Non-polynomial Worst-Case Analysis of Recursive Programs

Krishnendu Chatterjee[1], Hongfei Fu[2(✉)], and Amir Kafshdar Goharshady[1]

[1] IST Austria, Klosterneuburg, Austria
[2] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, People's Republic of China
fuhf@ios.ac.cn

Abstract. We study the problem of developing efficient approaches for proving worst-case bounds of non-deterministic recursive programs. Ranking functions are sound and complete for proving termination and worst-case bounds of non-recursive programs. First, we apply ranking functions to recursion, resulting in measure functions, and show that they provide a sound and complete approach to prove worst-case bounds of non-deterministic recursive programs. Our second contribution is the synthesis of measure functions in non-polynomial forms. We show that non-polynomial measure functions with logarithm and exponentiation can be synthesized through abstraction of logarithmic or exponentiation terms, Farkas' Lemma, and Handelman's Theorem using linear programming. While previous methods obtain worst-case polynomial bounds, our approach can synthesize bounds of the form $\mathcal{O}(n \log n)$ as well as $\mathcal{O}(n^r)$ where r is not an integer. We present experimental results to demonstrate that our approach can efficiently obtain worst-case bounds of classical recursive algorithms such as Merge-Sort, Closest-Pair, Karatsuba's algorithm and Strassen's algorithm.

1 Introduction

Automated analysis to obtain quantitative performance characteristics of programs is a key feature of static analysis. Obtaining precise worst-case complexity bounds is a topic of both wide theoretical and practical interest. The manual proof of such bounds can be cumbersome as well as require mathematical ingenuity, e.g., the book *The Art of Computer Programming* by Knuth presents several mathematically involved methods to obtain such precise bounds [52]. The derivation of such worst-case bounds requires a lot of mathematical skills and is not an automated method. However, the problem of deriving precise worst-case bounds is of huge interest in program analysis: (a) first, in applications such as hard real-time systems, guarantees of worst-case behavior are required; and (b) the bounds are useful in early detection of egregious performance problems in large

R. Majumdar and V. Kunčak (Eds.): CAV 2017, Part II, LNCS 10427, pp. 41–63, 2017.
DOI: 10.1007/978-3-319-63390-9_3

code bases. Works such as [36,37,40,41] provide an excellent motivation for the study of automatic methods to obtain worst-case bounds for programs.

Given the importance of the problem of deriving worst-case bounds, the problem has been studied in various different ways.

1. *WCET Analysis.* The problem of worst-case execution time (WCET) analysis is a large field of its own, that focuses on (but is not limited to) sequential loop-free code with low-level hardware aspects [67].
2. *Resource Analysis.* The use of abstract interpretation and type systems to deal with loop, recursion, data-structures has also been considered [1,37,50], e.g., using linear invariant generation to obtain disjunctive and non-linear bounds [19], potential-based methods for handling recursion and inductive data structures [40,41].
3. *Ranking Functions.* The notion of ranking functions is a powerful technique for termination analysis of (recursive) programs [8,9,20,25,58,61,64,68]. They serve as a sound and complete approach for proving termination of non-recursive programs [31], and they have also been extended as ranking supermatingales for analysis of probabilistic programs [12,14,16,29].

Given the many results above, two aspects of the problem have not been addressed.

1. *WCET Analysis of Recursive Programs Through Ranking Functions.* The use of ranking functions has been limited mostly to non-recursive programs, and their use to obtain worst-case bounds for recursive programs has not been explored in depth.
2. *Efficient Methods for Precise Bounds.* While previous works present methods for disjunctive polynomial bounds [37] (such as $\max(0,n)\cdot(1+max(n,m))$), or multivariate polynomial analysis [40], these works do not provide efficient methods to synthesize bounds such as $\mathcal{O}(n\log n)$ or $\mathcal{O}(n^r)$, where r is not an integer.

We address these two aspects, i.e., efficient methods for obtaining non-polynomial bounds such as $\mathcal{O}(n\log n)$, $\mathcal{O}(n^r)$ for recursive programs, where r is not an integer.

Our Contributions. Our main contributions are as follows:

1. First, we apply ranking functions to recursion, resulting in *measure* functions, and show that they provide a sound and complete method to prove termination and worst-case bounds of non-deterministic recursive programs.
2. Second, we present a sound approach for handling measure functions of specific forms. More precisely, we show that *non-polynomial* measure functions involving logarithm and exponentiation can be synthesized using *linear programming* through abstraction of logarithmic or exponentiation terms, Farkas' Lemma, and Handelman's Theorem.
3. A key application of our method is the worst-case analysis of recursive programs. Our procedure can synthesize non-polynomial bounds of the form

$\mathcal{O}(n \log n)$, as well as $\mathcal{O}(n^r)$, where r is not an integer. We show the applicability of our technique to obtain worst-case complexity bounds for several classical recursive programs:

- For *Merge-Sort* [24, Chap. 2] and the divide-and-conquer algorithm for the *Closest-Pair problem* [24, Chap. 33], we obtain $\mathcal{O}(n \log n)$ worst-case bound, and the bounds we obtain are asymptotically optimal. Note that previous methods are either not applicable, or grossly over-estimate the bounds as $\mathcal{O}(n^2)$.
- For *Karatsuba's algorithm* for polynomial multiplication (cf. [52]) we obtain a bound of $\mathcal{O}(n^{1.6})$, whereas the optimal bound is $n^{\log_2 3} \approx \mathcal{O}(n^{1.585})$, and for the classical *Strassen's algorithm* for fast matrix multiplication (cf. [24, Chap. 4]) we obtain a bound of $\mathcal{O}(n^{2.9})$ whereas the optimal bound is $n^{\log_2 7} \approx \mathcal{O}(n^{2.8074})$. Note that previous methods are either not applicable, or grossly over-estimate the bounds as $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively.

4. We present experimental results to demonstrate the effectiveness of our approach.

In general, our approach can be applied to (recursive) programs where the worst-case behaviour can be obtained by an analysis that involves only the structure of the program. For example, our approach cannot handle the Euclidean algorithm for computing the greatest common divisor of two given natural numbers, since the worst-case behaviour of this algorithm relies on Lamé's Theorem [52]. The key novelty of our approach is that we show how non-trivial non-polynomial worst-case upper bounds such as $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^r)$, where r is non-integral, can be soundly obtained, even for recursive programs, using linear programming. Moreover, as our computational tool is linear programming, the approach we provide is also a relatively scalable one (see Remark 2). Due to page limit, we omit the details for syntax, semantics, proofs, experiments and other technical parts. They can be found in the full version [15].

2 Non-deterministic Recursive Programs

In this work, our main contributions involve a new approach for non-polynomial worst-case analysis of recursive programs. To focus on the new contributions, we consider a simple programming language for non-deterministic recursive programs. In our language, (a) all scalar variables hold integers, (b) all assignments to scalar variables are restricted to linear expressions with floored operation, and (c) we do not consider return statements. The reason to consider such a simple language is that (i) non-polynomial worst-case running time often involves non-polynomial terms over integer-valued variables (such as array length) only, (ii) assignments to variables are often linear with possible floored expressions (in e.g. divide-and-conquer programs) and (iii) return value is often not related to worst-case behaviour of programs.

For a set A, we denote by $|A|$ the cardinality of A and $\mathbf{1}_A$ the indicator function on A. We denote by $\mathbb{N}$, $\mathbb{N}_0$, $\mathbb{Z}$, and $\mathbb{R}$ the sets of all positive integers, non-negative integers, integers, and real numbers, respectively. Below we fix a set $\mathcal{X}$ of *scalar* variables.

Arithmetic Expressions, Valuations, and Predicates. The set of *(linear) arithmetic expressions* $\mathfrak{e}$ over $\mathcal{X}$ is generated by the following grammar: $\mathfrak{e} ::= c \mid x \mid \left\lfloor \frac{\mathfrak{e}}{c} \right\rfloor \mid \mathfrak{e} + \mathfrak{e} \mid \mathfrak{e} - \mathfrak{e} \mid c * \mathfrak{e}$ where $c \in \mathbb{Z}$ and $x \in \mathcal{X}$. Informally, (i) $\frac{\cdot}{c}$ refers to division operation, (ii) $\lfloor \cdot \rfloor$ refers to the floored operation, and (iii) $+, -, *$ refer to addition, subtraction and multiplication operation over integers, respectively. In order to make sure that division is well-defined, we stipulate that every appearance of c in $\frac{\mathfrak{e}}{c}$ is non-zero. A *valuation* over $\mathcal{X}$ is a function ν from $\mathcal{X}$ into $\mathbb{Z}$. Informally, a valuation assigns to each scalar variable an integer. Under a valuation ν over $\mathcal{X}$, an arithmetic expression $\mathfrak{e}$ can be *evaluated* to an integer in the straightforward way. We denote by $\mathfrak{e}(\nu)$ the evaluation of $\mathfrak{e}$ under ν. The set of *propositional arithmetic predicates* ϕ over $\mathcal{X}$ is generated by the following grammar: $\phi ::= \mathfrak{e} \leq \mathfrak{e} \mid \mathfrak{e} \geq \mathfrak{e} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$ where $\mathfrak{e}$ represents an arithmetic expression. The satisfaction relation $\models$ between valuations and propositional arithmetic predicates is defined in the straightforward way through evaluation of arithmetic expressions. For each propositional arithmetic predicate ϕ, $\mathbf{1}_\phi$ is interpreted as the indicator function $\nu \mapsto \mathbf{1}_{\nu \models \phi}$ on valuations, where $\mathbf{1}_{\nu \models \phi}$ is 1 if $\nu \models \phi$ and 0 otherwise.

Syntax of the Programming Language. Due to page limit, we present a brief description of our syntax. The syntax is a subset of C programming language: in our setting, we have *scalar variables* which hold integers and *function names* which corresponds to functions (in programming-language sense); assignment statements are indicated by ':=', whose left-hand-side is a scalar variable and whose right-hand-side is a linear arithmetic expression; '**skip**' is the statement which does nothing; while-loops and conditional if-branches are indicated by '**while**' and '**if**' respectively, together with a propositional arithmetic predicate indicating the relevant condition (or guard); demonic non-deterministic branches are indicated by '**if**' and '$\star$'; function declarations are indicated by a function name followed by a bracketed list of non-duplicate scalar variables, while function calls are indicated by a function name followed by a bracketed list of linear arithmetic expressions; each function declaration is followed by a curly-braced compound statement as function body; finally, a program is a sequence of function declarations with their function bodies. Given a recursive program in our syntax, we assign a distinct natural number (called *label* in our context) to every assignment/skip statement, function call, if/while-statement and terminal line in the program. Each label serves as a program counter which indicates the next statement to be executed.

