Non-polynomial Worst-Case Analysis of Recursive Programs

KRISHNENDU CHATTERJEE, IST Austria (Institute of Science and Technology Austria), Austria

HONGFEI FU, Shanghai Jiao Tong University, P.R. China

AMIR KAFSHDAR GOHARSHADY, IST Austria, Austria

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. We show that measure functions 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 polynomial worst-case bounds, our approach can synthesize bounds of various forms including $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^r)$ 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 (i) Merge sort, Heap sort and the divide-and-conquer algorithm for the Closest Pair problem, where we obtain $\mathcal{O}(n \log n)$ worst-case bound, and (ii) Karatsuba's algorithm for polynomial multiplication and Strassen's algorithm for matrix multiplication, for which we obtain $\mathcal{O}(n^r)$ bounds such that r is not an integer and is close to the best-known bound for the respective algorithm. Besides the ability to synthesize non-polynomial bounds, we also show that our approach is equally capable of obtaining polynomial worst-case bounds for classical programs such as Quick sort and the dynamic programming algorithm for computing Fibonacci numbers.

CCS Concepts: • **Theory of computation** → **Program verification**.

Additional Key Words and Phrases: Recursive Programs, Worst-Case Analysis

ACM Reference Format:

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2019. Non-polynomial Worst-Case Analysis of Recursive Programs. *ACM Trans. Program. Lang. Syst.* 1, 1 (June 2019), 56 pages. https://doi.org/0000001.0000001

Corresponding Author: Hongfei Fu, fuhf@cs.sjtu.edu.cn
A conference version of this paper appeared in CAV [15].
Authors' addresses: Krishnendu Chatterjee, IST Austria (Institute of Science and Technology Austria), Klosterneuburg, Austria; Hongfei Fu, Shanghai Jiao Tong University, Shanghai, P.R. China; Amir Kafshdar Goharshady, IST Austria, Klosterneuburg, Austria.

0164-0925/2019/6-ART $15.00
https://doi.org/0000001.0000001

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 [54]. 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 code bases. Works such as [38, 39, 42, 43] 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 [71].
(2) *Resource Analysis.* The use of abstract interpretation and type systems to deal with loop, recursion and data-structures has also been considered [1, 39, 52], e.g., using linear invariant generation to obtain disjunctive and non-linear bounds [20], potential-based methods for handling recursion and inductive data structures [42, 43].
(3) *Ranking Functions.* The notion of ranking functions is a powerful technique for termination analysis of (recursive) programs [9, 10, 21, 26, 61, 65, 68, 72]. They serve as a sound and complete approach for proving termination of non-recursive programs [32], and they have also been extended as ranking supermatingales for analysis of probabilistic programs [13, 14, 17, 30].

Despite the many results mentioned 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 [39] (such as $\max(0, n) \cdot (1 + max(n, m))$), or multivariate polynomial analysis [42], 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 by providing 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. Note that the problem of termination

is undecidable in general, hence finding measure functions for arbitrary programs is undecidable, as well.

(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* [25, Chapter 2], *Heap-Sort* [25, Chapter 6], and the divide-and-conquer algorithm for the *Closest-Pair problem* [25, Chapter 33], we obtain $\mathcal{O}(n \log n)$ worst-case bounds. In these cases, the obtained bounds are asymptotically optimal. Note that previous automated methods are either not applicable, or grossly over-estimate the bounds as $\mathcal{O}(n^2)$.
   - For *Karatsuba's algorithm* for polynomial multiplication (cf. [54]) 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. [25, Chapter 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.

Applicability. 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 [54], which states that the number of divisions performed by the Euclidean algorithm is no more than 5 times the length of the smaller number. Besides non-polynomial bounds, our approach can also synthesize polynomial complexity bounds for (recursive) programs such as the dynamic-programming algorithm for computing Fibonacci numbers and Quick-Sort [25, Chapter 7].

Key Novelty. 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 ranking functions and linear programming. Moreover, as our computational tool is linear programming, the approach we provide is also a relatively scalable one (see Remark 12).

This article is an extended version of the conference paper [15]. The extension is as follows: (i) we have explained the technical steps of our algorithm in much more detail and illustrated them with examples, (ii) we have clarified that our approach can also be used to derive polynomial complexity bounds, (iii) we have consolidated the experimental results by more examples, (iv) we have automated the invariant generation by integrating our tool with the Stanford Invariant Generator (Sting) suite, hence making the tool much more user-friendly, and (v) we have provided a more extended comparison with previous works.

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 $\mathfrak{e}(\nu)$ in the straightforward inductive way:

- $c(\nu) := c$;
- $x(\nu) := \nu(x)$;
- $\left\lfloor \frac{\mathfrak{e}}{c} \right\rfloor(\nu) := \left\lfloor \frac{\mathfrak{e}(\nu)}{c} \right\rfloor$; (Note that $c \neq 0$ by our assumption.)
- $(\mathfrak{e} + \mathfrak{e}')(\nu) := \mathfrak{e}(\nu) + \mathfrak{e}'(\nu)$;
- $(\mathfrak{e} - \mathfrak{e}')(\nu) := \mathfrak{e}(\nu) - \mathfrak{e}'(\nu)$;
- $(c * \mathfrak{e}')(\nu) := c \cdot \mathfrak{e}'(\nu)$.

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 ν and propositional arithmetic predicates ϕ is defined in the straightforward way through evaluation of arithmetic expressions:

- $\nu \models \mathfrak{e} \bowtie \mathfrak{e}'$ $(\bowtie \in \{\leq, \geq\})$ if $\mathfrak{e}(\nu) \bowtie \mathfrak{e}'(\nu)$;
- $\nu \models \neg\phi$ iff $\nu \not\models \phi$;
- $\nu \models \phi_1 \wedge \phi_2$ iff $\nu \models \phi_1$ and $\nu \models \phi_2$;
- $\nu \models \phi_1 \vee \phi_2$ iff $\nu \models \phi_1$ or $\nu \models \phi_2$.

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. The syntax is essentially 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.

In the sequel, we fix a countable set of *scalar variables*; and we also fix a countable set of *function names*. W.l.o.g, these two sets are pairwise disjoint. Each scalar variable holds an integer upon instantiation.

The Syntax. The syntax of our recursive programs is illustrated by the grammar in Fig. 1. Below we briefly explain the grammar.

- *Variables:* Expressions $\langle pvar \rangle$ range over scalar variables.
- *Function Names:* Expressions $\langle fname \rangle$ range over function names.
- *Constants:* Expressions $\langle int \rangle$ range over integers represented as decimal numbers, while expressions $\langle nonzero \rangle$ range over non-zero integers represented as decimal numbers.
- *Arithmetic Expressions:* Expressions $\langle expr \rangle$ range over linear arithmetic expressions consisting of scalar variables, floor operation (cf. $\lfloor \langle expr \rangle / \langle nonzero \rangle \rfloor$) and arithmetic operations.
- *Parameters:* Expressions $\langle plist \rangle$ range over lists of scalar variables, and expressions $\langle vlist \rangle$ range over lists of $\langle expr \rangle$ expressions.
- *Boolean Expressions:* Expressions $\langle bexpr \rangle$ range over propositional arithmetic predicates over scalar variables.
- *Statements:* Various types of assignment statements are indicated by ':='; '**skip**' is the statement that does nothing; conditional branch or demonic non-determinism is indicated by the keyword '**if**', while $\langle bexpr \rangle$ indicates conditional branch and $\star$ indicates demonic non-determinism; while-loops are indicated by the keyword '**while**'; sequential compositions are indicated by semicolon; finally, function calls are indicated by $\langle fname \rangle$ ($\langle vlist \rangle$).
- *Programs:* Each recursive program $\langle prog \rangle$ is a sequence of function entities, for which each function entity $\langle func \rangle$ consists of a function name followed by a list of parameters (composing a function declaration) and a curly-braced statement.

Assumptions. W.l.o.g, we consider further syntactical restrictions for simplicity:

- *Function Entities:* we consider that every parameter list $\langle plist \rangle$ contains no duplicate scalar variables, and the function names from function entities are distinct.
- *Function Calls:* we consider that no function call involves some function name without function entity (i.e., undeclared function names).

$$
\begin{aligned}
&\langle prog\rangle ::= \langle func\rangle\langle prog\rangle \mid \langle func\rangle\\
&\langle func\rangle ::= \langle fname\rangle\text{`('}\langle plist\rangle\text{`)'`\{'}\langle stmt\rangle\text{`\}'}\\
&\langle plist\rangle ::= \langle pvar\rangle \mid \langle pvar\rangle\text{`,'}\langle plist\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle stmt\rangle ::= {}& \text{`}\mathbf{skip}\text{'} \mid \langle pvar\rangle\ \text{`:='}\ \langle expr\rangle\\
&\mid \langle fname\rangle\text{`('}\langle vlist\rangle\text{`)'}\\
&\mid \text{`}\mathbf{if}\text{'}\ \langle bexpr\rangle\ \text{`}\mathbf{then}\text{'}\ \langle stmt\rangle\ \text{`}\mathbf{else}\text{'}\ \langle stmt\rangle\ \text{`}\mathbf{fi}\text{'}\\
&\mid \text{`}\mathbf{if}\text{'}\ \star\ \text{`}\mathbf{then}\text{'}\ \langle stmt\rangle\ \text{`}\mathbf{else}\text{'}\ \langle stmt\rangle\ \text{`}\mathbf{fi}\text{'}\\
&\mid \text{`}\mathbf{while}\text{'}\ \langle bexpr\rangle\ \text{`}\mathbf{do}\text{'}\ \langle stmt\rangle\ \text{`}\mathbf{od}\text{'}\\
&\mid \langle stmt\rangle\ \text{`;'}\ \langle stmt\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle expr\rangle ::= {}& \langle int\rangle \mid \langle pvar\rangle\\
&\mid \langle expr\rangle\ \text{`}+\text{'}\ \langle expr\rangle \mid \langle expr\rangle\ \text{`}-\text{'}\ \langle expr\rangle\\
&\mid \langle int\rangle\ \text{`}*\text{'}\ \langle expr\rangle \mid \text{`}\Big\lfloor\text{'}\frac{\langle expr\rangle}{\langle nonzero\rangle}\text{`}\Big\rfloor\text{'},\\
\langle vlist\rangle ::= {}& \langle expr\rangle \mid \langle expr\rangle\text{`,'}\langle vlist\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\langle literal\rangle ::= \langle expr\rangle\ \text{`}\leq\text{'}\ \langle expr\rangle \mid \langle expr\rangle\ \text{`}\geq\text{'}\ \langle expr\rangle\\
&\langle bexpr\rangle ::= \langle literal\rangle \mid \neg\langle bexpr\rangle\\
&\mid \langle bexpr\rangle\ \text{`}\mathbf{or}\text{'}\ \langle bexpr\rangle \mid \langle bexpr\rangle\ \text{`}\mathbf{and}\text{'}\ \langle bexpr\rangle
\end{aligned}
$$

Fig. 1. Syntax of Recursive Programs

Statement Labeling. 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. Informally, 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.

Definition 2.1 (Control-Flow Graphs). A *control-flow graph* (CFG) is a triple which takes the form

$$(\dagger):\ \left(F, \left\{\left(L^{\mathsf{f}}, L^{\mathsf{f}}_{\mathrm{b}}, L^{\mathsf{f}}_{\mathrm{a}}, L^{\mathsf{f}}_{\mathrm{c}}, L^{\mathsf{f}}_{\mathrm{d}}, V^{\mathsf{f}}, \ell^{\mathsf{f}}_{\mathrm{in}}, \ell^{\mathsf{f}}_{\mathrm{out}}\right)\right\}_{\mathsf{f}\in F}, \{\to_{\mathsf{f}}\}_{\mathsf{f}\in F}\right)$$

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 $\to_{\mathsf{f}}$ 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.

Informally, a control-flow graph specifies how values for program variables and the program counter change in a program. We refer to the status of the program counter as a *label*, and assign an initial label and a terminal label to the function body of each function entity. Moreover, we have four types of labels, namely *branching*, *assignment*, *call* and *nondeterministic* labels. A branching label corresponds to a conditional-branching statement indicated by the keyword '**if**' or '**while**' together with some propositional arithmetic predicate ϕ, and leads to the next label in the current function body determined by ϕ without change on values. An assignment label corresponds to an assignment statement indicated by ':=' or **skip**, and leads to the next label right after the statement in the current function body with change of values specified by the update function determined at the right-hand-side of ':=', for which an update function gives the next valuation over program variables based on the current valuation and the sampled values; (**skip** is deemed as an assignment statement that does not change values). A call label corresponds to a function call with some function name g and initial values determined by the value-passing function specified by the call, leads to the label right after the call in the current function body, and does not change values in the original function body. Finally, a nondeterministic label corresponds to a demonic nondeterministic statement indicated by '**if**' and '$\star$', and leads to two labels specified by the '**then**' and the '**else**' branches. It is intuitively clear that every nondeterministic probabilistic recursive program can be equivalently transformed into a CFG.

Below we illustrate an example.

Example 2.2. We consider the running example in Figure 2 which abstracts the running time of Binary-Search. The CFG for this example is depicted in Figure 3. □

Now we establish the transformation from recursive programs to CFGs.

Intuitive Description. We first construct each $\to_{\mathsf{f}}$ (for $\mathsf{f} \in F$) (cf. (†)) for each of its function bodies and then group them together. To construct each $\to_{\mathsf{f}}$, we first construct the partial relation $\to_{P,\mathsf{f}}$ inductively on the structure of P for each statement P appearing in the function body of f, then define $\to_{\mathsf{f}}$ as $\to_{P_{\mathsf{f}},\mathsf{f}}$ for which P_{f} is the function body of f. Each relation $\to_{P,\mathsf{f}}$ involves two distinguished labels, namely $\ell^{P,\mathsf{f}}_{\mathrm{in}}$ and $\ell^{P,\mathsf{f}}_{\mathrm{out}}$, that intuitively represent the label assigned to the first instruction to be executed in P and the terminal program counter of P, respectively; after the inductive construction, $\ell^{\mathsf{f}}_{\mathrm{in}}, \ell^{\mathsf{f}}_{\mathrm{out}}$ are defined as $\ell^{P_{\mathsf{f}},\mathsf{f}}_{\mathrm{in}}, \ell^{P_{\mathsf{f}},\mathsf{f}}_{\mathrm{out}}$, respectively.

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

Fig. 2. A program for Binary-Search

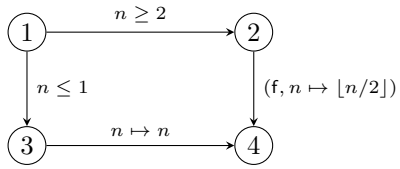

Fig. 3. The CFG for Figure 2

From Programs to CFG's. In this part, we demonstrate inductively how the control-flow graph of a recursive program can be constructed. Below we fix a recursive program W and denote by F the set of function names appearing in W. For each function name $\mathsf{f} \in F$, we define P_{f} to be the function body of f, and define V^{f} to be the set of scalar variables appearing in P_{f} and the parameter list of f.

The control-flow graph of W is constructed by first constructing the counterparts $\{\to_{\mathsf{f}}\}_{\mathsf{f}\in F}$ for each of its function bodies and then grouping them together. To construct each $\to_{\mathsf{f}}$, we first construct the partial relation $\to_{P,\mathsf{f}}$ inductively on the structure of P for each statement P which involves variables solely from V^{f}, then define $\to_{\mathsf{f}}$ as $\to_{P_{\mathsf{f}},\mathsf{f}}$.

Let $\mathsf{f} \in F$. Given an assignment statement of the form $x{:=}\mathfrak{e}$ involving variables solely from V^{f} and a valuation $\nu \in \mathit{Val}_{\mathsf{f}}$, we denote by $\nu[\mathfrak{e}/x]$ the valuation over V^{f} such that

$$\left(\nu[\mathfrak{e}/x]\right)(q) = \begin{cases} \nu(q) & \text{if } q \in V^{\mathsf{f}}\backslash\{x\} \\ \mathfrak{e}(\nu) & \text{if } q = x \end{cases}.$$

Given a function call $\mathsf{g}(\mathfrak{e}_1, \ldots, \mathfrak{e}_k)$ with variables solely from V^{f} and its declaration being $\mathsf{g}(q_1, \ldots, q_k)$, and a valuation $\nu \in \mathit{Val}_{\mathsf{f}}$, we define $\nu[\mathsf{g}, \{\mathfrak{e}_j\}_{1\le j\le k}]$ to be a valuation over V^{g} by:

$$\nu[\mathsf{g}, \{\mathfrak{e}_j\}_{1\le j\le k}](q) := \begin{cases} \mathfrak{e}_j(\nu) & \text{if } q = q_j \text{ for some } j \\ 0 & \text{if } q \in V^{\mathsf{g}}\backslash\{q_1, \ldots, q_k\} \end{cases}.$$

Now the inductive construction for each $\to_{P,\mathsf{f}}$ is demonstrated as follows. For each statement P which involves variables solely from V^{f}, the relation $\to_{P,\mathsf{f}}$ involves two distinguished labels, namely $\ell_{\text{in}}^{P,\mathsf{f}}$ and $\ell_{\text{out}}^{P,\mathsf{f}}$, that intuitively represent the label assigned to the first instruction to be executed in P and the terminal program counter of P, respectively. After the inductive construction, $\ell_{\text{in}}^{\mathsf{f}}, \ell_{\text{out}}^{\mathsf{f}}$ are defined as $\ell_{\text{in}}^{P_{\mathsf{f}},\mathsf{f}}, \ell_{\text{out}}^{P_{\mathsf{f}},\mathsf{f}}$, respectively.

(1) *Assignments.* For P of the form $x{:=}\mathfrak{e}$ or **skip**, $\to_{P,\mathsf{f}}$ involves a new assignment label $\ell_{\text{in}}^{P,\mathsf{f}}$ (as the initial label) and a new branching label $\ell_{\text{out}}^{P,\mathsf{f}}$ (as the terminal label), and contains a sole triple $\left(\ell_{\text{in}}^{P,\mathsf{f}}, \nu \mapsto \nu[\mathfrak{e}/x], \ell_{\text{out}}^{P,\mathsf{f}}\right)$ or $\left(\ell_{\text{in}}^{P,\mathsf{f}}, \nu \mapsto \nu, \ell_{\text{out}}^{P,\mathsf{f}}\right)$, respectively.

