Institute of Science and Technology

Termination and Worst-Case Analysis of Recursive Programs

1 Anonymous and 2 Anonymous and 3 Anonymous

Technical Report No. IST-2016-618-v1+1
Deposited at 15 Jul 2016 09:07
https://repository.ist.ac.at/618/1/popl2017a.pdf

IST Austria (Institute of Science and Technology Austria)
Am Campus 1
A-3400 Klosterneuburg, Austria

Termination and Worst-Case Analysis of Recursive Programs

Abstract

We study the problem of developing efficient approaches for proving termination of recursive programs with one-dimensional arrays. Ranking functions serve as a sound and complete approach for proving termination of non-recursive programs without array operations. First, we generalize ranking functions to the notion of measure functions, and prove that measure functions (i) provide a sound method to prove termination of recursive programs (with one-dimensional arrays), and (ii) is both sound and complete over recursive programs without array operations. Our second contribution is the synthesis of measure functions of specific forms in polynomial time. More precisely, we prove that (i) polynomial measure functions over recursive programs can be synthesized in polynomial time through Farkas' Lemma and Handelman's Theorem, and (ii) measure functions involving logarithm and exponentiation can be synthesized in polynomial time through abstraction of logarithmic or exponential terms and Handelman's Theorem. A key application of our method is the worst-case analysis of recursive programs. While previous methods obtain worst-case polynomial bounds of the form $\mathcal{O}(n^k)$, where k is an integer, our polynomial-time methods can synthesize bounds of the form $\mathcal{O}(n \log n)$, as well as $\mathcal{O}(n^x)$, where x is not an integer. We show the applicability of our automated technique to obtain worst-case complexity of classical recursive algorithms such as (i) Merge-Sort, 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, where we obtain $\mathcal{O}(n^x)$ bound, where x is not an integer and close to the best-known bounds for the respective algorithms. Finally, we present experimental results to demonstrate the effectiveness of our approach.

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 [39, 40]. The derivation of such worst-case bounds require 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 for hard real-time systems, guarantees of worst-case behavior is required; and (b) the bounds are useful in early detection of egregious performance problems in large code bases. Works such as [23, 24, 29, 30] 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 focus on (but not limited to) sequential loop-free code with low level hardware aspects [49].
2. *Resource analysis.* The use of abstract interpretation and type systems to deal with loop, recursion, data-structures has also been considered in details [4, 24, 38], such as using linear invariant generation to obtain disjunctive and non-linear bounds [13], as well as potential based methods for handling recursion and inductive data structures [29, 30].
3. *Ranking functions.* The notion of ranking functions is a powerful technique for termination analysis of non-recursive programs [7, 8, 12, 15, 43, 46, 48, 50]. They serve as a sound and complete approach for proving termination of non-recursive programs without array operations [19], and they have also been extended as ranking supermatingales for analysis of probabilistic programs [9, 10, 17].

Given the above wide array of results, two aspects of the problem has not been addressed yet.

1. *Termination of recursive programs through ranking functions.* The use of ranking functions has been limited mostly to non-recursive programs, and the use of ranking functions for recursive programs and their use to obtain worst-case analysis bounds has not been explored in depth.
2. *Efficient methods for precise bounds.* While previous works present methods for disjunctive polynomial bounds [24] (such as $\max(0, n) \cdot (1 + max(n, m))$), or multivariate polynomial analysis [29], these works do not provide efficient (polynomial-time) methods to synthesize bounds such as $\mathcal{O}(n \log n)$ or $\mathcal{O}(n^x)$, where x is not an integer.

In this work we address these two aspects, i.e., termination of recursive programs with ranking functions, and polynomial-time methods for obtaining bounds such as $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^x)$, where x is not integral.

Our contributions. Our main contributions are as follows:

1. First, we generalize the notion of ranking functions for non-recursive programs to the notion of *measure* functions for recursive programs. We show that measure functions (i) provide a sound method to prove termination of recursive programs, and (ii) is both sound and complete over recursive programs without array operations. Thus these results generalize the results for non-recursive programs.
2. Second, we present polynomial-time procedure for handling of measure functions of specific forms. More precisely, we prove that (i) polynomial measure functions over recursive programs can be synthesized in polynomial time through Farkas' Lemma and Handelman's Theorem, and (ii) measure functions involving logarithm and exponentiation can be synthesized in polynomial time through abstraction of logarithmic or exponential terms and Handelman's Theorem.
3. A key application of our method is the worst-case analysis of recursive programs. Our polynomial-time procedure can synthesize bounds of the form $\mathcal{O}(n \log n)$, as well as $\mathcal{O}(n^x)$, where x 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* [14, Chapter 2], the divide-and-conquer algorithm for the *Closest-Pair problem* [14, Chapter 33],

we obtain $\mathcal{O}(n \log n)$ worst-case bound, and the bounds we obtain are asymptotically optimal. Note that previous polynomial-time methods are either not applicable, or grossly over-estimate the bounds as $\mathcal{O}(n^2)$.
- For *Karatsuba's algorithm* for polynomial multiplication (cf. [41]) 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. [14, 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 polynomial-time methods are either not applicable, or grossly over-estimate the bounds as $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively.

4. Finally, we present experimental results to demonstrate the effectiveness of our approach.

Key novelty. The key novelty of our approach is that we show how non-trivial non-linear worst-case upper bounds such as $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^x)$, where x is non-integral, can be soundly obtained, even for recursive programs, in polynomial time using linear programming. Moreover, as our computational tool is linear programming the approach we provide is also relatively a scalable one. See Remark 3 for further details.

Due to space restrictions, several technical details are relegated to the Appendix (cf. the submitted supplementary material).

2. Recursive Programs

In this work our main contributions involve a new approach for analysis of termination of recursive programs. To focus on the new contributions, we consider a simple programming language for recursive programs, with the basic capabilities for recursion and array operation (we do not consider return statements). In our language, (a) all scalar variables hold integers and all array variables refer to finite sequences of integers in heap memory; (b) scalar variables are call-by-value and array variables are call-by-reference; (c) array indices count from 1, and out-of-range array indices are deemed as runtime errors. Moreover, all assignments to scalar variables are restricted to linear expressions and without array entries. We first introduce some basic notations and concepts, then illustrate the syntax of recursive programs and finally the semantics.

2.1 Basic Notations and Concepts

For a set A, we denote by $|A|$ the cardinality of 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. Throughout the paper, we use the special symbol $\bot$ to indicate either the null array (i.e., the array with length zero), runtime errors, or non-existing memory addresses. Moreover, we use *Seq* to denote the set of finite sequences of integers, and denote by $\|a\|$ the length of a for any $a \in$ *Seq*. For each $a \in$ *Seq* and $d \in \mathbb{Z}$, we define $a[d]$ as the d-th entry of a, given $1 \leq d \leq \|a\|$. We will denote by $\mathcal{X}, \mathcal{A}$ two disjoint countable sets of *scalar* and *array* variables, respectively.

Heaps. A *heap* is a function $\mathfrak{h}$ from $\mathbb{N}_0$ into *Seq* such that $\mathfrak{h}(0) = \bot$ and the set $\{d \mid \mathfrak{h}(d) \neq \bot\}$ is finite, where natural numbers are interpreted as memory addresses. Intuitively, each $\mathfrak{h}(d)$ refers to the array stored at the memory address indicated by d.

Arithmetic Array Expressions. The set of *arithmetic array expressions* $\mathfrak{e}$ over $\mathcal{X}$ and $\mathcal{A}$ is generated by the following grammar:

$$\mathfrak{e} ::= c \mid x \mid \|ar\| \mid ar[x] \mid \left\lfloor \frac{\mathfrak{e}}{c} \right\rfloor \mid \mathfrak{e} + \mathfrak{e} \mid \mathfrak{e} - \mathfrak{e} \mid \mathfrak{e} * \mathfrak{e}$$

where $c \in \mathbb{Z}$, $x \in \mathcal{X}$ and $ar \in \mathcal{A}$. Informally, (i) $\|ar\|$ refers to the length of the array indicated by ar, (ii) $ar[x]$ refers to the x-th entry of the array indicated by ar, (iii) $\frac{\cdot}{c}$ refers to division operation, (iv) $\lfloor \cdot \rfloor$ refers to the floor operation, and (v) $+, -, *$ refer to addition, subtraction and multiplication operation over integers, respectively.

Valuations. A *valuation* over $\mathcal{X}, \mathcal{A}$ is a function ν on $\mathcal{X} \cup \mathcal{A}$ such that $\nu(x) \in \mathbb{Z}$ and $\nu(ar) \in \mathbb{N}_0$ for all $x \in \mathcal{X}$ and $ar \in \mathcal{A}$. Informally, a valuation assigns to each scalar an integer and to each array variable a memory address. Under a valuation ν over $\mathcal{X}, \mathcal{A}$ and a heap $\mathfrak{h}$, an arithmetic array expression $\mathfrak{e}$ can be either evaluated to an integer in the straightforward way if no division by zero or out-of-range array-index occurs, or otherwise evaluated to $\bot$; and we denote by $\mathfrak{e}_\mathfrak{h}(\nu)$ the evaluation of $\mathfrak{e}$ under ν and $\mathfrak{h}$. A detailed description of evaluation of arithmetic array expressions is provided in Appendix A.

Propositional Array Predicates. The set of *propositional array predicates* ϕ over $\mathcal{X}$ and $\mathcal{A}$ 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 array expression. Under a heap $\mathfrak{h}$, the satisfaction relation $\models_\mathfrak{h}$ between valuations and propositional array predicates is defined in the straightforward way through evaluation of arithmetic array expressions (cf. Appendix B for details). We consider propositional array predicates in DNF (disjunctive normal form).

2.2 The Syntax

In the sequel, we fix a countable set of *scalar variables* and a countable set of *array variables*; and we also fix a countable set of *function names*. W.l.o.g, these three sets are pairwise disjoint. Each scalar variable (resp. array variable) holds an integer (resp. a one-dimensional array of integers) 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 $\langle pvar \rangle$ and $\langle arvar \rangle$ range over scalar and array variables, respectively.
- *Function Names:* Expressions $\langle fname \rangle$ range over function names.
- *Constants:* Expressions $\langle int \rangle$ range over integers represented as decimal numbers.
- *Arithmetic Expressions:* Expressions $\langle aexpr \rangle$ range over arithmetic array expressions consisting of scalar variables, array entries (cf. $\langle arvar \rangle [\langle pvar \rangle]$), floor operation (cf. $\lfloor \langle aexpr \rangle / \langle int \rangle \rfloor$), lengths of arrays (cf. $\|\langle arvar \rangle\|$) and arithmetic operations. Expressions $\langle pexpr \rangle$ are a subclass of $\langle aexpr \rangle$ which are linear expressions that does not involve array entries.
- *Parameters:* Expressions $\langle plist \rangle$ range over lists of scalar/array variables, and expressions $\langle vlist \rangle$ range over lists of $\langle pexpr \rangle$ expressions/array variables.
- *Boolean Expressions:* Expressions $\langle bexpr \rangle$ range over propositional array predicates over scalar and array variables.
- *Statements:* Various types of assignment statements are indicated by ':='; '**skip**' is the statement that does nothing; conditional branches are indicated by the keyword '**if**'; while-loops are indicated by the keyword '**while**'; sequential compositions are indicated by semicolon; finally, function calls are indicated by $fname\,(\langle vlist \rangle)$.
- *Programs:* Each recursive program $\langle prog \rangle$ is a sequence of function bodies, for which each function body $\langle func \rangle$ consists of a function name followed by a list of parameters (composing a function declaration) and a curly-bracketed statement.

Note that we do not specify a main function body as in e.g. C programming language. Later we will fix an initial function name.

$$\langle prog\rangle ::= \langle func\rangle\langle prog\rangle \mid \langle func\rangle$$

$$\langle func\rangle ::= \langle fname\rangle\text{`('}\langle plist\rangle\text{`)'`\{'}\langle stmt\rangle\text{`\}'}$$
$$\langle plist\rangle ::= \langle pvar\rangle \mid \langle arvar\rangle$$
$$\mid \langle pvar\rangle\text{`,'}\langle plist\rangle \mid \langle arvar\rangle\text{`,'}\langle plist\rangle$$

$$\langle stmt\rangle ::= \textbf{`skip'} \mid \langle pvar\rangle\ \text{`:='}\ \langle pexpr\rangle$$
$$\mid \langle arvar\rangle\text{`['}\langle pvar\rangle\text{`]'}\ \text{`:='}\ \langle aexpr\rangle$$
$$\mid \langle fname\rangle\text{`('}\langle vlist\rangle\text{`)'}$$
$$\mid \textbf{`if'}\ \langle bexpr\rangle\ \textbf{`then'}\ \langle stmt\rangle\ \textbf{`else'}\ \langle stmt\rangle\ \textbf{`fi'}$$
$$\mid \textbf{`while'}\ \langle bexpr\rangle\ \textbf{`do'}\ \langle stmt\rangle\ \textbf{`od'}$$
$$\mid \langle stmt\rangle\ \text{`;'}\ \langle stmt\rangle$$

$$\langle pexpr\rangle ::= \langle int\rangle \mid \langle pvar\rangle \mid \text{`}\|\text{'}\langle arvar\rangle\text{`}\|\text{'}$$
$$\mid \langle pexpr\rangle\ \text{`}+\text{'}\ \langle pexpr\rangle \mid \langle pexpr\rangle\ \text{`}-\text{'}\ \langle pexpr\rangle$$
$$\mid \langle int\rangle\ \text{`}*\text{'}\ \langle pexpr\rangle \mid \text{`}\Big\lfloor\text{'}, \frac{\langle pexpr\rangle}{\langle int\rangle}, \text{`}\Big\rfloor\text{'},$$
$$\langle aexpr\rangle ::= \langle int\rangle \mid \langle pvar\rangle$$
$$\mid \langle arvar\rangle\text{`['}\langle pvar\rangle\text{`]'} \mid \text{`}\|\text{'}\langle arvar\rangle\text{`}\|\text{'}$$
$$\mid \langle aexpr\rangle\ \text{`}+\text{'}\ \langle aexpr\rangle \mid \langle aexpr\rangle\ \text{`}-\text{'}\ \langle aexpr\rangle$$
$$\mid \langle aexpr\rangle\ \text{`}*\text{'}\ \langle aexpr\rangle \mid \text{`}\Big\lfloor\text{'}, \frac{\langle aexpr\rangle}{\langle int\rangle}, \text{`}\Big\rfloor\text{'},$$
$$\langle vlist\rangle ::= arvar \mid \langle pexpr\rangle$$
$$\mid \langle arvar\rangle\text{`,'}\langle vlist\rangle \mid \langle pexpr\rangle\text{`,'}\langle vlist\rangle$$

$$\langle literal\rangle ::= \langle aexpr\rangle\ \text{`}\leq\text{'}\ \langle aexpr\rangle \mid \langle aexpr\rangle\ \text{`}\geq\text{'}\ \langle aexpr\rangle$$
$$\langle bexpr\rangle ::= \langle literal\rangle \mid \neg\langle bexpr\rangle$$
$$\mid \langle bexpr\rangle\ \textbf{`or'}\ \langle bexpr\rangle \mid \langle bexpr\rangle\ \textbf{`and'}\ \langle bexpr\rangle$$

Figure 1. Syntax of Recursive Programs

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

- *Function Bodies:* we consider that every parameter list $\langle plist\rangle$ contains no duplicate scalar/array variables, and the function names from function bodies are distinct.
- *Function Calls:* we consider that no function call involves some function name without function body (i.e., undeclared function names), and values passed to each function call (cf. $\langle vlist\rangle$) match types of the corresponding parameter list.