Semantics Through CFGs. We use control-flow graphs (CFGs) to specify the semantics of recursive programs. Informally, a CFG specifies how values for scalar variables and the program counter change in a program.

Definition 1 (Control-Flow Graphs). *A* control-flow graph *(CFG) is a triple which takes the form* (†) $\left(F, \left\{\left(L^{\mathsf{f}}, L^{\mathsf{f}}_{\mathrm{b}}, L^{\mathsf{f}}_{\mathrm{a}}, L^{\mathsf{f}}_{\mathrm{c}}, L^{\mathsf{f}}_{\mathrm{d}}, V^{\mathsf{f}}, \ell^{\mathsf{f}}_{\mathrm{in}}, \ell^{\mathsf{f}}_{\mathrm{out}}\right)\right\}_{\mathsf{f}\in F}, \{\rightarrow_{\mathsf{f}}\}_{\mathsf{f}\in F}\right)$ *where:*

- F *is a finite set of* function names*;*
- *each* L^{f} *is a finite set of* labels *attached to the function name* f*, which is partitioned into (i) the set* $L^{\mathsf{f}}_{\mathrm{b}}$ *of* branching *labels, (ii) the set* $L^{\mathsf{f}}_{\mathrm{a}}$ *of* assignment *labels, (iii) the set* $L^{\mathsf{f}}_{\mathrm{c}}$ *of* call *labels and (iv) the set* $L^{\mathsf{f}}_{\mathrm{d}}$ *of* demonic non-deterministic *labels;*
- *each* V^{f} *is the set of* scalar variables *attached to* f*;*
- *each* $\ell^{\mathsf{f}}_{\mathrm{in}}$ *(resp.* $\ell^{\mathsf{f}}_{\mathrm{out}}$*) is the* initial label *(resp.* terminal label*) in* L^{f}*;*
- *each* $\rightarrow_{\mathsf{f}}$ *is a relation whose every member is a triple of the form* (ℓ, α, ℓ') *for which* ℓ *(resp.* ℓ'*) is the source label (resp. target label) of the triple such that* $\ell \in L^{\mathsf{f}}$ *(resp.* $\ell' \in L^{\mathsf{f}}$*), and* α *is (i) either a propositional arithmetic predicate* ϕ *over* V^{f} *(as the set of scalar variables) if* $\ell \in L^{\mathsf{f}}_{\mathrm{b}}$*, (ii) or an* update function *from the set of valuations over* V^{f} *into the set of valuations over* V^{f} *if* $\ell \in L^{\mathsf{f}}_{\mathrm{a}}$*, (iii) or a pair* (g, h) *with* $\mathsf{g} \in F$ *and* h *being a* value-passing function *which maps every valuation over* V^{f} *to a valuation over* V^{g} *if* $\ell \in L^{\mathsf{f}}_{\mathrm{c}}$*, (iv) or* $\star$ *if* $\ell \in L^{\mathsf{f}}_{\mathrm{d}}$*.*

W.l.o.g, we consider that all labels are natural numbers. We denote by $\mathit{Val}_{\mathsf{f}}$ the set of valuations over V^{f}, for each $\mathsf{f} \in F$. Informally, a function name f, a label $\ell \in L^{\mathsf{f}}$ and a valuation $\nu \in \mathit{Val}_{\mathsf{f}}$ reflects that the current status of a recursive program is under function name f, right before the execution of the statement labeled ℓ in the function body named f and with values specified by ν, respectively.

Example 1. We consider the running example in Fig. 1 which abstracts the running time of Binary-Search. The CFG for this example is depicted in Fig. 2. □

It is intuitive that every recursive program in our setting can be transformed into a CFG. Based on CFGs, the semantics models executions of a recursive program as runs, and is defined through the standard notion of call stack. Below we fix a recursive program P and its CFG taking the form (†). We first define the notion of *stack element* and *configurations* which captures all information within a function call.

```
f(n)  {
1:    if  n ≥ 2  then
2:          f(⌊n/2⌋)
3:        else  skip
      fi
4:  }
```

Fig. 1. A program for Binary-Search

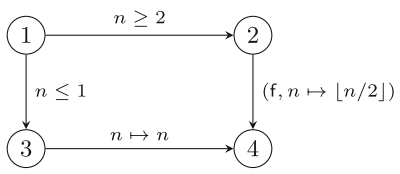

Fig. 2. The CFG for Fig. 1

Stack Elements and Configurations. A *stack element* $\mathfrak{c}$ (of P) is a triple (f, ℓ, ν) (treated as a letter) where $\mathsf{f} \in F$, $\ell \in L^{\mathsf{f}}$ and $\nu \in Val_{\mathsf{f}}$; $\mathfrak{c}$ is non-terminal if $\ell \in L^{\mathsf{f}} \setminus \{\ell^{\mathsf{f}}_{\text{out}}\}$. A *configuration* (of P) is a finite word of non-terminal stack elements (including the empty word ε). Thus, a stack element (f, ℓ, ν) specifies that the current function name is f, the next statement to be executed is the one labelled with ℓ and the current valuation w.r.t f is ν; a configuration captures the whole trace of the call stack.

Schedulers and Runs. To resolve non-determinism indicated by $\star$, we consider the standard notion of *schedulers*, which have the full ability to look into the whole history for decision. Formally, a scheduler π is a function that maps every sequence of configurations ending in a non-deterministic location to the next configuration. A stack element $\mathfrak{c}$ (as the initial stack element) and a scheduler π defines a unique infinite sequence $\{w_j\}_{j \in \mathbb{N}_0}$ of configurations as the execution starting from $\mathfrak{c}$ and under π, which is denoted as the *run* $\rho(\mathfrak{c}, \pi)$. This defines the semantics of recursive programs.

We now define the notion of termination time which corresponds directly to the running time of a recursive program. In our setting, execution of every step takes one time unit.

Definition 2 (Termination Time). *For each stack element* $\mathfrak{c}$ *and each scheduler* π*, the* termination time *of the run* $\rho(\mathfrak{c}, \pi) = \{w_j\}_{j \in \mathbb{N}_0}$*, denoted by* $T(\mathfrak{c}, \pi)$*, is defined as* $T(\mathfrak{c}, \pi) := \min\{j \mid w_j = \varepsilon\}$ *(i.e., the earliest time when the stack is empty) where* $\min \emptyset := \infty$*. For each stack element* $\mathfrak{c}$*, the* worst-case termination-time function $\overline{T}$ *is a function on the set of stack elements defined by:* $\overline{T}(\mathfrak{c}) := \sup\{T(\mathfrak{c}, \pi) \mid \pi$ *is a scheduler for* $P\}$.

Thus $\overline{T}$ captures the worst-case behaviour of the recursive program P.

3 Measure Functions

In this section, we introduce the notion of measure functions for recursive programs. We show that measure functions are sound and complete for nondeterministic recursive programs and serve as upper bounds for the worst-case termination-time function. In the whole section, we fix a recursive program P together with its CFG taking the form (†). We now present the standard notion of *invariants* which represent reachable stack elements. Due to page limit, we omit the intuitive notion of *reachable stack elements*. Informally, a stack element is *reachable* w.r.t an initial function name and initial valuations satisfying a prerequisite (as a propositional arithmetic predicate) if it can appear in the run under some scheduler.

Definition 3 (Invariants). *A (linear)* invariant I *w.r.t a function name* f^* *and a propositional arithmetic predicate* ϕ^* *over* V^{f^*} *is a function that upon any pair* (f, ℓ) *satisfying* $\mathsf{f} \in F$ *and* $\ell \in L^{\mathsf{f}} \setminus \{\ell^{\mathsf{f}}_{\text{out}}\}$*,* $I(\mathsf{f}, \ell)$ *is a propositional arithmetic predicate over* V^{f} *such that (i)* $I(\mathsf{f}, \ell)$ *is without the appearance of floored expressions (i.e.* $\lfloor \cdot \rfloor$*) and (ii) for all stack elements* (f, ℓ, ν) *reachable w.r.t* f^*, ϕ^**,* $\nu \models I(\mathsf{f}, \ell)$*. The invariant* I *is in* disjunctive normal form *if every* $I(\mathsf{f}, \ell)$ *is in disjunctive normal form.*

Obtaining invariants automatically is a standard problem in programming languages, and several techniques exist (such as abstract interpretation [26] or Farkas' Lemma [19]). In the rest of the section we fix a(n initial) function name $\mathsf{f}^* \in F$ and a(n initial) propositional arithmetic predicate ϕ^* over V^{f^*}. For each $\mathsf{f} \in F$ and $\ell \in L^{\mathsf{f}} \backslash \{\ell^{\mathsf{f}}_{\text{out}}\}$, we define $D_{\mathsf{f},\ell}$ to be the set of all valuations ν w.r.t f such that (f, ℓ, ν) is reachable w.r.t f^*, ϕ^*. Below we introduce the notion of measure functions.