(2) *Function Calls.* For P of the form $\mathtt{g}(\mathfrak{e}_1, \ldots, \mathfrak{e}_k)$, $\to_{P,\mathtt{f}}$ involves a new call label $\ell_{\text{in}}^{P,\mathtt{f}}$ and a new branching label $\ell_{\text{out}}^{P,\mathtt{f}}$, and contains a sole triple $\left(\ell_{\text{in}}^{P,\mathtt{f}}, (\mathtt{g}, \nu \mapsto \nu[\mathtt{g}, \{\mathfrak{e}_j\}_{1\leq j\leq k}]), \ell_{\text{out}}^{P,\mathtt{f}}\right)$.
(3) *Sequential Statements.* For $P{=}Q_1; Q_2$, we take the disjoint union of $\to_{Q_1,\mathtt{f}}$ and $\to_{Q_2,\mathtt{f}}$, while redefining $\ell_{\text{out}}^{Q_1,\mathtt{f}}$ to be $\ell_{\text{in}}^{Q_2,\mathtt{f}}$ and putting $\ell_{\text{in}}^{P,\mathtt{f}} := \ell_{\text{in}}^{Q_1,\mathtt{f}}$ and $\ell_{\text{out}}^{P,\mathtt{f}} := \ell_{\text{out}}^{Q_2,\mathtt{f}}$.
(4) *If-Branches.* For $P{=}$**if** ϕ **then** Q_1 **else** Q_2 **fi** with ϕ being a propositional arithmetic predicate, we first add two new branching labels $\ell_{\text{in}}^{P}, \ell_{\text{out}}^{P}$, then take the disjoint union of $\to_{Q_1,\mathtt{f}}$ and $\to_{Q_2,\mathtt{f}}$ while simultaneously identifying both $\ell_{\text{out}}^{Q_1,\mathtt{f}}$ and $\ell_{\text{out}}^{Q_2,\mathtt{f}}$ with $\ell_{\text{out}}^{P,\mathtt{f}}$, and finally obtain $\to_{P,\mathtt{f}}$ by adding two triples $(\ell_{\text{in}}^{P,\mathtt{f}}, \phi, \ell_{\text{in}}^{Q_1,\mathtt{f}})$ and $(\ell_{\text{in}}^{P,\mathtt{f}}, \neg\phi, \ell_{\text{in}}^{Q_2,\mathtt{f}})$ into the disjoint union of $\to_{Q_1,\mathtt{f}}$ and $\to_{Q_2,\mathtt{f}}$.
(5) *While-Loops.* For $P{=}$ **while** ϕ **do** Q **od**, we add a new branching label $\ell_{\text{out}}^{P,\mathtt{f}}$ as a terminal label and obtain $\to_{P,\mathtt{f}}$ by adding triples $(\ell_{\text{out}}^{Q,\mathtt{f}}, \phi, \ell_{\text{in}}^{Q,\mathtt{f}})$ and $(\ell_{\text{out}}^{Q,\mathtt{f}}, \neg\phi, \ell_{\text{out}}^{P,\mathtt{f}})$ into $\to_{Q,\mathtt{f}}$, and define $\ell_{\text{in}}^{P,\mathtt{f}} := \ell_{\text{out}}^{Q,\mathtt{f}}$.
(6) *Demonic Nondeterminism.* For $P{=}$**if** $\star$ **then** Q_1 **else** Q_2 **fi**, we first add a new demonic label ℓ_{in}^{P} and a new branching labels ℓ_{out}^{P}, then take the disjoint union of $\to_{Q_1,\mathtt{f}}$ and $\to_{Q_2,\mathtt{f}}$ while simultaneously identifying both $\ell_{\text{out}}^{Q_1,\mathtt{f}}$ and $\ell_{\text{out}}^{Q_2,\mathtt{f}}$ with $\ell_{\text{out}}^{P,\mathtt{f}}$, and finally obtain $\to_{P,\mathtt{f}}$ by adding two triples $(\ell_{\text{in}}^{P,\mathtt{f}}, \star, \ell_{\text{in}}^{Q_1,\mathtt{f}})$ and $(\ell_{\text{in}}^{P,\mathtt{f}}, \star, \ell_{\text{in}}^{Q_2,\mathtt{f}})$ into the disjoint union of $\to_{Q_1,\mathtt{f}}$ and $\to_{Q_2,\mathtt{f}}$.

Based on CFG, 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.

Stack Elements and Configurations. A *stack element* $\mathfrak{c}$ (of P) is a triple $(\mathtt{f}, \ell, \nu)$ (treated as a letter) where $\mathtt{f} \in F$, $\ell \in L^{\mathtt{f}}$ and $\nu \in \mathit{Val}_{\mathtt{f}}$; $\mathfrak{c}$ is non-terminal if $\ell \in L^{\mathtt{f}} \setminus \{\ell_{\text{out}}^{\mathtt{f}}\}$. A *configuration* (of P) is a finite word of non-terminal stack elements (including the empty word ε). Thus, a stack element $(\mathtt{f}, \ell, \nu)$ specifies that the current function name is $\mathtt{f}$, the next statement to be executed is the one labelled with ℓ and the current valuation w.r.t $\mathtt{f}$ 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 label. A non-terminal stack element $\mathfrak{c}$ (as the initial stack element) and a scheduler π defines inductively 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* $\rho(\mathfrak{c}, \pi)$, as follows:

- Initially, $w_0 = \mathfrak{c}$.
- *Termination:* If $w_j = \varepsilon$ then $w_{j+1} := \varepsilon$.
- *Inductive Step:* Assume that $w_j = (\mathtt{f}, \ell, \nu) \cdot w'$. Then:
  (1) *assignment:* if $\ell \in L_{\text{a}}^{\mathtt{f}}$, (ℓ, h, ℓ') is the only triple in $\to_{\mathtt{f}}$ and $\nu' = h(\nu)$, then (i) $w_{j+1} := (\mathtt{f}, \ell', \nu') \cdot w'$ whenever $\ell' \neq \ell_{\text{out}}^{\mathtt{f}}$ and (ii) $w_{j+1} := w'$ otherwise;
  (2) *branching:* if $\ell \in L_{\text{b}}^{\mathtt{f}}$ and (ℓ, ϕ, ℓ') is the only triple in $\to_{\mathtt{f}}$ such that $\nu \models \phi$, then (i) $w_{j+1} := (\mathtt{f}, \ell', \nu) \cdot w'$ whenever $\ell' \neq \ell_{\text{out}}^{\mathtt{f}}$ and (ii) $w_{j+1} := w'$ otherwise;
  (3) *call:* if $\ell \in L_{\text{c}}^{\mathtt{f}}$ and $(\ell, (\mathtt{g}, h), \ell')$ is the only triple in $\to_{\mathtt{f}}$, then (i) $w_{j+1} := (\mathtt{g}, \ell_{\text{in}}^{\mathtt{g}}, h(\nu)) \cdot (\mathtt{f}, \ell', \nu) \cdot w'$ whenever $\ell' \neq \ell_{\text{out}}^{\mathtt{f}}$ and (ii) $w_{j+1} := (\mathtt{g}, \ell_{\text{in}}^{\mathtt{g}}, h(\nu)) \cdot w'$ otherwise;

(4) *nondeterminism:* if $\ell \in L^{\mathsf{f}}_{\mathrm{d}}$ and $\ell' = \pi(w_0 \dots w_j)$, then (i) $w_{j+1} := (\mathsf{f}, \ell', \nu) \cdot w'$ whenever $\ell' \neq \ell^{\mathsf{f}}_{\mathrm{out}}$ and (ii) $w_{j+1} := w'$ otherwise.

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 2.3 (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: $\overline{T}(\mathfrak{c}) := \sup\{T(\mathfrak{c}, \pi) \mid \pi \text{ is a scheduler for } P\}$.

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. Note that the problem of termination is undecidable in its general form. Hence, obtaining measure functions for arbitrary programs is also undecidable. In the next section, we will focus on an algorithmic approach for obtaining measure functions of *specific forms*. In the sequel, we fix a recursive program P together with its CFG taking the form (†).

We first present the standard notion of *invariants* which captures *reachable stack elements*. To this end, we define the 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 3.1 (Reachability). Let f^* be a function name and ϕ^* be a propositional arithmetic predicate over V^{f^*}. A configuration w is *reachable* w.r.t f^*, ϕ^* if there exist a scheduler π and a stack element $(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\mathrm{in}}, \nu)$ such that (i) $\nu \models \phi^*$ and (ii) w appears in the run $\rho((\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\mathrm{in}}, \nu), \pi)$. A stack element $\mathfrak{c}$ is *reachable* w.r.t f^*, ϕ^* if there exists a configuration w reachable w.r.t f^*, ϕ^* such that $w = \mathfrak{c} \cdot w'$ for some configuration w'.

Then we define the notion of invariants.

Definition 3.2 (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 $\ell \in L^{\mathsf{f}} \backslash \{\ell^{\mathsf{f}}_{\mathrm{out}}\}$, $I(\mathsf{f}, \ell)$ 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 [27], Farkas' Lemma [20], or using a Positivstellensatz [16]). In the rest of the section, we fix an initial function name $\mathsf{f}^* \in F$ and an initial propositional arithmetic predicate ϕ^* over V^{f^*}. For each $\mathsf{f} \in F$ and $\ell \in L^{\mathsf{f}} \backslash \{\ell^{\mathsf{f}}_{\mathrm{out}}\}$, 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 3.3 (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 $\to_{\mathsf{f}}$ 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 $\to_{\mathsf{f}}$ 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 $\to_{\mathsf{f}}$ 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 $\to_{\mathsf{f}}$ 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 results stating that measure functions are sound and complete for the worst-case termination-time function.

Proposition 3.4 (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 $\overline{T}(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu) \leq g(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu)$.*

Proof. Define the function h from the set of configurations into $[0, \infty]$ as follows:

$$h(w) := \sum_{k=0}^{n} g(\mathsf{f}_k, \ell_k, \nu_k) \text{ for } w = \{(\mathsf{f}_k, \ell_k, \nu_k)\}_{0 \leq k \leq n}$$

where $h(\varepsilon) := 0$. We show that h can be deemed as a ranking function over the set of reachable configurations w.r.t f^*, ϕ^*.

Let $\nu \in \mathit{Val}_{\mathsf{f}^*}$ be any valuation such that $\nu \models \phi^*$ and π be any scheduler for P. Moreover, let $\rho\left((\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu), \pi\right) = \{w_n\}_{n \in \mathbb{N}_0}$. Since the case $g(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu) = \infty$ is trivial, we only consider the case $g(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu) < \infty$.

By Definition 3.1, every w_n is reachable w.r.t f^*, ϕ^*. Hence, by Definition 3.3, (a) for all $n \in \mathbb{N}_0$, if $w_n \neq \varepsilon$ then $h(w_n) \geq 1$, and $h(w_n) = 0$ otherwise. Furthermore, by Definition 3.3 and our semantics,one can easily verify that (b) for all $n \in \mathbb{N}_0$, if $w_n \neq \varepsilon$ then $h(w_{n+1}) \leq h(w_n) - 1$.

To see (b), consider for example the function-call case where $w_n = (\mathsf{f}, \ell, \nu) \cdot w'$, $\ell \in L^{\mathsf{f}}_{\text{c}}$ and $(\ell, (\mathsf{g}, f), \ell')$ is the only triple in $\to_{\mathsf{f}}$ with source label ℓ. Then by our semantics, $h(w_n) = g(\mathsf{f}, \ell, \nu) + h(w')$ and $h(w_{n+1}) = g(\mathsf{g}, \ell^{\mathsf{g}}_{\text{in}}, f(\nu)) + g(\mathsf{f}, \ell', \nu) + h(w')$. Thus by C3, $h(w_{n+1}) + 1 \leq h(w_n)$. The other cases (namely assignment, branching and nondeterminism) can be verified similarly through a direct investigation of our semantics and an application of C2, C4 or C5.

From (a), (b) and the fact that $h(w_0) = g(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu) < \infty$, one has that $m := T\left((\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu), \pi\right) < \infty$ (or otherwise (a) and (b) cannot hold simultaneously). Furthermore, from an easy inductive proof based on (b), one has that $h(w_m) \leq h(w_0) - m$. Together with $h(w_m) = 0$ (from (a)), one obtains that

$$T\left((\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu), \pi\right) = m \leq h(w_0) = g(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\text{in}}, \nu) \ .$$

Hence, $\overline{T}(\mathsf{f}^*, \ell_{\text{in}}^{\mathsf{f}^*}, \nu) \leq g(\mathsf{f}^*, \ell_{\text{in}}^{\mathsf{f}^*}, \nu)$. □

To prove completeness, we first do some technical preparations. We recall that for each $\mathsf{f} \in F$ and $\ell \in L^{\mathsf{f}} \backslash \{\ell_{\text{out}}^{\mathsf{f}}\}$, 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^*.

We introduce some notations for schedulers. Let π be a scheduler for P and $\mathfrak{c}$ be a non-terminal stack element. We define $\text{post}(\pi, \mathfrak{c})$ to be the scheduler such that for any non-empty finite word of configurations $w_0 \dots w_n$ with w_n being non-deterministic,

$$\text{post}(\pi, \mathfrak{c})(w_0 \dots w_n) = \pi(\mathfrak{c} \cdot w_0 \dots w_n) \ .$$

In the case that $\mathfrak{c}$ is not non-deterministic, we define $\text{pre}(\pi, \mathfrak{c})$ to be one of the schedulers such that for any non-empty finite word of configurations $w_0 \dots w_n$ with w_n being non-deterministic,

$$\text{pre}(\pi, \mathfrak{c})(\mathfrak{c} \cdot w_0 \dots w_n) = \pi(w_0 \dots w_n) \ ;$$

the decisions of $\text{pre}(\pi, \mathfrak{c})$ at finite words not starting with $\mathfrak{c}$ will be irrelevant. In the case that $\mathfrak{c} = (\mathsf{f}, \ell, \nu)$ is non-deterministic, for any given $\ell' \in L^{\mathsf{f}}$ such that $(\ell, \star, \ell') \in \to_{\mathsf{f}}$, we define $\text{pre}(\pi, (\mathfrak{c}, \ell'))$ to be one of the schedulers such that (i)

$$\text{pre}(\pi, (\mathfrak{c}, \ell'))(\mathfrak{c}) = \ell'$$

and (ii) for any non-empty finite word of configurations $w_0 \dots w_n$ with w_n being non-deterministic,

$$\text{pre}(\pi, (\mathfrak{c}, \ell'))(\mathfrak{c} \cdot w_0 \dots w_n) = \pi(w_0 \dots w_n) \ ;$$

again, the decisions of $\text{pre}(\pi, (\mathfrak{c}, \ell'))$ at finite words not starting with $\mathfrak{c}$ will be irrelevant.

Proposition 3.5 (Completeness). *$\overline{T}$ is a measure function w.r.t f^*, ϕ^*.*

Proof. We prove that $\overline{T}$ satisfies the conditions C1–C5 in Definition 3.3 with equality. Condition C1 follows directly from the definition. Below we prove that C2–C5 hold.

Let $\mathfrak{c} = (\mathsf{f}, \ell, \nu)$ be a non-terminal stack element such that $\nu \in D_{\mathsf{f},\ell}$. Below we clarify several cases on $\mathfrak{c}$.

Case 1 (cf. C2): $\ell \in L_{\text{a}}^{\mathsf{f}} \backslash \{\ell_{\text{out}}^{\mathsf{f}}\}$ and (ℓ, f, ℓ') is the only triple in $\to_{\mathsf{f}}$ with source label ℓ. Consider any scheduler π for W. By our semantics, one can prove easily that

- $T\left((\mathsf{f}, \ell, \nu), \pi\right) = 1 + T\left((\mathsf{f}, \ell', f(\nu)), \text{post}(\pi, \mathfrak{c})\right)$, and
- $T\left((\mathsf{f}, \ell, \nu), \text{pre}(\pi, \mathfrak{c})\right) = 1 + T\left((\mathsf{f}, \ell', f(\nu)), \pi\right)$.

Since π is arbitrarily, by taking the supremum at the both sides of the equalities above one has that

$$\overline{T}(\mathsf{f}, \ell', f(\nu)) + 1 = \overline{T}(\mathsf{f}, \ell, \nu) \ .$$

Case 2 (cf. C3): $\ell \in L_{\text{c}}^{\mathsf{f}} \backslash \{\ell_{\text{out}}^{\mathsf{f}}\}$ and $(\ell, (\mathsf{g}, f), \ell')$ is the only triple in $\to_{\mathsf{f}}$ with source label ℓ.