Statement Labelling. Given a recursive program, 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.

We illustrate with an example of a recursive program together with its labelling, which implements the Merge-Sort algorithm.

Example 1. *Fig. 2 depicts a recursive program for the Merge-Sort algorithm [14, Chapter 2]. The numbers on the leftmost side are the labels assigned to statements which represent program counters, where* mergesort *starts from label* 1 *and ends at* 7*, and* merge *starts from label* 1 *and ends at* 15*.*

```
mergesort(ar, i, j, tmp) {
1:   if 1 ≤ i and i ≤ j − 1 then
2:        k := i + ⌊(j−i+1)/2⌋ − 1;
3:        mergesort(ar, i, k, tmp);
4:        mergesort(ar, k + 1, j, tmp);
5:        merge(i, j, k, ar, tmp)
6:     else skip
     fi
7: }

merge(i, j, k, ar, tmp) {
1:   m := i;  2:  n := k + 1;  3:  l := i;
4:   while l ≤ j do
5:      if ar[m] ≤ ar[n] then
6:           tmp[l] := ar[m];
7:           m := m + 1
           else
8:           tmp[l] := ar[n];
9:           n := n + 1
        fi;
10:     l := l + 1
     od;
11:  l := i;
12:  while l ≤ j do
13:     ar[l] := tmp[l];
14:     l := l + 1
     od
15: }
```

Figure 2. A program that implements Merge-Sort

2.3 The Semantics

We use control-flow graphs (CFGs) to specify the semantics of recursive programs. The notion of CFGs is illustrated as follows.

Definition 1 (Control-Flow Graphs)**.** *A* control-flow graph *(CFG) is a triple which takes the form (†)*

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

where:

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

W.l.o.g, we consider that all labels are natural numbers.

Definition 2 (Valuations $\mathit{Val}_{\mathsf{f}}$)**.** *We denote by $\mathit{Val}_{\mathsf{f}}$ the set of valuations over $V_{\mathrm{p}}^{\mathsf{f}}, V_{\mathrm{ar}}^{\mathsf{f}}$, for each $\mathsf{f} \in F$. We say that a valuation $\nu \in \mathit{Val}_{\mathsf{f}}$ is* initial *if $\nu(q) = 0$ whenever $q \in V_{\mathrm{p}}^{\mathsf{f}} \cup V_{\mathrm{ar}}^{\mathsf{f}}$ and q does not appear in the parameter list of f.*

Informally, a function name f, a label $\ell \in L^{\mathsf{f}}$, a valuation $\nu \in \mathit{Val}_{\mathsf{f}}$ and a heap $\mathfrak{h}$ reflects that the current status of a recursive program is under function name f, right before the execution of the statement labelled ℓ in the function body named f, with values specified by ν and with heap content $\mathfrak{h}$, respectively. Labels in $L_{\mathrm{b}}^{\mathsf{f}}$ correspond to conditional-branching statements indicated by the keyword '**if**' or '**while**' in the function body with function name f; labels in $L_{\mathrm{a}}^{\mathsf{f}}$ correspond to assignment statements indicated by ':=' or **skip**; labels in $L_{\mathrm{c}}^{\mathsf{f}}$ correspond to function calls. Moreover, the label $\ell_{\mathrm{in}}^{\mathsf{f}}$ (resp. $\ell_{\mathrm{out}}^{\mathsf{f}}$) corresponds to the initial statement to be executed (resp. the terminal program counter) in the function body with function name f. Besides, each relation $\rightarrow_{\mathsf{f}}$ specifies the transitions (as triples) between labels within the function body named f, together with additional information specific to different types of labels; finally, each update function updates the current valuation and heap w.r.t its corresponding assignment label, and each value-passing function outputs the initial valuation to the function call to be executed.

It is intuitively clear that any recursive program can be transformed into a corresponding CFG: one first constructs each $\rightarrow_{\mathsf{f}}$ (for $\mathsf{f} \in F$) for each of its function bodies and then group them together. To construct each $\rightarrow_{\mathsf{f}}$, we first construct the partial relation $\rightarrow_{P,\mathsf{f}}$ inductively on the structure of P for each statement P appearing in the function body of f, then define $\rightarrow_{\mathsf{f}}$ as $\rightarrow_{P_{\mathsf{f}},\mathsf{f}}$ for which P_{f} is the function body of f. Each relation $\rightarrow_{P,\mathsf{f}}$ involves two distinguished labels, namely $\ell_{\mathrm{in}}^{P,\mathsf{f}}$ and $\ell_{\mathrm{out}}^{P,\mathsf{f}}$, 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, $\ell_{\mathrm{in}}^{\mathsf{f}}, \ell_{\mathrm{out}}^{\mathsf{f}}$ are defined as $\ell_{\mathrm{in}}^{P_{\mathsf{f}},\mathsf{f}}, \ell_{\mathrm{out}}^{P_{\mathsf{f}},\mathsf{f}}$, respectively.

Due to page limit, we put the detailed transformation in Appendix C. An example of CFGs is illustrated in Example 2.

Example 2. *The CFG of the* mergesort *program of Fig. 2 is depicted in Fig. 4 and Fig. 5, where*

$$\phi := 1 \leq i \wedge i \leq j - 1, \;\; \psi := l \leq j, \;\; \varphi := ar[m] \leq ar[n],$$

id indicates the identity function and f_i, g_i's are given in Fig. 3.

Based on CFGs, we illustrate the semantics of recursive programs as follows. 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 W and its CFG taking the form (†). Let P_{f} be the function body of an arbitrary function name f in F. We first define the notion of *stack element* which captures all information within a function call.

Definition 3 (Stack Elements)**.** *A* stack element *σ (of W) is a letter that represents a triple (f, ℓ, ν) where $\mathsf{f} \in F$, $\ell \in L^{\mathsf{f}}$ and $\nu \in \mathit{Val}_{\mathsf{f}}$.*

Thus, a stack element (f, ℓ, ν) specifies that the current function name is f, the next statement to be executed is the one labelled with ℓ and the current valuation of scalar and array variables w.r.t f is ν. Then we define the notion of *configuration* which captures all information needed to describe the current status of W, i.e., a configuration records both the whole trace of the call stacks and the current heap.

Definition 4 (Configurations)**.** *A* configuration *(of W) is a pair $(w, \mathfrak{h})$ where w is a finite word of stack elements (including the empty word ε) and $\mathfrak{h}$ is a heap.*

The Semantics. Given a stack element $\mathfrak{c}$ and a heap $\mathfrak{h}$, the *run* $\rho(\mathfrak{c}, \mathfrak{h})$ is an infinite sequence $\{(w_j, \mathfrak{h}_j)\}_{j \in \mathbb{N}_0}$ of configurations inductively defined as follows.

- *Initialization:* $w_0 := \mathfrak{c}$ and $\mathfrak{h}_0 := \mathfrak{h}$.
- *Termination:* If either $w_j = \varepsilon$ or w_j is a single letter $\sigma = (\mathsf{f}, \ell_{\mathrm{out}}^{\mathsf{f}}, \nu)$, for $\mathsf{f} \in F$ and $\nu \in \mathit{Val}_{\mathsf{f}}$, then $w_{j+1} := \varepsilon$ and $\mathfrak{h}_{j+1} := \mathfrak{h}_j$. Note that even though we start with some stack element, the termination ensures that the stack is empty.
- *Inductive Step:* Assume that $(w_j, \mathfrak{h}_j) = ((\mathsf{f}, \ell, \nu) \cdot w', \mathfrak{h})$. Then:
  1. *assignment:* if $\ell \in L_{\mathrm{a}}^{\mathsf{f}}$, (ℓ, f, ℓ') is the only triple in $\rightarrow_{\mathsf{f}}$ and $(\nu', \mathfrak{h}') = f(\nu, \mathfrak{h})$, then
     $$\text{(i) } (w_{j+1}, \mathfrak{h}_{j+1}) := ((\mathsf{f}, \ell', \nu') \cdot w', \mathfrak{h}')$$
     whenever $\ell' \neq \ell_{\mathrm{out}}^{\mathsf{f}}$ and otherwise
     $$\text{(ii) } (w_{j+1}, \mathfrak{h}_{j+1}) := (w', \mathfrak{h}');$$
  2. *conditional-branching:* if $\ell \in L_{\mathrm{b}}^{\mathsf{f}}$ and (ℓ, ϕ, ℓ') is the only triple in $\rightarrow_{\mathsf{f}}$ such that $\nu \models_{\mathfrak{h}} \phi$, then
     $$\text{(i) } (w_{j+1}, \mathfrak{h}_{j+1}) := ((\mathsf{f}, \ell', \nu) \cdot w', \mathfrak{h})$$
     whenever $\ell' \neq \ell_{\mathrm{out}}^{\mathsf{f}}$ and otherwise
     $$\text{(ii) } (w_{j+1}, \mathfrak{h}_{j+1}) := (w', \mathfrak{h});$$
  3. *function-call:* if $\ell \in L_{\mathrm{c}}^{\mathsf{f}}$ and $(\ell, (\mathsf{g}, f), \ell')$ is the only triple in $\rightarrow_{\mathsf{f}}$, then
     $$\text{(i) } (w_{j+1}, \mathfrak{h}_{j+1}) := ((\mathsf{g}, \ell_{\mathrm{in}}^{\mathsf{g}}, f(\nu, \mathfrak{h})) \cdot (\mathsf{f}, \ell', \nu) \cdot w', \mathfrak{h})$$
     whenever $\ell' \neq \ell_{\mathrm{out}}^{\mathsf{f}}$ and otherwise
     $$\text{(ii) } (w_{j+1}, \mathfrak{h}_{j+1}) := ((\mathsf{g}, \ell_{\mathrm{in}}^{\mathsf{g}}, f(\nu, \mathfrak{h})) \cdot w', \mathfrak{h}).$$

3. Termination over Recursive Programs

In this section, we describe the notion of termination time for recursive programs and the problem we study. Below we fix a recursive program W and its CFG taking the form (†). We first define the notion of termination time which corresponds directly to the running time of a recursive program.

Definition 5 (Termination Time)**.** *Let $\mathfrak{c}$ be a stack element and $\mathfrak{h}$ be a heap. The* termination time *of the run $\rho(\mathfrak{c}, \mathfrak{h}) = \{(w_j, \mathfrak{h}_j)\}_{j \in \mathbb{N}_0}$, denoted by $T(\mathfrak{c}, \mathfrak{h})$, is defined as*

$$T(\mathfrak{c}, \mathfrak{h}) := \min\{j \mid \textit{ either (i) } w_j = \varepsilon \textit{ or}$$
$$\textit{(ii) } w_j \textit{ is a single letter } \sigma = (\mathsf{f}, \ell_{\mathrm{out}}^{\mathsf{f}}, \nu) \textit{ for } \mathsf{f} \in F \textit{ and } \nu \in \mathit{Val}_{\mathsf{f}}\}$$

where $\min \emptyset := \infty$.

Thus, $T(\mathfrak{c}, \mathfrak{h})$ is the number of steps until termination starting from $\mathfrak{c}, \mathfrak{h}$. Note that in the definition above, along with empty stack, we also consider when there is the last function terminating. The component about last function terminating is technical, because it allows more elegant definitions of measure functions later.

Remark 1. *Definition 5 measures the termination time w.r.t an initial stack element and an initial heap in a stepwise fashion: execution of every triple in $\bigcup_{\mathsf{f} \in F} \rightarrow_{\mathsf{f}}$ takes one unit-time.*

A disadvantage of the notion of termination time is that it crucially depends on the actual content of the heap, which makes the task of analysis difficult. To overcome this difficulty, we focus on a variant of termination time that abstracts away the heap contents. To achieve this, we first define the notion of abstract valuations.

Definition 6 (Abstract Valuations $\overline{\mathit{Val}}_{\mathsf{f}}$)**.** *An* abstract valuation *w.r.t a function name f is a function μ on $V_{\mathrm{p}}^{\mathsf{f}} \cup V_{\mathrm{ar}}^{\mathsf{f}}$ such that $\mu(x) \in \mathbb{Z}$ for all $x \in V_{\mathrm{p}}^{\mathsf{f}}$ and $\mu(ar) \in \mathbb{N}_0$ for all $ar \in V_{\mathrm{ar}}^{\mathsf{f}}$, where $\mu(ar)$ is interpreted directly as a non-negative integer rather than a memory address. An abstract valuation μ w.r.t f is* initial *if $\mu(q) = 0$ whenever $q \in V_{\mathrm{p}}^{\mathsf{f}} \cup V_{\mathrm{ar}}^{\mathsf{f}}$ and q is not involved in the parameter list of f (i.e., a "local variable"). We denote by $\overline{\mathit{Val}}_{\mathsf{f}}$ the set of abstract valuations w.r.t f.*

i	f_i
1	$k \leftarrow \bar{i} + \lfloor \frac{\bar{i}-\bar{j}+1}{2} \rfloor - 1$
2	$(i,j) \leftarrow (\bar{i},\bar{k})$ $(ar,tmp) \leftarrow (\overline{ar},\overline{tmp})$
3	$(i,j) \leftarrow (\bar{k}+1,\bar{j})$ $(ar,tmp) \leftarrow (\overline{ar},\overline{tmp})$
4	$(i,j,k) \leftarrow (\bar{i},\bar{j},\bar{k})$ $(ar,tmp) \leftarrow (\overline{ar},\overline{tmp})$
i	g_i
1	$m \leftarrow \bar{i}$
2	$n \leftarrow \bar{k}+1$
3	$l \leftarrow \bar{i}$
4	$m \leftarrow \overline{m}+1$
5	$n \leftarrow \overline{n}+1$
6	$l \leftarrow \overline{l}+1$

Figure 3. Illustration for Fig. 4 and Fig. 5, where $\overline{q}$ is the concrete entity held by the scalar/array variable q under the valuation at runtime, and each function is represented in the form "$p \leftarrow q$" meaning "q assigned to p" where only the relevant variable is shown for assignment functions.

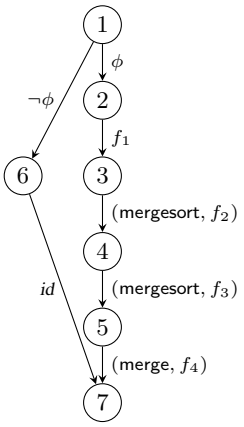

Figure 4. The part of CFG for mergesort in Fig. 2

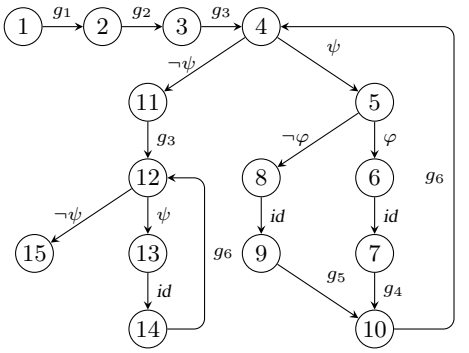

Figure 5. The part of CFG for merge in Fig. 2

Given a valuation $\nu \in \mathit{Val}_{\mathsf{f}}$ and a heap $\mathfrak{h}$, the abstract valuation $\mu[\nu,\mathfrak{h}]$ w.r.t f is defined such that $\mu[\nu,\mathfrak{h}](x) = \nu(x)$ and $\mu[\nu,\mathfrak{h}](ar) = \|\mathfrak{h}(\nu(ar))\|$ for all $x \in V_{\mathrm{p}}^{\mathsf{f}}$ and $ar \in V_{\mathrm{ar}}^{\mathsf{f}}$. The purpose to introduce the notion of abstract valuation is to abstract away array entries and to record solely the length of the array. Since the satisfaction of a propositional array predicate ϕ without the appearance of array entries can be defined directly on abstract valuations μ, we write $\mu \models \phi$ if there exist a valuation ν and a heap $\mathfrak{h}$ such that $\mu = \mu[\nu,\mathfrak{h}]$ and $\nu \models_{\mathfrak{h}} \phi$.