Definition 4 (Measure Functions). *A* measure function *w.r.t* f^*, ϕ^* *is a function* g *from the set of stack elements into* $[0, \infty]$ *such that for all stack elements* (f, ℓ, ν)*, the following conditions hold:*

- ***C1:*** *if* $\ell = \ell^{\mathsf{f}}_{\text{out}}$*, then* $g(\mathsf{f}, \ell, \nu) = 0$*;*
- ***C2:*** *if* $\ell \in L^{\mathsf{f}}_{\text{a}} \backslash \{\ell^{\mathsf{f}}_{\text{out}}\}$*,* $\nu \in D_{\mathsf{f},\ell}$ *and* (ℓ, h, ℓ') *is the only triple in* $\rightarrow_{\mathsf{f}}$ *with source label* ℓ *and update function* h*, then* $g(\mathsf{f}, \ell', h(\nu)) + 1 \leq g(\mathsf{f}, \ell, \nu)$*;*
- ***C3:*** *if* $\ell \in L^{\mathsf{f}}_{\text{c}} \backslash \{\ell^{\mathsf{f}}_{\text{out}}\}$*,* $\nu \in D_{\mathsf{f},\ell}$ *and* $(\ell, (\mathsf{g}, h), \ell')$ *is the only triple in* $\rightarrow_{\mathsf{f}}$ *with source label* ℓ *and value-passing function* h*, then* $1 + g(\mathsf{g}, \ell^{\mathsf{g}}_{\text{in}}, h(\nu)) + g(\mathsf{f}, \ell', \nu) \leq g(\mathsf{f}, \ell, \nu)$*;*
- ***C4:*** *if* $\ell \in L^{\mathsf{f}}_{\text{b}} \backslash \{\ell^{\mathsf{f}}_{\text{out}}\}$*,* $\nu \in D_{\mathsf{f},\ell}$ *and* $(\ell, \phi, \ell_1), (\ell, \neg\phi, \ell_2)$ *are namely two triples in* $\rightarrow_{\mathsf{f}}$ *with source label* ℓ*, then* $\mathbf{1}_{\nu \models \phi} \cdot g(\mathsf{f}, \ell_1, \nu) + \mathbf{1}_{\nu \models \neg\phi} \cdot g(\mathsf{f}, \ell_2, \nu) + 1 \leq g(\mathsf{f}, \ell, \nu)$*;*
- ***C5:*** *if* $\ell \in L^{\mathsf{f}}_{\text{d}} \backslash \{\ell^{\mathsf{f}}_{\text{out}}\}$*,* $\nu \in D_{\mathsf{f},\ell}$ *and* $(\ell, \star, \ell_1), (\ell, \star, \ell_2)$ *are namely two triples in* $\rightarrow_{\mathsf{f}}$ *with source label* ℓ*, then* $\max\{g(\mathsf{f}, \ell_1, \nu), g(\mathsf{f}, \ell_2, \nu)\} + 1 \leq g(\mathsf{f}, \ell, \nu)$*.*

Intuitively, a measure function is a non-negative function whose values strictly decrease along the executions regardless of the choice of the demonic scheduler. By applying ranking functions to configurations, one can prove the following theorem stating that measure functions are sound and complete for the worst-case termination-time function.

Theorem 1 (Soundness and Completeness). *(1)* (Soundness). *For all measure functions* g *w.r.t* f^*, ϕ^**, it holds that for all valuations* $\nu \in \mathit{Val}_{\mathsf{f}^*}$ *such that* $\nu \models \phi^*$*, we have* $\overline{T}(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu) \leq g(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu)$*. (2)* (Completeness). $\overline{T}$ *is a measure function w.r.t* f^*, ϕ^**.*

By Theorem 1, to obtain an upper bound on the worst-case termination-time function, it suffices to synthesize a measure function. Below we show that it suffices to synthesize measure functions at cut-points (which we refer as *significant* labels).

Definition 5 (Significant Labels). *Let* $\mathsf{f} \in F$*. A label* $\ell \in L^{\mathsf{f}}$ *is* significant *if either* $\ell = \ell^{\mathsf{f}}_{\text{in}}$ *or* ℓ *is the initial label to some while-loop appearing in the function body of* f*.*

We denote by $L^{\mathsf{f}}_{\text{s}}$ the set of significant locations in L^{f}. Informally, a significant label is a label where valuations cannot be easily deduced from other labels, namely valuations at the start of the function-call and at the initial label of a while loop.

The Expansion Construction (from g to $\widehat{g}$). Let g be a function from $\left\{(\mathsf{f}, \ell, \nu) \mid \mathsf{f} \in F, \ell \in L^{\mathsf{f}}_{\mathrm{s}}, \nu \in \mathit{Val}_{\mathsf{f}}\right\}$ into $[0, \infty]$. One can obtain from g a function $\widehat{g}$ from the set of all stack elements into $[0, \infty]$ in a straightforward way through iterated application of the equality forms of C1–C5.

4 The Synthesis Algorithm

By Theorem 1, measure functions are a sound approach for upper bounds of the worst-case termination-time function, and hence synthesis of measure functions of specific forms provide upper bounds for worst-case behaviour of recursive programs. We first define the synthesis problem of measure functions and then present the synthesis algorithm, where the initial stack element is integrated into the input invariant. Informally, the input is a recursive program, an invariant for the program and technical parameters for the specific form of a measure function, and the output is a measure function if the algorithm finds one, and fail otherwise.

The RecTermBou **Problem.** The RecTermBou problem is defined as follows:

- *Input:* a recursive program P, an invariant I in disjunctive normal form and a quadruple (d, op, r, k) of technical parameters;
- *Output:* a measure function h w.r.t the quadruple (d, op, r, k).

The quadruple (d, op, r, k) specifies the form of a measure function in the way that $d \in \mathbb{N}$ is the degree of the measure function to be synthesized, $\mathrm{op} \in \{\log, \exp\}$ signals either logarithmic (when $\mathrm{op} = \log$) (e.g., $n \ln n$) or exponential (when $\mathrm{op} = \exp$) (e.g., $n^{1.6}$) measure functions, r is a rational number greater than 1 which specifies the exponent in the measure function (i.e., n^r) when $\mathrm{op} = \exp$ and $k \in \mathbb{N}$ is a technical parameter required by Theorem 3. In the input for RecTermBou we fix the exponent r when $\mathrm{op} = \exp$. However, iterating with binary search over an input bounded range we can obtain a measure function in the given range as precise as possible. Moreover, the invariants can be obtained automatically through e.g. [19]. Below we present our algorithm SynAlgo for synthesizing measure functions for the RecTermBou problem. The algorithm is designed to synthesize one function over valuations at each function name and appropriate significant labels so that C1–C5 are fulfilled. Due to page limit, we only illustrate the main conceptual details of our algorithm. In the following, we fix an input from the RecTermBou problem to our algorithm.

Overview. We present the overview of our solution which has the following five steps.

1. *Step 1.* Since one key aspect of our result is to obtain bounds of the form $\mathcal{O}(n \log n)$ as well as $\mathcal{O}(n^r)$, where r is not an integer, we first consider general form of upper bounds that involve logarithm and exponentiation (Step 1(a)), and then consider templates with the general form of upper bounds for significant labels (Step 1(b)).

2. *Step 2.* The second step considers the template generated in Step 1 for significant labels and generate templates for all labels. This step is relatively straightforward.
3. *Step 3.* The third step establishes constraint triples according to the invariant given by the input and the template obtained in Step 2. This step is also straightforward.
4. *Step 4.* The fourth step is the significant step which involves transforming the constraint triples generated in Step 3 into ones without logarithmic and exponentiation terms. The first substep (Step 4(a)) is to consider abstractions of logarithmic, exponentiation, and floored expressions as fresh variables. The next step (Step 4(b)) requires to obtain linear constraints over the abstracted variables. We use Farkas' lemma and Lagrange's Mean-Value Theorem (LMVT) to obtain sound linear inequalities for those variables.
5. *Step 5.* The final step is to solve the unknown coefficients of the template from the constraint triples (without logarithm or exponentiation) obtained from Step 4. This requires the solution of positive polynomials over polyhedrons through the sound form of Handelman's Theorem (Theorem 3) to transform into a linear program.

We first present an informal illustration of the key ideas through a simple example.

Example 2. Consider the task to synthesize a measure function for Karatsuba's algorithm [52] for polynomial multiplication which runs in $c \cdot n^{1.6}$ steps, where c is a coefficient to be synthesized and n represents the maximal degree of the input polynomials and is a power of 2. We describe informally how our algorithm tackles Karatsuba's algorithm. Let n be the length of the two input polynomials and $c \cdot n^{1.6}$ be the template. Since Karatsuba's algorithm involves three submultiplications and seven additions/subtractions, the condition C3 becomes (*) $c \cdot n^{1.6} - 3 \cdot c \cdot \left(\frac{n}{2}\right)^{1.6} - 7 \cdot n \geq 0$ for all $n \geq 2$. The algorithm first abstracts $n^{1.6}$ as a stand-alone variable u. Then the algorithm generates the following inequalities through properties of exponentiation: (**) $u \geq 2^{1.6}, u \geq 2^{0.6} \cdot n$. Finally, the algorithm transforms (*) into (***) $c \cdot u - 3 \cdot \left(\frac{1}{2}\right)^{1.6} \cdot c \cdot u - 7 \cdot n \geq 0$ and synthesizes a value for c through Handelman's Theorem to ensure that (***) holds under $n \geq 2$ and (**). One can verify that $c = 1000$ is a feasible solution since

$$\left(1000 - 3000 \cdot (1/2)^{1.6}\right) \cdot u - 7 \cdot n =$$
$$\frac{7}{2^{0.6}} \cdot \left(u - 2^{0.6} \cdot n\right) + \frac{1000 \cdot 2^{1.6} - 3014}{2^{1.6}} \cdot \left(u - 2^{1.6}\right) + \left(1000 \cdot 2^{1.6} - 3014\right) \cdot 1.$$

Hence, Karatsuba's algorithm runs in $\mathcal{O}(n^{1.6})$ time. □

4.1 Step 1 of SynAlgo

Step 1(a): General Form of a Measure Function

Extended Terms. In order to capture non-polynomial worst-case complexity of recursive programs, our algorithm incorporates two types of extensions of terms.

1. *Logarithmic Terms.* The first extension, which we call log-extension, is the extension with terms from $\ln x, \ln(x-y+1)$ where x, y are scalar variables appearing in the parameter list of some function name and $\ln(\cdot)$ refers to the natural logarithm function with base e. Our algorithm will take this extension when op is log.
2. *Exponentiation Terms.* The second extension, which we call exp-extension, is with terms from $x^r, (x-y+1)^r$ where x, y are scalar variables appearing in the parameter list of some function name. The algorithm takes this when op = exp.

The intuition is that x (resp. $x-y+1$) may represent a positive quantity to be halved iteratively (resp. the length between array indexes y and x).

General Form. The general form for any coordinate function $\eta(\mathsf{f}, \ell, \cdot)$ of a measure function η (at function name f and $\ell \in L^{\mathsf{f}}_{\mathrm{s}}$) is a finite sum

$$\mathfrak{e} = \textstyle\sum_i c_i \cdot g_i \tag{1}$$

where (i) each c_i is a constant scalar and each g_i is a finite product of no more than d terms (i.e., with degree at most d) from scalar variables in V^{f} and logarithmic/exponentiation extensions (depending on op), and (ii) all g_i's correspond to all finite products of no more than d terms. Analogous to arithmetic expressions, for any such finite sum $\mathfrak{e}$ and any valuation $\nu \in \mathit{Val}_{\mathsf{f}}$, we denote by $\mathfrak{e}(\nu)$ the real number evaluated through replacing any scalar variable x appearing in $\mathfrak{e}$ with $\nu(x)$, provided that $\mathfrak{e}(\nu)$ is well-defined.