We first consider the case $\overline{T}(\mathsf{g}, \ell_{\text{in}}^{\mathsf{g}}, \nu) = \infty$, meaning that schedulers π can make $T\left((\mathsf{g}, \ell_{\text{in}}^{\mathsf{g}}, \nu), \pi\right)$ arbitrarily large. Since for any scheduler π, $T\left((\mathsf{f}, \ell, \nu), \text{pre}(\pi, \mathfrak{c})\right) \geq 1 + T\left((\mathsf{g}, \ell_{\text{in}}^{\mathsf{g}}, \nu), \pi\right)$, one has that

$$\overline{T}(\mathsf{f}, \ell, \nu) = 1 + \overline{T}(\mathsf{g}, \ell_{\text{in}}^{\mathsf{g}}, f(\nu)) + \overline{T}(\mathsf{f}, \ell', \nu) (= \infty) \ .$$

Then we consider the case that $\overline{T}(\mathsf{g}, \ell_{\text{in}}^{\mathsf{g}}, \nu) < \infty$. On one hand, let π be any scheduler for W. Since $T\left((\mathsf{g}, \ell_{\text{in}}^{\mathsf{g}}, \nu), \text{post}(\pi, \mathfrak{c})\right) < \infty$, one can find a (unique) finite prefix γ of the

run $\rho\left((\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, \nu), \mathrm{post}(\pi, \mathfrak{c})\right)$ consisting of only non-empty (i.e. not ε) configurations which describes the finite execution of the function call g under the scheduler $\mathrm{post}(\pi, \mathfrak{c})$. By our semantics, one can prove easily that

$$T\left((\mathsf{f}, \ell, \nu), \pi\right) = 1 + T\left((\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, f(\nu)), \mathrm{post}(\pi, \mathfrak{c})\right) + T\left((\mathsf{f}, \ell', \nu), \pi'\right)$$

where π' is the scheduler such that for any non-empty finite word of configurations $w_0 \dots w_n$ with w_n being non-deterministic, $\pi'(w_0 \dots w_n) = \pi(\mathfrak{c} \cdot \gamma \cdot w_0 \dots w_n)$. It follows from the arbitrary choice of π that

$$\overline{T}(\mathsf{f}, \ell, \nu) \leq 1 + \overline{T}(\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, f(\nu)) + \overline{T}(\mathsf{f}, \ell', \nu) \ .$$

On the other hand, let π_1, π_2 be any two schedulers for P. Since $\overline{T}\left(\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, \nu\right) < \infty$, one can find a (unique) finite prefix γ of the run $\rho\left((\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, \nu), \pi_1\right)$ consisting of only non-empty (i.e. not ε) configurations which describes the finite execution of the function call g under the scheduler π_1. Then

$$T\left((\mathsf{f}, \ell, \nu), \pi\right) = 1 + T\left((\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, f(\nu)), \pi_1\right) + T\left((\mathsf{f}, \ell', \nu), \pi_2\right)$$

where π is one of the schedulers such that (i) $\pi(\mathfrak{c} \cdot \beta) = \pi_1(\beta)$ whenever β ends at a non-deterministic configuration and is a prefix of γ (including γ) and (ii) $\pi(\mathfrak{c} \cdot \gamma \cdot \beta) = \pi_2(\beta)$ whenever β is non-empty and ends at a non-deterministic configuration. Thus, by the arbitrary choice of π_1, π_2, one has that

$$1 + \overline{T}\left(\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, f(\nu)\right) + \overline{T}\left(\mathsf{f}, \ell', \nu\right) \leq \overline{T}\left(\mathsf{f}, \ell, \nu\right) \ .$$

In either case, we have

$$1 + \overline{T}\left(\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, f(\nu)\right) + \overline{T}\left(\mathsf{f}, \ell', \nu\right) = \overline{T}\left(\mathsf{f}, \ell, \nu\right) \ .$$

Case 3 (cf. C4): $\ell \in L^{\mathsf{f}}_{\mathrm{b}} \backslash \{\ell^{\mathsf{f}}_{\mathrm{out}}\}$ and $(\ell, \phi, \ell_1), (\ell, \neg\phi, \ell_2)$ are namely two triples in $\rightarrow_{\mathsf{f}}$ with source label ℓ. By our semantics, one can easily prove that

$$T\left((\mathsf{f}, \ell, \nu), \pi\right) = 1 + \mathbf{1}_{\nu \models \phi} \cdot T\left((\mathsf{f}, \ell_1, \nu), \mathrm{post}(\pi, \mathfrak{c})\right) + \mathbf{1}_{\nu \models \neg\phi} \cdot T\left((\mathsf{f}, \ell_2, \nu), \mathrm{post}(\pi, \mathfrak{c})\right)$$

for any scheduler π for W, and

$$T\left((\mathsf{f}, \ell, \nu), \pi\right) = 1 + \mathbf{1}_{\nu \models \phi} \cdot T\left((\mathsf{f}, \ell_1, \nu), \pi_1\right) + \mathbf{1}_{\nu \models \neg\phi} \cdot T\left((\mathsf{f}, \ell_2, \nu), \pi_2\right)$$

for any schedulers π_1, π_2 for P, where π is either $\mathrm{pre}\left(\pi_1, \mathfrak{c}\right)$ if $\nu \models \phi$, or $\mathrm{pre}\left(\pi_2, \mathfrak{c}\right)$ if $\nu \models \neg\phi$. By taking the supremum at the both sides of the equalities above, one has

$$\overline{T}\left(\mathsf{f}, \ell, \nu\right) = 1 + \mathbf{1}_{\nu \models \phi} \cdot \overline{T}\left(\mathsf{f}, \ell_1, \nu\right) + \mathbf{1}_{\nu \models \neg\phi} \cdot \overline{T}\left(\mathsf{f}, \ell_2, \nu\right) \ .$$

Case 4 (cf. C5): $\ell \in L^{\mathsf{f}}_{\mathrm{d}} \backslash \{\ell^{\mathsf{f}}_{\mathrm{out}}\}$ and $(\ell, \star, \ell_1), (\ell, \star, \ell_2)$ are namely two triples in $\rightarrow_{\mathsf{f}}$ with source label ℓ. By our semantics, one easily proves that

$$T\left((\mathsf{f}, \ell, \nu), \pi\right) = 1 + \mathbf{1}_{\pi(\mathfrak{c}) = \ell_1} \cdot T\left((\mathsf{f}, \ell_1, \nu), \mathrm{post}(\pi, \mathfrak{c})\right) + \mathbf{1}_{\pi(\mathfrak{c}) = \ell_2} \cdot T\left((\mathsf{f}, \ell_2, \nu), \mathrm{post}(\pi, \mathfrak{c})\right) \ .$$

for any scheduler π (for P), and

$$T\left((\mathsf{f}, \ell, \nu), \pi\right) = 1 + \max\{T\left((\mathsf{f}, \ell_1, \nu), \pi_1\right), T\left((\mathsf{f}, \ell_2, \nu), \pi_2\right)\} \ .$$

for any schedulers π, π_1, π_2 such that π is either $\mathrm{pre}(\pi_1, (\mathfrak{c}, \ell_1))$ if $T\left((\mathsf{f}, \ell_1, \nu), \pi_1\right) \geq T\left((\mathsf{f}, \ell_2, \nu), \pi_2\right)$, or $\mathrm{pre}(\pi_2, (\mathfrak{c}, \ell_2))$ if otherwise. By taking the supremum at the both sides of the equalities above, one obtains

$$\overline{T}\left(\mathsf{f}, \ell, \nu\right) = 1 + \max\left\{\overline{T}\left(\mathsf{f}, \ell_1, \nu\right), \overline{T}\left(\mathsf{f}, \ell_2, \nu\right)\right\} \ .$$

□

By the previous two propositions, one directly obtains the soundness and completeness of measure functions.

Theorem 3.6 (Soundness and Completeness). *Measure functions are sound and complete for worst-case termination time of recursive programs.*

By Theorem 3.6, 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 to as *significant* labels).

Definition 3.7 (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. By introducing significant labels, one can define a measure function only at significant labels and then expand it to all labels.

The Expansion Construction (from g to $\widehat{g}$). Let g be a function from

$$\left\{(\mathsf{f}, \ell, \nu) \mid \mathsf{f} \in F, \ell \in L^{\mathsf{f}}_{\text{s}}, \nu \in \mathit{Val}_{\mathsf{f}}\right\}$$

into $[0, \infty]$. It is intuitively clear that 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. Below we illustrate the details. Let g be a function from $\left\{(\mathsf{f}, \ell, \nu) \mid \mathsf{f} \in F, \ell \in L^{\mathsf{f}}_{\text{s}}, \nu \in \mathit{Val}_{\mathsf{f}}\right\}$ into $[0, \infty]$. The function *expanded from* g, denoted by $\widehat{g}$, is a function from the set of all stack elements into $[0, \infty]$ inductively defined through the procedure described as follows.

(1) *Initial Step.* If $\ell \in L^{\mathsf{f}}_{\text{s}}$, then $\widehat{g}(\mathsf{f}, \ell, \nu) := g(\mathsf{f}, \ell, \nu)$.
(2) *Termination.* If $\ell = \ell^{\mathsf{f}}_{\text{out}}$, then $\widehat{g}(\mathsf{f}, \ell, \nu) := 0$.
(3) *Assignment.* If $\ell \in L^{\mathsf{f}}_{\text{a}} \backslash L^{\mathsf{f}}_{\text{s}}$ with (ℓ, h, ℓ') being the only triple in $\to_{\mathsf{f}}$ and $\widehat{g}(\mathsf{f}, \ell', \centerdot)$ is already defined, then $\widehat{g}(\mathsf{f}, \ell, \nu) := 1 + \widehat{g}(\mathsf{f}, \ell', h(\nu))$.
(4) *Branching.* If $\ell \in L^{\mathsf{f}}_{\text{b}} \backslash L^{\mathsf{f}}_{\text{s}}$ with (ℓ, ϕ, ℓ_1), $(\ell, \neg\phi, \ell_2)$ being namely the two triples in $\to_{\mathsf{f}}$ and both $\widehat{g}(\mathsf{f}, \ell_1, \centerdot)$ and $\widehat{g}(\mathsf{f}, \ell_2, \centerdot)$ is already defined, then $\widehat{g}(\mathsf{f}, \ell, \nu) := \mathbf{1}_{\nu \models \phi} \cdot \widehat{g}(\mathsf{f}, \ell_1, \nu) + \mathbf{1}_{\nu \models \neg\phi} \cdot \widehat{g}(\mathsf{f}, \ell_2, \nu) + 1$;
(5) *Call.* If $\ell \in L^{\mathsf{f}}_{\text{c}} \backslash L^{\mathsf{f}}_{\text{s}}$ with $(\ell, (\mathsf{g}, h), \ell')$ being the only triple in $\to_{\mathsf{f}}$ and $\widehat{g}(\mathsf{f}, \ell', \centerdot)$ is already defined, then $\widehat{g}(\mathsf{f}, \ell, \nu) := g(\mathsf{g}, \ell^{\mathsf{g}}_{\text{in}}, h(\nu)) + \widehat{g}(\mathsf{f}, \ell', \nu) + 1$.
(6) *Non-determinism.* If $\ell \in L^{\mathsf{f}}_{\text{d}} \backslash L^{\mathsf{f}}_{\text{s}}$ with $(\ell, \star, \ell_1)$, $(\ell, \star, \ell_2)$ being namely the two triples in $\to_{\mathsf{f}}$ with source label ℓ and both $\widehat{g}(\mathsf{f}, \ell_1, \centerdot)$ and $\widehat{g}(\mathsf{f}, \ell_2, \centerdot)$ is already defined, then

$$\widehat{g}(\mathsf{f}, \ell, \nu) := \max\left\{\widehat{g}(\mathsf{f}, \ell_1, \nu), \widehat{g}(\mathsf{f}, \ell_2, \nu)\right\} + 1.$$

Note that in the previous expansion construction, we have not technically shown that $\widehat{g}$ is defined over all stack elements. The following technical lemma shows that the function $\widehat{g}$ is indeed well-defined.

Lemma 3.8. *For each function g from $\left\{(\mathsf{f}, \ell, \nu) \mid \mathsf{f} \in F, \ell \in L^{\mathsf{f}}_{\text{s}}, \nu \in \mathit{Val}_{\mathsf{f}}\right\}$ into $[0, \infty]$, the function $\widehat{g}$ is well-defined.*

Proof. Suppose that $\widehat{g}$ is not well-defined, i.e., there exists some $\mathsf{f} \in F$ and $\ell_0 \in L^{\mathsf{f}}$ such that $\widehat{g}(\mathsf{f}, \ell_0, \centerdot)$ remains undefined. Then by the inductive procedure, there exists a triple (ℓ_0, α, ℓ_1) in $\to_{\mathsf{f}}$ such that $\widehat{g}(\mathsf{f}, \ell_1, \centerdot)$ remains undefined. With the same reasoning, one can inductively

construct an infinite sequence $\{\ell_j\}_{j\in\mathbb{N}_0}$ such that each $\widehat{g}(\mathsf{f},\ell_j,\cdot)$ remains undefined. Since L^{f} is finite, there exist j_1, j_2 such that $j_1 \neq j_2$ and $\ell_{j_1} = \ell_{j_2}$. It follows from our semantics that there exists j^* such that $j_1 \leq j^* \leq j_2$ and ℓ_{j^*} corresponds to the initial label of a while-loop in W. Contradiction to the fact that $\ell_{j^*} \in L^{\mathsf{f}}_{\mathrm{s}}$. □

With significant labels, we can apply conditions C1–C5 directly to significant labels, as is shown by the next proposition.

Proposition 3.9. *Let g be a function from $\big\{(\mathsf{f},\ell,\nu) \mid \mathsf{f}\in F, \ell\in L^{\mathsf{f}}_{\mathrm{s}}, \nu\in \mathit{Val}_{\mathsf{f}}\big\}$ into $[0,\infty]$. Let I be an invariant w.r.t f^*,ϕ^*. Consider that for all stack elements (f,ℓ,ν) such that $\ell\in L^{\mathsf{f}}_{\mathrm{s}}$ and $\nu\models I(\mathsf{f},\ell)$, the following conditions hold:*

- ***C2':*** *if $\ell\in L^{\mathsf{f}}_{\mathrm{a}}$ and (ℓ,f,ℓ') is the only triple in $\to_{\mathsf{f}}$ with source label ℓ, then $\widehat{g}(\mathsf{f},\ell',f(\nu))+1\leq\widehat{g}(\mathsf{f},\ell,\nu)$;*
- ***C3':*** *if $\ell\in L^{\mathsf{f}}_{\mathrm{c}}$ and $(\ell,(\mathsf{g},f),\ell')$ is the only triple in $\to_{\mathsf{f}}$ with source label ℓ, then $1+\widehat{g}(\mathsf{g},\ell^{\mathsf{g}}_{\mathrm{in}},f(\nu))+\widehat{g}(\mathsf{f},\ell',\nu)\leq\widehat{g}(\mathsf{f},\ell,\nu)$;*
- ***C4':*** *if $\ell\in L^{\mathsf{f}}_{\mathrm{b}}$ and $(\ell,\phi,\ell_1),(\ell,\neg\phi,\ell_2)$ are namely two triples in $\to_{\mathsf{f}}$ with source label ℓ, then $\mathbf{1}_{\nu\models\phi}\cdot\widehat{g}(\mathsf{f},\ell_1,\nu)+\mathbf{1}_{\nu\models\neg\phi}\cdot\widehat{g}(\mathsf{f},\ell_2,\nu)+1\leq\widehat{g}(\mathsf{f},\ell,\nu)$;*
- ***C5':*** *if $\ell\in L^{\mathsf{f}}_{\mathrm{d}}$ and $(\ell,\star,\ell_1),(\ell,\star,\ell_2)$ are namely two triples in $\to_{\mathsf{f}}$ with source label ℓ, then $\max\{\widehat{g}(\mathsf{f},\ell_1,\nu),\widehat{g}(\mathsf{f},\ell_2,\nu)\}+1\leq\widehat{g}(\mathsf{f},\ell,\nu)$.*

Then $\widehat{g}$ is a measure function w.r.t f^*,ϕ^*.

Proof. The proof follows directly from the fact that (i) all valuations in $D_{\mathsf{f},\ell}$ satisfy $I(\mathsf{f},\ell)$, for all $\mathsf{f}\in F$ and $\ell\in L^{\mathsf{f}}\backslash\{\ell^{\mathsf{f}}_{\mathrm{out}}\}$, (ii) C2'-C5' directly implies C2-C5 for $\ell\in L^{\mathsf{f}}_{\mathrm{s}}$ and (iii) C1-C5 are automatically satisfied for $\ell\notin L^{\mathsf{f}}_{\mathrm{s}}$ by the expansion construction of significant labels. □

4 THE SYNTHESIS ALGORITHM

By Theorem 3.6, finding general measure functions is a sound and complete approach for upper bounds of the worst-case termination-time function. However, it is well-known that the termination problem is undecidable in general. Hence, we focus on the synthesis of measure functions of *specific forms*, which lead to a sound method for finding upper bounds for worst-case behavior 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 as follows:

- $d\in\mathbb{N}$ 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 op = exp, and
- $k \in \mathbb{N}$ is a technical parameter required by Theorem 4.8.

Remark 1. In the input for RecTermBou we fix the exponent r when op = exp. However, by iterating with binary search over an input bounded range, we can obtain an arbitrarily-precise measure function in the given range. Moreover, the invariants can be obtained automatically through e.g. [16, 20].

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 significant label so that C1–C5 are fulfilled. Below we fix an input to our algorithm.

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

(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)).
(Step 2) The second step considers the template generated in Step 1 for significant labels and generates templates for all labels. This step is relatively straightforward.
(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.
(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.
(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 4.8) to transform into a linear program.

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

Example 4.1. Consider the task to synthesize a measure function for Karatsuba's algorithm [54] 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 to Karatsuba's algorithm 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 sub-multiplications 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. □

Remark 2 (Complexity). In the following, we analyze the complexity of each step of our approach w.r.t the size $|P|$ of the input program, i.e., the length of the program where the decimals are represented in binary. We assume that the maximal number of conditional/non-deterministic branches and function calls in one subroutine are bounded and its corresponding invariant length is constant. We also assume that the conditions in conditional branches have bounded length and consider the parameters r and d to be constants.

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 of the form $\ln x, \ln(x - y + 1)$ 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 of the form $x^r, (x - y + 1)^r$ 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

$$\mathfrak{e} = \textstyle\sum_i c_i \cdot \bar{\mathrm{g}}_i \tag{1}$$

where (i) each c_i is a constant scalar and each $\bar{\mathrm{g}}_i$ 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 $\bar{\mathrm{g}}_i$'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 several previous works (cf. [13, 17, 21, 26, 42, 61, 65, 72]), 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. Although r is fixed when $\mathrm{op} = \exp$, one can perform a binary search over a bounded range for a suitable or optimal r.

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 4.2 (Farkas' Lemma [29, 64]). *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 $\{\mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b}\} \neq \emptyset$. Then $\{\mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b}\} \subseteq \{\mathbf{x} \mid \mathbf{c}^{\mathrm{T}}\mathbf{x} \leq d\}$ iff there exists $\mathbf{y} \in \mathbb{R}^m$ such that $\mathbf{y} \geq \mathbf{0}$, $\mathbf{A}^{\mathrm{T}}\mathbf{y} = \mathbf{c}$ and $\mathbf{b}^{\mathrm{T}}\mathbf{y} \leq d$.*

By Farkas' Lemma, there exists an algorithm that infers whether $x - 1 \geq 0$ (or $x - y \geq 0$) holds under $I(\mathsf{f}, \ell)$ in polynomial time through emptiness checking of polyhedra (cf. [63]) since $I(\mathsf{f}, \ell)$ involves only linear (degree-1) polynomials in our setting.