Definition 7 (Function $\overline{f}$ for f)**.** *Given an update function f appearing in the CFG of W and an abstract valuation μ, we define $\overline{f}(\mu) := \mu[f(\nu,\mathfrak{h})]$ for some $\nu,\mathfrak{h}$ such that $\mu = \mu[\nu,\mathfrak{h}]$; and given a value-passing function f appearing in the CFG of W and an abstract valuation μ, we define $\overline{f}(\mu) := \mu[f(\nu,\mathfrak{h}),\mathfrak{h}]$ for some $\nu,\mathfrak{h}$ such that $\mu = \mu[\nu,\mathfrak{h}]$.*

Note that both the $\overline{f}(\mu)$'s are well-defined due to our setting on assignment statements and function-calls. Now we define the abstract stack elements and abstract termination time as follows.

Definition 8 (Abstract Stack Elements and Abstract Termination Time)**.** *An* abstract stack element *is a triple (f,ℓ,μ) where $\mathsf{f} \in F$, $\ell \in L^{\mathsf{f}}$ and $\mu \in \overline{\mathit{Val}}_{\mathsf{f}}$. For each abstract stack element (f,ℓ,μ), the* abstract termination time $\overline{T}(\mathsf{f},\ell,\mu)$ *is defined by*

$$\overline{T}(\mathsf{f},\ell,\mu) := \sup\{T((\mathsf{f},\ell,\nu),\mathfrak{h}) \mid \nu \in \mathit{Val}_{\mathsf{f}}, \mathfrak{h} \text{ is a heap and } \mu = \mu[\nu,\mathfrak{h}]\} \ .$$

Informally, $\overline{T}(\boldsymbol{.})$ abstracts $T(\boldsymbol{.})$ by taking the supremum over all possible array entries, thus naturally provides an upper bound for $T(\boldsymbol{.})$.

In this paper, we study the problem of providing upper bounds for $\overline{T}(\boldsymbol{.})$. A very useful notion to achieve this is an *invariant* for the input recursive program, where an invariant specifies a logical formulae which all "reachable" abstract stack elements must satisfy. We define reachable stack elements and invariants below.

Definition 9 (Reachable stack elements)**.** *A pair $(\mathfrak{c},\mathfrak{h})$, where $\mathfrak{c}$ is a stack element and $\mathfrak{h}$ is a heap, is* reachable *w.r.t a function name f^* and a propositional array predicate ϕ^* over $V_{\mathrm{p}}^{\mathsf{f}^*}, V_{\mathrm{ar}}^{\mathsf{f}^*}$ if there exist a stack element $(\mathsf{f}^*,\ell_{\mathrm{in}}^{\mathsf{f}^*},\nu')$, a finite word w of stack elements, and a heap $\mathfrak{h}'$ such that $\nu' \models_{\mathfrak{h}'} \phi^*$ and the configuration $(\mathfrak{c}\cdot w,\mathfrak{h})$ appears in the run $\rho((\mathsf{f}^*,\ell_{\mathrm{in}}^{\mathsf{f}^*},\nu'),\mathfrak{h}')$. An abstract stack element (f,ℓ,μ) is* reachable *w.r.t $\mathsf{f}^* \in F$ and propositional array predicate ϕ^* over $V_{\mathrm{p}}^{\mathsf{f}^*}, V_{\mathrm{ar}}^{\mathsf{f}^*}$ if there exists a reachable pair $((\mathsf{f},\ell,\nu),\mathfrak{h})$ w.r.t f^*,ϕ^* such that $\mu = \mu[\nu,\mathfrak{h}]$.*

Definition 10 (Invariants)**.** *An* invariant *I w.r.t a function name f^* and a propositional array predicate ϕ^* over $V_{\mathrm{p}}^{\mathsf{f}^*}, V_{\mathrm{ar}}^{\mathsf{f}^*}$ is a function which maps every pair (f,ℓ) satisfying $\mathsf{f} \in F$ and $\ell \in L^{\mathsf{f}} \setminus \{\ell_{\mathrm{out}}^{\mathsf{f}}\}$ to a propositional array predicate over $V_{\mathrm{p}}^{\mathsf{f}}, V_{\mathrm{ar}}^{\mathsf{f}}$ which is without the appearance of array entries (i.e., $ar[x]$'s) or floored expressions (i.e. $\lfloor\boldsymbol{.}\rfloor$), such that for all abstract stack elements (f,ℓ,μ) reachable w.r.t f^*,ϕ^*, we have $\mu \models I(\mathsf{f},\ell)$ whenever $\ell \neq \ell_{\mathrm{out}}^{\mathsf{f}}$.*

The RECTERMBOU **problem.** In this work we consider the algorithmic problem of providing upper bounds for termination time of recursive programs, which we refer to as the RECTERMBOU problem, formally, defined as follows:

- *Input:* a recursive program W, a function name f^*, a propositional array predicate ϕ^* over $V_{\mathrm{p}}^{\mathsf{f}^*}, V_{\mathrm{ar}}^{\mathsf{f}^*}$ without array entries and an invariant I w.r.t f^*,ϕ^*;
- *Output:* a function $h : \overline{\mathit{Val}}_{\mathsf{f}} \to [0,\infty]$ (in certain finite representation) from abstract valuations to $[0,\infty]$ such that $\overline{T}(\mathsf{f}^*,\ell_{\mathrm{in}}^{\mathsf{f}^*},\mu) \leq h(\mu)$ for all $\mu \in \overline{\mathit{Val}}_{\mathsf{f}}$ such that $\mu \models \phi^*$.

4. Measure Functions

In this section we introduce the notion of measure functions for recursive programs. The main contributions of this section are as follows: (1) The notion of ranking functions for non-recursive programs is generalized to the notion of measure functions to present upper bounds on (abstract) termination time for recursive programs. (2) We present soundness (Theorem 1), as well as completeness when there are no array operations (Theorem 2). (3) Finally, we present the notion of *significant labels* which reduces the task of synthesizing measure functions instead of the whole program to the case of significant labels only. From our soundness result (Theorem 1) it follows that it suffices to synthesize measure functions to solve the RECTERMBOU problem, and we consider the synthe-

sis problem in the following section. We start with the notion of measure functions.

In the whole section, we fix a recursive program W together with its CFG taking the form (†), a(n initial) function name $\mathsf{f}^* \in F$ and a propositional array predicate ϕ^* over $V_{\mathrm{p}}^{\mathsf{f}^*}, V_{\mathrm{ar}}^{\mathsf{f}^*}$ without array entries. For each $\mathsf{f} \in F$ and $\ell \in L^{\mathsf{f}} \backslash \{\ell_{\mathrm{out}}^{\mathsf{f}}\}$, we define $D_{\mathsf{f},\ell}$ to be the set of all abstract valuations μ w.r.t f such that (f, ℓ, μ) is reachable w.r.t f^*, ϕ^*.

Definition 11 (Measure Functions)**.** *A* measure function *w.r.t* f^*, ϕ^* *is a function g from the set of abstract stack elements into $[0, \infty]$ such that for all abstract stack elements (f, ℓ, μ), the following conditions hold:*

- ***C1:*** *if $\ell = \ell_{\mathrm{out}}^{\mathsf{f}}$, then $g(\mathsf{f}, \ell, \mu) = 0$;*
- ***C2:*** *if $\ell \in L_{\mathrm{a}}^{\mathsf{f}} \backslash \{\ell_{\mathrm{out}}^{\mathsf{f}}\}$, $\mu \in D_{\mathsf{f},\ell}$ and (ℓ, f, ℓ') is the only triple in $\rightarrow_{\mathsf{f}}$ with source label ℓ, then*

$$g(\mathsf{f}, \ell', \overline{f}(\mu)) + 1 \leq g(\mathsf{f}, \ell, \mu);$$

  (recall that $\overline{f}$ for f in Definition 7);
- ***C3:*** *if $\ell \in L_{\mathrm{c}}^{\mathsf{f}} \backslash \{\ell_{\mathrm{out}}^{\mathsf{f}}\}$, $\mu \in D_{\mathsf{f},\ell}$ and $(\ell, (\mathsf{g}, f), \ell')$ is the only triple in $\rightarrow_{\mathsf{f}}$ with source label ℓ, then*

$$1 + g(\mathsf{g}, \ell_{\mathrm{in}}^{\mathsf{g}}, \overline{f}(\mu)) + g(\mathsf{f}, \ell', \mu) \leq g(\mathsf{f}, \ell, \mu);$$

- ***C4:*** *if $\ell \in L_{\mathrm{b}}^{\mathsf{f}} \backslash \{\ell_{\mathrm{out}}^{\mathsf{f}}\}$, $\mu \in D_{\mathsf{f},\ell}$ and $(\ell, \phi, \ell_1), (\ell, \neg\phi, \ell_2)$ are namely two triples in $\rightarrow_{\mathsf{f}}$ with source label ℓ, then (i)*

$$\mathbf{1}_{\mu \models \phi} \cdot g(\mathsf{f}, \ell_1, \mu) + \mathbf{1}_{\mu \models \neg\phi} \cdot g(\mathsf{f}, \ell_2, \mu) + 1 \leq g(\mathsf{f}, \ell, \mu)$$

  whenever ϕ does not involve array entries and (ii) otherwise

$$\max\{g(\mathsf{f}, \ell_1, \mu), g(\mathsf{f}, \ell_2, \mu)\} + 1 \leq g(\mathsf{f}, \ell, \mu).$$

Informally, C1 is the condition for terminal labels; C2 is the one for assignment labels; C3 is for function-call labels and C4 is for conditional-branching labels. Intuitively, a measure function is a function whose values do not increase along the execution (or run) of a recursive program.

4.1 Soundness and Partial-completeness

In this section we establish soundness and partial-completeness of measure functions. First, we establish the soundness of measure functions, i.e., they provide an upper bound on (abstract) termination time (proof in Appendix D of the supplementary material). The proof is basically a case analysis and inductive argument.

Theorem 1 (Soundness)**.** *For all measure functions g w.r.t f^*, ϕ^*, it holds that for all abstract valuations μ w.r.t f^* such that $\mu \models \phi^*$, we have $\overline{T}(\mathsf{f}^*, \ell_{\mathrm{in}}^{\mathsf{f}^*}, \mu) \leq g(\mathsf{f}^*, \ell_{\mathrm{in}}^{\mathsf{f}^*}, \mu)$.*

Remark 2. *The notion of measure function is a direct generalization of ranking functions to recursion. This can be observed from the fact that once condition C3 is omitted, then Definition 11 coincides with a ranking function for non-recursive programs.*

Below we show that when W does not have array variables, then the abstract termination time function is a measure function for W (proof in Appendix D).

Theorem 2 (Partial Completeness)**.** *If W does not involve array variables, then there exists a measure function g w.r.t f^*, ϕ^* satisfying that for all abstract valuations μ w.r.t f^* such that $\mu \models \phi^*$, we have $\overline{T}(\mathsf{f}^*, \ell_{\mathrm{in}}^{\mathsf{f}^*}, \mu) = g(\mathsf{f}^*, \ell_{\mathrm{in}}^{\mathsf{f}^*}, \mu)$.*

4.2 Significant labels

By Theorem 1, to obtain an upper bound on abstract termination time, an algorithm only needs to synthesize a measure function in a particular form. The computational problem is to synthesize a measure function, and an algorithm needs to synthesize a function over abstract valuations at all labels of all function names. This computational step can be expensive when the input program is "large". In this section we show that the algorithm only needs to synthesize them at *significant labels* only.

Definition 12 (Significant Labels $L_{\mathrm{s}}^{\mathsf{f}}$)**.** *Let $\mathsf{f} \in F$. A label $\ell \in L^{\mathsf{f}}$ is* significant *if it corresponds to one of the following two cases: (i) $\ell = \ell_{\mathrm{in}}^{\mathsf{f}}$ or (ii) $\ell = \ell_{\mathrm{in}}^{P,\mathsf{f}}$ for some while-loop P appearing in the function body of f. We denote by $L_{\mathrm{s}}^{\mathsf{f}}$ the set of significant locations in L^{f}.*

Informally, a significant label is a label where (abstract) 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. In the following definition, we illustrate how one can obtain a measure function from a function defined only on significant labels.

Definition 13 (Transformation from g to $\widehat{g}$)**.** *Let g be a function from*

$$\left\{(\mathsf{f}, \ell, \mu) \mid \mathsf{f} \in F, \ell \in L_{\mathrm{s}}^{\mathsf{f}}, \mu \in \overline{\mathit{Val}}_{\mathsf{f}}\right\}$$

into $[0, \infty]$. The function expanded from g*, denoted by $\widehat{g}$, is a function from the set of all abstract stack elements into $[0, \infty]$ inductively defined through the procedure described as follows.*

1. Initial Step. *If $\ell \in L_{\mathrm{s}}^{\mathsf{f}}$, then $\widehat{g}(\mathsf{f}, \ell, \mu) := g(\mathsf{f}, \ell, \mu)$.*
2. Termination. *If $\ell = \ell_{\mathrm{out}}^{\mathsf{f}}$, then $\widehat{g}(\mathsf{f}, \ell, \mu) := 0$.*
3. Assignment. *If $\ell \in L_{\mathrm{a}}^{\mathsf{f}} \backslash L_{\mathrm{s}}^{\mathsf{f}}$ with (ℓ, f, ℓ') being the only triple in $\rightarrow_{\mathsf{f}}$ and $\widehat{g}(\mathsf{f}, \ell', \boldsymbol{.})$ is already defined, then*

$$\widehat{g}(\mathsf{f}, \ell, \mu) := 1 + \widehat{g}(\mathsf{f}, \ell', \overline{f}(\mu)) \ .$$

   Recall that $\overline{f}$ for f is defined in Definition 7.
4. Conditional-Branching. *If $\ell \in L_{\mathrm{b}}^{\mathsf{f}} \backslash L_{\mathrm{s}}^{\mathsf{f}}$ with (ℓ, ϕ, ℓ_1), $(\ell, \neg\phi, \ell_2)$ being namely the two triples in $\rightarrow_{\mathsf{f}}$ and both $\widehat{g}(\mathsf{f}, \ell_1, \boldsymbol{.})$ and $\widehat{g}(\mathsf{f}, \ell_2, \boldsymbol{.})$ is already defined, then*

$$\widehat{g}(\mathsf{f}, \ell, \mu) := \mathbf{1}_{\mu \models \phi} \cdot \widehat{g}(\mathsf{f}, \ell_1, \mu) + \mathbf{1}_{\mu \models \neg\phi} \cdot \widehat{g}(\mathsf{f}, \ell_2, \mu) + 1$$

   in the case that ϕ does not involve array entries, and otherwise

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

5. Function-Call. *If $\ell \in L_{\mathrm{c}}^{\mathsf{f}} \backslash L_{\mathrm{s}}^{\mathsf{f}}$ with $(\ell, (\mathsf{g}, f), \ell')$ being the only triple in $\rightarrow_{\mathsf{f}}$ and $\widehat{g}(\mathsf{f}, \ell', \boldsymbol{.})$ is already defined, then*

$$\widehat{g}(\mathsf{f}, \ell, \mu) := g(\mathsf{g}, \ell_{\mathrm{in}}^{\mathsf{g}}, \overline{f}(\mu)) + \widehat{g}(\mathsf{f}, \ell', \mu) + 1 \ .$$

Note that in Definition 13, we have not technically shown that $\widehat{g}$ is defined over all abstract stack elements. The following technical lemma shows that the function $\widehat{g}$ is indeed well-defined (proof in Appendix E).