Semantics of General Form. A finite sum $\mathfrak{e}$ at f and $\ell \in L^{\mathsf{f}}_{\mathrm{s}}$ in the form (1) defines a function $[\![\mathfrak{e}]\!]$ on $\mathit{Val}_{\mathsf{f}}$ in the way that for each $\nu \in \mathit{Val}_{\mathsf{f}}$: $[\![\mathfrak{e}]\!](\nu) := \mathfrak{e}(\nu)$ if $\nu \models I(\mathsf{f}, \ell)$, and $[\![\mathfrak{e}]\!](\nu) := 0$ otherwise. Note that in the definition of $[\![\mathfrak{e}]\!]$, we do not consider the case when log or exponentiation is undefined. However, we will see in Step 1(b) below that log or exponentiation will always be well-defined.

Step 1(b): Templates. As in all previous works (cf. [12,16,20,25,40,58,61,68]), we consider a template for measure function determined by the triple (d, op, r) from the input parameters. Formally, the template determined by (d, op, r) assigns to every function name f and $\ell \in L^{\mathsf{f}}_{\mathrm{s}}$ an expression in the form (1) (with degree d and extension option op). Note that a template here only restricts (i) the degree and (ii) log or exp extension for a measure function, rather than its specific form. In detail, the algorithm sets up a template η for a measure function by assigning to each function name f and significant label $\ell \in L^{\mathsf{f}}_{\mathrm{s}}$ an expression $\eta(\mathsf{f}, \ell)$ in a form similar to (1), except for that c_i's in (1) are interpreted as distinct *template variables* whose actual values are to be synthesized. In order to ensure that logarithm and exponentiation are well-defined over each $I(\mathsf{f}, \ell)$, we impose the following restriction (§) on our template: $\ln x, x^r$ (resp. $\ln(x-y+1), (x-y+1)^r$) appear in $\eta(\mathsf{f}, \ell)$ only when $x-1 \geq 0$ (resp. $x-y \geq 0$) can be inferred from the invariant $I(\mathsf{f}, \ell)$. To infer $x-1 \geq 0$ or $x-y \geq 0$ from $I(\mathsf{f}, \ell)$, we utilize Farkas' Lemma.

Theorem 2 (Farkas' Lemma [28,60]). *Let* $\mathbf{A} \in \mathbb{R}^{m\times n}$, $\mathbf{b} \in \mathbb{R}^m$, $\mathbf{c} \in \mathbb{R}^n$ *and* $d \in \mathbb{R}$. *Assume that* $\{\mathbf{x} \mid \mathbf{A}\mathbf{x} \le \mathbf{b}\} \neq \emptyset$. *Then* $\{\mathbf{x} \mid \mathbf{A}\mathbf{x} \le \mathbf{b}\} \subseteq \{\mathbf{x} \mid \mathbf{c}^{\mathrm{T}}\mathbf{x} \le d\}$ *iff there exists* $\mathbf{y} \in \mathbb{R}^m$ *such that* $\mathbf{y} \ge \mathbf{0}$, $\mathbf{A}^{\mathrm{T}}\mathbf{y} = \mathbf{c}$ *and* $\mathbf{b}^{\mathrm{T}}\mathbf{y} \le d$.

By Farkas' Lemma, there exists an algorithm that infers whether $x - 1 \ge 0$ (or $x - y \ge 0$) holds under $I(\mathsf{f}, \ell)$ in polynomial time through emptiness checking of polyhedra (cf. [59]) since $I(\mathsf{f}, \ell)$ involves only linear (degree-1) polynomials in our setting.

Then η naturally induces a function $[\![\eta]\!]$ from $\{(\mathsf{f}, \ell, \nu) \mid \mathsf{f} \in F, \ell \in L^{\mathsf{f}}_{\mathrm{s}}, \nu \in \mathit{Val}_{\mathsf{f}}\}$ into $[0, \infty]$ parametric over template variables such that $[\![\eta]\!](\mathsf{f}, \ell, \nu) = [\![\eta(\mathsf{f}, \ell)]\!](\nu)$ for all appropriate stack elements (f, ℓ, ν). Note that $[\![\eta]\!]$ is well-defined since logarithm and exponentiation is well-defined over satisfaction sets given by I.

4.2 Step 2 of SynAlgo

Step 2: Computation of $\widehat{[\![\eta]\!]}$. Let η be the template constructed from Step 1. This step computes $\widehat{[\![\eta]\!]}$ from η by the expansion construction of significant labels (Sect. 3) which transforms a function g into $\widehat{g}$. Recall the function $[\![\mathfrak{e}]\!]$ for $\mathfrak{e}$ is defined in Step 1(a). Formally, based on the template η from Step 1, the algorithm computes $\widehat{[\![\eta]\!]}$, with the exception that template variables appearing in η are treated as undetermined constants. Then $\widehat{[\![\eta]\!]}$ is a function parametric over the template variables in η.

By an easy induction, each $\widehat{[\![\eta]\!]}(\mathsf{f}, \ell, \centerdot)$ can be represented by an expression in the form

$$\max\left\{ \textstyle\sum_j \mathbf{1}_{\phi_{1j}} \cdot h_{1j},\ \ \ldots,\ \ \sum_j \mathbf{1}_{\phi_{mj}} \cdot h_{mj} \right\} \tag{2}$$

where each ϕ_{ij} is a propositional arithmetic predicate over V^{f} such that for each i, $\bigvee_j \phi_{ij}$ is tautology and $\phi_{ij_1} \wedge \phi_{ij_2}$ is unsatisfiable whenever $j_1 \neq j_2$, and each h_{ij} takes the form similar to (1) with the difference that (i) each c_i is either a scalar or a template variable appearing in η and (ii) each g_i is a finite product whose every multiplicand is either some $x \in V^{\mathsf{f}}$, or some $\lfloor \mathfrak{e} \rfloor$ with $\mathfrak{e}$ being an instance of $\langle \mathit{expr} \rangle$, or some $\ln \mathfrak{e}$ (or $\mathfrak{e}^r$, depending on op) with $\mathfrak{e}$ being an instance of $\langle \mathit{expr} \rangle$. For this step we use the fact that all propositional arithmetic predicates can be put in disjunctive normal form.

4.3 Step 3 of SynAlgo

This step generates constraint triples from $\widehat{[\![\eta]\!]}$ computed in Step 2. By applying non-negativity and C2–C5 to $\widehat{[\![\eta]\!]}$ (computed in Step 2), the algorithm establishes constraint triples which will be interpreted as universally-quantified logical formulas later.

Constraint Triples. A *constraint triple* is a triple $(\mathsf{f}, \phi, \mathfrak{e})$ where (i) $\mathsf{f} \in F$, (ii) ϕ is a propositional arithmetic predicate over V^{f} which is a conjunction of atomic formulae of the form $\mathfrak{e}' \ge 0$ with $\mathfrak{e}'$ being an arithmetic expression, and (iii) $\mathfrak{e}$ is an expression taking the form similar to (1) with the difference that

(i) each c_i is either a scalar, or a template variable c appearing in η, or its reverse $-c$, and (ii) each g_i is a finite product whose every multiplicand is either some $x \in V^{\mathsf{f}}$, or some $\lfloor \mathfrak{e} \rfloor$ with $\mathfrak{e}$ being an instance of $\langle expr \rangle$, or some $\ln \mathfrak{e}$ (or $\mathfrak{e}^r$, depending on op) with $\mathfrak{e}$ being an instance of $\langle expr \rangle$. For each constraint triple $(\mathsf{f}, \phi, \mathfrak{e})$, the function $[\![\mathfrak{e}]\!]$ on Val_{f} is defined in the way such that each $[\![\mathfrak{e}]\!](\nu)$ is the evaluation result of $\mathfrak{e}$ when assigning $\nu(x)$ to each $x \in V^{\mathsf{f}}$; under (§) (of Step 1(b)), logarithm and exponentiation will always be well-defined.

Semantics of Constraint Triples. A constraint triple $(\mathsf{f}, \phi, \mathfrak{e})$ encodes the following logical formula: $\forall \nu \in Val_{\mathsf{f}}. (\nu \models \phi \rightarrow [\![\mathfrak{e}]\!](\nu) \geq 0)$. Multiple constraint triples are grouped into a single logical formula through conjunction.

Step 3: Establishment of Constraint Triples. Based on $\widehat{[\![\eta]\!]}$ (computed in the previous step), the algorithm generates constraint triples at each significant label, then group all generated constraint triples together in a conjunctive way. To be more precise, at every significant label ℓ of some function name f, the algorithm generates constraint triples through non-negativity of measure functions and conditions C2–C5; after generating the constraint triples for each significant label, the algorithm groups them together in the conjunctive fashion to form a single collection of constraint triples.

Example 3. Consider our running example (cf. Example 1). Let the input quadruple be $(1, \log, -, 1)$ and invariant (at label 1) be $n \geq 1$ (length of array should be positive). In Step 1, the algorithm assigns the template $\eta(\mathsf{f}, 1, n) = c_1 \cdot n + c_2 \cdot \ln n + c_3$ at label 1 and $\eta(\mathsf{f}, 4, n) = 0$ at label 4. In Step 2, the algorithm computes template at other labels and obtains that $\eta(\mathsf{f}, 2, n) = 1 + c_1 \cdot \lfloor n/2 \rfloor + c_2 \cdot \ln \lfloor n/2 \rfloor + c_3$ and $\eta(\mathsf{f}, 3, n) = 1$. In Step 3, the algorithm establishes the following three constraint triples $\mathfrak{q}_1, \mathfrak{q}_2, \mathfrak{q}_3$:

- $\mathfrak{q}_1 := (\mathsf{f}, n - 1 \geq 0, c_1 \cdot n + c_2 \cdot \ln n + c_3)$ from the logical formula $\forall n.(n \geq 1) \rightarrow c_1 \cdot n + c_2 \cdot \ln n + c_3 \geq 0$ for non-negativity of measure functions;
- $\mathfrak{q}_2 := (\mathsf{f}, n - 1 \geq 0 \wedge 1 - n \geq 0, c_1 \cdot n + c_2 \cdot \ln n + c_3 - 2)$ and $\mathfrak{q}_3 := (\mathsf{f}, n - 2 \geq 0, c_1 \cdot (n - \lfloor n/2 \rfloor) + c_2 \cdot (\ln n - \ln \lfloor n/2 \rfloor) - 2$ from resp. logical formulae
  - $\forall n.(n \geq 1 \wedge n \leq 1) \rightarrow c_1 \cdot n + c_2 \cdot \ln n + c_3 \geq 2$ and
  - $\forall n.(n \geq 2) \rightarrow c_1 \cdot n + c_2 \cdot \ln n + c_3 \geq c_1 \cdot \lfloor n/2 \rfloor + c_2 \cdot \ln \lfloor n/2 \rfloor + c_3 + 2$

  for C4 (at label 1). □

4.4 Step 4 of SynAlgo

Step 4: Solving Constraint Triples. To check whether the logical formula encoded by the generated constraint triples is valid, the algorithm follows a sound method which abstracts each multiplicand other than scalar variables in the form (2) as a stand-alone variable, and transforms the validity of the formula into a system of linear inequalities over template variables appearing in η through Handelman's Theorem and linear programming. The main idea is that the algorithm establishes tight linear inequalities for those abstraction variables by investigating properties for the abstracted arithmetic expressions, and use

linear programming to solve the formula based on the linear inequalities for abstraction variables. We note that validity of such logical formulae are generally undecidable since they involve non-polynomial terms such as logarithm [33].