Then η naturally induces a function $[\![\eta]\!]$ 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 $[0, \infty]$ 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.

Remark 3 (Complexity). Note that the length of the template $\eta(\mathsf{f}, \ell)$, generated according to the general form above at each label of the program P, is only dependent on the number of variables in P and the degree d of the template, where d is constant. Therefore, the template generated at each label has polynomial length and the total runtime for generating all templates is polynomial in $|P|$. Moreover, each application of Farkas' Lemma over an invariant $I(\mathsf{f}, \ell)$ leads to a linear programming instance of size $O(|I(\mathsf{f}, \ell)|)$, which can be solved in polynomial time. Therefore, the total runtime is polynomial in $|P|$.

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 (Section 3) which transforms a function g into $\widehat{g}$[1]. 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 symbolically computes $\widehat{[\![\eta]\!]}$, i.e. template variables appearing in η are treated as undetermined constants. Then $\widehat{[\![\eta]\!]}$ is a function parametric over the template variables in η.

[1]In general, we use η to signify a measure function with unknown coefficients, i.e. c-variables, that need to be synthesized, and use g to denote a concrete measure function.

For the computation, we use the fact that all propositional arithmetic predicates can be put in disjunctive normal form. We also use the fact that for real-valued functions $\{f_i\}_{1\le i\le m}$, $\{g_j\}_{1\le j\le n}$ and h, it holds that

$$\max\{f_i\}_i + \max\{g_j\}_j = \max\{f_i + g_j\}_{i,j}$$

and

$$h \cdot \max\{f_i\}_i = \max\{h \cdot f_i\}_i$$

provided that h is everywhere non-negative, where the maximum function over a finite set of functions is defined in pointwise fashion. Moreover, we use the facts that (i)

$$\mathbf{1}_{\nu\models\phi_1} \cdot \mathbf{1}_{\nu\models\phi_2} = \mathbf{1}_{\nu\models\phi_1\wedge\phi_2}$$

for all propositional arithmetic predicates ϕ_1, ϕ_2 and valuations ν, and (ii)

$$\left(\sum_{i=1}^{m} \mathbf{1}_{\phi_i} \cdot g_i\right) + \left(\sum_{j=1}^{n} \mathbf{1}_{\psi_j} \cdot h_j\right) = \sum_{i=1}^{m}\sum_{j=1}^{n} \mathbf{1}_{\phi_i\wedge\psi_j} \cdot (h_i + g_j)$$

provided that (a) $\bigvee_i \phi_i, \bigvee_j \psi_j$ are both tautology and (b) $\phi_{i_1} \wedge \phi_{i_2}, \psi_{j_1} \wedge \psi_{j_2}$ are both unsatisfiable whenever $i_1 \neq i_2$ and $j_1 \neq j_2$, and (iii) propositional arithmetic predicates are closed under substitution of expressions in $\langle expr\rangle$ for scalar variables. Then by an easy induction on the computation steps, each $\widehat{[\![\eta]\!]}(\mathsf{f}, \ell, \centerdot)$ can be represented by an expression in the form:

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

(1) 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
(2) 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 $\bar{\mathrm{g}}_i$ 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 $\langle expr\rangle$.

Remark 4 (Complexity). Note that, as in Remark 3, each $\eta(\mathsf{f}, \ell)$ has polynomial length. Moreover, the maximal number of conditional/non-deterministic branches and function calls in one subroutine (that may cause exponential blow up in the size of $\widehat{[\![\eta]\!]}$) is bounded. Therefore, computing $\widehat{[\![\eta]\!]}$ takes polynomial time in the size $|P|$ of the program.

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 $\mathfrak{e}' \geq 0$ 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 $\bar{\mathrm{g}}_i$ 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: $\forall \nu \in Val_{\mathsf{f}}. (\nu \models \phi \to [\![\mathfrak{e}]\!](\nu) \geq 0)$. 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 groups 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 conjunctive fashion to form a single collection of constraint triples.

Let ℓ be any significant label at any function name f, the algorithm generates constraint triples at f, ℓ as follows: W.l.o.g, let $I(\mathsf{f}, \ell) = \bigvee_l \Psi_l$ where each Ψ_l is a conjunction of atomic formulae of the form $\mathfrak{e} \geq 0$. The algorithm first generates constraint triples related to non-negativity of measure functions.

- **Non-negativity**: The algorithm generates the collection of constraint triples

$$\{(\mathsf{f}, \Psi_l, \eta(\mathsf{f}, \ell))\}_l \quad .$$

Then the algorithm generates constraint triples through C2'-C5' as follows.

- **Case 1 (C2')**: $\ell \in L^{\mathsf{f}}_{\mathrm{s}} \cap L^{\mathsf{f}}_{\mathrm{a}}$ and (ℓ, f, ℓ') is the sole triple in $\to_{\mathsf{f}}$ with source label ℓ. As $\widehat{[\![\eta]\!]}(\mathsf{f}, \ell', \centerdot)$ can be represented in the form (2), $\widehat{[\![\eta]\!]}(\mathsf{f}, \ell', f(\centerdot))$ can also be represented by an expression

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

  in the form (2). Let a disjunctive normal form of each formula $I(\mathsf{f}, \ell) \wedge \phi_{ij}$ be $\bigvee_l \Phi^l_{ij}$, where each Φ^l_{ij} is a conjunction of atomic formulae of the form $\mathfrak{e}' \geq 0$. Then the algorithm generates the collection of constraint triples

$$\left\{\left(\mathsf{f}, \Phi^l_{ij}, \eta(\mathsf{f}, \ell) - h_{ij} - 1\right)\right\}_{i,j,l} \quad .$$

- **Case 2 (C3')**: $\ell \in L^{\mathsf{f}}_{\mathrm{c}}$ and $(\ell, (\mathsf{g}, f), \ell')$ is the sole triple in $\to_{\mathsf{f}}$ with source label ℓ. Let $\widehat{[\![\eta]\!]}\,(\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, f(\centerdot)) + \widehat{[\![\eta]\!]}(\mathsf{f}, \ell', \centerdot)$ be represented by the expression

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

  in the form (2). Let a disjunctive normal form of each formula $I(\mathsf{f}, \ell) \wedge \phi_{ij}$ be $\bigvee_l \Phi^l_{ij}$, where each Φ^l_{ij} is a conjunction of atomic formulae of the form $\mathfrak{e}' \geq 0$. Then the algorithm generates the collection of constraint triples

$$\left\{\left(\mathsf{f}, \Phi^l_{ij}, \eta(\mathsf{f}, \ell) - h_{ij} - 1\right)\right\}_{i,j,l} \quad .$$

- **Case 3 (C4')**: $\ell \in L^{\mathsf{f}}_{\mathrm{b}}$ and $(\ell, \phi, \ell_1), (\ell, \neg\phi, \ell_2)$ are namely two triples in $\to_{\mathsf{f}}$ with source label ℓ. Let h be the function (parametric on template variables)

$$\mathbf{1}_{\phi} \cdot \widehat{[\![\eta]\!]}(\mathsf{f}, \ell_1, \centerdot) + \mathbf{1}_{\neg\phi} \cdot \widehat{[\![\eta]\!]}(\mathsf{f}, \ell_2, \centerdot) \ .$$

  Then h can be represented by an expression

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

  in the form (2). Let a disjunctive normal form of each formula $I(\mathsf{f}, \ell) \wedge \phi_{ij}$ be $\bigvee_l \Phi^l_{ij}$, where each Φ^l_{ij} is a conjunction of atomic formulae of the form $\mathfrak{e}' \geq 0$. Then the algorithm generates the collection of constraint triples

$$\left\{ \left(\mathsf{f}, \Phi^l_{ij}, \eta(\mathsf{f}, \ell) - h_{ij} - 1\right) \right\}_{i,j,l} \ .$$

- **Case 4 (C5')** $\ell \in L^{\mathsf{f}}_{\mathrm{d}}$ and $(\ell, \star, \ell_1), (\ell, \star, \ell_2)$ are namely two triples in $\to_{\mathsf{f}}$ with source label ℓ. Let h be the function (parametric on template variables)

$$\max\left\{ \widehat{[\![\eta]\!]}(\mathsf{f}, \ell_1, \centerdot), \widehat{[\![\eta]\!]}(\mathsf{f}, \ell_2, \centerdot) \right\}$$

  otherwise. Then h can be represented by an expression

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

  in the form (2). Let a disjunctive normal form of each formula $I(\mathsf{f}, \ell) \wedge \phi_{ij}$ be $\bigvee_l \Phi^l_{ij}$. Then the algorithm generates the collection of constraint triples

$$\left\{ \left(\mathsf{f}, \Phi^l_{ij}, \eta(\mathsf{f}, \ell) - h_{ij} - 1\right) \right\}_{i,j,l} \ .$$

After generating the constraint triples for each significant label, the algorithm groups them together in a conjunctive fashion to form a single collection of constraint triples.

Remark 5 (Complexity). Given that the $\widehat{[\![\eta]\!]}$ at each label has polynomial size (see Remark 4), each of the cases above has a polynomial running time in $|P|$. Hence, this step runs in polynomial time and generates at most $O(|P|)$ triples of polynomial size.

Example 4.3. Consider our running example (cf. Example 2.2). 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 $\forall n.(n \geq 1) \to c_1 \cdot n + c_2 \cdot \ln n + c_3 \geq 0$ 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
  – $\forall n.(n \geq 1 \wedge n \leq 1) \to c_1 \cdot n + c_2 \cdot \ln n + c_3 \geq 2$ and
  – $\forall n.(n \geq 2) \to c_1 \cdot n + c_2 \cdot \ln n + c_3 \geq c_1 \cdot \lfloor n/2 \rfloor + c_2 \cdot \ln \lfloor n/2 \rfloor + c_3 + 2$
  for C4 (at label 1). □

4.4 Step 4 of SynAlgo

Step 4: Solving Constraint Triples. When solving the constraint triples generated in the previous step to find concrete coefficients for the template variables so that the constraint triples hold. A major barrier is that the expression $\mathfrak{e}$ in a constraint triple $(\mathsf{f}, \phi, \mathfrak{e})$ involves non-polynomial terms, and validity with non-polynomial terms is generally undecidable [35]. To overcome this difficulty, we first abstract every non-polynomial term with a stand-alone variable. Such abstraction loses the detailed information of the non-polynomial terms, so in the second step we establish linear inequalities over these variables to recover the linear relationships between these variables. The final solution of the constraint triples is given in the next step (Step 5). In detail, the algorithm follows a sound method which abstracts each multiplicand other than scalar variables in the form (2) as a stand-alone variable. The main idea is that the algorithm establishes tight linear inequalities for those abstraction variables by investigating properties of the abstracted arithmetic expressions.

Below, we describe the main aspect in this step on 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 $\mathsf{Sat}(\Gamma) := \{\mathbf{x} \in \mathbb{R}^n \mid f(\mathbf{x}) \geq 0 \text{ for all } f \in \Gamma\}$. 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 4.8.

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, Exponentiation 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.
   - $\mathcal{E}_E := \{\mathfrak{e} \mid \ \mathfrak{e}^r \text{ appears in } \mathfrak{e}^* \text{ (as sub-expression)}\}$ upon op = exp.
   - $\mathcal{E}_F := \{\mathfrak{e} \mid \ \mathfrak{e} \text{ appears in } \mathfrak{e}^* \text{ and takes the form } \lfloor \frac{\star}{c} \rfloor\}$.

   Let $\mathcal{E} := \mathcal{E}_L \cup \mathcal{E}_E \cup \mathcal{E}_F$.
(3) *Variables for Logarithmic, Exponentiation 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 $\mathfrak{e} \in \mathcal{E}_L$;
   - two fresh variables $v_\mathfrak{e}, v'_\mathfrak{e}$ such that $v_\mathfrak{e}$ indicates $\mathfrak{e}^r$ and $v'_\mathfrak{e}$ for $\mathfrak{e}^{r-1}$ for $\mathfrak{e} \in \mathcal{E}_E$;
   - a fresh variable $w_\mathfrak{e}$ indicating $\mathfrak{e}$ for $\mathfrak{e} \in \mathcal{E}_F$.

   We note that $v'_\mathfrak{e}$ is introduced in order to have a more accurate approximation for $v_\mathfrak{e}$ later. 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 $\mathfrak{e}^*$ 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 $\ln \mathfrak{e}'$, (ii) every possible $v_{\mathfrak{e}'}$ for $(\mathfrak{e}')^r$ and (iii) 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 deemed as a polynomial over scalar and freshly-added variables. Then any $\mathfrak{e}(\mathbf{x})$ or $\widetilde{\mathfrak{e}}(\mathbf{x})$ is the result of polynomial evaluation under the correspondence between variables and coordinates of $\mathbf{x}$ 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
$$\forall \mathbf{x} \in \mathbb{R}^N. \left(\mathbf{x} \in \mathsf{Sat}(\Gamma) \to \widetilde{\mathfrak{e}'}(\mathbf{x}) \geq t_{\mathfrak{e}'}\right)$$
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. [63]). 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 4.4. We continue with Example 4.3. 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) to approximate logarithmic and exponentiation terms.

Theorem 4.5 (Lagrange's Mean-Value Theorem [7, Chapter 6]). *Let $f : [a, b] \to \mathbb{R}$ (for $a < b$) be a function continuous on $[a, b]$ and differentiable on (a, b). Then there exists a real number $\xi \in (a, b)$ such that $f'(\xi) = \frac{f(b)-f(a)}{b-a}$.*

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 $\mathcal{E}'$.* For each $\mathfrak{e} \in \mathcal{E}'$, we find the largest constant $t_{\mathfrak{e}} \in \mathbb{R}$ such that the logical formula $\forall \mathbf{x} \in \mathbb{R}^N. (\mathbf{x} \in \mathsf{Sat}(\Gamma) \to \widetilde{\mathfrak{e}}(\mathbf{x}) \geq t_{\mathfrak{e}})$ 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 $\mathcal{E}'$.* For each pair $(\mathfrak{e}, \mathfrak{e}') \in \mathcal{E}' \times \mathcal{E}'$ 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

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

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

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 $\mathcal{E}'$.* For each pair $(\mathfrak{e}, \mathfrak{e}') \in \mathcal{E}' \times \mathcal{E}'$ 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

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

holds. The algorithm first finds the minimal value $\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}$ over all feasible $(\mathsf{r}_{\mathfrak{e},\mathfrak{e}'}, \mathsf{b}_{\mathfrak{e},\mathfrak{e}'})$'s, then finds the minimal $\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}$ over all feasible $(\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}, \mathsf{b}_{\mathfrak{e},\mathfrak{e}'})$'s. If such $\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}$ does not exists, the algorithm simply leaves $\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}$ undefined. Note that once $\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}$ exists and $\mathsf{Sat}(\Gamma)$ is non-empty, then $\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}$ exists since $\mathsf{b}_{\mathfrak{e},\mathfrak{e}'}$ cannot be arbitrarily small once $\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}$ is fixed.

(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) *Constraints from Exponentiation.* For each variable $v_{\mathfrak{e}}$, the algorithm adds into Γ polynomial expressions $v_{\mathfrak{e}} - t_{\mathfrak{e}}^{r-1} \cdot \widetilde{\mathfrak{e}}$ and $v_{\mathfrak{e}} - t_{\mathfrak{e}}^{r}$ due to the definition of $t_{\mathfrak{e}}$. And for each variable $v'_{\mathfrak{e}}$, the algorithm adds (i) $v'_{\mathfrak{e}} - t_{\mathfrak{e}}^{r-1}$ and (ii) either $v'_{\mathfrak{e}} - t_{\mathfrak{e}}^{r-2} \cdot \tilde{\mathfrak{e}}$ when $r \geq 2$ or $\widetilde{\mathfrak{e}} - t_{\mathfrak{e}}^{2-r} \cdot v'_{\mathfrak{e}}$ when $1 < r < 2$.

(6) *Mutual No-Smaller-Than Inequalities over $u_{\mathfrak{e}}$'s.* For each pair $(\mathfrak{e}, \mathfrak{e}') \in \mathcal{E}' \times \mathcal{E}'$ 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 (Theorem 4.5) 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 $t_{\mathfrak{e}'}$.

(7) *Mutual No-Greater-Than Inequalities over $u_{\mathfrak{e}}$'s.* For each pair $(\mathfrak{e},\mathfrak{e}') \in \mathcal{E}' \times \mathcal{E}'$ such that $\mathfrak{e} \neq \mathfrak{e}'$ and $\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}, \mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}$ are successfully found and $\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'} > 0$, the algorithm adds

$$u_{\mathfrak{e}'} + \ln \mathsf{r}^*_{\mathfrak{e},\mathfrak{e}} - u_{\mathfrak{e}} + \mathbf{1}_{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'} \geq 0} \cdot t_{\mathfrak{e}'}^{-1} \cdot \frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}}$$

into Γ. This is because $\left(\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'} \cdot [\![\mathfrak{e}']\!] + \mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}\right) - [\![\mathfrak{e}]\!] \geq 0$ implies

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

where the last step is obtained from Lagrange's Mean-Value Theorem and by distinguishing whether $\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'} \geq 0$ or not. Note that one has

$$t_{\mathfrak{e}'} + \frac{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}}{\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}} \geq 1$$

due to the maximal choice of $t_{\mathfrak{e}'}$ and the fact that $\widetilde{\mathfrak{e}}$ (as a polynomial function) is everywhere greater than or equal to 1 under $\mathsf{Sat}(\Gamma)$ (cf. (§)).