Lemma 1. *For each function g from*

$$\left\{(\mathsf{f}, \ell, \mu) \mid \mathsf{f} \in F, \ell \in L_{\mathrm{s}}^{\mathsf{f}}, \mu \in \overline{\mathit{Val}}_{\mathsf{f}}\right\}$$

into $[0, \infty]$, the function $\widehat{g}$ is well-defined.

Example 3. *In Fig. 2, label* 1 *in function-call* mergesort *and labels* $1, 4, 12$ *in* merge *are significant labels.*

With Definition 13, it is possible to obtain a measure function by imposing constraints on a function defined at significant labels only. In order to avoid direct manipulation of $D_{\mathsf{f},\ell}$'s (reachable abstract valuations), we further fix an invariant w.r.t f^* and use it to over-approximate $D_{\mathsf{f},\ell}$'s. Then we obtain the following proposition (proof in Appendix F).

Proposition 1. *Let g be a function from*

$$\left\{(\mathsf{f}, \ell, \mu) \mid \mathsf{f} \in F, \ell \in L_{\mathrm{s}}^{\mathsf{f}}, \mu \in \overline{\mathit{Val}}_{\mathsf{f}}\right\}$$

into $[0, \infty]$ and $\widehat{g}$ be defined as in Definition 13. Let I be an invariant w.r.t f^*, ϕ^*. Consider that for all abstract stack elements (f, ℓ, μ) such that $\ell \in L_{\mathrm{s}}^{\mathsf{f}}$ and $\mu \models I(\mathsf{f}, \ell)$, the following conditions hold:

- ***C2':*** *if $\ell \in L^{\mathsf{f}}_{\mathrm{a}}$ and (ℓ, f, ℓ') is the only triple in $\rightarrow_{\mathsf{f}}$ with source label ℓ, then $\widehat{g}(\mathsf{f}, \ell', \overline{f}(\mu)) + 1 \leq \widehat{g}(\mathsf{f}, \ell, \mu)$;*
- ***C3':*** *if $\ell \in L^{\mathsf{f}}_{\mathrm{c}}$ and $(\ell, (\mathsf{g}, f), \ell')$ is the only triple in $\rightarrow_{\mathsf{f}}$ with source label ℓ, then $1 + \widehat{g}(\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, \overline{f}(\mu)) + \widehat{g}(\mathsf{f}, \ell', \mu) \leq \widehat{g}(\mathsf{f}, \ell, \mu)$;*
- ***C4':*** *if $\ell \in L^{\mathsf{f}}_{\mathrm{b}}$ and $(\ell, \phi, \ell_1), (\ell, \neg\phi, \ell_2)$ are namely two triples in $\rightarrow_{\mathsf{f}}$ with source label ℓ, then (i)*

  $$\mathbf{1}_{\mu \models \phi} \cdot \widehat{g}(\mathsf{f}, \ell_1, \mu) + \mathbf{1}_{\mu \models \neg\phi} \cdot \widehat{g}(\mathsf{f}, \ell_2, \mu) + 1 \leq \widehat{g}(\mathsf{f}, \ell, \mu)$$

  whenever ϕ does not involve array entries and (ii) otherwise

  $$\max\{\widehat{g}(\mathsf{f}, \ell_1, \mu), \widehat{g}(\mathsf{f}, \ell_2, \mu)\} + 1 \leq \widehat{g}(\mathsf{f}, \ell, \mu).$$

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

Note that from Definition 13, conditions C2'-C4' essentially specify constraints on g. This allows an algorithm to synthesize a measure function only at significant labels.

5. The Synthesis Algorithm

In this section, we present our algorithm, namely SYNALGO, for synthesizing measure functions, which by Theorem 1 is also a sound approach for the RECTERMBOU problem. The synthesis algorithm is designed to synthesize one function over abstraction valuations at each function name and appropriate significant label, so that conditions C2'-C4' in Proposition 1 are fulfilled.We will present the main conceptual details of our algorithm, and some technical details are relegated to Appendix G. We first present an overview of our solution.

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

1. *Step 1.* Since one key aspect of our result is to obtain bounds of the form $\mathcal{O}(n \log n)$ as well as $\mathcal{O}(n^x)$, where x is not an integer, we first consider general form of upper bounds that involve logarithm and exponentiation (Step 1(a)), and then consider templates with the general form of upper bounds for significant labels (Step 1(b)).
2. *Step 2.* The second step considers the template generated in Step 1 for significant labels and generate templates for all labels. This step relatively straightforward.
3. *Step 3.* The third step establishes constraint triples according to the invariant given by the input and the template obtained in Step 2. This step is also straightforward.
4. *Step 4.* The fourth step is the significant step which involves solving the constraint triples generated in Step 3. The first substep (Step 4(a)) is to consider abstractions of logarithm, exponentiation, and floor expressions. The next step (Step 4(b)) requires to obtain linear constraints over the abstract variables. We use Farkas' lemma for a lower bound for abstract variables and use Lagrange's Mean-Value Theorem (LMVT) to obtain sound and linear constraints for abstract logarithmic and exponentiation variables. The next (and the final) step (Step 4(c)) requires to solve the unknown coefficients of the template. This requires the solution of positive polynomial over a polyhedron, and we use Handelman's theorem to solve this step.

Informal illustration of the conceptual steps. Since the most interesting conceptual contributions are in Step 4, we present an illustration of the key ideas on simple examples.

Example 4. *Consider that the task is to synthesize a measure function for Merge-Sort which takes a form similar to $2 \cdot n \cdot \log n$ at some label, where n is the variable representing the length of the array. Then due to condition C3 our algorithm needs to detect that*

$$2 \cdot n \cdot \ln n - 2 \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \ln \left\lfloor \frac{n}{2} \right\rfloor - 2 \cdot \left\lceil \frac{n}{2} \right\rceil \cdot \ln \left\lceil \frac{n}{2} \right\rceil - n \geq 0 \quad (1)$$

under the invariant condition $n \geq 2$. The algorithm first abstracts the logarithm and floored expression as independent fresh variables as follows:

$$w_1 := \left\lceil \frac{n}{2} \right\rceil, w_2 := \left\lfloor \frac{n}{2} \right\rfloor, w_3 := n$$
$$u_1 := \ln \left\lceil \frac{n}{2} \right\rceil, u_2 := \ln \left\lfloor \frac{n}{2} \right\rfloor, u_3 := \ln n$$

where w_3 is introduced only for convenience. By applying Farkas' Lemma with $n \geq 2$ and properties of the floor operation, our algorithm generates the following inequalities:

$$w_1 \geq 1, \;\; w_2 \geq 1, \;\; \frac{w_3 - 1}{2} \leq w_2 \leq \frac{w_3}{2} \leq w_1 \leq \frac{w_3 + 1}{2}. \quad (2)$$

Below we present a set of constraints, which we refer as (‡) (and we will use them later as well, in Example 9). Based on (2), the algorithm generates the following inequalities (‡) through (a) properties of logarithm, (b) the fact that the global minima of the function $x \mapsto \frac{x}{\log x}$ over $(0, \infty)$ is attained at constant e, and (c) applying Lagrange's Mean-Value Theorem (LMVT) over the logarithmic function $x \mapsto \ln x$.

- $u_1 \geq 0, u_2 \geq 0, u_3 \geq \ln 2$*;*
- $w_j \geq e \cdot u_j$ *(for $1 \leq j \leq 3$);*
- $u_2 \leq u_3 - \ln 2$ *and* $u_3 \leq \ln 2 + u_2 + \frac{1}{2}$*;*
- $u_1 \leq u_3 - \ln 2 + \frac{1}{2}$ *and* $u_3 \leq u_1 + \ln 2$*;*
- $u_2 \leq u_1$ *and* $u_1 \leq u_2 + 1$.

The first inequality above of (‡) use Farkas' Lemma, the second is the global minimality, and all the rests use LMVT. While above we have transcendental numbers $\ln 2$ and e, for our purpose it suffices to replace them by approximate constants, such as constants 2.7182 and 2.7183 (resp., 0.6931 and 0.6932) for lower and upper approximations of e (resp., $\ln 2$). Finally, the algorithm transforms (1) into

$$2 \cdot w_3 \cdot u_3 - 2 \cdot w_1 \cdot u_1 - 2 \cdot w_2 \cdot u_2 - w_3 \geq 0 \quad (3)$$

and then checks whether (3) holds under the generated inequalities. Observe that (3) is non-linear, which we show how to solve in polynomial time through the sound form of Handelman's Theorem.

Example 5. *Consider that our task is to synthesize a measure function for Karatsuba's algorithm [41] for polynomial multiplication which takes a form similar to $2 \cdot n^{1.8}$ at some label, where n represents the maximal degree of the input polynomials and is a power of 2. Then due to condition C3, our algorithm needs to detect:*

$$2 \cdot n^{1.6} - 4 \cdot \left(\frac{n}{2}\right)^{1.6} - n \geq 0 \quad (4)$$

under the invariant condition $n \geq 2$. The algorithm first abstracts $n^{1.6}, n^{0.6}$ as stand-alone variables u, v, respectively. Then the algorithm generates the following inequalities using Farkas' Lemma (the first two inequalities) and through properties of exponentiation (the rest of them):

$$u \geq 2^{1.6}, v \geq 2^{0.6}, u \geq 2^{0.6} \cdot n, u \geq 2 \cdot v, n \geq 2^{0.4} \cdot v. \quad (5)$$

Finally, the algorithm transforms (4) into

$$2 \cdot u - 4 \cdot \left(\frac{1}{2}\right)^{1.6} \cdot u - n \geq 0 \quad (6)$$

and checks whether (6) holds under (5) together with $n \geq 2$ through the sound form of Handelman's Theorem. Also note that above we have constants such as $2^{1.8}$ and $2^{0.8}$, but again we can use appropriate lower and upper rational constants for approximation.

Below we fix an input recursive program W, with its CFG taking the form (†), an input function name $\mathsf{f}^* \in F$, an input propositional array predicate ϕ^* and an input invariant I w.r.t f^*, ϕ^*. We will present our algorithm in four steps, and let us first consider our running example.

Example 6. *Consider the recursive program* mergesort *as our running example for this section. Alongside the input program, the other inputs are as follows: (i)* $\mathsf{f}^* :=$ mergesort*; (ii)* $\phi^*(i,j) := 1 \leq i \wedge i \leq j$*; and (iii) the input invariant* I *is given by*

- $I(\mathsf{mergesort}, 1) = I(\mathsf{merge}, 1) := \phi^*(i,j)$*, and*
- $I(\mathsf{merge}, 4) = I(\mathsf{merge}, 12) := \phi^*(i,j) \wedge l \leq j+1$.

Note that due to Proposition 1, it suffices to specify invariant at significant labels only.

5.1 Step 1 of SYNALGO

Step 1(a): General Form of Upper-Bound Functions. In order to capture worst-case complexity upper-bounds of recursive programs, the general form of an upper-bound function incorporates terms from program variables and array lengths (i.e, $\|\centerdot\|$). Moreover, in order to capture more intricate complexity upper-bound related to positive quantities (such as array lengths), our algorithm incorporates two types of extensions of terms.

1. *Logarithmic terms.* The first extension, which we call $\log$-extension, is the extension with terms from
$$\ln x, \quad \ln(x-y+1), \quad \ln\|ar\| \tag{7}$$
where x, y are program variables appearing in the parameter list of f, ar is an array variable appearing in the parameter list of f and $\ln(\centerdot)$ is the natural logarithm function with base e.
2. *Exponentiation terms.* The second extension, which we call exp-extension, is the extension with terms from
$$x^k, \quad (x-y+1)^k, \quad \|ar\|^k \tag{8}$$
where x, y are program variables appearing in the parameter list of f, ar is an array variable appearing in the parameter list of f and k is a positive *rational* (i.e., not necessarily integer) exponent.

The intuition is that x (resp. $x-y+1$) may represent the length between array indices 1 and x or other positive quantities (resp. the length between array indices y and x).

Setup of General Form of Upper-Bound Functions. Then we set the general form of an upper-bound function for any function name f and any $\ell \in L^{\mathsf{f}}_{\mathrm{s}}$ to be a function on $\overline{\mathit{Val}}_{\mathsf{f}}$ defined through a finite sum
$$\sum_i c_i \cdot g_i \tag{9}$$
where each c_i is a constant rational scalar and each g_i is a finite product of terms from program variables in $V^{\mathsf{f}}_{\mathrm{p}}$, lengths of array variables in $\left\{\|ar\| \mid ar \in V^{\mathsf{f}}_{\mathrm{ar}}\right\}$ and extensions from either (7) or (8).

Function $[\![\mathfrak{e}]\!]$ *from* $\mathfrak{e}$. A finite sum $\mathfrak{e}$ in the form (9) defines a function $[\![\mathfrak{e}]\!]$ on $\overline{\mathit{Val}}_{\mathsf{f}}$ in the way that for each $\mu \in \overline{\mathit{Val}}_{\mathsf{f}}$:

1. $[\![\mathfrak{e}]\!](\mu) := 0$ if $\mu \not\models I(\mathsf{f}, \ell)$;
2. otherwise, $[\![\mathfrak{e}]\!](\mu)$ is defined as the evaluation result of $\mathfrak{e}$ when assigning $\mu(x)$ to each $x \in V^{\mathsf{f}}_{\mathrm{p}}$ and $\mu(ar)$ to each occurrence of $\|ar\|$.

Note that in the definition of $[\![\mathfrak{e}]\!]$, we do not consider the case when $\log$ or exponentiation is undefined. However, we will see later that $\log$ and exponentiation will always be well-defined.

Step 1(b): Templates.

For complexity bounds, as in all previous works (such as [9, 10, 12, 15, 43, 46, 50]), we consider that the template for the form of upper bounds is chosen manually, (where the template consists of the term extension from either $\log$-extension or exp-extension with exponent k and a natural number $m \geq 1$). The algorithm first sets up a template η for a measure function w.r.t f^*, ϕ^*. In detail, the algorithm builds η by assigning to each function name f and significant label $\ell \in L^{\mathsf{f}}_{\mathrm{s}}$ an expression $\eta(\mathsf{f}, \ell)$ in the form (9), except for that (i) c_i's in (9) are interpreted as distinct *scalar variables* whose actual values are to be synthesized and (ii) g_i's in (9) range over all finite products of terms with at most m multiplicands; and in order to ensure that logarithm and exponentiation are well-defined over $I(\mathsf{f}, \ell)$, we consider that:

> (§) $\ln x, x^k$ (resp. $\ln(x-y+1), (x-y+1)^k$) appear in $\eta(\mathsf{f}, \ell)$ only when $x-1 \geq 0$ (resp. $x-y \geq 0$) can be inferred from the propositional array predicate $I(\mathsf{f}, \ell)$.

To infer $x-1 \geq 0$ or $x-y \geq 0$ from $I(\mathsf{f}, \ell)$, we utilize Farkas' Lemma [16].

Theorem 3 (Farkas' Lemma [16, 45])**.** *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. [44]) provided that $I(\mathsf{f}, \ell)$ involves only linear (degree-1) polynomials.

