \text{`}\wedge\text{'}\ \langle bexpr \rangle \\
\langle literal \rangle & ::= & \langle expr \rangle\ \text{`}<\text{'}\ \langle expr \rangle \\
 & & |\langle expr \rangle\ \text{`}\leq\text{'}\ \langle expr \rangle \\
 & & |\langle expr \rangle\ \text{`}\geq\text{'}\ \langle expr \rangle \\
 & & |\langle expr \rangle\ \text{`}>\text{'}\ \langle expr \rangle \\
\langle expr \rangle & ::= & \langle var \rangle \\
 & & |\langle constant \rangle \\
 & & |\langle expr \rangle\ \text{`}+\text{'}\ \langle expr \rangle \\
 & & |\langle expr \rangle\ \text{`}-\text{'}\ \langle expr \rangle \\
 & & |\langle expr \rangle\ \text{`}*\text{'}\ \langle expr \rangle
\end{array}
$$

Fig. 5. Detailed Syntax of Non-deterministic Recursive Programs

Syntactic Assumptions. We assume that each function in **F** is defined exactly once in the program, function headers do not contain duplicate variables, and each function call statement provides exactly as many parameters as defined in the header of the function that is being called. Moreover, we assume that no variable appears in both sides of a function call statement.

B EXPERIMENTAL RESULTS

B.1 The Invariant Synthesized for Our Running Example

Our weak invariant generation algorithm, i.e. WeakInvSynth, generates the following inductive invariant for the running example of Figure 2:

ℓ	$\mathsf{Pre}(\ell)$	$\mathsf{Ind}(\ell)$
1	$\bar{n} = n, n \geq 1$	$0.13 - 0.01 \cdot \bar{n} - 0.05 \cdot r - 0.07 \cdot s - 0.24 \cdot i + 0.06 \cdot n + 0.16 \cdot \bar{n}^2 - 0.08 \cdot \bar{n} \cdot r$ $+0.11 \cdot r^2 - 0.13 \cdot \bar{n} \cdot s + 0.18 \cdot r \cdot s + 0.15 \cdot s^2 - 0.13 \cdot \bar{n} \cdot i + 0.16 \cdot r \cdot i$ $+0.24 \cdot i \cdot s + 0.23 \cdot i^2 + 0.07 \cdot \bar{n} \cdot n - 0.52 \cdot r \cdot n - 0.67 \cdot n \cdot s - 0.66 \cdot i \cdot n + 1.10 \cdot n^2 > 0$
2	**true**	$0.09 - 0.01 \cdot \bar{n} - 0.18 \cdot r - 0.30 \cdot s + 0.09 \cdot i + 0.11 \cdot n + 0.03 \cdot \bar{n}^2 + 0.01 \cdot \bar{n} \cdot r$ $+0.13 \cdot r^2 - 0.03 \cdot \bar{n} \cdot s + 0.16 \cdot r \cdot s + 0.23 \cdot s^2 - 0.01 \cdot \bar{n} \cdot i - 0.18 \cdot r \cdot i$ $-0.30 \cdot i \cdot s + 0.09 \cdot i^2 + 0.01 \cdot \bar{n} \cdot n + 0.11 \cdot i \cdot n > 0$
3	**true**	$0.49 + 0.01 \cdot \bar{n} + 0.11 \cdot r - 0.59 \cdot s - 0.30 \cdot i - 0.29 \cdot n$ $+0.13 \cdot r^2 - 0.01 \cdot \bar{n} \cdot s - 0.35 \cdot r \cdot s + 0.60 \cdot s^2 + 0.05 \cdot r \cdot i$ $-0.01 \cdot i \cdot s + 0.16 \cdot i^2 - 0.04 \cdot r \cdot n - 0.06 \cdot n \cdot s - 0.04 \cdot i \cdot n + 0.18 \cdot n^2 > 0$
4	**true**	$0.20 - 0.12 \cdot \bar{n} + 0.01 \cdot r - 0.01 \cdot s - 0.22 \cdot i - 0.07 \cdot n$ $+1.08 \cdot \bar{n}^2 + 0.10 \cdot \bar{n} \cdot r + 0.15 \cdot r^2 - 0.49 \cdot \bar{n} \cdot s - 0.08 \cdot r \cdot s + 0.10 \cdot s^2 - 0.65 \cdot \bar{n} \cdot i - 0.15 \cdot r \cdot i$ $+0.16 \cdot i \cdot s + 0.14 \cdot i^2 - 0.57 \cdot \bar{n} \cdot n - 0.11 \cdot r \cdot n + 0.13 \cdot n \cdot s + 0.24 \cdot i \cdot n + 0.22 \cdot n^2 > 0$
5	**true**	$0.22 - 0.12 \cdot \bar{n} + 0.01 \cdot r - 0.02 \cdot s - 0.05 \cdot i - 0.28 \cdot n + 1.08 \cdot \bar{n}^2 + 0.10 \cdot \bar{n} \cdot r$ $+0.15 \cdot r^2 - 0.49 \cdot \bar{n} \cdot s - 0.08 \cdot r \cdot s + 0.10 \cdot s^2 - 0.63 \cdot \bar{n} \cdot i - 0.13 \cdot r \cdot i$ $+0.16 \cdot i \cdot s + 0.14 \cdot i^2 - 0.60 \cdot \bar{n} \cdot n - 0.12 \cdot r \cdot n + 0.13 \cdot n \cdot s + 0.22 \cdot i \cdot n + 0.24 \cdot n^2 > 0$
6	**true**	$0.22 - 0.12 \cdot \bar{n} + 0.01 \cdot r - 0.02 \cdot s - 0.05 \cdot i - 0.28 \cdot n + 1.08 \cdot \bar{n}^2 + 0.10 \cdot \bar{n} \cdot r$ $+0.15 \cdot r^2 - 0.49 \cdot \bar{n} \cdot s - 0.08 \cdot r \cdot s + 0.10 \cdot s^2 - 0.63 \cdot \bar{n} \cdot i - 0.13 \cdot r \cdot i$ $+0.16 \cdot i \cdot s + 0.14 \cdot i^2 - 0.60 \cdot \bar{n} \cdot n - 0.12 \cdot r \cdot n + 0.13 \cdot n \cdot s + 0.22 \cdot i \cdot n + 0.24 \cdot n^2 > 0$
7	**true**	$0.15 - 0.09 \cdot \bar{n} - 0.12 \cdot r + 0.03 \cdot s - 0.07 \cdot i - 0.13 \cdot n + 0.23 \cdot \bar{n}^2 + 0.48 \cdot \bar{n} \cdot r$ $+0.35 \cdot r^2 - 0.31 \cdot \bar{n} \cdot s - 0.40 \cdot r \cdot s + 0.13 \cdot s^2 + 0.16 \cdot \bar{n} \cdot i + 0.13 \cdot r \cdot i$ $-0.09 \cdot i \cdot s + 0.06 \cdot i^2 - 0.24 \cdot \bar{n} \cdot n - 0.24 \cdot r \cdot n + 0.18 \cdot n \cdot s - 0.10 \cdot i \cdot n + 0.14 \cdot n^2 > 0$
8	**true**	$0.18 - 0.11 \cdot \bar{n} + 0.01 \cdot r + 0.50 \cdot i - 0.79 \cdot n + 1.09 \cdot \bar{n}^2 + 0.11 \cdot \bar{n} \cdot r$ $+0.15 \cdot r^2 - 0.48 \cdot \bar{n} \cdot s - 0.08 \cdot r \cdot s + 0.10 \cdot s^2 - 0.57 \cdot \bar{n} \cdot i - 0.09 \cdot r \cdot i$ $+0.16 \cdot i \cdot s + 0.18 \cdot i^2 - 0.66 \cdot \bar{n} \cdot n - 0.16 \cdot r \cdot n + 0.12 \cdot n \cdot s + 0.13 \cdot i \cdot n + 0.27 \cdot n^2 > 0$
9	**true**	$1 + 0.5 \cdot \bar{n} - r + 0.5 \cdot \bar{n}^2 > 0$