Below we describe how the algorithm transforms a constraint triple into one without logarithmic or exponentiation term. Given any finite set Γ of polynomials over n variables, we define $\mathsf{Sat}(\Gamma) := \{\boldsymbol{x} \in \mathbb{R}^n \mid h(\boldsymbol{x}) \geq 0 \text{ for all } h \in \Gamma\}$. In the whole step, we let $(\mathsf{f}, \phi, \mathfrak{e}^*)$ be any constraint triple such that $\phi = \bigwedge_j \mathfrak{e}_j \geq 0$; moreover, we maintain a finite set Γ of linear (degree-1) polynomials over scalar and freshly-added variables. Intuitively, Γ is related to both the set of all $\mathfrak{e}_j$'s (so that $\mathsf{Sat}(\Gamma)$ is somehow the satisfaction set of ϕ) and the finite subset of polynomials in Theorem 3. Due to lack of space, we only illustrate the part of the algorithm for logarithmic terms (i.e., the case when op = log); exponentiation terms can be treated in a similar fashion.

Step 4(a): Abstraction of Logarithmic, Exponentiation, and Floored Expressions. The first sub-step involves the following computational steps, where Items 2–4 handle variables for abstraction, and Item 6 is approximation of floored expressions, and other steps are straightforward.

1. *Initialization.* First, the algorithm maintains a finite set of linear (degree-1) polynomials Γ and sets it initially to the empty set.
2. *Logarithmic and Floored Expressions.* Next, the algorithm computes the following subsets of $\langle expr \rangle$:
   - $\mathcal{E}_L := \{\mathfrak{e} \mid \ln \mathfrak{e} \text{ appears in } \mathfrak{e}^* \text{ (as sub-expression)}\}$ upon op = log .
   - $\mathcal{E}_F := \{\mathfrak{e} \mid \mathfrak{e} \text{ appears in } \mathfrak{e}^* \text{ and takes the form } \lfloor \frac{\bullet}{c} \rfloor\}$.

   Let $\mathcal{E} := \mathcal{E}_L \cup \mathcal{E}_F$.
3. *Variables for Logarithmic and Floored Expressions.* Next, for each $\mathfrak{e} \in \mathcal{E}$, the algorithm establishes fresh variables as follows:
   - a fresh variable $u_{\mathfrak{e}}$ which represents $\ln \mathfrak{e}$ for $\mathfrak{e} \in \mathcal{E}_L$;
   - a fresh variable $w_{\mathfrak{e}}$ indicating $\mathfrak{e}$ for $\mathfrak{e} \in \mathcal{E}_F$.

   After this step, the algorithm sets N to be the number of all variables (i.e., all scalar variables and all fresh variables added up to this point). In the rest of this section, we consider an implicit linear order over all scalar and freshly-added variables so that a valuation of these variables can be treated as a vector in $\mathbb{R}^N$.
4. *Variable Substitution (from $\mathfrak{e}$ to $\widetilde{\mathfrak{e}}$).* Next, for each $\mathfrak{e}$ which is either $\mathfrak{t}$ or some $\mathfrak{e}_j$ or some expression in $\mathcal{E}$, the algorithm computes $\tilde{\mathfrak{e}}$ as the expression obtained from $\mathfrak{e}$ by substituting (i) every possible $u_{\mathfrak{e}'}$ for $\ln \mathfrak{e}'$, and (ii) every possible $w_{\mathfrak{e}'}$ for $\mathfrak{e}'$ such that $\mathfrak{e}'$ is a sub-expression of $\mathfrak{e}$ which does not appear as sub-expression in some other sub-expression $\mathfrak{e}'' \in \mathcal{E}_F$ of $\mathfrak{e}$. From now on, any $\mathfrak{e}$ or $\widetilde{\mathfrak{e}}$ or is treated as a polynomial over scalar and freshly-added variables. Then any $\mathfrak{e}(\boldsymbol{x})$ or $\widetilde{\mathfrak{e}}(\boldsymbol{x})$ is the result of polynomial evaluation under the correspondence between variables and coordinates of $\boldsymbol{x}$ specified by the linear order.
5. *Importing ϕ into Γ.* The algorithm adds all $\widetilde{\mathfrak{e}_j}$ into Γ.
6. *Approximation of Floored Expressions.* For each $\mathfrak{e} \in \mathcal{E}_F$ such that $\mathfrak{e} = \lfloor \frac{\mathfrak{e}'}{c} \rfloor$, the algorithm adds linear constraints for $w_{\mathfrak{e}}$ recursively on the nesting depth of floor operation as follows.

- *Base Step.* If $\mathfrak{e} = \lfloor \frac{\mathfrak{e}'}{c} \rfloor$ and $\mathfrak{e}'$ involves no nested floored expression, then the algorithm adds into Γ either (i) $\widetilde{\mathfrak{e}'} - c \cdot w_{\mathfrak{e}}$ and $c \cdot w_{\mathfrak{e}} - \widetilde{\mathfrak{e}'} + c - 1$ when $c \geq 1$, which is derived from $\frac{\mathfrak{e}'}{c} - \frac{c-1}{c} \leq \mathfrak{e} \leq \frac{\mathfrak{e}'}{c}$, or (ii) $c \cdot w_{\mathfrak{e}} - \widetilde{\mathfrak{e}'}$ and $\widetilde{\mathfrak{e}'} - c \cdot w_{\mathfrak{e}} - c - 1$ when $c \leq -1$, which follows from $\frac{\mathfrak{e}'}{c} - \frac{c+1}{c} \leq \mathfrak{e} \leq \frac{\mathfrak{e}'}{c}$. Second, given the current Γ, the algorithm finds the largest constant $t_{\mathfrak{e}'}$ through Farkas' Lemma such that $\forall \boldsymbol{x} \in \mathbb{R}^N. \left(\boldsymbol{x} \in \mathsf{Sat}(\Gamma) \rightarrow \widetilde{\mathfrak{e}'}(\boldsymbol{x}) \geq t_{\mathfrak{e}'}\right)$ holds; if such $t_{\mathfrak{e}'}$ exists, the algorithm adds the constraint $w_{\mathfrak{e}} \geq \left\lfloor \frac{t_{\mathfrak{e}'}}{c} \right\rfloor$ into Γ.
- *Recursive Step.* If $\mathfrak{e} = \lfloor \frac{\mathfrak{e}'}{c} \rfloor$ and $\mathfrak{e}'$ involves some nested floored expression, then the algorithm proceeds almost in the same way as for the Base Step, except that $\widetilde{\mathfrak{e}'}$ takes the role of $\mathfrak{e}'$. (Note that $\widetilde{\mathfrak{e}'}$ does not involve nested floored expresions.)

7. *Emptiness Checking.* The algorithm checks whether $\mathsf{Sat}(\Gamma)$ is empty or not in polynomial time in the size of Γ (cf. [59]). If $\mathsf{Sat}(\Gamma) = \emptyset$, then the algorithm discards this constraint triple with no linear inequalities generated, and proceeds to other constraint triples; otherwise, the algorithm proceeds to the remaining steps.

Example 4. We continue with Example 3. In Step 4(a), the algorithm first establishes fresh variables $u := \ln n$, $v := \ln \lfloor n/2 \rfloor$ and $w := \lfloor n/2 \rfloor$, then finds that (i) $n - 2 \cdot w \geq 0$, (ii) $2 \cdot w - n + 1 \geq 0$ and (iii) $n - 2 \geq 0$ (as Γ) implies that $w - 1 \geq 0$. After Step 4(a), the constraint triples after variable substitution and their Γ's are as follows:

- $\widetilde{\mathfrak{q}_1} = (\mathsf{f}, n - 1 \geq 0, c_1 \cdot n + c_2 \cdot u + c_3)$ and $\Gamma_1 = \{n - 1\}$;
- $\widetilde{\mathfrak{q}_2} = (\mathsf{f}, n - 1 \geq 0 \wedge 1 - n \geq 0, c_1 \cdot n + c_2 \cdot u + c_3 - 2)$ and $\Gamma_2 = \{n - 1, 1 - n\}$;
- $\widetilde{\mathfrak{q}_3} := (\mathsf{f}, n - 2 \geq 0, c_1 \cdot (n - w) + c_2 \cdot (u - v) - 2)$ and $\Gamma_3 = \{n - 2, n - 2 \cdot w, 2 \cdot w - n + 1, w - 1\}$. □

For the next sub-step we will use Lagrange's Mean-Value Theorem (LMVT) [6, Chap. 6] to approximate logarithmic and exponentiation terms.

Step 4(b): Linear Constraints for Abstracted Variables. The second sub-step consists of the following computational steps which establish into Γ linear constraints for logarithmic or exponentiation terms. Below we denote by $\mathcal{E}'$ either the set $\mathcal{E}_L$ when op = log or $\mathcal{E}_E$ when op = exp. Recall the $\widetilde{\mathfrak{e}}$ notation is defined in the Variable Substitution (Item 4) of Step 4(a).