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

Example 7. *Consider our running example* mergesort *depicted in Fig. 2. The algorithm establishes an expression in form (9) for each* $\eta(\mathsf{mergesort}, 1, \centerdot)$, $\eta(\mathsf{merge}, 1, \centerdot)$, $\eta(\mathsf{merge}, 4, \centerdot)$ *and* $\eta(\mathsf{merge}, 12, \centerdot)$. *For the sake of succinct illustration, we consider the following simplified sub-forms:*

- $\eta(\mathsf{mergesort}, 1, \centerdot)$ *is represented by*
$$\mathbf{1}_{I(\mathsf{mergesort},1)} \cdot \mathfrak{t}_1(i,j) + \mathbf{1}_{\neg I(\mathsf{mergesort},1)} \cdot 0$$
where $\mathfrak{t}_1(i,j) := c_{1,1} + c_{1,2} \cdot (j-i+1) \cdot \ln(j-i+1)$.
- $\eta(\mathsf{merge}, 1, \centerdot)$ *is represented by*
$$\mathbf{1}_{I(\mathsf{merge},1)} \cdot \mathfrak{t}_2(i,j) + \mathbf{1}_{\neg I(\mathsf{merge},1)} \cdot 0$$
where $\mathfrak{t}_2(i,j) := c'_{1,1} + c'_{1,2} \cdot (j-i+1)$.
- $\eta(\mathsf{merge}, 4, \centerdot)$ *is represented by*
$$\mathbf{1}_{I(\mathsf{merge},4)} \cdot \mathfrak{t}_3(i,j,l) + \mathbf{1}_{\neg I(\mathsf{merge},4)} \cdot 0$$
where $\mathfrak{t}_3(i,j,l) := c'_{4,1} + c'_{4,2} \cdot (j-l+1) + c'_{4,3} \cdot (j-i+1)$.
- $\eta(\mathsf{merge}, 12, \centerdot)$ *is represented by*
$$\mathbf{1}_{I(\mathsf{merge},12)} \cdot \mathfrak{t}_4(j,l) + \mathbf{1}_{\neg I(\mathsf{merge},12)} \cdot 0$$
where $\mathfrak{t}_4(j,l) := c'_{12,1} + c'_{12,2} \cdot (j-l+1)$.

In the forms above, all $c_{m,n}, c'_{m,n}$*'s are scalar variables.*

5.2 Step 2 of SYNALGO

Step 2: Computation of $\widehat{[\![\eta]\!]}$**.** Recall the transformation of a function g to $\hat{g}$ in Definition 13, and the function $[\![\mathfrak{e}]\!]$ for $\mathfrak{e}$. We now consider a template η, and the above two transformations give us $\widehat{[\![\eta]\!]}$. Formally, based on the template η, the algorithm computes $\widehat{[\![\eta]\!]}$ by Definition 13, while treating scalar variables appearing in η as undetermined constants. Then $\widehat{[\![\eta]\!]}$ is a function parametric over the

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

$$\max\left\{\sum_j \mathbf{1}_{\phi_{1j}} \cdot h_{1j},\ \ldots,\ \sum_j \mathbf{1}_{\phi_{mj}} \cdot h_{mj}\right\} \tag{10}$$

where

1. each ϕ_{ij} is a propositional array predicate over $V^{\mathsf{f}}_{\mathrm{p}}, V^{\mathsf{f}}_{\mathrm{ar}}$ such that (i) $\mathbf{1}_{\phi_{ij}}$ is interpreted as the indicator function $\mu \mapsto \mathbf{1}_{\mu \models \phi}$ on $\overline{\mathit{Val}}_{\mathsf{f}}$ and (ii) 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 (9) with the difference that (i) each c_i is either a scalar or a scalar variable appearing in η and (ii) each g_i is a finite product whose every multiplicand is either some $x \in V^{\mathsf{f}}_{\mathrm{p}}$, or some $\|ar\|$, or some $\lfloor \mathfrak{e} \rfloor$ with $\mathfrak{e}$ being an instance of $\langle \mathit{pexpr} \rangle$, or some $\ln \mathfrak{e}$ (or $\mathfrak{e}^k$) with $\mathfrak{e}$ being an instance of $\langle \mathit{pexpr} \rangle$.

For this step we use that array predicates are in DNF. For detailed description see Appendix G.

Example 8. *Consider again our running example* mergesort *(cf. Fig. 2). After the computation of $\widehat{[\![\eta]\!]}$, we obtain the following:*

- $\widehat{[\![\eta]\!]}(\mathsf{mergesort}, 2, \centerdot)$ *is (represented) by*

$$4 + \mathbf{1}_{\phi^*(i,j)} \cdot \mathfrak{t}_2(i,j) + \mathbf{1}_{\phi^*(i+w,j)} \cdot \mathfrak{t}_1(i+w,j) + \mathbf{1}_{\phi^*(i,i+w-1)} \cdot \mathfrak{t}_1(i,i+w-1)$$

  with w representing the floored expression $\lfloor \frac{j-i+1}{2} \rfloor$, and is further transformed by the algorithm equivalently into

$$4 + \mathbf{1}_{\varphi_1} \cdot (\mathfrak{t}_2(i,j) + \mathfrak{t}_1(i+w,j) + \mathfrak{t}_1(i,i+w-1)) + \mathbf{1}_{\varphi_2 \vee \varphi_3} \cdot \mathfrak{t}_1(i+w,j) + \mathbf{1}_{\varphi_4} \cdot \mathfrak{t}_1(i,i+w-1)$$

  with
  - $\varphi_1 := 1 \leq i \wedge i + w \leq j \wedge 1 \leq w$, *and*
  - $\varphi_2 := i \leq 0 \wedge 1 \leq i + w \wedge i + w \leq j$, *and*
  - $\varphi_3 := i \geq j + 1 \wedge 1 \leq i + w \wedge i + w \leq j \wedge w \leq 0$, *and*
  - $\varphi_4 := i \geq j + 1 \wedge 1 \leq i \wedge 1 \leq w$,

  which conforms to form (10), where the constant 4 lies outside every summand, every summand being identically zero is omitted and some simplification over logical formulae which preserves satisfaction relation are carried out.
- $\widehat{[\![\eta]\!]}(\mathsf{mergesort}, 6, \centerdot)$ *is the function with constant value 1;*
- $\widehat{[\![\eta]\!]}(\mathsf{merge}, 2, \centerdot)$ *by* $2 + \mathbf{1}_{1 \leq i \wedge i \leq j \wedge i \leq j+1} \cdot \mathfrak{t}_3(i,j,i)$*;*
- $\widehat{[\![\eta]\!]}(\mathsf{merge}, 5, \centerdot)$ *by* $4 + \mathbf{1}_{1 \leq i \wedge i \leq j \wedge l+1 \leq j+1} \cdot \mathfrak{t}_3(i,j,l+1)$*;*
- $\widehat{[\![\eta]\!]}(\mathsf{merge}, 11, \centerdot)$ *by* $1 + \mathbf{1}_{1 \leq i \wedge i \leq j \wedge i \leq j+1} \cdot \mathfrak{t}_4(j,i)$*;*
- $\widehat{[\![\eta]\!]}(\mathsf{merge}, 13, \centerdot)$ *by* $2 + \mathbf{1}_{1 \leq i \wedge i \leq j \wedge l+1 \leq j+1} \cdot \mathfrak{t}_4(j,l+1)$*;*
- $\widehat{[\![\eta]\!]}(\mathsf{merge}, 15, \centerdot)$ *is the function with constant value 0.*

5.3 Step 3 of SYNALGO

Illustration of Constraint Triples. By applying non-negativity and C2'-C4' to $\widehat{[\![\eta]\!]}$, the algorithm establishes constraints over scalar variables appearing in η. We organize each constraint as a triple $(\mathsf{f}, \phi, \mathfrak{e})$ where

- $\mathsf{f} \in F$,
- ϕ is a propositional array predicate over $V^{\mathsf{f}}_{\mathrm{p}}, V^{\mathsf{f}}_{\mathrm{ar}}$ which is without array entries and is a conjunction of atomic formulae (i.e., formulae of the form $\mathfrak{e}' \geq 0$ with $\mathfrak{e}'$ being an arithmetic array expression without array entries), and
- $\mathfrak{e}$ is an expression taking the form similar to (9) with the difference that (i) each c_i is either a scalar, or a scalar variable c appearing in η, or its reverse $-c$, and (ii) each g_i is a finite product whose every multiplicand is either some $x \in V^{\mathsf{f}}_{\mathrm{p}}$, or some $\|ar\|$ with $ar \in V^{\mathsf{f}}_{\mathrm{ar}}$, or some $\lfloor \mathfrak{e} \rfloor$ with $\mathfrak{e}$ being an instance of $\langle \mathit{pexpr} \rangle$, or some $\ln \mathfrak{e}$ (or $\mathfrak{e}^k$) with $\mathfrak{e}$ being an instance of $\langle \mathit{pexpr} \rangle$ over $V^{\mathsf{f}}_{\mathrm{p}}$.

For each expression taking the form similar to (9), the function $[\![\mathfrak{e}]\!]$ on $\overline{\mathit{Val}}_{\mathsf{f}}$ is defined in the way such that each $[\![\mathfrak{e}]\!](\mu)$ is the evaluation result of $\mathfrak{e}$ when assigning $\mu(x)$ to each $x \in V^{\mathsf{f}}_{\mathrm{p}}$ and $\mu(ar)$ to each occurrence of $\|ar\|$; under (§) (of Step 1(b)), logarithm and exponentiation will always be well-defined.

A constraint triple $(\mathsf{f}, \phi, \mathfrak{e})$ encodes the following logical formula

$$\forall \mu \in \overline{\mathit{Val}}_{\mathsf{f}}.\ (\mu \models \phi \rightarrow [\![\mathfrak{e}]\!](\mu) \geq 0) \quad .$$

Multiple constraint triples are grouped into a single logical formula through conjunction.

Step 3: Establishment of Constraint Triples. Based on $\widehat{[\![\eta]\!]}$, the algorithm generates constraint triples at each significant label of some function name, then group all generated constraint triples together in a conjunctive way. To be more precise, at every significant label ℓ of some function name f, the algorithm generates constraint triples through non-negativity of measure functions and condition C2'–C4'; after generating the constraint triples for each significant label, the algorithm group them together in the conjunctive fashion to form a single collection of constraint triples. For a detailed procedure and illustration see Appendix G.

5.4 Step 4 of SYNALGO

Step 4: Solving Constraint Triples. To check whether the logical formula encoded by generated constraint triples is valid, the algorithm follows a sound method which treats each multiplicand, other than program variables, in the form (10) as a stand-alone variable, and transforms the validity of the formula into a system of linear equalities over scalar variables appearing in η through Handelman's Theorem.

Notations for the algorithm. In the following, we describe how the algorithm transforms a constraint triple $(\mathsf{f}, \phi, \mathfrak{t})$ into a collection of linear equalities involving scalar variables in η. Below given any finite set Γ of polynomials over n variables, we denote by $\mathsf{Sat}(\Gamma) := \{\mathbf{x} \in \mathbb{R}^n \mid f(\mathbf{x}) \geq 0 \text{ for all } f \in \Gamma\}$. Let $(\mathsf{f}, \phi, \mathfrak{t})$ be any constraint triple such that $\phi = \bigwedge_j \mathfrak{e}_j \geq 0$.

Step 4(a): Abstraction of Logarithm, Exponentiation, and Floored Expressions. The first substep involves the following computational steps, where item 2-4 use variables for abstraction, and item 7 is approximation of floor expression, 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. *Logarithm, Exponentiation, and Floored expressions.* Next, the algorithm computes the following subsets of $\langle \mathit{pexpr} \rangle$:
   - $\mathcal{E}_L := \{\mathfrak{e} \mid \ln \mathfrak{e} \text{ appears in } \mathfrak{t} \text{ (as sub-expression)}\}$.
   - $\mathcal{E}_E := \{\mathfrak{e} \mid \mathfrak{e}^k \text{ appears in } \mathfrak{t} \text{ (as sub-expression)}\}$.
   - $\mathcal{E}_F := \{\mathfrak{e} \mid \mathfrak{e} \text{ appears in } \mathfrak{t} \text{ and takes the form } \lfloor \frac{\centerdot}{c} \rfloor\}$.

   Let $\mathcal{E} := \mathcal{E}_L \cup \mathcal{E}_E \cup \mathcal{E}_F$.
3. *Variables for Array-Lengths.* Next, for each $ar \in V^{\mathsf{f}}_{\mathrm{ar}}$ that appears in $\mathfrak{t}$, the algorithm introduces a fresh variable l_{ar} which indicates $\|ar\|$.
4. *Variables for Logarithm, Exponentiation, Floor expressions.* Next, for each $\mathfrak{e} \in \mathcal{E}$, the algorithm establishes fresh variables:
   - 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 $\mathfrak{e}^k$ and $v'_{\mathfrak{e}}$ indicates $\mathfrak{e}^{k-1}$ for $\mathfrak{e} \in \mathcal{E}_E$.
   - Fresh variable $w_{\mathfrak{e}}$ indicating $\mathfrak{e}$ for $\mathfrak{e} \in \mathcal{E}_F$.

   After this step, the algorithm sets N to be the number of all variables (i.e., all scalar variables and all fresh variables) and

consider an implicit linear order over the variables so that a valuation of these variables can be treated as a vector in $\mathbb{R}^N$.

5. *Variable Substitution: from $\mathfrak{e}$ to $\tilde{\mathfrak{e}}$.* Next, for each $\mathfrak{e}$ which is either (i) $\mathfrak{t}$, or some (ii) $\mathfrak{e}_j$, or (iii) some expression in $\mathcal{E}$, the algorithm computes $\tilde{\mathfrak{e}}$ as the expression obtained from $\mathfrak{e}$ by substituting every l_{ar} for $\|ar\|$, every $u_{\mathfrak{e}'}$ for $\ln \mathfrak{e}'$, every $v_{\mathfrak{e}'}$ for $(\mathfrak{e}')^k$, and every $w_{\mathfrak{e}'}$ satisfying that there is an appearance of $\mathfrak{e}'$ (at some syntactical sub-part) in $\mathfrak{e}$ such that this appearance of $\mathfrak{e}'$ does not appear in some another $\mathfrak{e}'' \in \mathcal{E}_2$ appearing in $\mathfrak{e}$.
6. *Importing ϕ into Γ.* The algorithm adds all $\widetilde{\mathfrak{e}_j}$ into Γ.
7. *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 on $w_{\mathfrak{e}}$ recursively as follows:
   - *Base Step.* If $\mathfrak{e}'$ involves no nested floored expression, then the algorithm into Γ either (i) adds $\widetilde{\mathfrak{e}'} - c \cdot w_{\mathfrak{e}}$ 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) adds $c \cdot w_{\mathfrak{e}} - \widetilde{\mathfrak{e}'}$ 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, the algorithm finds the largest $t_{\mathfrak{e}'}$ through Farkas' Lemma such that
   $$\forall \mathbf{x} \in \mathbb{R}^N. \left(\mathbf{x} \in \mathsf{Sat}(\Gamma) \to \tilde{\mathfrak{e}'}(\mathbf{x}) \geq t_{\mathfrak{e}'}\right)$$
   holds; if such $t_{\mathfrak{e}'}$ exists, then the algorithm adds the constraint $w_{\mathfrak{e}} \geq \left\lfloor \frac{t_{\mathfrak{e}'}}{c} \right\rfloor$ into Γ.
   - *Recursive Step.* If $\mathfrak{e}'$ involves some nested floored expression, then the algorithm proceeds with $\tilde{\mathfrak{e}'}$ recursively in the same way as for the Base Step.
8. *Emptiness Checking.* The algorithm checks whether $\mathsf{Sat}(\Gamma)$ is empty or not in polynomial time in the size of Γ (cf. [44]). If $\mathsf{Sat}(\Gamma) = \emptyset$, then the algorithm stops with no linear inequalities generated; otherwise, the algorithm proceeds to the remaining steps.