(8) *Mutual No-Smaller-Than Inequalities over $v_{\mathfrak{e}}$'s.* For each pair of variables of the form $(v_{\mathfrak{e}}, v_{\mathfrak{e}'})$ such that $\mathfrak{e} \neq \mathfrak{e}'$, $r^*_{\mathfrak{e},\mathfrak{e}'}, b^*_{\mathfrak{e},\mathfrak{e}'}$ are successfully found and $r^*_{\mathfrak{e},\mathfrak{e}'} > 0, b^*_{\mathfrak{e},\mathfrak{e}'} \geq 0$, the algorithm adds

$$v_{\mathfrak{e}} - \left(r^*_{\mathfrak{e},\mathfrak{e}'}\right)^r \cdot \left(v_{\mathfrak{e}'} + r \cdot \frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}} \cdot v'_{\mathfrak{e}'} \right)$$

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

$$\begin{aligned}
[\![\mathfrak{e}]\!]^r &\geq \left(r^*_{\mathfrak{e},\mathfrak{e}'}\right)^r \cdot \left([\![\mathfrak{e}']\!] + \frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}} \right)^r \\
&\geq \left(r^*_{\mathfrak{e},\mathfrak{e}'}\right)^r \cdot \left([\![\mathfrak{e}']\!]^r + \left([\![\mathfrak{e}']\!] + \frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}} \right)^r - [\![\mathfrak{e}']\!]^r \right) \\
&\geq \left(r^*_{\mathfrak{e},\mathfrak{e}'}\right)^r \cdot \left([\![\mathfrak{e}']\!]^r + r \cdot [\![\mathfrak{e}']\!]^{r-1} \cdot \frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}} \right) .
\end{aligned}$$

where the last step is obtained from Lagrange's Mean-Value Theorem.

(9) *Mutual No-Greater-Than Inequalities over $v_{\mathfrak{e}}$'s.* For each pair of variables of the form $(v_{\mathfrak{e}}, v_{\mathfrak{e}'})$ such that $\mathfrak{e} \neq \mathfrak{e}'$, $\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}, \mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}$ are successfully found and $\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'} > 0, \mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'} \geq 0$, the

algorithm adds

$$\left(\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}\right)^r \cdot \left(v_{\mathfrak{e}'} + \left(\mathbf{1}_{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'} \le 0} + \mathbf{1}_{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'} > 0} \cdot M^{r-1} \right) \cdot r \cdot \frac{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}}{\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}} \cdot v'_{\mathfrak{e}'} \right) - v_{\mathfrak{e}}$$

into Γ, where $M := \frac{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}}{\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'} \cdot t_{\mathfrak{e}'}} + 1$. This is due to the fact that $\left(\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'} \cdot [\![\mathfrak{e}']\!] + \mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}\right) - [\![\mathfrak{e}]\!] \ge 0$ implies

$$\begin{aligned}
[\![\mathfrak{e}]\!]^r &\le \left(\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}\right)^r \cdot \left([\![\mathfrak{e}']\!] + \frac{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}}{\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}} \right)^r \\
&\le \left(\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}\right)^r \cdot \left([\![\mathfrak{e}']\!]^r + \left([\![\mathfrak{e}']\!] + \frac{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}}{\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}} \right)^r - [\![\mathfrak{e}']\!]^r \right) \\
&\le \left(\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}\right)^r \cdot \left([\![\mathfrak{e}']\!]^r + \left(\mathbf{1}_{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'} \le 0} + \mathbf{1}_{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'} > 0} \cdot M^{r-1} \right) \cdot r \cdot \frac{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}}{\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}} \cdot [\![\mathfrak{e}']\!]^{r-1} \right)
\end{aligned}$$

where the last step is obtained from Lagrange's Mean-Value Theorem and the fact that $[\![\mathfrak{e}']\!] \ge t_{\mathfrak{e}'}$ implies $[\![\mathfrak{e}']\!] + \frac{\mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'}}{\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}} \le M \cdot [\![\mathfrak{e}']\!]$.

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 $(\mathsf{f}, \bigwedge_{h \in \Gamma} h \ge 0, \widetilde{\mathfrak{e}^*})$.

Example 4.6. We continue with Example 4.4. In Step 4(b), the algorithm establishes the following non-trivial inequalities:

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

After Step 4(b), Γ_i's ($1 \le i \le 3$) 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 6. 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. □

Remark 7. The aim of Step 4(b) is to approximate logarithmic and exponentiation terms by linear inequalities they satisfy. In the final step, those linear inequalities suffice to solve our problem. As to extensibility, Step 4(b) may be extended to other non-polynomial terms by constructing similar linear inequalities they satisfy. □

Remark 8 (Complexity). As established in Remark 5, this step of the algorithm has to process polynomially-many constraint triples, each of which has a polynomial size. For each triple, the number of inequalities generated by the algorithm is also polynomially-bounded. To generate each inequality, the algorithm might also call an LP-solver for solving a linear program of

polynomial size (due to Farkas' Lemma). Hence, in total, this step of the algorithm takes polynomial time in $O(|P|)$. Moreover, for each generated Γ, we have $|\Gamma|$ is polynomial in $|P|$.

4.5 Step 5 of SynAlgo

In the previous step, we eliminated all instances of non-polynomial terms by replacing them with stand-alone variables and linear inequalities over those variables. Hence, we only have polynomials in the constraint triples. However, to cope with polynomial constraints we need to solve formulas in the first-order theory of reals in general, resulting in high computational complexity. To avoid this, we transform the validity of the formula into a system of linear inequalities over template variables through Handelman's Theorem, and then use linear programming to solve the coefficients in the template. Handelman's Theorem is a characterization of positive polynomials over polytopes. Here, we utilize its sound form as a sufficient but not complete method to prove the validity of the constraint triples. The use of Handelman's Theorem reduces the solution of the constraint triples to linear programming, thus ensuring polynomial-time complexity for our algorithm.

Below we first present Handelman's Theorem.

Definition 4.7 (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: $\mathit{Monoid}(\Gamma) := \left\{ \prod_{i=1}^{k} h_i \mid k \in \mathbb{N}_0 \text{ and } h_1, \ldots, h_k \in \Gamma \right\}$.

Theorem 4.8 (Handelman's Theorem [40]). *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) $g(\mathbf{x}) > 0$ for all $\mathbf{x} \in \mathsf{Sat}(\Gamma)$, 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 $u_1, \ldots, u_n \in \mathit{Monoid}(\Gamma)$.

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. The following proposition shows that Eq. (3) encompasses a simple proof system for non-negative polynomials over polyhedra.

Proposition 4.9. *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. Let the collection of deduction systems $\{\vdash_k\}_{k \in \mathbb{N}}$ be generated by the following rules:*

$$\frac{h \in \Gamma}{\vdash_1 h \geq 0} \qquad \frac{c \in \mathbb{R}, c \geq 0}{\vdash_1 c \geq 0} \qquad \frac{\vdash_k h \geq 0, c \in \mathbb{R}, c \geq 0}{\vdash_k c \cdot h \geq 0}$$

$$\frac{\vdash_k h_1 \geq 0, \vdash_k h_2 \geq 0}{\vdash_k h_1 + h_2 \geq 0} \qquad \frac{\vdash_{k_1} h_1 \geq 0, \vdash_{k_2} h_2 \geq 0}{\vdash_{k_1+k_2} h_1 \cdot h_2 \geq 0}.$$

Then for all $k \in \mathbb{N}$ and polynomials $g \in \mathfrak{R}[x_1, \ldots, x_m]$, if $\vdash_k g \geq 0$ then $g = \sum_{i=1}^{n} c_i \cdot u_i$ for some $n \in \mathbb{N}$, non-negative real numbers $c_1, \ldots, c_n \geq 0$ and $u_1, \ldots, u_n \in \mathit{Monoid}(\Gamma)$ such that every u_i is a product of no more than k polynomials in Γ.

Proof. By an easy induction on k. □

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.

Example 4.10. We continue with Example 4.6. 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 3.6, $\overline{T}(\mathsf{f}, 1, n) \leq \eta(\mathsf{f}, 1, n) = \frac{2}{\ln 2} \cdot \ln n + 2$. It follows that Binary-Search runs in $\mathcal{O}(\log n)$ in worst-case. □

Remark 9 (Complexity). As established in Remark 8, we have $|\Gamma|$ is polynomial. Moreover, in Eq. (3), we are only considering those monoid elements whose degree is bounded by the constant k. Hence, we have polynomially-many distinct monoid elements. Therefore, this step generates polynomially many linear (in)equalities for each constraint triple. Given that we have polynomially constraint triples (Remark 5), the overall size of the linear program generated is polynomial in $O(|P|)$.

Remark 10 (Overall Complexity). Based on Remarks 3, 4, 5, 8 and 9, the overall complexity of our approach is polynomial in $O(|P|)$ plus the time needed for solving linear programming instance of size polynomial in $O(|P|)$. It is well-known that LPs can be solved in polynomial time. Hence, our approach is efficient and runs in polynomial time as well.

4.6 Soundness of SynAlgo

We now state the soundness of our approach for synthesis of measure functions.

Theorem 4.11. *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.*

Proof. The proof follows from the facts that (i) once the logical formulae encoded by the constraint triples (cf. semantics of constraint triples specified in Step 3) are fulfilled by template variables, then $\widehat{[\![\eta]\!]}$ (obtained in Step 2, with η being the template established in Step 1) satisfies the conditions specified in Proposition 3.9, (ii) the variable abstraction (i.e., the linear inequalities) in Step 4 is a sound over-approximation for floored expressions, logarithmic terms and exponentiation terms, and (iii) Handelman's Theorem provides a sound form for positive polynomials over polyhedra. □

Remark 11. While Step 4 transforms logarithmic and exponentiation terms to polynomials, in Step 5 we need a sound method to solve polynomials with linear programming. We achieve this with Handelman's Theorem.

Remark 12. We remark three aspects of our algorithm.

(1) *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. In particular, as shown in Remark 10, our approach only needs to solve LP instance of size $O(|P|)$ and has an additional linear, i.e. $O(|P|)$, overhead for performing all other steps.
(2) *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.
(3) *Polynomial Bounds.* Besides polynomial bounds, our approach can also be applied to synthesis of normal polynomial complexity bounds. This is easily done by a degeneration of our approach such that (i) in Step 1 of SynAlgo non-polynomial terms are dropped in a template and (ii) in Step 4 one does not need abstraction of non-polynomial terms and extra linear constraints for abstraction variables.
(4) *While Loops.* Although we focus on recursion, our approach also covers iterative algorithms with while loops. When restricted to non-recursive programs, our approach coincides with the approach of ranking functions, and is thus sound and complete in general. Additionally, our approach can synthesize both polynomial and non-polynomial ranking functions, as opposed to previous approaches that can only synthesize polynomial ranking functions (e.g. [26]).

5 ILLUSTRATION ON MERGE-SORT

In this section, we present a step-by-step application of our entire method (from syntax, to semantics, to all the steps of the algorithm) to the classical Merge-Sort algorithm. This illustrates the quite involved aspects of our approach.

5.1 Program Implementation

Figure 4 represents an implementation for the Merge-Sort algorithm [25, Chapter 2] in our language. The numbers on the leftmost side are the labels assigned to statements which represent program counters, where the function mergesort starts from label 1 and ends at 7, and merge starts from label 8 and ends at 22.

```
mergesort(i, j) {
1:   if 1 ≤ i and i ≤ j − 1 then
2:       k := ⌊(i+j)/2⌋;
3:       mergesort(i, k);
4:       mergesort(k + 1, j);
5:       merge(i, j, k)
6:     else skip
     fi
7: }
   merge(i, j, k) {
8:   m := i;
9:   n := k + 1;
10:  l := i;
11:   while l ≤ j do
12:     if ⋆ then
13:         skip;
14:         m := m + 1
       else
15:         skip;
16:         n := n + 1
       fi;
17:    l := l + 1
     od;
18: l := i;
19: while l ≤ j do
20:    skip;
21:    l := l + 1
     od
22: }
```

Fig. 4. A program that implements Merge-Sort

In detail, the scalar variable i (resp. j) in both the parameter list of mergesort and that of merge represents the start (resp. the end) of the array index; the scalar variable k in the parameter list of merge represents the separating index for merging two sub-arrays between i, j; moreover, in merge, the demonic nondeterministic branch (at program counter 12) abstracts the comparison between array entries and the **skip**'s at program counters $13, 15, 20$ represent corresponding array assignment statements in a real implementation for Merge-Sort.

The replacement of array-relevant operations with either **skip** or demonic non-determinism preserves worst-case complexity as demonic non-determinism over-approximates conditional branches involving array entries.

i	f_i
1	$k \leftarrow \lfloor(\overline{i}+\overline{j})/2\rfloor$
2	$(i,j) \leftarrow (\overline{i},\overline{k})$
3	$(i,j) \leftarrow (\overline{k}+1,\overline{j})$
4	$(i,j,k) \leftarrow (\overline{i},\overline{j},\overline{k})$

i	g_i
1	$m \leftarrow \overline{i}$
2	$n \leftarrow \overline{k}+1$
3	$l \leftarrow \overline{i}$
4	$m \leftarrow \overline{m}+1$
5	$n \leftarrow \overline{n}+1$
6	$l \leftarrow \overline{l}+1$

Fig. 5. Illustration for Fig. 6 and Fig. 7

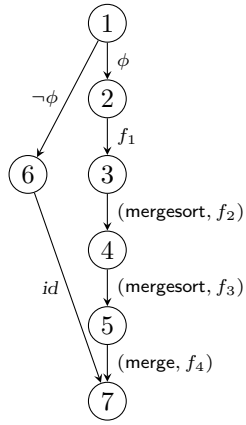

Fig. 6. The part of CFG for mergesort in Fig. 4

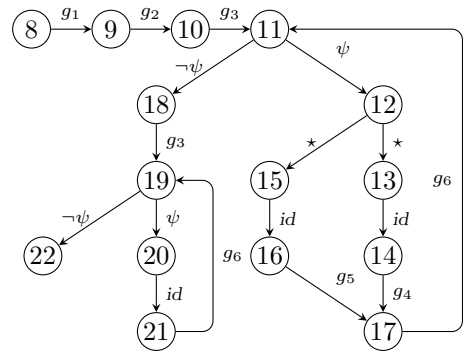

Fig. 7. The part of CFG for merge in Fig. 4

5.2 Control Flow Graph and Significant Labels

To begin the analysis, the algorithm must first obtain the control-flow graph (CFG) of the program. This is depicted in Figures 5, 6 and 7.

For brevity we define $\phi := 1 \leq i \wedge i \leq j-1$, $\psi := l \leq j$, and let id indicate the identity function. The update functions (f_i's and g_i's) are given in Figure 5. Moreover, in Figure 5, any $\overline{q}$ is the concrete entity held by the scalar variable q under the valuation at runtime, and every function is represented in the form "$p \leftarrow q$" meaning "q assigned to p", where only the relevant variable is shown for assignment functions.

Figure 6 shows that mergesort (cf. Fig. 4) starts from the if-branch with guard ϕ (label 1). Then if ϕ is satisfied (i.e., the length of the array is greater than one), the program steps into the recursive step which is composed of labels 2–5; otherwise, the program proceeds to terminal label 7 through 6 with nothing done.

Figure 7 shows that merge (cf. Fig. 4) starts from a series of assignments (labels 8–10) and then enters the while-loop (labels 11–17) which does the merging of two sub-arrays; after the while-loop starting from 11, the program enters the part for copying array-content back (labels 18–21) and finally enters the terminal label 22.

The types of labels are straightforward. In mergesort, labels $2, 6$ are assignment labels, label 1 is a branching label, labels 3–5 are call labels. In merge, labels 8–$10, 13$–$18, 20$–21 are assignment labels, labels $11, 19$ are branching labels, and label 12 is a demonic non-deterministic point.

In this program, significant labels are 1 and 8 (beginning points of functions) and 11 and 19 (while loop headers).

f	ℓ	$I(\mathsf{f},\ell)$
mergesort	1	$i \geq 0 \wedge j \geq i$
merge	8	$i \geq 0 \wedge j \geq i$
merge	11	$l \geq i \wedge l \leq j+1$
merge	19	$l \geq i \wedge l \leq j+1$

Table 1. Invariants Used for Significant Labels

f	ℓ	$\eta(\mathsf{f},\ell)$
mergesort	1	$c_1(j-i+1)\ln(j-i+1)+c_2$
merge	8	$c_3(j-i+1)+c_4$
merge	11	$c_5 l + c_6 j + c_7 i + c_8$
merge	19	$c_9 l + c_{10} j + c_{11}$

Table 2. Templates at Significant Labels

5.3 Step 1: Invariants and Templates

As mentioned in Section 3, automatic obtaining of invariants is a standard problem with several known techniques[20, 27]. Since invariant generation is not a main part of our algorithm, we simply use the straightforward invariants in Table 1 to demonstrate our algorithm.

The algorithm constructs a template $\eta(\mathsf{f},\ell,\boldsymbol{.})$ of the form shown in (1) at every significant label. Due to the large amount of space needed to illustrate the method on the complete template, for the sake of this example, we restrict our templates to the forms presented in Table 2 instead. Since a template with this restricted form is feasible, the whole template consisting of all products of pairs of terms is also feasible. Here the c_i's are variables that the algorithm needs to synthesize.

Note that the results reported in Section 6 were obtained by our implementation in the general case, where the templates were not given as part of input and were generated automatically by our algorithm in their full form according to (1).

5.4 Step 2: Computation of $\widehat{[\![\eta]\!]}$

In this step, the algorithm expands the η generated for significant labels in the previous step to obtain $\widehat{[\![\eta]\!]}$ for all labels. This results in a function of form (2). In order to make equations easier to read, we use the notation $\binom{\phi}{p}$ to denote $\mathbf{1}_\phi \cdot p$ in this section. We also use $\widehat{[\![\eta]\!]}(\mathsf{f},\ell)[p \leftarrow \mathfrak{e}]$ to denote the result of replacing every occurrence of the variable p in $\widehat{[\![\eta]\!]}(\mathsf{f},\ell)$ with the expression $\mathfrak{e}$.

End points of functions. The algorithm sets $\widehat{[\![\eta]\!]}(\mathsf{mergesort},7)$ and $\widehat{[\![\eta]\!]}(\mathsf{merge},22)$ to 0, since these correspond to the terminal labels of their respective functions.