1. *Lower-Bound for Expressions in $\mathcal{E}_L$.* For each $\mathfrak{e} \in \mathcal{E}_L$, we find the largest constant $t_{\mathfrak{e}} \in \mathbb{R}$ such that the logical formula $\forall \boldsymbol{x} \in \mathbb{R}^N. (\boldsymbol{x} \in \mathsf{Sat}(\Gamma) \rightarrow \widetilde{\mathfrak{e}}(\boldsymbol{x}) \geq t_{\mathfrak{e}})$ holds, This can be solved by Farkas' Lemma and linear programming, since $\widetilde{\mathfrak{e}}$ is linear. Note that as long as $\mathsf{Sat}(\Gamma) \neq \emptyset$, it follows from (§) (in Step 1(b)) that $t_{\mathfrak{e}}$ is well-defined (since $t_{\mathfrak{e}}$ cannot be arbitrarily large) and $t_{\mathfrak{e}} \geq 1$.
2. *Mutual No-Smaller-Than Inequalities over $\mathcal{E}_L$.* For each pair $(\mathfrak{e}, \mathfrak{e}') \in \mathcal{E}_L \times \mathcal{E}_L$ such that $\mathfrak{e} \neq \mathfrak{e}'$, the algorithm finds real numbers $r_{(\mathfrak{e},\mathfrak{e}')}, b_{(\mathfrak{e},\mathfrak{e}')}$ through Farkas'

Lemma and linear programming such that (i) $r_{(\mathfrak{e},\mathfrak{e}')} \geq 0$ and (ii) both the logical formulae

$$\forall \boldsymbol{x} \in \mathbb{R}^N. \left[\boldsymbol{x} \in \mathsf{Sat}(\Gamma) \to \widetilde{\mathfrak{e}}(\boldsymbol{x}) - \left(r_{\mathfrak{e},\mathfrak{e}'} \cdot \widetilde{\mathfrak{e}'}(\boldsymbol{x}) + b_{\mathfrak{e},\mathfrak{e}'}\right) \geq 0\right] \quad \text{and}$$

$$\forall \boldsymbol{x} \in \mathbb{R}^N. \left[\boldsymbol{x} \in \mathsf{Sat}(\Gamma) \to r_{\mathfrak{e},\mathfrak{e}'} \cdot \widetilde{\mathfrak{e}'}(\boldsymbol{x}) + b_{\mathfrak{e},\mathfrak{e}'} \geq 1\right]$$

hold. The algorithm first finds the maximal value $r^*_{\mathfrak{e},\mathfrak{e}'}$ over all feasible $(r_{\mathfrak{e},\mathfrak{e}'}, b_{\mathfrak{e},\mathfrak{e}'})$'s, then finds the maximal $b^*_{\mathfrak{e},\mathfrak{e}'}$ over all feasible $(r^*_{\mathfrak{e},\mathfrak{e}'}, b_{\mathfrak{e},\mathfrak{e}'})$'s. If such $r^*_{\mathfrak{e},\mathfrak{e}'}$ does not exist, the algorithm simply leaves $r^*_{\mathfrak{e},\mathfrak{e}'}$ undefined. Note that once $r^*_{\mathfrak{e},\mathfrak{e}'}$ exists and $\mathsf{Sat}(\Gamma) \neq \emptyset$, then $b^*_{\mathfrak{e},\mathfrak{e}'}$ exists since $b_{\mathfrak{e},\mathfrak{e}'}$ cannot be arbitrarily large once $r^*_{\mathfrak{e},\mathfrak{e}'}$ is fixed.

3. *Mutual No-Greater-Than Inequalities over $\mathcal{E}_L$.* For each pair $(\mathfrak{e},\mathfrak{e}') \in \mathcal{E}_L \times \mathcal{E}_L$ such that $\mathfrak{e} \neq \mathfrak{e}'$, the algorithm finds real numbers $\mathsf{r}_{(\mathfrak{e},\mathfrak{e}')}, \mathsf{b}_{(\mathfrak{e},\mathfrak{e}')}$ through Farkas' Lemma and linear programming such that (i) $\mathsf{r}_{(\mathfrak{e},\mathfrak{e}')} \geq 0$ and (ii) the logical formula

$$\forall \boldsymbol{x} \in \mathbb{R}^N. \left[\boldsymbol{x} \in \mathsf{Sat}(\Gamma) \to \left(\mathsf{r}_{\mathfrak{e},\mathfrak{e}'} \cdot \widetilde{\mathfrak{e}'}(\boldsymbol{x}) + \mathsf{b}_{\mathfrak{e},\mathfrak{e}'}\right) - \widetilde{\mathfrak{e}}(\boldsymbol{x}) \geq 0\right]$$

holds. The algorithm then finds the minimal value $(\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'},\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'})$ similarly as above.

4. *Constraints from Logarithm.* For each variable $u_{\mathfrak{e}}$, the algorithm adds into Γ first the polynomial expression $\widetilde{\mathfrak{e}} - \left(\mathbf{1}_{t_{\mathfrak{e}} \leq e} \cdot e + \mathbf{1}_{t_{\mathfrak{e}} > e} \cdot \frac{t_{\mathfrak{e}}}{\ln t_{\mathfrak{e}}}\right) \cdot u_{\mathfrak{e}}$ from the fact that the function $z \mapsto \frac{z}{\ln z}$ $(z \geq 1)$ has global minima at e (so that the inclusion of this polynomial expression is sound), and then the polynomial expression $u_{\mathfrak{e}} - \ln t_{\mathfrak{e}}$ due to the definition of $t_{\mathfrak{e}}$.
5. *Mutual No-Smaller-Than Inequalities over $u'_{\mathfrak{e}}s$.* For each pair $(\mathfrak{e},\mathfrak{e}') \in \mathcal{E}_L \times \mathcal{E}_L$ such that $\mathfrak{e} \neq \mathfrak{e}'$ and $r^*_{\mathfrak{e},\mathfrak{e}'}, b^*_{\mathfrak{e},\mathfrak{e}'}$ are successfully found and $r^*_{\mathfrak{e},\mathfrak{e}'} > 0$, the algorithm adds $u_{\mathfrak{e}} - \ln r^*_{\mathfrak{e},\mathfrak{e}'} - u_{\mathfrak{e}'} + \mathbf{1}_{b^*_{\mathfrak{e},\mathfrak{e}'}<0} \cdot \left(t_{\mathfrak{e}'} + \frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}}\right)^{-1} \cdot \left(-\frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}}\right)$ into Γ. This is due to the fact that $[\![\mathfrak{e}]\!] - \left(r^*_{\mathfrak{e},\mathfrak{e}'} \cdot [\![\mathfrak{e}']\!] + b^*_{\mathfrak{e},\mathfrak{e}'}\right) \geq 0$ implies the following:

$$\begin{aligned}
\ln [\![\mathfrak{e}]\!] &\geq \ln r^*_{\mathfrak{e},\mathfrak{e}'} + \ln\left([\![\mathfrak{e}']\!] + (b^*_{\mathfrak{e},\mathfrak{e}'}/r^*_{\mathfrak{e},\mathfrak{e}'})\right) \\
&= \ln r^*_{\mathfrak{e},\mathfrak{e}'} + \ln [\![\mathfrak{e}']\!] + \left(\ln\left([\![\mathfrak{e}']\!] + (b^*_{\mathfrak{e},\mathfrak{e}'}/r^*_{\mathfrak{e},\mathfrak{e}'})\right) - \ln [\![\mathfrak{e}']\!]\right) \\
&\geq \ln r^*_{\mathfrak{e},\mathfrak{e}'} + \ln [\![\mathfrak{e}']\!] - \mathbf{1}_{b^*_{\mathfrak{e},\mathfrak{e}'}<0} \cdot \left(t_{\mathfrak{e}'} + (b^*_{\mathfrak{e},\mathfrak{e}'}/r^*_{\mathfrak{e},\mathfrak{e}'})\right)^{-1} \cdot \left(-b^*_{\mathfrak{e},\mathfrak{e}'}/r^*_{\mathfrak{e},\mathfrak{e}'}\right),
\end{aligned}$$

where the last step is obtained from LMVT and by distinguishing whether $b^*_{\mathfrak{e},\mathfrak{e}'} \geq 0$ or not, using the fact that the derivative of the natural-logarithm is the reciprocal function. Note that one has $t_{\mathfrak{e}'} + \frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}} \geq 1$ due to the maximal choice of $t_{\mathfrak{e}'}$.
6. *Mutual No-Greater-Than Inequalities over $u'_{\mathfrak{e}}s$.* Similar to the previous item, the algorithm establishes mutual no-greater-than inequalities over $u_{\mathfrak{e}}$'s.

Although in Item 4 and Item 6 above, we have logarithmic terms such as $\ln t_{\mathfrak{e}}$ and $\ln r^*_{\mathfrak{e},\mathfrak{e}'}$, both $t_{\mathfrak{e}}$ and $r^*_{\mathfrak{e},\mathfrak{e}'}$ are already determined constants, hence their approximations can be used. After Step 4, the constraint triple $(\mathsf{f}, \phi, \mathfrak{e}^*)$ is transformed into $(\mathsf{f}, \bigwedge_{h \in \Gamma} h \geq 0, \widetilde{\mathfrak{e}^*})$.

Example 5. We continue with Example 4. In Step 4(b), the algorithm establishes the following non-trivial inequalities:

- (*From Item 2,3 in Step 4(b) for* $\widetilde{\mathfrak{q}_3}$) $w \geq 0.5 \cdot n - 0.5, w \leq 0.5 \cdot n$ and $n \geq 2 \cdot w, n \leq 2 \cdot w + 1$;
- (*From Item 4 in Step 4(b) for* $\widetilde{\mathfrak{q}_1}, \widetilde{\mathfrak{q}_2}$) $n - e \cdot u \geq 0$ and $u \geq 0$;
- (*From Item 4 in Step 4(b) for* $\widetilde{\mathfrak{q}_3}$) $n - e \cdot u \geq 0, u - \ln 2 \geq 0$ and $w - e \cdot v \geq 0, v \geq 0$;
- (*From Item 6,7 in Step 4(b) for* $\widetilde{\mathfrak{q}_3}$) $u - v - \ln 2 \geq 0$ and $v - u + \ln 2 + \frac{1}{2} \geq 0$.