For the next step we will use Lagrange's Mean-Value Theorem (LMVT) to abstract logarithmic and exponential terms.

Theorem 4 (Lagrange's Mean-Value Theorem [5, 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 handles extra linear constraints to be added into Γ. We present the details for logarithm and similar technical details for exponentiation terms are in Appendix G. Below we denote by $\mathcal{E}_1$ the set $\mathcal{E}_L \cup \mathcal{E}_E$. In the first three items we use the $\tilde{\mathfrak{e}}$ notation introduced in the Variable substitution (item 5) of Step 4(a).

1. *Lower-Bound for Expressions in $\mathcal{E}_1$.* For each $\mathfrak{e} \in \mathcal{E}_1$, we find the largest $t_{\mathfrak{e}} \in \mathbb{R}$ such that the formula
   $$\forall \mathbf{x} \in \mathbb{R}^N. (\mathbf{x} \in \mathsf{Sat}(\Gamma) \to \tilde{\mathfrak{e}}(\mathbf{x}) \geq t_{\mathfrak{e}})$$
   holds, where $\tilde{\mathfrak{e}}$ is deemed as a polynomial over all variables and $\tilde{\mathfrak{e}}(\mathbf{x})$ is the result of polynomial evaluation under the correspondence between variables and coordinates of $\mathbf{x}$ specified by the linear order This can be solved by Farkas' Lemma and linear programming, since $\tilde{\mathfrak{e}}$ is linear. Note that as long as $\mathsf{Sat}(\Gamma) \neq \emptyset$ holds, it follows from (§) (of 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 $\mathcal{E}_1$.* For each pair $(\mathfrak{e}, \mathfrak{e}') \in \mathcal{E}_1 \times \mathcal{E}_1$ 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 formulae
   $$\forall \mathbf{x} \in \mathbb{R}^N. \left[\mathbf{x} \in \mathsf{Sat}(\Gamma) \to \widetilde{\mathfrak{e}}(\mathbf{x}) - \left(r_{\mathfrak{e},\mathfrak{e}'} \cdot \widetilde{\mathfrak{e}'}(\mathbf{x}) + b_{\mathfrak{e},\mathfrak{e}'}\right) \geq 0\right]$$
   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 $[\![\Gamma]\!]$ is non-empty, 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 $\mathcal{E}_1$.* For each pair $(\mathfrak{e}, \mathfrak{e}') \in \mathcal{E}_1 \times \mathcal{E}_1$ 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 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 then finds the minimal value $(\mathsf{r}^*_{\mathfrak{e},\mathfrak{e}'}, \mathsf{b}^*_{\mathfrak{e},\mathfrak{e}'})$ similarly as above.
4. *Constraints for Logarithm.* For each variable $u_{\mathfrak{e}}$, the algorithm adds into Γ the polynomial expression
   $$\tilde{\mathfrak{e}} - \left(\mathbf{1}_{t_{\mathfrak{e}} \leq e} \cdot e + \mathbf{1}_{t_{\mathfrak{e}} > e} \cdot \frac{t_{\mathfrak{e}}}{\ln t_{\mathfrak{e}}}\right) \cdot u_{\mathfrak{e}}$$
   due to the fact that the function $d \mapsto \frac{d}{\ln d}$ ($d \geq 1$) has global minima at e, and the polynomial expression $u_{\mathfrak{e}} - \ln t_{\mathfrak{e}}$ due to the definition of $t_{\mathfrak{e}}$.
5. *Constraints for Exponentiation.* The details are in Appendix G.
6. *Mutual No-Smaller-Than Inequalities over $u_{\mathfrak{e}}$'s.* For each pair $(\mathfrak{e}, \mathfrak{e}') \in \mathcal{E}_1 \times \mathcal{E}_1$ 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 (recall $[\![\mathfrak{e}]\!]$ from $\mathfrak{e}$ defined before Step 1(b))) the following:
   $$\begin{aligned}
   \ln [\![\mathfrak{e}]\!] &\geq \ln r^*_{\mathfrak{e},\mathfrak{e}'} + \ln \left([\![\mathfrak{e}']\!] + \frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}}\right) \\
   &= \ln r^*_{\mathfrak{e},\mathfrak{e}'} + \ln [\![\mathfrak{e}']\!] \\
   &\qquad + \left(\ln \left([\![\mathfrak{e}']\!] + \frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}}\right) - \ln [\![\mathfrak{e}']\!]\right) \\
   &\geq \ln r^*_{\mathfrak{e},\mathfrak{e}'} + \ln [\![\mathfrak{e}']\!] \\
   &\qquad - \mathbf{1}_{b^*_{\mathfrak{e},\mathfrak{e}'} < 0} \cdot \left(t_{\mathfrak{e}'} + \frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}}\right)^{-1} \cdot \left(-\frac{b^*_{\mathfrak{e},\mathfrak{e}'}}{r^*_{\mathfrak{e},\mathfrak{e}'}}\right),
   \end{aligned}$$
   where the last step is obtained from Lagrange's Mean-Value Theorem and by distinguishing whether $b^*_{\mathfrak{e},\mathfrak{e}'} \geq 0$ or not. 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.* Similar to the previous item, the algorithm establishes mutual no-greater-than inequalities over $u_{\mathfrak{e}}$'s.

Note that in item 4 and item 6 above, we have logarithmic terms such as $\ln t_{\mathfrak{e}}$ and $\ln r^*_{\mathfrak{e},\mathfrak{e}'}$, but note that both $t_{\mathfrak{e}}$ and $r^*_{\mathfrak{e},\mathfrak{e}'}$ are already determined constants, and hence their approximations can be used (recall Example 4).

Example 9. *Consider again our running example in Fig. 2. We pick the generated constraint triple*

$$(\mathsf{mergesort}, i-1 \geq 0 \wedge j-i-w \geq 0 \wedge w-1 \geq 0, \mathfrak{t})$$

with $\mathfrak{t} := \mathfrak{t}_1(i,j) - \mathfrak{t}_2(i,j) - \mathfrak{t}_1(i+w,j) - \mathfrak{t}_1(i,i+w-1) - 5$ *as an example on the generation of linear (in-)equalities. First, the algorithm assigns a variable* w *to represent the floored expression* $\left\lfloor \frac{j-i+1}{2} \right\rfloor$. *Second, the algorithm establishes* u_i *to represent* $\ln \mathfrak{e}_i$ $(1 \leq i \leq 3)$, *where* $\mathfrak{e}_1 := j-i-\left\lfloor \frac{j-i+1}{2} \right\rfloor + 1$, *and* $\mathfrak{e}_2 := \left\lfloor \frac{j-i+1}{2} \right\rfloor$, *and* $\mathfrak{e}_3 := j-i+1$. *Note that now one has that*

$$\begin{aligned}\widetilde{\mathfrak{t}} :=& c_{1,1} + c_{1,2} \cdot \widetilde{\mathfrak{e}_3} \cdot u_3 - \left(c'_{1,1} + c'_{1,2} \cdot (j-i+1)\right) \\ & - (c_{1,1} + c_{1,2} \cdot \widetilde{\mathfrak{e}_1} \cdot u_1) - (c_{1,1} + c_{1,2} \cdot \widetilde{\mathfrak{e}_2} \cdot u_2) \quad .\end{aligned}$$

Third, the algorithm establishes the inequalities

$$w \leq \frac{j-i+1}{2}, \qquad w \geq \frac{j-i}{2}, \qquad w \geq 1$$

for the floored expression $\left\lfloor \frac{j-i+1}{2} \right\rfloor$. *Next, the algorithm obtains* $t_{\mathfrak{e}_j}$*'s by deducing that* $\widetilde{\mathfrak{e}_1} \geq 1$, $\widetilde{\mathfrak{e}_2} \geq 1$, *and* $\widetilde{\mathfrak{e}_3} \geq 2$. *Then, the algorithm deduces the following constraints:*

1. $\widetilde{\mathfrak{e}_2} \leq \frac{1}{2} \cdot \widetilde{\mathfrak{e}_3}$ *and* $\widetilde{\mathfrak{e}_3} \leq 2 \cdot \widetilde{\mathfrak{e}_2} + 1$;
2. $\widetilde{\mathfrak{e}_2} \geq \frac{1}{2} \cdot \widetilde{\mathfrak{e}_3} - \frac{1}{2}$ *and* $\widetilde{\mathfrak{e}_3} \geq 2 \cdot \widetilde{\mathfrak{e}_2}$;
3. $\widetilde{\mathfrak{e}_1} \leq \frac{1}{2} \cdot \widetilde{\mathfrak{e}_3} + \frac{1}{2}$ *and* $\widetilde{\mathfrak{e}_3} \leq 2 \cdot \widetilde{\mathfrak{e}_1}$;
4. $\widetilde{\mathfrak{e}_1} \geq \frac{1}{2} \cdot \widetilde{\mathfrak{e}_3}$ *and* $\widetilde{\mathfrak{e}_3} \geq 2 \cdot \widetilde{\mathfrak{e}_1} - 1$;
5. $\widetilde{\mathfrak{e}_2} \leq \widetilde{\mathfrak{e}_1}$ *and* $\widetilde{\mathfrak{e}_1} \leq \widetilde{\mathfrak{e}_2} + 1$;
6. $\widetilde{\mathfrak{e}_2} \geq \widetilde{\mathfrak{e}_1} - 1$ *and* $\widetilde{\mathfrak{e}_1} \geq \widetilde{\mathfrak{e}_2}$.

Next, the algorithm adds the constraints formulated under (‡) in Example 4 into Γ, *with each* $\widetilde{\mathfrak{e}_j}$ *being represented by* w_j. *Finally, the algorithm establishes linear equalities through Handelman's Theorem on* Γ *and* $\mathfrak{t}$.

Step 4(c): Solving Unknown Coefficients in the Template. For this step, we use Handelman's Theorem, which we present below.

Definition 14 (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:*

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

Theorem 5 (Handelman's Theorem [25])**.** *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* $\mathbf{x} \in Y$, *then*

$$g = \sum_{i=1}^{n} c_i \cdot u_i \tag{11}$$

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. (11) which provides a sound form for a non-negative polynomial over a general (i.e. possibly unbounded) polyhedron. In Proposition 2 (in Appendix G), we show that Eq. (11) encompasses a simple proof system for non-negative polynomials over polyhedra.

Then the final step involves the following computational steps.

1. *Application of Handelman's Theorem.* First, the algorithm reads a natural number m, which is the maximal number of multiplicands in each summand at the right-hand-side of Eq. (11). Then, the algorithm establish a fresh scalar variable λ_h for each polynomial h in Monoid(Γ) with no more than m multiplicands from Γ, and establish linear (in-)equalities over scalar variables λ_h's and those in η by equating coefficients of the same monomials at the left- and right-hand-side of the following polynomial equality $\widetilde{\mathfrak{t}} = \sum_h \lambda_h \cdot h$ and incorporating all constraints of the form $\lambda_h \geq 0$.
2. *Solving Scalar Variables.* The algorithm collects all linear inequalities extracted from constraint triples conjunctively as a single system of linear inequalities and solve it through linear-programming algorithms.

```
randwalk(i, j) {
if 2*i + 3*j ≤ 100
  then
    i := i − 1;
    j := j + 1;
    randwalk(i, j)
  else skip
fi }
```

```
nestedloop(i, j, m, n) {
if i ≤ m then
  if j ≤ n then j := j + 1
  else i := i + 1; j := 0
  fi;
  nestedloop(i, j, m, n)
  else skip
fi }
```

Figure 6. Programs for Sect 6.1

We now state our main result for synthesis of measure functions (proof in Appendix G).

Theorem 6. *Our algorithm,* SYNALGO*, is a polynomial-time algorithm and a sound approach for the* RECTERMBOU *problem, i.e., if* SYNALGO *succeeds to synthesize a function* g *on* $\left\{(\mathsf{f}, \ell, \mu) \mid \mathsf{f} \in F, \ell \in L^{\mathsf{f}}_{\mathrm{s}}, \mu \in \overline{\mathit{Val}_{\mathsf{f}}}\right\}$, *then* $\hat{g}$ *is a measure function and an upper bound on the abstract termination time.*

Remark 3. *We remark two 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 relatively a scalable one.*
2. Novelty. *A key novelty of our approach is to obtain non-trivial non-linear bounds (such as* $\mathcal{O}(n \log n)$ *,* $\mathcal{O}(n^x)$*, where* x *is not an integer) by using only 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 abstract variables; and (c) use of Handelman's Theorem to solve the constraints in polynomial time.*

6. Applications and Experimental Results

In this section we explain how our sound algorithm is applicable for termination analysis of recursive programs as well as to obtain non-trivial worst-case bounds.

6.1 Termination of recursive programs

For termination of recursive programs we consider two examples, described below.

Example 1: Walk on two-dimensional plane. We first consider a simple deterministic walk on a two-dimensional plane, until a boundary is reached. We consider the walk as a recursive procedure given in the left part of Fig. 6. With a linear template, our algorithm synthesizes a linear measure function.

Example 2: Nested loop. We consider a recursive procedure that implements a nested loop, given in the right part of Fig. 6, and with a quadratic template, our algorithm synthesizes a quadratic measure function.

The above two examples are simple representative of recursive programs, and our polynomial-time algorithm can synthesize linear and quadratic bounds for the termination of recursive programs.

6.2 Non-trivial worst-case bounds

For worst-case upper bounds of non-trivial form, we consider four classical examples from the literature.

Merge-Sort. We have already illustrated (as running example), how our algorithm can synthesize $\mathcal{O}(n \log n)$ bound for Merge-Sort.

Closest-pair. The closest pair problem consider a set n of two-dimensional points and asks for the pair of points that have shortest Euclidean distance between them. We consider the closest-pair problem algorithm from (cf. [14, Chapter 33]) (also see Appendix H for the pseudo-code). Similar, to the Merge-Sort problem, we obtain an $\mathcal{O}(n \log n)$ bound for the algorithm.

Strassen's algorithm. We consider one of the classic sub-cubic algorithm for Matrix multiplication. For simplicity, we consider matrices given as one dimensional arrays along with the row and column number n. The Strassen algorithm (cf. [14, 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 H. Using a template of $n^{2.9}$, our algorithm synthesizes a measure function (basically, using constraints as illustrated in Example 5).

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 computational problem asks to compute the coefficients of the polynomial obtained by multiplication of p_1 and p_2, and considers that n is a power of 2. While the most naive algorithm is quadratic, Karatsuba's algorithm (cf. [41]) is a classical 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 H. Using a template of $n^{1.6}$, our algorithm synthesizes a measure function (basically, using constraints as illustrated in Example 5).

The above four examples show that our sound approach can synthesize non-trivial worst-case complexity bounds for several classical algorithms.