Significant Labels. For each significant label ℓ of a function f, the algorithm, by definition, sets $\widehat{[\![\eta]\!]}(\mathsf{f},\ell)$ to:

$$\binom{I(\mathsf{f},\ell)}{\eta(\mathsf{f},\ell)} + \binom{\neg I(\mathsf{f},\ell)}{0}.$$

Therefore we have:

$$\widehat{[\![\eta]\!]}(\mathsf{mergesort},1) = \binom{i \geq 0 \wedge j \geq i}{c_1(j-i+1)\ln(j-i+1)+c_2} + \binom{i<0 \vee j<i}{0},$$
$$\widehat{[\![\eta]\!]}(\mathsf{merge},8) = \binom{i \geq 0 \wedge j \geq i}{c_3(j-i+1)+c_4} + \binom{i<0 \vee j<i}{0},$$
$$\widehat{[\![\eta]\!]}(\mathsf{merge},11) = \binom{l \geq i \wedge l \leq j+1}{c_5 l + c_6 j + c_7 i + c_8} + \binom{l<i \vee l>j+1}{0},$$
$$\widehat{[\![\eta]\!]}(\mathsf{merge},19) = \binom{l \geq i \wedge l \leq j+1}{c_9 l + c_{10} j + c_{11}} + \binom{l<i \vee l>j+1}{0}.$$

Expansion to Other Labels. By applying C1-C5 the algorithm calculates $\widehat{[\![\eta]\!]}$ for all labels in the following order:

$$\begin{aligned}
&\widehat{[\![\eta]\!]}(\mathsf{mergesort},6) = 1 + \widehat{[\![\eta]\!]}(\mathsf{mergesort},7),\\
&\widehat{[\![\eta]\!]}(\mathsf{mergesort},5) = 1 + \widehat{[\![\eta]\!]}(\mathsf{mergesort},7) + \widehat{[\![\eta]\!]}(\mathsf{merge},8),\\
&\widehat{[\![\eta]\!]}(\mathsf{mergesort},4) = 1 + \widehat{[\![\eta]\!]}(\mathsf{mergesort},5) + \widehat{[\![\eta]\!]}(\mathsf{mergesort},1)[i \leftarrow k+1],\\
&\widehat{[\![\eta]\!]}(\mathsf{mergesort},3) = 1 + \widehat{[\![\eta]\!]}(\mathsf{mergesort},4) + \widehat{[\![\eta]\!]}(\mathsf{mergesort},1)[j \leftarrow k],\\
&\widehat{[\![\eta]\!]}(\mathsf{mergesort},2) = 1 + \widehat{[\![\eta]\!]}(\mathsf{mergesort},3)[k \leftarrow \lfloor (i+j)/2 \rfloor],
\end{aligned}$$

$$\begin{aligned}
&\widehat{[\![\eta]\!]}(\mathsf{merge},21) = 1 + \widehat{[\![\eta]\!]}(\mathsf{merge},19)[l \leftarrow l+1],\\
&\widehat{[\![\eta]\!]}(\mathsf{merge},20) = 1 + \widehat{[\![\eta]\!]}(\mathsf{merge},21),\\
&\widehat{[\![\eta]\!]}(\mathsf{merge},18) = 1 + \widehat{[\![\eta]\!]}(\mathsf{merge},19)[l \leftarrow i],\\
&\widehat{[\![\eta]\!]}(\mathsf{merge},17) = 1 + \widehat{[\![\eta]\!]}(\mathsf{merge},11)[l \leftarrow l+1],\\
&\widehat{[\![\eta]\!]}(\mathsf{merge},16) = 1 + \widehat{[\![\eta]\!]}(\mathsf{merge},17)[n \leftarrow n+1],\\
&\widehat{[\![\eta]\!]}(\mathsf{merge},15) = 1 + \widehat{[\![\eta]\!]}(\mathsf{merge},16),\\
&\widehat{[\![\eta]\!]}(\mathsf{merge},14) = 1 + \widehat{[\![\eta]\!]}(\mathsf{merge},17)[m \leftarrow m+1],\\
&\widehat{[\![\eta]\!]}(\mathsf{merge},13) = 1 + \widehat{[\![\eta]\!]}(\mathsf{merge},14),\\
&\widehat{[\![\eta]\!]}(\mathsf{merge},12) = 1 + \max\left\{\widehat{[\![\eta]\!]}(\mathsf{merge},13), \widehat{[\![\eta]\!]}(\mathsf{merge},15)\right\},\\
&\widehat{[\![\eta]\!]}(\mathsf{merge},10) = 1 + \widehat{[\![\eta]\!]}(\mathsf{merge},11)[l \leftarrow i],\\
&\widehat{[\![\eta]\!]}(\mathsf{merge},9) = 1 + \widehat{[\![\eta]\!]}(\mathsf{merge},10)[n \leftarrow k+1].
\end{aligned}$$

The concrete values of $\widehat{[\![\eta]\!]}$'s calculated as above are too long to present in the paper. Therefore we only include some of them to illustrate the method:

$$\widehat{[\![\eta]\!]}(\mathsf{mergesort},6) = 1,$$
$$\widehat{[\![\eta]\!]}(\mathsf{merge},21) = \binom{l+1 \geq i \wedge l \leq j}{c_9 l + c_9 + c_{10} j + c_{11} + 1} + \binom{l+1<i \vee l>j}{1},$$

$$\widehat{[\![\eta]\!]}(\mathsf{merge}, 20) = \begin{pmatrix} l+1 \geq i \wedge l \leq j \\ c_9 l + c_9 + c_{10} j + c_{11} + 2 \end{pmatrix} + \begin{pmatrix} l+1 < i \vee l > j \\ 2 \end{pmatrix},$$
$$\widehat{[\![\eta]\!]}(\mathsf{merge}, 18) = \begin{pmatrix} i \geq i \wedge i \leq j+1 \\ c_9 i + c_{10} j + c_{11} + 1 \end{pmatrix} + \begin{pmatrix} i < i \vee i > j+1 \\ 1 \end{pmatrix}.$$

The algorithm utilizes Farkas lemma to simplify the $\widehat{[\![\eta]\!]}$'s and remove unnecessary terms. Hence, at the end of this step we will have:

$$\widehat{[\![\eta]\!]}(\mathsf{merge}, 18) = \begin{pmatrix} i \leq j+1 \\ c_9 i + c_{10} j + c_{11} + 1 \end{pmatrix} + \begin{pmatrix} i > j+1 \\ 1 \end{pmatrix}.$$

5.5 Step 3: Establishment of Constraint Triples

A constraint triple $(\mathsf{f}, \phi, \mathfrak{e})$ denotes $\forall \nu \in \mathit{Val}_{\mathsf{f}}.\, (\nu \models \phi \rightarrow [\![\mathfrak{e}]\!](\nu) \geq 0)$, i.e., each triple consists of a function name, a precondition ϕ and an expression $\mathfrak{e}$. This means that the unknown variables (c_i's) should be assigned values in a way that whenever ϕ is satisfied, we have $\mathfrak{e} \geq 0$. When several triples are obtained, we group them together in a conjunctive manner. Note that $(\mathsf{f}, \phi \vee \psi, \mathfrak{e})$ can be broken into two triples $(\mathsf{f}, \phi, \mathfrak{e})$ and $(\mathsf{f}, \psi, \mathfrak{e})$.

The function $\widehat{[\![\eta]\!]}$, by definition, satisfies the conditions of a measure function in all non-significant labels. In order to obtain a correct measure function, we need to make sure that these conditions are fulfilled in significant labels, too. Concretely, the algorithm has to find c_i's such that the following inequalities hold and therefore converts each of these inequalities to a series of constraint triples.

$\widehat{[\![\eta]\!]}(\mathsf{mergesort}, 1) \geq 0,$
$\widehat{[\![\eta]\!]}(\mathsf{merge}, 8) \geq 0,$
$\widehat{[\![\eta]\!]}(\mathsf{merge}, 11) \geq 0,$
$\widehat{[\![\eta]\!]}(\mathsf{merge}, 19) \geq 0,$

$\widehat{[\![\eta]\!]}(\mathsf{mergesort}, 1) \geq 1 + \mathbf{1}_{1 \leq i \leq j-1} \cdot \widehat{[\![\eta]\!]}(\mathsf{mergesort}, 2) + \mathbf{1}_{i<1 \vee i>j-1} \widehat{[\![\eta]\!]}(\mathsf{mergesort}, 6),$
$\widehat{[\![\eta]\!]}(\mathsf{merge}, 8) \geq 1 + \widehat{[\![\eta]\!]}(\mathsf{merge}, 9)[m \leftarrow i],$
$\widehat{[\![\eta]\!]}(\mathsf{merge}, 11) \geq 1 + \mathbf{1}_{l \leq j} \widehat{[\![\eta]\!]}(\mathsf{merge}, 12) + \mathbf{1}_{l>j} \widehat{[\![\eta]\!]}(\mathsf{merge}, 18),$
$\widehat{[\![\eta]\!]}(\mathsf{merge}, 19) \geq 1 + \mathbf{1}_{l \leq j} \widehat{[\![\eta]\!]}(\mathsf{merge}, 20) + \mathbf{1}_{l>j} \widehat{[\![\eta]\!]}(\mathsf{merge}, 22).$

The first group of inequalities above dictate the non-negativity of the obtained measure function and the second group assure that it satisfies conditions C2'–C4'.

In this case, our algorithm creates a lot of triples and then uses Farkas lemma to simplify them and ignore triples that have contradictory preconditions or expressions that are always non-negative. Here we illustrate how some of these triples are obtained to cover the main ideas.

For example, since we must have $\widehat{[\![\eta]\!]}(\mathsf{merge}, 8) \geq 0$, therefore:

$$\begin{pmatrix} i \geq 0 \wedge j \geq i \\ c_3(j - i + 1) + c_4 \end{pmatrix} + \begin{pmatrix} i < 0 \vee j < i \\ 0 \end{pmatrix} \geq 0.$$

This leads to the creation of three triples. The first part leads to

$$T_1 := (\mathsf{merge}, i \geq 0 \wedge j \geq i, c_3(j-i+1)+c_4),$$

and the second part leads to the following two triples that both get ignored because their expression is nonnegative even without considering the precondition: $(\mathsf{merge}, i < 0, 0), (\mathsf{merge}, j < i, 0)$.

As a more involved example, the following is a term in $\widehat{[\![\eta]\!]}(\mathsf{merge}, 2)$:

$$\tau := \begin{pmatrix}\phi_\tau \\ \mathfrak{e}_\tau\end{pmatrix} := \begin{pmatrix} j-d-1 \geq 0 \wedge d-i \geq 0 \wedge i \geq 0 \\ c_1u_3d - c_1u_3i + c_1u_3 + 2c_2 + c_1u_4j - c_1u_4d + c_3j - c_3i + c_3 + c_4 + 4\end{pmatrix},$$

here $d := \lfloor (i+j)/2 \rfloor$, $u_3 := \ln(d-i+1)$ and $u_4 := \ln(j-d)$. Note that at this stage, the term above is stored and used in its full expanded form by the algorithm and these definitions are only used in this illustration to shorten the length of this term. In step 4, the algorithm will create these variables and use the shortened representation from there on.

Since the inequality $\widehat{[\![\eta]\!]}(\mathsf{mergesort}, 1) \geq 1 + \mathbf{1}_{1\leq i\leq j-1} \cdot \widehat{[\![\eta]\!]}(\mathsf{mergesort}, 2)$ should be satisfied, we get:

$$\begin{pmatrix} 0 \leq i \leq j \\ c_1(j-i+1)\ln(j-i+1)+c_2\end{pmatrix} + \begin{pmatrix} i<0 \vee j<i \\ 0\end{pmatrix} \geq \mathbf{1}_{1\leq i\leq j-1}(\tau+1).$$

This leads to the following constraint triples:

$$(\mathsf{mergesort}, 0 \leq i \leq j \wedge 1 \leq i \leq j-1 \wedge \phi_\tau, c_1(j-i+1)\ln(j-i+1)+c_2-\mathfrak{e}_\tau-1),$$

$$(\mathsf{mergesort}, i<0 \wedge 1 \leq i \leq j-1 \wedge \phi_\tau, -\mathfrak{e}_\tau-1),$$

$$(\mathsf{mergesort}, j<i \wedge 1 \leq i \leq j-1 \wedge \phi_\tau, -\mathfrak{e}_\tau-1).$$

The last two triples are discarded because the algorithm uses Farkas Lemma and deduces that their conditions are unsatisfiable. The first triple is also simplified using Farkas Lemma. This leads to the following final triple:

$$T_2 := (\mathsf{mergesort}, 1 \leq i \leq j-1 \wedge \phi_\tau, c_1(j-i+1)\ln(j-i+1)+c_2-\mathfrak{e}_\tau-1).$$

All other inequalities are processed in a similar manner. In this case, at the end of this stage, the algorithm has generated 19 simplified constraint triples.

5.6 Step 4: Converting Constraint Triples to Linear Inequalities

This is the main step of the algorithm. Now that the triples are obtained, we need to solve the system of triples to get concrete values for c_i's and hence upper-bounds on the runtime of functions, but this is a non-linear optimization problem. At this step, we introduce new variables and use them to obtain linear inequalities.

Step 4(a): Abstraction of Logarithmic, Exponentiation and Floored Expressions. At this stage the algorithm defines the following variables and replaces them in all constraint triples to

obtain a short representation like the one we used for τ:

$$\begin{aligned} d &:= \lfloor \tfrac{i+j}{2} \rfloor, \\ u_0 &:= \ln(j-i+1), \\ u_1 &:= \ln(j-k), \\ u_2 &:= \ln(k-i+1), \\ u_3 &:= \ln(d-i+1), \\ u_4 &:= \ln(j-d). \end{aligned}$$

In the description we also use w to denote $i+j$, but the algorithm does not add this variable. Note that all logarithmic, floored and exponentiation terms are replaced by the new variables above and hence all triples become linear, but in order for them to reflect the original triples, new conditions should be added to them. To each triple T, we assign a set Γ of linear polynomials such that their non-negativity captures the desired conditions. For example, for the triple T_1 above, we have $\Gamma_1 = \{i, j-i\}$, because the only needed conditions are $i \geq 0$ and $j \geq i$ and since no new variable has appeared in this triple, there is no need for any additional constraint.

On the other hand, for T_2, we start by setting $\Gamma_2 = \{i-1, j-d-1, d-i\}$. Note that we could add $j-i-1$ and i to Γ_2, too, but these can be deduced from the rest and hence the algorithm simplifies Γ_2 and discards them.

Since the variables u_0, u_3 and u_4 appear in T_2 and are non-negative logarithmic terms, the algorithm adds u_0, u_3, u_4 to Γ_2.

Then the algorithm adds $w - 2d = i + j - 2d$ and $2d - w + 1 = 2d - i - j + 1$ to Γ_2. This is due to the definition of d as $\lfloor w/2 \rfloor$. Then it performs an emptiness checking to discard the triple if it has already become infeasible.

Step 4(b): Linear Constraints for Abstracted (Logarithmic) Variables. We find constraints for every logarithmic variable. As an example, since $j \geq d+1$ and $d \geq i$, it can be inferred that $j - i + 1 \geq 2$. This is the tightest obtainable bound and the algorithm finds it using Farkas Lemma. Now since $2 < e$ and $u_0 = \ln(j-i+1)$, the algorithm adds $(j-i+1) - eu_0$ to Γ_2 according to part (4) of Step 4(b). Similar constraints are added for other variables.

Next the algorithm adds constraints on the relation between u_i's to Γ_2 as in parts (6) and (7). For example, since $\Gamma_2 \geq 0$ implies $(j-i+1) - 2(j-d) \geq 0$ and $2(j-d) \geq 1$, the algorithm infers that $u_0 - \ln 2 - u_4$ is non-negative and adds it to Γ_2 according to part (6).

Other constraints, and constraints for exponentiation variables, if present, will also be added according to step 4(b).

5.7 Step 5: Solving Unknown Coefficients in the Template (c_i's)

After creating Γ's, the algorithm attempts to find suitable values for the variables c_i such that for each triple $T = (\mathfrak{f}, \phi, \mathfrak{e})$ and its corresponding Γ, it is the case that $\Gamma \geq 0$ implies $\mathfrak{e} \geq 0$. Since all elements of Γ are linear, we can use Handelman's theorem to reduce this problem to an equivalent system of linear inequalities. The algorithm does this for every triple and then appends all the resulting systems of linear inequalities together in a conjunctive manner and uses an LP-Solver to solve it. In this case the final result, obtained from `lpsolve` is as follows:

variable	value	variable	value
c_1	40.9650	c_7	-3
c_2	3	c_8	12
c_3	9	c_9	-3
c_4	6	c_{10}	3
c_5	-6	c_{11}	4
c_6	9		

This means that the algorithm successfully obtained the upper-bound

$$40.9650(j - i + 1)\ln(j - i + 1) + 3$$

for the given Merge-Sort implementation.

For simplicity of illustration we do not show how to obtain a better leading constant. In the illustration above we show for constant 40.9650, whereas our approach and implementation can obtain the constant 25.02 (as reported in Table 5).

6 EXPERIMENTAL RESULTS

Our approach can handle a wide variety of programs. Specifically, even though a major novelty of our method is that it enables synthesis of non-polynomial bounds, it is equally effective in finding polynomial bounds. In this section, we first report on an implementation of our approach and then provide three categories of experimental results obtained from this implementation: (i) bounds for recursive programs, (ii) polynomial bounds and (iii) non-polynomial bounds. These examples show that our sound approach can synthesize non-trivial worst-case complexity bounds for several classical algorithms.

6.1 Implementation and Experimental Details

Implementation. We implemented our approach in Java. The code and instructions for executing it are available at http://ist.ac.at/~akafshda/NPWCARP. Our code generates a set of linear constraints and then uses lp_solve [58] through JavaILP [59] for solving linear programs. Our implementation successfully passed artifact evaluation phase of the International Conference on Computer-Aided Verification (CAV) [15].

Invariants. Our tool uses the Stanford Invariant Generator (Sting) [62] for automated generation of linear invariants. Sting implements several algorithms for linear invariant generation and provides a tradeoff between runtime and the quality of invariants. In our experimental results, we use the invariants obtained through the BHRZ method [6], which sacrifices precision in order to have polynomial runtime. However, as the experimental results show, the resulting simple invariants are sufficient for our approach.

Pseudo-codes. Appendix A contains pseudo-codes for the more complex examples, i.e. Karatsuba, Closest pair and Strassen. Codes of all other examples can be found in the webpage above.

Machine. All reported results were obtained on an Ubuntu 18.04 machine using an Intel Core i5-5300U (2.9 GHz) processor and 12GB of RAM.