After Step 4(b), Γ_i's $(1 \leq i \leq 3)$ are updated as follows:

- $\Gamma_1 = \{n - 1, n - e \cdot u, u\}$ and $\Gamma_2 = \{n - 1, 1 - n, n - e \cdot u, u\}$;
- $\Gamma_3 = \{n - 2, n - 2 \cdot w, 2 \cdot w - n + 1, w - 1, n - e \cdot u, u - \ln 2, w - e \cdot v, v, u - v - \ln 2, v - u + \ln 2 + \frac{1}{2}\}$. □

Remark 1. The key difficulty is to handle logarithmic and exponentiation terms. In Step 4(a) we abstract such terms with fresh variables and perform sound approximation of floored expressions. In Step 4(b) we use Farkas' Lemma and LMVT to soundly transform logarithmic or exponentiation terms to polynomials. □

4.5 Step 5 of SynAlgo

This step is to solve the template variables in the template established in Step 1, based on the sets Γ computed in Step 4. While Step 4 transforms logarithmic and exponentiation terms to polynomials, we need a sound method to solve polynomials with linear programming. We achieve this with Handelman's Theorem.

Definition 6 (Monoid). *Let Γ be a finite subset of some polynomial ring $\mathfrak{R}[x_1, \ldots, x_m]$ such that all elements of Γ are polynomials of degree* 1. *The* monoid *of Γ is defined by: Monoid(Γ) $:= \left\{\prod_{i=1}^{k} h_i \mid k \in \mathbb{N}_0 \text{ and } h_1, \ldots, h_k \in \Gamma\right\}$.*

Theorem 3 (Handelman's Theorem [38]). *Let $\mathfrak{R}[x_1, \ldots, x_m]$ be the polynomial ring with variables $x_1, \ldots, x_m$ (for $m \geq 1$). Let $g \in \mathfrak{R}[x_1, \ldots, x_m]$ and Γ be a finite subset of $\mathfrak{R}[x_1, \ldots, x_m]$ such that all elements of Γ are polynomials of degree* 1. *If (i) the set $\mathsf{Sat}(\Gamma)$ is compact and non-empty and (ii) $g(\boldsymbol{x}) > 0$ for all $\boldsymbol{x} \in \mathsf{Sat}(\Gamma)$, then*

$$g = \textstyle\sum_{i=1}^{n} c_i \cdot u_i \tag{3}$$

for some $n \in \mathbb{N}$, non-negative real numbers $c_1, \ldots, c_n \geq 0$ and $u_1, \ldots, u_n \in$ Monoid(Γ).

Basically, Handelman's Theorem gives a characterization of positive polynomials over polytopes. In this paper, we concentrate on Eq. (3) which provides a sound form for a non-negative polynomial over a general (i.e. possibly unbounded) polyhedron.

Step 5: Solving Unknown Coefficients in the Template. Now we use the input parameter k as the maximal number of multiplicands in each summand at the right-hand-side of Eq. (3). For any constraint triple $(\mathsf{f}, \phi, \mathfrak{e}^*)$ which is generated in Step 3 and passes the emptiness checking in Item 7 of Step 4(a), the algorithm performs the following steps.

1. *Preparation for Eq. (3).* The algorithm reads the set Γ for $(\mathsf{f}, \phi, \mathfrak{e}^*)$ computed in Step 4, and computes $\widetilde{\mathfrak{e}^*}$ from Item 4 of Step 4(a).
2. *Application of Handelman's Theorem.* First, the algorithm establishes a fresh coefficient variable λ_h for each polynomial h in Monoid(Γ) with no more than k multiplicands from Γ. Then, the algorithm establishes linear equalities over coefficient variables λ_h's and template variables in the template η established in Step 1 by equating coefficients of the same monomials at the left- and right-hand-side of the following polynomial equality $\widetilde{\mathfrak{e}^*} = \sum_h \lambda_h \cdot h$. Second, the algorithm incorporates all constraints of the form $\lambda_h \geq 0$.

Then the algorithm collects all linear equalities and inequalities established in Item 2 above conjunctively as a single system of linear inequalities and solves it through linear-programming algorithms; if no feasible solution exists, the algorithm fails without output, otherwise the algorithm outputs the function $\widehat{[\![\eta]\!]}$ where all template variables in the template η are resolved by their values in the solution. We now state the soundness of our approach for synthesis of measure functions.

Theorem 4. *Our algorithm,* SynAlgo, *is a sound approach for the* RecTermBou *problem, i.e., if* SynAlgo *succeeds to synthesize a function g on $\{(\mathsf{f}, \ell, \nu) \mid \mathsf{f} \in F, \ell \in L^{\mathsf{f}}_{\mathrm{s}}, \nu \in \mathit{Val}_{\mathsf{f}}\}$, then $\widehat{g}$ is a measure function and hence an upper bound on the termination-time function.*

Example 6. Continue with Example 5. In the final step (Step 5), the unknown coefficients c_i's ($1 \leq i \leq 3$) are to be resolved through (3) so that logical formulae encoded by $\widetilde{\mathfrak{q}_i}$'s are valid (w.r.t updated Γ_i's). Since to present the whole technical detail would be too cumbersome, we present directly a feasible solution for c_i's and how they fulfill (3). Below we choose the solution that $c_1 = 0$, $c_2 = \frac{2}{\ln 2}$ and $c_3 = 2$. Then we have that

- (From $\widetilde{\mathfrak{q}_1}$) $c_2 \cdot u + c_3 = \lambda_1 \cdot u + \lambda_2$ where $\lambda_1 := \frac{2}{\ln 2}$ and $\lambda_2 := 2$;
- (From $\widetilde{\mathfrak{q}_2}$) $c_2 \cdot u + c_3 - 2 = \lambda_1 \cdot u$;
- (From $\widetilde{\mathfrak{q}_3}$) $c_2 \cdot (u - v) - 2 = \lambda_1 \cdot (u - v - \ln 2)$.

Hence by Theorem 1, $\overline{T}(\mathsf{f}, 1, n) \leq \eta(\mathsf{f}, 1, n) = \frac{2}{\ln 2} \cdot \ln n + 2$. It follows that BinarySearch runs in $\mathcal{O}(\log n)$ in worst-case. □

Remark 2. We remark two aspects of our algorithm. (i) *Scalability.* Our algorithm only requires solving linear inequalities. Since linear-programming solvers have been widely studied and experimented, the scalability of our approach directly depends on the linear-programming solvers. Hence the approach we present is a relatively scalable one. (ii) *Novelty.* A key novelty of our approach

is to obtain non-polynomial bounds (such as $\mathcal{O}(n \log n)$, $\mathcal{O}(n^r)$, where r is not integral) through linear programming. The novel technical steps are: (a) use of abstraction variables; (b) use of LMVT and Farkas' lemma to obtain sound linear constraints over abstracted variables; and (c) use of Handelman's Theorem to solve the unknown coefficients in polynomial time. □

5 Experimental Results

We apply our approach to four classical recursive algorithms, namely Merge-Sort [24, Chap. 2], the divide-and-conquer algorithm for the Closest-Pair problem [24, Chap. 33], Strassen's Algorithm for matrix multiplication [24, Chap. 4] and Karatsuba's Algorithm for polynomial multiplication [52]. We implement our algorithm that basically generates a set of linear constraints, where we use lp_solve [56] for solving linear programs. Our experimental results are presented in Table 1, where all numbers are rounded to 10^{-2} and n represents the input length of those algorithms. All results were obtained on an Intel i3-4130 CPU 3.4 GHz 8 GB of RAM. For Merge-Sort and Closest-Pair, our algorithm obtains the worst-case bound $\mathcal{O}(n \log n)$. For Strassen's Algorithm, we use a template with $n^{2.9}$ so that our algorithm synthesizes a measure function, hence proving that its worst-case behaviour is no greater than $\mathcal{O}(n^{2.9})$, near the worst-case complexity $\mathcal{O}(n^{2.8074})$. For Karatsuba's Algorithm, we use a template with $n^{1.6}$ so that our algorithm synthesizes a measure function (basically, using constraints as illustrated in Example 2), hence proving that $\mathcal{O}(n^{1.6})$ is an upper bound on its worst-case behaviour, which is near the worst-case complexity $\mathcal{O}(n^{1.5850})$. In the experiments we derive simple invariants from the programs directly from the prerequisites of procedures and guards of while-loops. Alternatively, they can be derived automatically using [19].

Table 1. Experimental results where $\eta(\ell_0)$ is the part of measure function at the initial label.

Example	Time (in seconds)	$\eta(\ell_0)$
Merge-Sort	6	$25.02 \cdot n \cdot \ln n + 21.68 \cdot n - 20.68$
Closest-Pair	11	$128.85 \cdot n \cdot \ln n + 108.95 \cdot n - 53.31$
Karatsuba	3	$2261.55 \cdot n^{1.6} + 1$
Strassen	7	$954.20 \cdot n^{2.9} + 1$

6 Related Work

The termination of recursive programs or other temporal properties has already been extensively studied [5,21–23,27,53–55,66]. Our work is most closely related to automatic amortized analysis [4,32,40,42–46,50,51,63], as well as the SPEED

project [35–37]. There are two key differences of our methods as compared to previous works. First, our methods are based on extension of ranking functions to non-deterministic recursive programs, whereas previous works either use potential functions, abstract interpretation, or size-change. Second, while none of the previous methods can derive non-polynomial bounds such as $\mathcal{O}(n^r)$, where r is not an integer, our approach can derive such non-polynomial bounds through linear programming.

Ranking functions for intra-procedural analysis have been widely studied [8,9,20,25,58,61,64,68], and have been extended to ranking supermartingales [12–14,16,17,29] for probabilistic programs. Most works focus on linear or polynomial ranking functions/supermartingales [12,14,16,20,25,58,61,64,68]. Polynomial ranking functions alone can only derive polynomial bounds, and needs additional structures (e.g., evaluation trees) to derive non-polynomial bounds such as $\mathcal{O}(2^n)$ (cf. [10]). In contrast, we directly synthesize non-polynomial ranking functions without additional structures. The approach of recurrence relations for worst-case analysis is explored in [1–3,30,34]. A related result is by Albert *et al.* [2] who considered using evaluation trees for solving recurrence relations, which can derive the worst-case bound for Merge-Sort. Their method relies on specific features such as branching factor and height of an evaluation tree, and cannot derive bounds like $\mathcal{O}(n^r)$ where r is not an integer. Another approach through theorem proving is explored in [65]. This approach is to iteratively generate control-flow paths and then to obtain worst-case bounds over generated paths through theorem proving (with arithmetic theorems). Several other works present proof rules for deterministic programs [39] as well as for probabilistic programs [49,57]. None of these works can be automated. Other related approaches are sized types [18,47,48], and polynomial resource bounds [62]. Again none of these approaches can yield bounds like $\mathcal{O}(n \log n)$ or $\mathcal{O}(n^r)$, for r non-integral.