6.3 Experimental results

Below we present experimental results on the examples explained in the above two subsections. We implement our algorithm that basically generates a set of linear constraints, which we use lp_solve [1] for solving linear programming. Our experimental results are presented in Table 1. The maximal number of multiplicands in Eq. 11 was set as 2. Moreover, since Merge-Sort is essentially similar to Closest-Pair we only report the results for Merge-Sort. For detailed description of approximation constants (as mentioned in Example 4) see Appendix H. Moreover, all results were obtained quite efficiently (within few minutes), and were obtained on 2.5GHz AMD processor over Debian 3.2.78 OS. Note that our main contribution is algorithmic, and our implementation can be made more efficient with optimizations. Thus we report the final outcome of our result.

Example	$\eta(\ell_0,)$
Deterministic Walk	$4 \cdot (102 - 2 \cdot i - 3 \cdot j)$
Nested Loop	$8 \cdot (m - i + 1) \cdot n + 2$
Merge-Sort	$1.293 \cdot n \cdot \ln n - 1.293n + 7.7069$
Karatsuba	$23.0314 \cdot n^{1.6} - 21.0314$
Strassen	$1200.2125 \cdot n^{2.9} + 10.1467 \cdot n^2 + 7.4789$

Table 1. Experimental Results

7. Related Work

In this section we discuss the related work. Our work is most closely related to automatic amortized analysis [20, 27–29, 31–33, 37, 38], as well as the SPEED project [22–24]. All these works focus on worst-case bounds for programs. There are two key differences to our methodology. First, our methods are based on extension of ranking functions to recursive programs, whereas previous works either use potential functions, or abstract interpretation. Second, our approach gives a polynomial-time algorithm to derive bounds such as $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^x)$, where x is not integer, whereas none of the previous methods can derive such bounds in polynomial time.

Other approaches for bounds analysis involve recurrence relations, such as, [2–4, 18, 21]. Even for relatively simple programs the recurrence are quite complex, and cannot be solved using standard techniques. In contrast, our approach can synthesize non-trivial complexity bounds using linear programming.

Ranking functions for intraprocedural analysis has been widely studied [7, 8, 12, 15, 43, 46, 48, 50]. Most works have focussed on linear or polynomial ranking functions [12, 15, 43, 46, 48, 50]. Such ranking functions can only derive linear or polynomial bounds for programs without recursion. In contrast, we can derive much more complex bounds for recursive programs. The notion of ranking functions have been extended to ranking supermartingales [9, 10, 17] for probabilistic programs without recursion. These works cannot derive non-polynomial bounds for termination.

Several other works present proof rules for deterministic programs [26] as well as for probabilistic programs [36, 42]. None of these works can be automated.

Other related approaches are sized types [11, 34, 35], and polynomial resource bounds [47]. Again none of these approaches can yield bounds like $\mathcal{O}(n \log n)$ or $\mathcal{O}(n^x)$, for x non-integral.

8. Conclusion

In this paper, we presented two major contributions. First, we present the notion of measure function (cf. Sect. 4) and present soundness and partial completeness for proving termination of recursive programs. Second, we present a synthesis technique for measure functions with logarithm or exponentiation through (i) abstraction of logarithmic and exponential terms and (ii) Farkas' Lemma, LMVT, and Handelman's Theorem (cf. Sect. 5). Since Farkas' Lemma, LMVT, and Handelman's Theorem reduce our synthesis problems into linear programming, our technique leads to an efficient (polynomial-time) algorithm for synthesizing measure functions even with logarithmic or exponential terms. We illustrated by experiments that our technique can capture worst-case complexity of classical recursive programs with non-trivial worst-case bounds: our technique detects $\mathcal{O}(n \log n)$-complexity for both Merge-Sort and the divide-and-conquer algorithm for the Closest-Pair problem ($\mathcal{O}(n \log n)$), $\mathcal{O}(n^{1.6})$ for Karatsuba's algorithm for polynomial multiplication, and $\mathcal{O}(n^{2.9})$ for Strassen's algorithm for matrix multiplication. The bound we obtain for Karatsuba's and Strassen's algorithm are close to the optimal bounds known. In this work we focussed on recursive programs with arrays. An interesting direction of future work is to extend our technique to other data-structures.

References

[1] lp_solve 5.5.2.3. http://lpsolve.sourceforge.net/5.5/, 2016.

[2] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In R. D. Nicola, editor, *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*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2007. ISBN 978-3-540-71314-2. doi: 10.1007/978-3-540-71316-6_12. URL http://dx.doi.org/10.1007/978-3-540-71316-6_12.

[3] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In M. Alpuente and G. Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2008. ISBN 978-3-540-69163-1. doi: 10.1007/978-3-540-69166-2_15. URL http://dx.doi.org/10.1007/978-3-540-69166-2_15.

[4] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. V. Ramírez-Deantes, G. Román-Díez, and D. Zanardini. Termination and cost analysis with COSTA and its user interfaces. *Electr. Notes Theor. Comput. Sci.*, 258(1):109–121, 2009. doi: 10.1016/j.entcs.2009.12.008. URL http://dx.doi.org/10.1016/j.entcs.2009.12.008.

[5] R. G. Bartle and D. R. Sherbert. *Introduction to Real Analysis*. John Wiley & Sons, Inc., 4th edition, 2011.

[6] R. Bodík and R. Majumdar, editors. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016. ACM. ISBN 978-1-4503-3549-2. URL http://dl.acm.org/citation.cfm?id=2837614.

[7] O. Bournez and F. Garnier. Proving positive almost-sure termination. In *RTA*, pages 323–337, 2005.

[8] A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 491–504. Springer, 2005. ISBN 3-540-27231-3. doi: 10.1007/11513988_48.

[9] A. Chakarov and S. Sankaranarayanan. Probabilistic program analysis with martingales. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 511–526. Springer, 2013. ISBN 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8_34.

[10] K. Chatterjee, H. Fu, P. Novotný, and R. Hasheminezhad. Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In Bodík and Majumdar [6], pages 327–342. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837639. URL http://doi.acm.org/10.1145/2837614.2837639.

[11] W. Chin and S. Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001. doi: 10.1023/A:1012996816178. URL http://dx.doi.org/10.1023/A:1012996816178.

[12] M. Colón and H. Sipma. Synthesis of linear ranking functions. In T. Margaria and W. Yi, editors, *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*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2001. ISBN 3-540-41865-2. doi: 10.1007/3-540-45319-9_6.

[13] M. Colón, S. Sankaranarayanan, and H. Sipma. Linear invariant generation using non-linear constraint solving. In W. A. H. Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 2003. ISBN 3-540-40524-0. doi: 10.1007/978-3-540-45069-6_39. URL http://dx.doi.org/10.1007/978-3-540-45069-6_39.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms.

[15] P. Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2005. ISBN 3-540-24297-X. doi: 10.1007/978-3-540-30579-8_1.

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

[17] L. M. F. Fioriti and H. Hermanns. Probabilistic termination: Soundness, completeness, and compositionality. In S. K. Rajamani and D. Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 489–501. ACM, 2015. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2677001.

[18] P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithm. *Theor. Comput. Sci.*, 79(1):37–109, 1991. doi: 10.1016/0304-3975(91)90145-R. URL http://dx.doi.org/10.1016/0304-3975(91)90145-R.

[19] R. W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19:19–33, 1967.

[20] S. Gimenez and G. Moser. The complexity of interaction. In Bodík and Majumdar [6], pages 243–255. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837646. URL http://doi.acm.org/10.1145/2837614.2837646.

[21] B. Grobauer. Cost recurrences for DML programs. In B. C. Pierce, editor, *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001.*, pages 253–264. ACM, 2001. ISBN 1-58113-415-0. doi: 10.1145/507635.507666. URL http://doi.acm.org/10.1145/507635.507666.

[22] B. S. Gulavani and S. Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In A. Gupta and S. Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2008. ISBN 978-3-540-70543-7. doi: 10.1007/978-3-540-70545-1_35. URL http://dx.doi.org/10.1007/978-3-540-70545-1_35.

[23] S. Gulwani. SPEED: symbolic complexity bound analysis. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 2009. ISBN 978-3-642-02657-7. doi: 10.1007/978-3-642-02658-4_7. URL http://dx.doi.org/10.1007/978-3-642-02658-4_7.

[24] S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 127–139. ACM, 2009. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480898. URL http://doi.acm.org/10.1145/1480881.1480898.

[25] D. Handelman. Representing polynomials by positive linear functions on compact convex polyhedra. *Pacific J. Math.*, 132:35–62, 1988.

[26] W. H. Hesselink. Proof rules for recursive procedures. *Formal Asp. Comput.*, 5(6):554–570, 1993. doi: 10.1007/BF01211249. URL http://dx.doi.org/10.1007/BF01211249.

[27] J. Hoffmann and M. Hofmann. Amortized resource analysis with polymorphic recursion and partial big-step operational semantics. In K. Ueda, editor, *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings*, volume 6461 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 2010. ISBN 978-3-642-17163-5. doi: 10.1007/978-3-642-17164-2_13. URL http://dx.doi.org/10.1007/978-3-642-17164-2_13.

[28] J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential. In A. D. Gordon, editor, *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*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010. ISBN 978-3-642-11956-9. doi: 10.1007/978-3-642-11957-6_16. URL http://dx.doi.org/10.1007/978-3-642-11957-6_16.

[29] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012. doi: 10.1145/2362389.2362393. URL http://doi.acm.org/10.1145/2362389.2362393.

[30] J. Hoffmann, K. Aehlig, and M. Hofmann. Resource aware ML. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 781–786. Springer, 2012. ISBN 978-3-642-31423-0. doi: 10.1007/978-3-642-31424-7_64. URL http://dx.doi.org/10.1007/978-3-642-31424-7_64.

[31] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In A. Aiken and G. Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 185–197. ACM, 2003. ISBN 1-58113-628-5. doi: 10.1145/640128.604148. URL http://doi.acm.org/10.1145/640128.604148.

[32] M. Hofmann and S. Jost. Type-based amortised heap-space analysis. In P. Sestoft, editor, *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*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2006. ISBN 3-540-33095-X. doi: 10.1007/11693024_3. URL http://dx.doi.org/10.1007/11693024_3.

[33] M. Hofmann and D. Rodriguez. Efficient type-checking for amortised heap-space analysis. In E. Grädel and R. Kahle, editors, *Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009. Proceedings*, volume 5771 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2009. ISBN 978-3-642-04026-9. doi: 10.1007/978-3-642-04027-6_24. URL http://dx.doi.org/10.1007/978-3-642-04027-6_24.

[34] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In D. Rémi and P. Lee, editors, *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999.*, pages 70–81. ACM, 1999. ISBN 1-58113-111-9. doi: 10.1145/317636.317785. URL http://doi.acm.org/10.1145/317636.317785.

[35] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In H. Boehm and G. L. S. Jr., editors, *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*, pages 410–423. ACM Press, 1996. ISBN 0-89791-769-3. doi: 10.1145/237721.240882. URL http://doi.acm.org/10.1145/237721.240882.

[36] C. Jones. *Probabilistic Non-Determinism*. PhD thesis, The University of Edinburgh, 1989.

[37] S. Jost, H. Loidl, K. Hammond, N. Scaife, and M. Hofmann. "carbon credits" for resource-bounded computations using amortised analysis. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 354–369. Springer, 2009. ISBN 978-3-642-05088-6. doi: 10.1007/978-3-642-05089-3_23. URL http://dx.doi.org/10.1007/978-3-642-05089-3_23.

[38] S. Jost, K. Hammond, H. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In M. V. Hermenegildo and J. Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 223–236. ACM, 2010. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706327. URL http://doi.acm.org/10.1145/1706299.1706327.

[39] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973. ISBN 0-201-03803-X.

[40] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

[41] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms, 2nd Edition*. Addison-Wesley, 1981. ISBN 0-201-03822-6.

[42] F. Olmedo, B. L. Kaminski, J.-P. Katoen, and C. Matheja. Reasoning about recursive probabilistic programs. In *LICS 2016*, 2016, to appear.

[43] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer, 2004. ISBN 3-540-20803-8. doi: 10.1007/978-3-540-24622-0_20.

[44] A. Schrijver. *Theory of Linear and Integer Programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999. ISBN 978-0-471-98232-6.

[45] A. Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003. ISBN 978-3-540-44389-6.

[46] L. Shen, M. Wu, Z. Yang, and Z. Zeng. Generating exact nonlinear ranking functions by symbolic-numeric hybrid method. *J. Systems Science & Complexity*, 26(2):291–301, 2013. doi: 10.1007/s11424-013-1004-1.

[47] O. Shkaravska, R. van Kesteren, and M. C. J. D. van Eekelen. Polynomial size analysis of first-order functions. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 351–365. Springer, 2007. ISBN 978-3-540-73227-3. doi: 10.1007/978-3-540-73228-0_25. URL http://dx.doi.org/10.1007/978-3-540-73228-0_25.

[48] K. Sohn and A. V. Gelder. Termination detection in logic programs using argument sizes. In D. J. Rosenkrantz, editor, *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 29-31, 1991, Denver, Colorado, USA*, pages 216–226. ACM Press, 1991. ISBN 0-89791-430-9. doi: 10.1145/113413.113433.

[49] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3), 2008. doi: 10.1145/1347375.1347389. URL http://doi.acm.org/10.1145/1347375.1347389.

[50] L. Yang, C. Zhou, N. Zhan, and B. Xia. Recent advances in program verification through computer algebra. *Frontiers of Computer Science in China*, 4(1):1–16, 2010. doi: 10.1007/s11704-009-0074-7. URL http://dx.doi.org/10.1007/s11704-009-0074-7.

A. Evaluation of Arithmetic Array Expressions

Below we fix a countable set $\mathcal{X}$ of scalar variables and a countable set $\mathcal{A}$ of array variables such that $\mathcal{X} \cap \mathcal{A} = \emptyset$. We also fix a heap $\mathfrak{h}$. Given an arithmetic expression $\mathfrak{e}$ over $\mathcal{X}, \mathcal{A}$ and a valuation on $\mathcal{X}, \mathcal{A}$, the element $\mathfrak{e}_\mathfrak{h}(\nu)$ is defined inductively on the structure of $\mathfrak{e}$ as follows:

- $c_\mathfrak{h}(\nu) := c$;
- $x_\mathfrak{h}(\nu) := \nu(x)$;
- $\|ar\|_\mathfrak{h}(\nu) := \|\mathfrak{h}(\nu(ar))\|$;
- $(ar[x])_\mathfrak{h}(\nu) := \mathfrak{h}(\nu(ar))[\nu(x)]$ if $1 \leq \nu(x) \leq \|\mathfrak{h}(\nu(ar))\|$, and $(ar[x])_\mathfrak{h}(\nu) := \bot$ otherwise;
- $\left\lfloor \frac{\mathfrak{e}}{c} \right\rfloor_\mathfrak{h}(\nu) := \left\lfloor \frac{\mathfrak{e}_\mathfrak{h}(\nu)}{c} \right\rfloor$ if $c \neq 0$ and $\mathfrak{e}_\mathfrak{h}(\nu) \neq \bot$, and $\left\lfloor \frac{\mathfrak{e}}{c} \right\rfloor_\mathfrak{h}(\nu) := \bot$ otherwise;
- $(\mathfrak{e} + \mathfrak{e}')_\mathfrak{h}(\nu) := \mathfrak{e}_\mathfrak{h}(\nu) + \mathfrak{e}'_\mathfrak{h}(\nu)$ if $\mathfrak{e}(\nu), \mathfrak{e}'_\mathfrak{h}(\nu) \in \mathbb{Z}$, and $(\mathfrak{e} + \mathfrak{e})_\mathfrak{h}(\nu) := \bot$ otherwise;
- $(\mathfrak{e} - \mathfrak{e}')_\mathfrak{h}(\nu) := \mathfrak{e}_\mathfrak{h}(\nu) - \mathfrak{e}'_\mathfrak{h}(\nu)$ if $\mathfrak{e}_\mathfrak{h}(\nu), \mathfrak{e}'_\mathfrak{h}(\nu) \in \mathbb{Z}$, and $(\mathfrak{e} - \mathfrak{e}')_\mathfrak{h}(\nu) := \bot$ otherwise;
- $(\mathfrak{e} * \mathfrak{e}')_\mathfrak{h}(\nu) := \mathfrak{e}_\mathfrak{h}(\nu) \cdot \mathfrak{e}'_\mathfrak{h}(\nu)$ if $\mathfrak{e}_\mathfrak{h}(\nu), \mathfrak{e}'_\mathfrak{h}(\nu) \in \mathbb{Z}$, and $(\mathfrak{e} * \mathfrak{e}')_\mathfrak{h}(\nu) := \bot$ otherwise.