6.2 Simple Recursive Programs

We begin with simple examples of recursive programs. The results are reported in Table 3.

Factorial. We consider a function that calculates $n!$ by recursively computing $(n-1)!$ and then multiplying it by n. We obtain an $\mathcal{O}(n)$ bound.

Number of Permutations. We compute the number of k-permutations of an n-set, i.e. $P(n,k) = \frac{n!}{(n-k)!}$ by two calls to the factorial function above. Our algorithm finds the bound $10n - 5k + 8$, illustrating that it can handle results containing more than one variable.

Table 3. Experimental results on simple recursive programs. Here $\eta(\ell_0)$ is the part of measure function at the initial label. All values are rounded up to two digits.

Example	Time (in Seconds)	$\eta(\ell_0)$
Factorial	0.57	$5 \cdot n + 2$
$P(n,k)$	0.63	$10 \cdot n - 5 \cdot k + 8$

6.3 Polynomial Bounds

The examples in this section illustrate that our approach is applicable for polynomial bounds. The obtained bounds are reported in Table 4.

Fibonacci. We consider the classic dynamic-programming approach to computing the first n Fibonacci numbers. An array is maintained with values of $f_0, f_1, \ldots, f_i$ and f_{i+1} is computed as $f_i + f_{i-1}$ and added to the array. Our approach synthesizes a bound of $\mathcal{O}(n)$.

Nested Loop. We take two nested while loops, each iterating n times. Our algorithm finds a bound of $\mathcal{O}(n^2)$.

Recursive Nested Loop. We consider a recursive function $\texttt{loop}(i,j,m,n)$ that mimics a nested loop by calling $\texttt{loop}(i,j+1,m,n)$ if $j \leq n$ and $\texttt{loop}(i+1,0,m,n)$ otherwise. The recursive calls end when $i > m$. Our synthesis algorithm is able to find a complicated quadratic bound containing four variables.

Quadratic Sorting Algorithms. As real-world examples, we executed our implementation over Insertion Sort, Bubble Sort and Quick Sort [25]. We used the standard pseudo-codes of these sorting algorithms as in [25]. Our algorithm was able to synthesize a quadratic bound for all cases. Note that all these algorithms have a quadratic worst-case runtime and hence our approach is producing asymptotically tight bounds.

6.4 Non-polynomial Bounds

In this section, we provide experimental results that led to non-polynomial measure functions. This is the main novelty of our approach. The final results are shown in Table 5.

Bivariate Example. Our approach is able to automatically synthesize non-polynomial bounds containing several variables. To demonstrate this, we consider the program in Figure 8. For this program, our approach reports a runtime of $\mathcal{O}(x + \log y)$.

Table 4. Experimental results with polynomial bounds. Here $\eta(\ell_0)$ is the part of measure function at the initial label. All values are rounded up to two digits.

Example	Time (in Seconds)	$\eta(\ell_0)$
Fibonacci	0.65	$3 \cdot n + 4$
Nested Loop	0.72	$n^2 + 5 \cdot n + 4$
Recursive Nested Loop	0.94	$(3 \cdot m - 3 \cdot i + 6) \cdot n + 9 \cdot m - 3 \cdot j - 8 \cdot i + 12$
Insertion Sort	0.75	$2.5 \cdot n^2 + 12.5 \cdot n$
Bubble Sort	0.91	$10.5 \cdot n^2 + 4.5$
Quick Sort	1.29	$9.34 \cdot n^2 + 8.67$

```
bivariate(x, y) {
while y > 1 do
    y := y/2;
    x := x + 1
od;
while x > 0 do
    x := x − 1
od
}
```

Fig. 8. An example program whose runtime is non-polynomial and dependent on more than one variable.

Binary Search. We consider the classical binary search function in which an array of sorted integers is being searched for a specific given value and each time the same function is called recursively on half of the original array. Our approach obtains an $\mathcal{O}(\log n)$ bound.

Merge Sort. We consider the classical Merge Sort algorithm [25, Chapter 2] in which an array of integers is divided into two equal subarrays, each of them is sorted recursively and then the two sorted subarrays are merged together. We obtain an asymptotically tight $\mathcal{O}(n \log n)$ bound.

Heap Sort. Another well-known sub-quadratic sorting algorithm is Heap Sort [25, Chapter 7] in which all elements of an array are inserted into a heap and then removed one by one in order from largest to smallest. Our approach leads to an asymptotically optimal bound of $\mathcal{O}(n \log n)$ in this case.

Closest Pair of Points. As a classic example from computational geometry, we take the closest pair problem which given a set of n two-dimensional points asks for a pair of points that have shortest Euclidean distance between them (cf. [25, Chapter 33]). Our method is able to automatically produce an $\mathcal{O}(n \log n)$ bound which matches the best known manual theoretical results.

Strassen's Algorithm. We consider one of the classic sub-cubic algorithms for Matrix multiplication. The Strassen algorithm (cf. [25, Chapter 4]) has a worst-case running time of $n^{\log_2 7}$. We present the pseudo-code of Strassen's algorithm in our programming language in Appendix A. Our algorithm synthesizes a measure function using a template with $n^{2.9}$.

Karatsuba's Algorithm. We consider two polynomials $p_1 = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^{n-1}$ and $p_2 = b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^{n-1}$, where the coefficients a_i's and b_i's are represented as arrays. The problem is to compute the coefficients of the polynomial obtained by multiplication of p_1 and p_2 considering that n is a power of 2. While the most naïve algorithm is quadratic, Karatsuba's algorithm (cf. [54]) is a well-studied sub-quadratic algorithm for the problem with running time $n^{\log_2 3}$. We present the pseudo-code Karatsuba's algorithm in our programming language in Appendix A. Using a template with $n^{1.6}$, our algorithm synthesizes a measure function (basically, using constraints as illustrated in Example 4.1).

Remark 13. Note that the obtained bounds for Strassen's and Karatsuba's algorithms are slightly worse than the optimal bounds. This is due to the fact that the optimal bounds have irrational exponents. In order to have finitely-representable numbers, our approach uses rational exponents. This being said, our approach is able to synthesize bounds of the form $O(n^r)$ for rational numbers r that are *arbitrarily* close to optimal.

Table 5. Experimental results with non-polynomial bounds. Here $\eta(\ell_0)$ is the part of measure function at the initial label. All values are rounded up to two digits.

Example	Time (in Seconds)	$\eta(\ell_0)$
Bivariate	1.53	$4.33 \cdot \ln y + 2 \cdot x + 4$
Binary Search	0.78	$10.1 \cdot \ln n + 1$
Merge Sort	4.99	$25.02 \cdot n \cdot \ln n + 21.68 \cdot n - 20.68$
Heap Sort	3.91	$23.09 \cdot n \cdot \ln n + 35.09 \cdot n - 33.08$
Closest Pair	10.66	$128.85 \cdot n \cdot \ln n + 108.95 \cdot n - 53.31$
Strassen	5.29	$954.2 \cdot n^{2.9} + 1$
Karatsuba	2.21	$2261.55 \cdot n^{1.6} + 1$

7 RELATED WORK

In this section we discuss the related work. The termination of recursive programs or other temporal properties has already been extensively studied [5, 22–24, 28, 55–57, 70]. Our work is most closely related to automatic amortized analysis [4, 34, 42, 44–48, 52, 53, 67], as well as the SPEED project [37–39]. 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 gives an algorithm to derive such non-polynomial bounds, and surprisingly using linear programming.

The most classical method for obtaining worst-case bounds on runtimes of recursive programs is to apply the Master Theorem (MT) [25]. We remark two main differences between our approach and MT:

- First, our approach is able to synthesize multivariate bounds. For example, it finds a non-polynomial bound based on two different variables for the "Bivariate" example in Section 6.4 and a polynomial bound based on four variables for the "Recursive Nested Loop" example of Section 6.3. On the other hand, MT can only find univariate bounds.

- Second, in the area of automated complexity analysis of programs, the template-based approaches, such as this work, as well as [26, 42], present an automated methodology, whereas MT requires manual generation of recurrence relations. Our main novelty w.r.t other template-based approaches is that we can handle non-polynomial bounds, whereas previous approaches only consider polynomial bounds.

The approach of recurrence relations for worst-case analysis is explored in [1–3, 31, 36]. 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. Another approach through theorem proving is explored in [69]. The approach is to iteratively generate control-flow paths and then to obtain worst-case bounds over generated paths through theorem proving (with arithmetic theorems).

Ranking functions for intra-procedural analysis has been widely studied [9, 10, 21, 26, 61, 65, 68, 72]. Most works have focused on linear or polynomial ranking functions [21, 26, 61, 65, 68, 72]. Such approach alone can only derive polynomial bounds for programs. When integrated with evaluation trees, polynomial ranking functions can derive exponential bounds such as $\mathcal{O}(2^n)$ [11]. In contrast, we directly synthesize non-polynomial ranking functions without the help of evaluation trees. Ranking functions have been extended to ranking supermartingales [13, 14, 17, 18, 30, 33] for probabilistic programs without recursion. These works cannot derive non-polynomial bounds.

Several other works present proof rules for deterministic programs [41] as well as for probabilistic programs [51, 60]. None of these works can be automated. Other related approaches are sized types [19, 49, 50], and polynomial resource bounds [66]. Again none of these approaches can yield bounds like $\mathcal{O}(n \log n)$ or $\mathcal{O}(n^r)$, for r non-integral.

Below we compare three most related works [2, 11, 37].

Comparison with [2]. First, [2] uses synthesis of linear ranking functions in order to bound the number of nodes or the height of an evaluation tree for a cost relation system. We use expression abstraction to synthesize non-linear bounds. Second, [2] uses branching factor of a cost relation system to bound the number of nodes, which typically leads to exponential bounds. Our approach produces efficient bounds. Third, [2] treats Merge-sort in a very specific way: first, there is a ranking function with a discount-factor 2 to bound the height of an evaluation tree logarithmically, then there is a linear bound for levels in an evaluation tree which does not increase when the levels become deeper, and multiplying them together produces $\mathcal{O}(n \log n)$ bound. We do not rely on these heuristics. Finally, [2] is not applicable to non-direct-recursive cost relations, while our approach is applicable to all recursive programs.

Comparison with [11]. To derive non-polynomial bounds, [11] integrate polynomial ranking functions with evaluation trees for recursive programs so that exponential bounds such as $\mathcal{O}(2^n)$ can be derived for Fibonacci numbers. In contrast, we directly synthesize non-polynomial ranking functions without the help of evaluation trees.

Comparison with [37]. [37] generates bounds through abstract interpretation using inference systems over expression abstraction with logarithm, maximum, exponentiation, etc. In contrast, we employ different method through ranking functions, also with linear-inequality system over expression abstraction with logarithm and exponentiation. The key difference w.r.t expression abstraction is that [37] handles in extra maximum and square root, while we

consider in extra floored expressions and finer linear inequalities between e.g., $\log n, \log(n+1)$ or $n^{1.6}, (n+1)^{1.6}$ through Lagrange's Mean-Value Theorem.

8 CONCLUSION

In this paper, we developed an 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, which 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 Merge-Sort, Heapsort 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 known. Besides, our approach can also synthesize normal polynomial bounds for quicksort, the dynamic programming version for Fibonacci, etc. An interesting future direction is to extend our technique to data-structures. Another future direction is to investigate the application of our approach to invariant generation.

Acknowledgments. We thank reviewers of both the conference version and the current version for their valuable comments that significantly improved the presentation of this work. 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 61802254, the CDZ project CAP (GZ 1023), the IBM PhD Fellowship program and a DOC Fellowship of the Austrian Academy of Sciences.

REFERENCES

[1] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, German Puebla, Diana V. Ramírez-Deantes, Guillermo Román-Díez, and Damiano Zanardini. 2009. Termination and Cost Analysis with COSTA and its User Interfaces. *Electr. Notes Theor. Comput. Sci.* 258, 1 (2009), 109–121. https://doi.org/10.1016/j.entcs.2009.12.008

[2] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings (Lecture Notes in Computer Science)*, María Alpuente and Germán Vidal (Eds.), Vol. 5079. Springer, 221–237. https://doi.org/10.1007/978-3-540-69166-2_15

[3] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2007. Cost Analysis of Java Bytecode. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science)*, Rocco De Nicola (Ed.), Vol. 4421. Springer, 157–172. https://doi.org/10.1007/978-3-540-71316-6_12

[4] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS 2010 (LNCS)*, Radhia Cousot and Matthieu Martel (Eds.), Vol. 6337. Springer, 117–133. https://doi.org/10.1007/978-3-642-15769-1_8

[5] Rajeev Alur and Swarat Chaudhuri. 2010. Temporal Reasoning for Procedural Programs. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings (Lecture Notes in Computer Science)*, Gilles Barthe and Manuel V. Hermenegildo (Eds.), Vol. 5944. Springer, 45–60. https://doi.org/10.1007/978-3-642-11319-2_7

[6] Roberto Bagnara, Patricia M Hill, Elisa Ricci, and Enea Zaffanella. 2003. Precise widening operators for convex polyhedra. In *International Static Analysis Symposium*. Springer, 337–354.

[7] Robert G. Bartle and Donald R. Sherbert. 2011. *Introduction to Real Analysis* (4th ed.). John Wiley & Sons, Inc.

[8] Rastislav Bodík and Rupak Majumdar (Eds.). 2016. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM. http://dl.acm.org/citation.cfm?id=2837614

[9] Olivier Bournez and Florent Garnier. 2005. Proving Positive Almost-Sure Termination. In *RTA*. Springer, 323–337.

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings (Lecture Notes in Computer Science)*, Kousha Etessami and Sriram K. Rajamani (Eds.), Vol. 3576. Springer, 491–504. https://doi.org/10.1007/11513988_48

[11] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2016. Analyzing Runtime and Size Complexity of Integer Programs. *ACM Trans. Program. Lang. Syst.* 38, 4 (2016), 13:1–13:50. http://dl.acm.org/citation.cfm?id=2866575

[12] Giuseppe Castagna and Andrew D. Gordon (Eds.). 2017. *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM. https://doi.org/10.1145/3009837

[13] Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science)*, Natasha Sharygina and Helmut Veith (Eds.), Vol. 8044. Springer, 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

[14] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2016. Termination Analysis of Probabilistic Programs Through Positivstellensatz's. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9779. Springer, 3–22. https://doi.org/10.1007/978-3-319-41528-4_1

[15] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2017. Non-polynomial Worst-Case Analysis of Recursive Programs. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10427. Springer, 41–63.

[16] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2019. Polynomial Invariant Generation for Non-deterministic Recursive Programs. *CoRR* abs/1902.04373 (2019). arXiv:1902.04373 http://arxiv.org/abs/1902.04373

[17] Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs, See [8], 327–342. https://doi.org/10.1145/2837614.2837639

[18] Krishnendu Chatterjee, Petr Novotný, and Đorđe Žikelić. 2017. Stochastic invariants for probabilistic termination, See [12], 145–160. https://doi.org/10.1145/3009837

[19] Wei-Ngan Chin and Siau-Cheng Khoo. 2001. Calculating Sized Types. *Higher-Order and Symbolic Computation* 14, 2-3 (2001), 261–300. https://doi.org/10.1023/A:1012996816178

[20] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. 2003. Linear Invariant Generation Using Non-linear Constraint Solving. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings (Lecture Notes in Computer Science)*, Warren A. Hunt Jr. and Fabio Somenzi (Eds.), Vol. 2725. Springer, 420–432. https://doi.org/10.1007/978-3-540-45069-6_39

[21] Michael Colón and Henny Sipma. 2001. Synthesis of Linear Ranking Functions. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science)*, Tiziana Margaria and Wang Yi (Eds.), Vol. 2031. Springer, 67–81. https://doi.org/10.1007/3-540-45319-9_6

[22] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination proofs for systems code. In *PLDI 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 415–426. https://doi.org/10.1145/1133981.1134029

[23] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2009. Summarization for termination: no return! *Formal Methods in System Design* 35, 3 (2009), 369–387. https://doi.org/10.1007/s10703-009-0087-8

[24] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *TACAS 2013 (LNCS)*, Nir Piterman and Scott A. Smolka (Eds.), Vol. 7795. Springer, 47–61. https://doi.org/10.1007/978-3-642-36742-7_4

[25] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3. ed.)*. MIT Press. http://mitpress.mit.edu/books/introduction-algorithms

[26] Patrick Cousot. 2005. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings (Lecture Notes in Computer Science)*, Radhia Cousot (Ed.), Vol. 3385. Springer, 1–24. https://doi.org/10.1007/978-3-540-30579-8_1

[27] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[28] Patrick Cousot and Radhia Cousot. 2012. An abstract interpretation framework for termination. In *POPL 2012*, John Field and Michael Hicks (Eds.). ACM, 245–258. https://doi.org/10.1145/2103656.2103687

[29] J. Farkas. 1894. A Fourier-féle mechanikai elv alkalmazásai (Hungarian). *Mathematikaiés Természettudományi Értesitö* 12 (1894), 457–472.

[30] Luis María Ferrer Fioriti and Holger Hermanns. 2015. Probabilistic Termination: Soundness, Completeness, and Compositionality. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 489–501. https://doi.org/10.1145/2676726.2677001

[31] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. 1991. Automatic Average-Case Analysis of Algorithm. *Theor. Comput. Sci.* 79, 1 (1991), 37–109. https://doi.org/10.1016/0304-3975(91)90145-R

[32] Robert W. Floyd. 1967. Assigning meanings to programs. *Mathematical Aspects of Computer Science* 19 (1967), 19–33.

[33] Hongfei Fu and Krishnendu Chatterjee. 2019. Termination of Nondeterministic Probabilistic Programs. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science)*, Constantin Enea and Ruzica Piskac (Eds.), Vol. 11388. Springer, 468–490. https://doi.org/10.1007/978-3-030-11245-5_22

[34] Stéphane Gimenez and Georg Moser. 2016. The complexity of interaction, See [8], 243–255. https://doi.org/10.1145/2837614.2837646

[35] Kurt Gödel, Stephen C. Kleene, and J. B. Rosser. 1934. On Undecidable Propositions of Formal Mathematical Systems. *Institute for Advanced Study Princeton, NJ* (1934).