B.2 Recursive Examples

We used the following recursive examples as benchmarks. Desired assertions are shown in brackets. Pre-conditions are enclosed in # signs. In all cases our algorithm synthesizes an inductive invariant that contains the desired assertions.

```
recursive-sum(n) {
    # n ≥ 0 #
    if n ≤ 0 then
        return n
    else
        m := n − 1;
        s := recursive-sum(m);
        if ⋆ then
            s := s + n
        else
            skip
        fi;
        return s
    fi
 [ret^recursive-sum < 0.5 · n̄² + 0.5 · n̄ + 1 ]}
```

```
recursive-square-sum(n) {
    # n ≥ 0 #
    if n ≤ 0 then
        return n
    else
        m := n − 1;
        s := recursive-sum(m);
        if ⋆ then
            s := s + n ∗ n
        else
            skip
        fi;
        return s
    fi
 [ret^recursive-square-sum < 0.34 · n̄³ + 0.5 · n̄² + 0.17 · n̄ + 1 ]}
```

```
recursive-cube-sum(n) {
    # n ≥ 0 #
    if n ≤ 0 then
        return n
    else
        m := n − 1;
        s := recursive-sum(m);
        if ⋆ then
            s := s + n ∗ n ∗ n
        else
            skip
        fi;
        return s
    fi
 [ret^recursive-cube-sum < 0.25 · n̄² · (n̄ + 1)² + 1 ]}
```

```
pw2(x) {
    // computes the largest power of 2 that is ≤ x
    # x ≥ 1 #
    if x ≥ 2 then
        y := 0.5 ∗ x;
        return 2 ∗ pw2(y)
    else
        return 1
    fi
 [ret^pw2 ≤ x̄ ∧ 2 · ret^pw2 > x̄]}
```

```
merge-sort(s, e)  // sorts and returns number of inversions in[s..e]
{
    # e ≥ s #
    if s ≥ e then
        return 0
    else
        i := 0.5 ∗ s + 0.5 ∗ e;
        j := ⌊i⌋;
        i := j + 1;
        r := merge-sort(s, j);
        ans := merge-sort(i, e);
        ans := ans + r;
        k := s;

        while i ≤ e do
            while k ≤ j do
                if ⋆ then  // array[k] ≤ array[i]
                    k := k + 1;
                    skip  // temp.push_back(array[k])
                else  // array[k] > array[i]
                    ans := ans + j − k + 1;  // add inversions
                    i := i + 1;
                    skip  // temp.push_back(array[i])
                fi
            od;
            skip;  // temp.push_back(array[i])
            i := i + 1
        od;

        while s ≤ e do
            skip;  // copy from temp to array
            s := s + 1
        od;

        return ans
    fi
[ ret^merge-sort < 0.5 · (ē − s̄) · (ē − s̄ + 1) + 1 ]}
```