7 Conclusion

In this paper, we developed a ranking-function based approach to obtain non-polynomial worst-case bounds for recursive programs through (i) abstraction of logarithmic and exponentiation terms and (ii) Farkas' Lemma, LMVT, and Handelman's Theorem. Moreover our approach obtains such bounds using linear programming, thus is an efficient approach. Our approach obtains non-trivial worst-case complexity bounds for classical recursive programs: $\mathcal{O}(n \log n)$-complexity for both Merge-Sort and the divide-and-conquer Closest-Pair algorithm, $\mathcal{O}(n^{1.6})$ for Karatsuba's algorithm for polynomial multiplication, and $\mathcal{O}(n^{2.9})$ for Strassen's algorithm for matrix multiplication. The bounds we obtain for Karatsuba's and Strassen's algorithm are close to the optimal bounds. An interesting future direction is to extend our technique to data-structures. Other future directions include investigating the application of our approach to invariant generation and using integer linear programming instead in our approach.

Acknowledgements. We thank all reviewers for valuable comments. The research is partially supported by Vienna Science and Technology Fund (WWTF) ICT15-003, Austrian Science Fund (FWF) NFN Grant No. S11407-N23 (RiSE/SHiNE), ERC Start grant (279307: Graph Games), the Natural Science Foundation of China (NSFC) under Grant No. 61532019 and the CDZ project CAP (GZ 1023).

References

1. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G., Ramírez-Deantes, D.V., Román-Díez, G., Zanardini, D.: Termination and cost analysis with COSTA and its user interfaces. Electr. Notes Theor. Comput. Sci. **258**(1), 109–121 (2009)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic inference of upper bounds for recurrence relations in cost analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008). doi:10.1007/978-3-540-69166-2_15
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java bytecode. In: Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71316-6_12
4. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15769-1_8
5. Alur, R., Chaudhuri, S.: Temporal reasoning for procedural programs. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 45–60. Springer, Heidelberg (2010). doi:10.1007/978-3-642-11319-2_7
6. Bartle, R.G., Sherbert, D.R.: Introduction to Real Analysis, 4th edn. Wiley, Hoboken (2011)
7. Bodík, R., Majumdar, R. (eds.): POPL. ACM, New York (2016)
8. Bournez, O., Garnier, F.: Proving positive almost-sure termination. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 323–337. Springer, Heidelberg (2005). doi:10.1007/978-3-540-32033-3_24
9. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005). doi:10.1007/11513988_48
10. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing runtime and size complexity of integer programs. ACM Trans. Program. Lang. Syst. **38**(4), 13:1–13:50 (2016)
11. Castagna, G., Gordon, A.D. (eds.): POPL. ACM, New York (2017)
12. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39799-8_34
13. Chatterjee, K., Fu, H.: Termination of nondeterministic recursive probabilistic programs. CoRR abs/1701.02944 (2017). http://arxiv.org/abs/1701.02944
14. Chatterjee, K., Fu, H., Goharshady, A.K.: Termination analysis of probabilistic programs through Positivstellensatzs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 3–22. Springer, Cham (2016). doi:10.1007/978-3-319-41528-4_1

15. Chatterjee, K., Fu, H., Goharshady, A.K.: Non-polynomial worst-case analysis of recursive programs. CoRR abs/1705.00317 (2017). https://arxiv.org/abs/1705.00317
16. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: Bodík and Majumdar [7], pp. 327–342
17. Chatterjee, K., Novotný, P., Žikelić, Đ.: Stochastic invariants for probabilistic termination. In: Castagna and Gordon [11], pp. 145–160
18. Chin, W., Khoo, S.: Calculating sized types. Higher-Order Symbolic Comput. **14**(2–3), 261–300 (2001)
19. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt, W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45069-6_39
20. Colón, M.A., Sipma, H.B.: Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001). doi:10.1007/3-540-45319-9_6
21. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Schwartzbach, M.I., Ball, T. (eds.) PLDI, pp. 415–426. ACM (2006)
22. Cook, B., Podelski, A., Rybalchenko, A.: Summarization for termination: no return!. Form. Methods Syst. Des. **35**(3), 369–387 (2009)
23. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 47–61. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36742-7_4
24. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
25. Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 1–24. Springer, Heidelberg (2005). doi:10.1007/978-3-540-30579-8_1
26. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) POPL, pp. 238–252. ACM (1977)
27. Cousot, P., Cousot, R.: An abstract interpretation framework for termination. In: Field, J., Hicks, M. (eds.) POPL, pp. 245–258. ACM (2012)
28. Farkas, J.: A fourier-féle mechanikai elv alkalmazásai (Hungarian). Mathematikaiés Természettudományi Értesitö **12**, 457–472 (1894)
29. Fioriti, L.M.F., Hermanns, H.: Probabilistic termination: soundness, completeness, and compositionality. In: Rajamani, S.K., Walker, D. (eds.) POPL, pp. 489–501. ACM (2015)
30. Flajolet, P., Salvy, B., Zimmermann, P.: Automatic average-case analysis of algorithm. Theor. Comput. Sci. **79**(1), 37–109 (1991)
31. Floyd, R.W.: Assigning meanings to programs. Math. Aspects Comput. Sci. **19**, 19–33 (1967)
32. Gimenez, S., Moser, G.: The complexity of interaction. In: Bodík and Majumdar [7], pp. 243–255
33. Gödel, K., Kleene, S.C., Rosser, J.B.: On undecidable propositions of formal mathematical systems. Institute for Advanced Study Princeton, NJ (1934)
34. Grobauer, B.: Cost recurrences for DML programs. In: Pierce, B.C. (ed.) ICFP, pp. 253–264. ACM (2001)

35. Gulavani, B.S., Gulwani, S.: A numerical abstract domain based on *expression abstraction* and *max operator* with application in timing analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 370–384. Springer, Heidelberg (2008). doi:10.1007/978-3-540-70545-1_35
36. Gulwani, S.: SPEED: symbolic complexity bound analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 51–62. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02658-4_7
37. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: precise and efficient static estimation of program computational complexity. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 127–139. ACM (2009)
38. Handelman, D.: Representing polynomials by positive linear functions on compact convex polyhedra. Pacific J. Math. **132**, 35–62 (1988)
39. Hesselink, W.H.: Proof rules for recursive procedures. Formal Asp. Comput. **5**(6), 554–570 (1993)
40. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. ACM Trans. Program. Lang. Syst. **34**(3), 14 (2012)
41. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 781–786. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31424-7_64
42. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 172–187. Springer, Heidelberg (2010). doi:10.1007/978-3-642-17164-2_13
43. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polynomial potential. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 287–306. Springer, Heidelberg (2010). doi:10.1007/978-3-642-11957-6_16
44. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: Aiken, A., Morrisett, G. (eds.) POPL, pp. 185–197. ACM (2003)
45. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006). doi:10.1007/11693024_3
46. Hofmann, M., Rodriguez, D.: Efficient type-checking for amortised heap-space analysis. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 317–331. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04027-6_24
47. Hughes, J., Pareto, L.: Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In: Rémi, D., Lee, P. (eds.) ICFP. pp. 70–81. ACM (1999)
48. Hughes, J., Pareto, L., Sabry, A.: Proving the correctness of reactive systems using sized types. In: Boehm, H., Jr., G.L.S. (eds.) POPL. pp. 410–423. ACM Press (1996)
49. Jones, C.: Probabilistic non-determinism. Ph.D. thesis, The University of Edinburgh (1989)
50. Jost, S., Hammond, K., Loidl, H., Hofmann, M.: Static determination of quantitative resource usage for higher-order programs. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL, pp. 223–236. ACM (2010)
51. Jost, S., Loidl, H.-W., Hammond, K., Scaife, N., Hofmann, M.: "Carbon Credits" for resource-bounded computations using amortised analysis. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 354–369. Springer, Heidelberg (2009). doi:10.1007/978-3-642-05089-3_23

52. Knuth, D.E.: The Art of Computer Programming, vols. I–III. Addison-Wesley, Reading (1973)
53. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 392–411. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54833-8_21
54. Lee, C.S.: Ranking functions for size-change termination. ACM Trans. Program. Lang. Syst. **31**(3), 10:1–10:42 (2009)
55. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Hankin, C., Schmidt, D. (eds.) POPL, pp. 81–92. ACM (2001)
56. lp_solve 5.5.2.3 (2016). http://lpsolve.sourceforge.net/5.5/
57. Olmedo, F., Kaminski, B.L., Katoen, J., Matheja, C.: Reasoning about recursive probabilistic programs. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) LICS, pp. 672–681. ACM (2016)
58. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24622-0_20
59. Schrijver, A.: Theory of Linear and Integer Programming. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, Hoboken (1999)
60. Schrijver, A.: Combinatorial Optimization - Polyhedra and Efficiency. Springer, Heidelberg (2003)
61. Shen, L., Wu, M., Yang, Z., Zeng, Z.: Generating exact nonlinear ranking functions by symbolic-numeric hybrid method. J. Syst. Sci. Complex. **26**(2), 291–301 (2013)
62. Shkaravska, O., Kesteren, R., Eekelen, M.: Polynomial size analysis of first-order functions. In: Rocca, S.R. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 351–365. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73228-0_25
63. Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 745–761. Springer, Cham (2014). doi:10.1007/978-3-319-08867-9_50
64. Sohn, K., Gelder, A.V.: Termination detection in logic programs using argument sizes. In: Rosenkrantz, D.J. (ed.) PODS, pp. 216–226. ACM Press (1991)
65. Srikanth, A., Sahin, B., Harris, W.R.: Complexity verification using guided theorem enumeration. In: Castagna and Gordon [11], pp. 639–652
66. Urban, C.: The abstract domain of segmented ranking functions. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 43–62. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38856-9_5
67. Wilhelm, R., et al.: The worst-case execution-time problem - overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 1–53 (2008)
68. Yang, L., Zhou, C., Zhan, N., Xia, B.: Recent advances in program verification through computer algebra. Front. Comput. Sci. China **4**(1), 1–16 (2010)