[36] Bernd Grobauer. 2001. Cost Recurrences for DML Programs. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, Benjamin C. Pierce (Ed.). ACM, 253–264. https://doi.org/10.1145/507635.507666

[37] Bhargav S. Gulavani and Sumit Gulwani. 2008. A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings (Lecture Notes in Computer Science)*, Aarti Gupta and Sharad Malik (Eds.), Vol. 5123. Springer, 370–384. https://doi.org/10.1007/978-3-540-70545-1_35

[38] Sumit Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science)*, Ahmed Bouajjani and Oded Maler (Eds.), Vol. 5643. Springer, 51–62. https://doi.org/10.1007/978-3-642-02658-4_7

[39] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: precise and efficient static estimation of program computational complexity. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 127–139. https://doi.org/10.1145/1480881.1480898

[40] D. Handelman. 1988. Representing Polynomials by Positive Linear Functions on Compact Convex Polyhedra. *Pacific J. Math.* 132 (1988), 35–62.

[41] Wim H. Hesselink. 1993. Proof Rules for Recursive Procedures. *Formal Asp. Comput.* 5, 6 (1993), 554–570. https://doi.org/10.1007/BF01211249

[42] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 14. https://doi.org/10.1145/2362389.2362393

[43] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science)*, P. Madhusudan and Sanjit A. Seshia (Eds.), Vol. 7358. Springer, 781–786. https://doi.org/10.1007/978-3-642-31424-7_64

[44] Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polymorphic Recursion and Partial Big-Step Operational Semantics. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings (Lecture Notes in Computer Science)*, Kazunori Ueda (Ed.), Vol. 6461. Springer, 172–187. https://doi.org/10.1007/978-3-642-17164-2_13

[45] Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science)*, Andrew D. Gordon (Ed.), Vol. 6012. Springer, 287–306. https://doi.org/10.1007/978-3-642-11957-6_16

[46] Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 185–197. https://doi.org/10.1145/640128.604148

[47] Martin Hofmann and Steffen Jost. 2006. Type-Based Amortised Heap-Space Analysis. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science)*, Peter Sestoft (Ed.), Vol. 3924. Springer, 22–37. https://doi.org/10.1007/11693024_3

[48] Martin Hofmann and Dulma Rodriguez. 2009. Efficient Type-Checking for Amortised Heap-Space Analysis. In *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings (Lecture Notes in Computer Science)*, Erich Grädel and Reinhard Kahle (Eds.), Vol. 5771. Springer, 317–331. https://doi.org/10.1007/978-3-642-04027-6_24

[49] John Hughes and Lars Pareto. 1999. Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.*, Didier Rémi and Peter Lee (Eds.). ACM, 70–81. https://doi.org/10.1145/317636.317785

[50] John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 410–423. https://doi.org/10.1145/237721.240882

[51] Claire Jones. 1989. *Probabilistic Non-Determinism.* Ph.D. Dissertation. The University of Edinburgh.

[52] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 223–236. https://doi.org/10.1145/1706299.1706327

[53] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. 2009. "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings (Lecture Notes in Computer Science)*, Ana Cavalcanti and Dennis Dams (Eds.), Vol. 5850. Springer, 354–369. https://doi.org/10.1007/978-3-642-05089-3_23

[54] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume I–III.* Addison-Wesley.

[55] Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. 2014. Automatic Termination Verification for Higher-Order Functional Programs. In *ESOP 2014 (LNCS)*, Zhong Shao (Ed.), Vol. 8410.

Springer, 392–411. https://doi.org/10.1007/978-3-642-54833-8_21
[56] Chin Soon Lee. 2009. Ranking functions for size-change termination. *ACM Trans. Program. Lang. Syst.* 31, 3 (2009), 10:1–10:42. https://doi.org/10.1145/1498926.1498928
[57] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *POPL 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM, 81–92. https://doi.org/10.1145/360204.360210
[58] lpsolve 2016. lp_solve 5.5.2.3. http://lpsolve.sourceforge.net/5.5/.
[59] Martin Lukasiewycz. 2008. Java ILP - Java Interface to ILP Solvers. http://javailp.sourceforge.net/.
[60] Federico Olmedo, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Christoph Matheja. 2016. Reasoning about Recursive Probabilistic Programs. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, Martin Grohe, Eric Koskinen, and Natarajan Shankar (Eds.). ACM, 672–681. https://doi.org/10.1145/2933575.2935317
[61] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings (Lecture Notes in Computer Science)*, Bernhard Steffen and Giorgio Levi (Eds.), Vol. 2937. Springer, 239–251. https://doi.org/10.1007/978-3-540-24622-0_20
[62] Sriram Sankaranarayanan, Henny B Sipma, and Zohar Manna. 2004. Constraint-based linear-relations analysis. In *International Static Analysis Symposium*. Springer, 53–68.
[63] Alexander Schrijver. 1999. *Theory of Linear and Integer Programming*. Wiley.
[64] Alexander Schrijver. 2003. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer.
[65] Liyong Shen, Min Wu, Zhengfeng Yang, and Zhenbing Zeng. 2013. Generating exact nonlinear ranking functions by symbolic-numeric hybrid method. *J. Systems Science & Complexity* 26, 2 (2013), 291–301. https://doi.org/10.1007/s11424-013-1004-1
[66] Olha Shkaravska, Ron van Kesteren, and Marko C. J. D. van Eekelen. 2007. Polynomial Size Analysis of First-Order Functions. In *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings (Lecture Notes in Computer Science)*, Simona Ronchi Della Rocca (Ed.), Vol. 4583. Springer, 351–365. https://doi.org/10.1007/978-3-540-73228-0_25
[67] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *CAV 2014 (LNCS)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 745–761. https://doi.org/10.1007/978-3-319-08867-9_50
[68] Kirack Sohn and Allen Van Gelder. 1991. Termination Detection in Logic Programs using Argument Sizes. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 29-31, 1991, Denver, Colorado, USA*, Daniel J. Rosenkrantz (Ed.). ACM Press, 216–226. https://doi.org/10.1145/113413.113433
[69] Akhilesh Srikanth, Burak Sahin, and William R. Harris. 2017. Complexity verification using guided theorem enumeration, See [12], 639–652. https://doi.org/10.1145/3009837
[70] Caterina Urban. 2013. The Abstract Domain of Segmented Ranking Functions. In *SAS 2013 (LNCS)*, Francesco Logozzo and Manuel Fähndrich (Eds.), Vol. 7935. Springer, 43–62. https://doi.org/10.1007/978-3-642-38856-9_5
[71] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.* 7, 3 (2008), 36:1–36:53. https://doi.org/10.1145/1347375.1347389
[72] 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. https://doi.org/10.1007/s11704-009-0074-7

A EXPERIMENTAL DETAILS

In this part, we present the details for our experimental results.

Pseudo-codes for Our Examples. Figures 9 and 10 together account for Karatsuba's Algorithm. Figures 11 – 14 demonstrate the divide-and-conquer algorithm for Closest-Pair problem, and Figures 15 – 17 show the Strassen's algorithm. Invariants are bracketed (i.e., $[\dots]$) at significant labels in the programs.

Remark 14 (Approximation constants.). We use approximation of constants up to four digits of precision. For example, we use the interval $[2.7182, 2.7183]$ (resp. $[0.6931, 0.6932]$) for tight approximation of e (resp. $\ln 2$). We use similar approximation for constants such as $2^{0.6}$ and $2^{0.9}$. □

Remark 15 (Input specifications.). We note that as input specifications other than the input program, the invariants can be obtained automatically [20, 27]. In the examples, the invariants are even simpler, and obtained from the guards of branching labels. Besides the above we have quadruple (d, op, r, k). We discuss these parameter for the examples below.

- The type of bound to be synthesized is denoted by op: For Merge-Sort and Closest-Pair it is log (denoting logarithmic terms in expression) and for Strassen's and Karatsuba's algorithms, it is exp (denoting non-polynomial bounds with non-integral exponent).
- The number of terms multiplied together in each summand of the general form as in (1) is d. For example, (i) for $n^{4.5}$ there is only one term, and hence $d = 1$; (ii) for $n^{3.9} \cdot \log n$ we have two terms, and hence $d = 2$. Therefore, even for worst-case non-polynomial bounds of higher degrees, the maximum degree for template is still small. In all our examples d is at most 2.
- Recall that r is an upper bound on the degree of exponent of the asymptotic bound of the measure function. Also observe that if r is not specified, we can search for r in a desired interval automatically using binary search. For example, for Strassen's algorithm, the desired interval of the exponent is between $[2, 3]$. With $r = 2$, our approach on Strassen's algorithm reports failure. Thus a binary search for r in the interval can obtain the exponent as 2.9.
- In all our examples $k = 2$ for the parameter for the Handelmann's Theorem.

Thus the input for our algorithm is quite simple. □

Remark 16 (Complexity). Given d and k for the template are constants the complexity of our algorithm is polynomial. While our algorithm is exponential in these parameters, in all our examples the above parameters are at most 2. The approach we present is polynomial time (using linear programming) for several non-trivial examples, and therefore a scalable one. □

```
//Initialize all array entries to be zero.
initialize(i, j) {
   [i ≤ j]
   l := i;
   [l ≤ j + 1]
   while l ≤ j do
      skip; l := l + 1
   od
}

//Copy one array into another.
copy(i, j, m, n) {
   [i ≤ j ∧ m ≤ n]
   k := i; l := m;
   [k ≤ j + 1 ∧ l ≤ n + 1]
   while k ≤ j ∧ l ≤ n do
      skip; k := k + 1; l := l + 1
   od
}

//Add two arrays entrywise.
add(i, j, m, n) {
   [i ≤ j ∧ m ≤ n]
   k := i; l := m;
   [k ≤ j + 1 ∧ l ≤ n + 1]
   while k ≤ j ∧ l ≤ n do
      skip; k := k + 1; l := l + 1
   od
}

//Subtract two arrays entrywise.
subtract(i, j, m, n) {
   [i ≤ j ∧ m ≤ n]
   k := i; l := m;
   [k ≤ j + 1 ∧ l ≤ n + 1]
   while k ≤ j ∧ l ≤ n do
      skip; k := k + 1; l := l + 1
   od
}
```

Fig. 9. Auxiliary Function Calls for Karatsuba's Algorithm

```
//The program calculates the product of two polynomials.
//The degree should be arranged in increasing order.
//Array index starts from 1.
//n is the length of the arrays and should be a power of 2.
//The quadruple of input parameters is (1, exp, 1.6, 2).

karatsuba(n) {
  [n ≥ 1]
  if n ≥ 2 then
     t := ⌊n/2⌋;

     // checking whether n is even
     if 2 * t ≤ n and 2 * t ≥ n then

        //sub-dividing arrays
        copy(1, t, 1, t);
        copy(t + 1, n, 1, t);
        copy(1, t, 1, t);
        copy(t + 1, n, 1, t);

        //adding the sub-arrays
        copy(1, t, 1, t); add(1, t, 1, t);
        copy(1, t, 1, t); add(1, t, 1, t);

        //recursive calls
        karatsuba(t);
        karatsuba(t);
        karatsuba(t);

        //combining step
        subtract(1, n − 1, 1, n − 1);
        subtract(1, n − 1, 1, n − 1);

        initialize(1, 2 * n − 1);
        add(1, n − 1, 1, n − 1);
        add(1, n − 1, n, 2 * n − 2);
        add(t + 1, n + t − 1, 1, n − 1)
     else skip //If n is not even, simply fail.
     fi
  else //trivial case
     skip
  fi
}
```

Fig. 10. Main Function Call for Karatsuba's Algorithm

```
//Copy one array into another.
copy(i, j, m, n) {
  [i ≤ j ∧ m ≤ n]
  k := i; l := m;
  [k ≤ j + 1 ∧ l ≤ n + 1]
  while k ≤ j ∧ l ≤ n do
    skip; k := k + 1; l := l + 1
  od
}

//sorting one array while adjusting another accordingly

mergesort(i, j) {
  [i ≤ j]
  if i ≤ j − 1 then
    k := i + ⌊(j−i+1)/2⌋ − 1;
    mergesort(i, k);
    mergesort(k + 1, j);
    merge(i, j, k)
  else
    skip
  fi
}

merge(i, j, k) {
  [i ≤ j]
  m := i; n := k + 1; l := i;
  [l ≤ j + 1]
  while l ≤ j do
    if ⋆ then
      skip; m := m + 1
    else
      skip; n := n + 1
    fi;
    l := l + 1
  od;
  l := i;
  [l ≤ j + 1]
  while l ≤ j do
    skip; l := l + 1
  od
}
```

Fig. 11. Merge-Sort and Copy for Closest-Pair

```
//Caculates the shortest distance of a finite set of points.
//One array stores x-coordinates and another stores y-coordinates.
//The quadruple of input parameters is (2, log, −, 2).

clst_pair_main(i, j) {
    [ i ≤ j ]

    //copying arrays
    copy(i, j, i, j); copy(i, j, i, j);

    //sorting arrays in x-coordinate
    mergesort(i, j);

    //sorting arrays in y-coordinate
    mergesort(i, j);

    //solving the result
    clst_pair(i, j)
}
```

Fig. 12. Main Function Call for Closest-Pair

```
//principal recursive function call for solving Closest-Pair

clst_pair(i, j) {
  [i ≤ j]
  if i ≤ j − 3 then
    //recursive case where there are at least 4 points

    k := i + ⌊(j−i+1)/2⌋ − 1;
    clst_pair(i, k);
    clst_pair(k + 1, j);

    //taking the minimum distance from the previous recursive calls
    skip;

    //fetch and scan the mid-line
    fetch&scan(i, j)
  else
    //base case (fewer than 4 points)
    skip
  fi
}
```

Fig. 13. Principal Recursive Function Call for Closest-Pair

```
//fetch and scan the mid-line
fetch&scan(i, j) {
  //fetching the points on the mid-line
  [i ≤ j − 3]
  l := i; p := i;

  [i ≤ j − 3 ∧ p ≤ j + 1 ∧ l ≤ j + 1]
  while p ≤ j do
    if ⋆ then l := l + 1 else skip fi;
    p := p + 1
  od

  if l ≥ i + 1 and l ≤ j + 1 then
    p := i;

    //scanning the points on the mid-line
    [p ≤ l]
    while p ≤ l − 1 do
      m := p + 1;

      //checking 7 points ahead on the mid-line
      [m ≤ p + 8]
      while m − p ≤ 7 and m ≤ l − 1 do
        skip; m := m + 1
      od;
      p := p + 1
    od
  else skip fi
}
```

Fig. 14. Other Function Calls for Closest-Pair

```
//The program calculates the product of two matrices.
//n is the row/column size of both matrices and should be a power of 2.
//Each of the matrices is stored in a two-dimensional array of dimension n.
//Array indices starts from 1.
//The quadruple of input parameters is (2, exp, 1.9, 2) .

strassen(n) {
  [n ≥ 1]
  if n ≥ 2 then
     t := ⌊n/2⌋ ;

     // checking whether n is even
     if 2 ∗ t ≤ n and 2 ∗ t ≥ n then
        //sub-dividing matrices A
        matrixtoblocks(n, t) ; matrixtoblocks(n, t) ;

        //sums of matrices
        copy(t) ; add(t) ; copy(t) ; add(t) ;
        copy(t) ; add(t) ; copy(t) ; subtract(t) ;
        copy(t) ; add(t) ; copy(t) ; add(t) ;
        copy(t) ; subtract(t) ; copy(t) ; add(t) ;
        copy(t) ; subtract(t) ; copy(t) ; add(t) ;

        //recursive calls
        strassen(t) ;
        strassen(t) ;
        strassen(t) ;
        strassen(t) ;
        strassen(t) ;
        strassen(t) ;
        strassen(t) ;

        //combining stage
        copy(t) ; add(t) ; subtract(t) ; add(t) ;
        copy(t) ; add(t) ; copy(t) ; add(t) ;

        copy(t) ; add(t) ; subtract(t) ; add(t) ;
        blockstomatrix(n, t)
     else skip //If n is not even, simply fail.
     fi
  else skip //trivial case
  fi
}
```

Fig. 15. Main Function Call for Strassen's Algorithm

```
//Partition a matrix into block matrices.
matrixtoblocks(n, t) {
  [t ≥ 1]
  i := 1;
  [t ≥ 1 ∧ i ≤ t + 1]
  while i ≤ t do
    j := 1;

    [t ≥ 1 ∧ i ≤ t ∧ j ≤ t + 1]
    while j ≤ t do
      skip; j := j + 1
    od;
    i := i + 1
  od
}

//Construct a matrix from blocks.
blockstomatrix(n, t) {
  [t ≥ 1]
  i := 1;
  [t ≥ 1 ∧ i ≤ t + 1]
  while i ≤ t do
    j := 1;

    [t ≥ 1 ∧ i ≤ t ∧ j ≤ t + 1]
    while j ≤ t do
      skip; j := j + 1
    od;
    i := i + 1
  od
}
```

```
//Copy a square matrix into another.
copy(n) {
[n ≥ 1]
i := 1;
[n ≥ 1 ∧ i ≤ n + 1]
while i ≤ n do
j := 1;
[n ≥ 1 ∧ i ≤ n ∧ j ≤ n + 1]
while j ≤ n do
skip; j := j + 1
od;
i := i + 1
od
}
```

Fig. 16. Auxiliary Function Calls for Strassen's Algorithm

```
//Add two matrices (entrywise).
add(n) {
  [n ≥ 1]
  i := 1;
  [n ≥ 1 ∧ i ≤ n + 1]
  while i ≤ n do
    j := 1;
    [n ≥ 1 ∧ i ≤ n ∧ j ≤ n + 1]
    while j ≤ n do
      skip; j := j + 1
    od;
    i := i + 1
  od
}

//Subtract two matrices (entrywise).
subtract(n) {
  [n ≥ 1]
  i := 1;
  [n ≥ 1 ∧ i ≤ n + 1]
  while i ≤ n do
    j := 1;
    [n ≥ 1 ∧ i ≤ n ∧ j ≤ n + 1]
    while j ≤ n do
      skip;
      j := j + 1
    od;
    i := i + 1
  od
}
```

Fig. 17. Matrix Addition and Subtraction for Strassen's Algorithm