B. Semantics of Propositional Array Predicates

Let $\mathcal{X}, \mathcal{A}$ be disjoint countable sets of scalar and array variables, respectively. And let $\mathfrak{h}$ be a heap. The satisfaction relation $\models_\mathfrak{h}$ between valuations ν and propositional array predicates ϕ is defined inductively as follows:

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

Conversion to DNF. By standard operations, propositional array predicates can be converted into DNF (disjunctive normal form). This is because since all variables are integer, and predicates $\mathfrak{e} > \mathfrak{e}'$ or $\mathfrak{e} < \mathfrak{e}'$ in some DNF can be equivalently rewritten into $\mathfrak{e} \geq \mathfrak{e}' + 1$ or $\mathfrak{e} \leq \mathfrak{e}' - 1$, respectively. We will use the DNF form in Step 2 of SYNALGO.

C. Control-Flow Graphs for Recursive Programs

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 $V^\mathsf{f}_\mathrm{p}, V^\mathsf{f}_\mathrm{ar}$ be the set of scalar and resp. array 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 $V^\mathsf{f}_\mathrm{p} \cup V^\mathsf{f}_\mathrm{ar}$, then define $\to_\mathsf{f}$ as $\to_{P_\mathsf{f},\mathsf{f}}$.

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

$$\left(\mathfrak{h}_\nu[\mathfrak{e}/ar[x]]\right)(d) = \begin{cases} \mathfrak{h}(d) & \text{if } d \neq \nu(ar) \\ \mathfrak{h}(d)[\mathfrak{e}_\mathfrak{h}(\nu)/\nu(x)] & \text{if } d = \nu(ar), \mathfrak{e}_\mathfrak{h}(\nu) \neq \bot \\ & \text{and } \nu(x) \in [1, \|\mathfrak{h}(\nu(ar))\|] \cap \mathbb{N} \\ \bot & \text{otherwise} \end{cases}$$

where $\mathfrak{h}(d)[\mathfrak{e}_\mathfrak{h}(\nu)/\nu(x)]$ is the array obtained from $\mathfrak{h}(d)$ by simply changing $\mathfrak{h}(d)[\nu(x)]$ to $\mathfrak{e}_\mathfrak{h}(\nu)$. Moreover, given an assignment statement of the form $x{:=}\mathfrak{e}$ involving variables solely from $V^\mathsf{f}_\mathrm{p} \cup V^\mathsf{f}_\mathrm{ar}$, a valuation $\nu \in \mathit{Val}_\mathsf{f}$ and a heap $\mathfrak{h}$, we denote by $\nu_\mathfrak{h}[\mathfrak{e}/x]$ the valuation over $V^\mathsf{f}_\mathrm{p}, V^\mathsf{f}_\mathrm{ar}$ such that

$$\left(\nu_\mathfrak{h}[\mathfrak{e}/x]\right)(q) = \begin{cases} \nu(q) & \text{if } q \in (V^\mathsf{f}_\mathrm{p} \backslash \{x\}) \cup V^\mathsf{f}_\mathrm{ar} \\ \mathfrak{e}_\mathfrak{h}(\nu) & \text{if } q = x \text{ and } \mathfrak{e}_\mathfrak{h}(\nu) \neq \bot \\ 0 & \text{if } q = x \text{ and } \mathfrak{e}_\mathfrak{h}(\nu) = \bot \end{cases}.$$

Finally, given a function call $\mathsf{g}(q'_1, \ldots, q'_k)$ with variables solely from $V^\mathsf{f}_\mathrm{p} \cup V^\mathsf{f}_\mathrm{ar}$ and its declaration being $\mathsf{g}(q_1, \ldots, q_k)$, a valuation $\nu \in \mathit{Val}_\mathsf{f}$ and a heap $\mathfrak{h}$, we define $\nu_\mathfrak{h}[\mathsf{g}, \{q'_j\}_{1 \leq j \leq k}]$ to be a valuation over $V^\mathsf{g}_\mathrm{p}, V^\mathsf{g}_\mathrm{ar}$ by:

$$\nu_\mathfrak{h}[\mathsf{g}, \{q'_j\}_{1 \leq j \leq k}](q_j) := \begin{cases} 0 & \text{if } q_j \in V^\mathsf{g}_\mathrm{p} \text{ and } (q'_j)_\mathfrak{h}(\nu) = \bot \\ (q'_j)_\mathfrak{h}(\nu) & \text{if } q_j \in V^\mathsf{g}_\mathrm{p} \text{ and } (q'_j)_\mathfrak{h}(\nu) \neq \bot \\ \nu(q'_j) & \text{if } q_j \in V^\mathsf{g}_\mathrm{ar} \end{cases}$$

and $\nu_\mathfrak{h}[\mathsf{g}, \{q'_j\}_{1 \leq j \leq k}](q) := 0$ for each $q \in V^\mathsf{g}_\mathrm{p} \backslash \{q_1, \ldots, q_k\}$.

Now the inductive construction for each $\to_{P,\mathsf{f}}$ is demonstrated as follows. For each statement P which involves variables solely from $V^\mathsf{f}_\mathrm{p} \cup V^\mathsf{f}_\mathrm{ar}$, the 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.

1. *Assignments and Skips.* For P of the form
$$x{:=}\mathfrak{e},\ ar[x]{:=}\mathfrak{e} \text{ or } \textbf{skip},$$
$\to_{P,\mathsf{f}}$ involves a new assignment label $\ell^{P,\mathsf{f}}_\mathrm{in}$ (as the initial label) and a new conditional-branching label $\ell^{P,\mathsf{f}}_\mathrm{out}$ (as the terminal label), and contains a sole triple
$$\left(\ell^{P,\mathsf{f}}_\mathrm{in}, (\nu, \mathfrak{h}) \mapsto (\nu_\mathfrak{h}[\mathfrak{e}/x], \mathfrak{h}), \ell^{P,\mathsf{f}}_\mathrm{out}\right)$$
or
$$\left(\ell^{P,\mathsf{f}}_\mathrm{in}, (\nu, \mathfrak{h}) \mapsto (\nu, \mathfrak{h}_\nu[\mathfrak{e}/ar[x]]), \ell^{P,\mathsf{f}}_\mathrm{out}\right)$$
or
$$\left(\ell^{P,\mathsf{f}}_\mathrm{in}, (\nu, \mathfrak{h}) \mapsto (\nu, \mathfrak{h}), \ell^{P,\mathsf{f}}_\mathrm{out}\right),$$
respectively.
2. *Function Calls.* For P of the form
$$\mathsf{g}(q'_1, \ldots, q'_k),$$
$\to_{P,\mathsf{f}}$ involves a new function-call label $\ell^{P,\mathsf{f}}_\mathrm{in}$ and a new conditional-branching label $\ell^{P,\mathsf{f}}_\mathrm{out}$, and contains a sole triple
$$\left(\ell^{P,\mathsf{f}}_\mathrm{in}, \left(\mathsf{g}, (\nu, \mathfrak{h}) \mapsto \nu_\mathfrak{h}[\mathsf{g}, \{q'_j\}_{1 \leq j \leq k}]\right), \ell^{P,\mathsf{f}}_\mathrm{out}\right).$$
3. *Sequential Statements.* For
$$P{=}Q_1; Q_2,$$
we take the disjoint union of $\to_{Q_1,\mathsf{f}}$ and $\to_{Q_2,\mathsf{f}}$, while redefining $\ell^{Q_1,\mathsf{f}}_\mathrm{out}$ to be $\ell^{Q_2,\mathsf{f}}_\mathrm{in}$ and putting $\ell^{P,\mathsf{f}}_\mathrm{in} := \ell^{Q_1,\mathsf{f}}_\mathrm{in}$ and $\ell^{P,\mathsf{f}}_\mathrm{out} := \ell^{Q_2,\mathsf{f}}_\mathrm{out}$.

4. *Conditional-Branching Statements.* For

$$P{=}\textbf{if } \phi \textbf{ then } Q_1 \textbf{ else } Q_2 \textbf{ fi}$$

with ϕ being a propositional array predicate, we first add two new conditional-branching labels $\ell^P_{\mathrm{in}}, \ell^P_{\mathrm{out}}$, then take the disjoint union of $\to_{Q_1,\mathsf{f}}$ and $\to_{Q_2,\mathsf{f}}$ while simultaneously identifying both $\ell^{Q_1,\mathsf{f}}_{\mathrm{out}}$ and $\ell^{Q_2,\mathsf{f}}_{\mathrm{out}}$ with $\ell^{P,\mathsf{f}}_{\mathrm{out}}$, and finally obtain $\to_{P,\mathsf{f}}$ by adding two triples $(\ell^{P,\mathsf{f}}_{\mathrm{in}}, \phi, \ell^{Q_1,\mathsf{f}}_{\mathrm{in}})$ and $(\ell^{P,\mathsf{f}}_{\mathrm{in}}, \neg\phi, \ell^{Q_2,\mathsf{f}}_{\mathrm{in}})$ into the disjoint union of $\to_{Q_1,\mathsf{f}}$ and $\to_{Q_2,\mathsf{f}}$.
5. *While Statements.* For

$$P{=}\textbf{while } \phi \textbf{ do } Q \textbf{ od},$$

we add a new conditional-branching label $\ell^{P,\mathsf{f}}_{\mathrm{out}}$ as a terminal label and obtain $\to_{P,\mathsf{f}}$ by adding triples $(\ell^{Q,\mathsf{f}}_{\mathrm{out}}, \phi, \ell^{Q,\mathsf{f}}_{\mathrm{in}})$ and $(\ell^{Q,\mathsf{f}}_{\mathrm{out}}, \neg\phi, \ell^{P,\mathsf{f}}_{\mathrm{out}})$ into $\to_{Q,\mathsf{f}}$, and define $\ell^{P,\mathsf{f}}_{\mathrm{in}} := \ell^{Q,\mathsf{f}}_{\mathrm{out}}$.

D. Proofs for Theorem 1 and Theorem 2

Theorem 1. For all measure functions g w.r.t f^*, ϕ^*, it holds that for all abstract valuations μ w.r.t f^* such that $\mu \models \phi^*$, $\overline{T}(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\mathrm{in}}, \mu) \leq g(\mathsf{f}^*, \ell^{\mathsf{f}^*}_{\mathrm{in}}, \mu)$.

Proof. Let g be a measure function w.r.t f^*. For each non-negative integer n, define

$$T_n(\mathfrak{c}, \mathfrak{h}) := \min\{n, T(\mathfrak{c}, \mathfrak{h})\}$$

and

$$\overline{T}_n(\mathsf{f}, \ell, \mu) := \min\left\{n, \overline{T}(\mathsf{f}, \ell, \mu)\right\}$$

for all stack elements $\mathfrak{c}$, heaps $\mathfrak{h}$ and abstract stack elements (f, ℓ, μ). Clearly, for all abstract stack elements (f, ℓ, μ), it holds that $\lim_{n\to\infty} \overline{T}_n(\mathsf{f}, \ell, \mu) = \overline{T}(\mathsf{f}, \ell, \mu)$ and

$$(\ddagger)\quad \overline{T}_n(\mathsf{f}, \ell, \mu) = \sup\{T_n((\mathsf{f}, \ell, \nu), \mathfrak{h}) \mid \nu \in \mathit{Val}_{\mathsf{f}}, \ \mathfrak{h} \text{ is a heap and } \mu = \mu[\nu, \mathfrak{h}]\}$$

for all $n \in \mathbb{N}_0$.

We prove by induction on n that

$$\overline{T}_n(\mathsf{f}, \ell, \mu) \leq g(\mathsf{f}, \ell, \mu)$$

for all $n \in \mathbb{N}_0$ and for all abstract stack elements (f, ℓ, μ) such that either $\ell = \ell^{\mathsf{f}}_{\mathrm{out}}$ or $\mu \in D_{\mathsf{f},\ell}$, which directly implies the theorem.

Base Step The base step $n \in \{0, 1\}$ is straightforward since one has for all abstract stack elements (f, ℓ, μ),

1. $\overline{T}_0(\mathsf{f}, \ell, \mu) = 0$ and $g(\mathsf{f}, \ell, \mu) \geq 0$ from definition, and
2. $\overline{T}_1(\mathsf{f}, \ell, \mu) = \mathbf{1}_{\ell \neq \ell^{\mathsf{f}}_{\mathrm{out}}}$ and

$$g(\mathsf{f}, \ell, \mu) \cdot \mathbf{1}_{\ell \neq \ell^{\mathsf{f}}_{\mathrm{out}} \wedge \mu \in D_{\mathsf{f},\ell}} \geq \mathbf{1}_{\ell \neq \ell^{\mathsf{f}}_{\mathrm{out}} \wedge \mu \in D_{\mathsf{f},\ell}}$$

from non-negativity and conditions C2–C4 of measure functions.

Inductive Step Assume that $n \geq 1$. Consider any abstract stack element (f, ℓ, μ) such that either $\ell = \ell^{\mathsf{f}}_{\mathrm{out}}$ or $\mu \in D_{\mathsf{f},\ell}$. The case $\ell = \ell^{\mathsf{f}}_{\mathrm{out}}$ is straightforward since $\overline{T}_{n+1}(\mathsf{f}, \ell, \mu) = 0$ by definition. Below we assume that $\mu \in D_{\mathsf{f},\ell}$ and let $((\mathsf{f}, \ell, \nu), \mathfrak{h})$ be any reachable pair w.r.t f^*, ϕ^* such that $\mu = \mu[\nu, \mathfrak{h}]$. We clarify several cases below.

1. **Case 1 (Assignment):** $\ell \in L^{\mathsf{f}}_{\mathrm{a}}$ and (ℓ, f, ℓ') is the sole triple in $\to_{\mathsf{f}}$ with source label ℓ. Then one easily obtains from the semantics that

$$T_{n+1}((\mathsf{f}, \ell, \nu), \mathfrak{h}) = 1 + T_n((\mathsf{f}, \ell', \nu'), \mathfrak{h}')$$

where $(\nu', \mathfrak{h}') := f(\nu, \mathfrak{h})$. It follows from $(\ddagger)$ and arbitrary choice of $(\nu, \mathfrak{h})$ that

$$\overline{T}_{n+1}(\mathsf{f}, \ell, \mu) \leq 1 + \overline{T}_n(\mathsf{f}, \ell', \overline{f}(\mu)).$$

By induction hypothesis, C2 and the fact that $\overline{f}(\mu) \in D_{\mathsf{f},\ell'}$ whenever $\ell' \neq \ell^{\mathsf{f}}_{\mathrm{out}}$, one obtains $\overline{T}_{n+1}(\mathsf{f}, \ell, \mu) \leq g(\mathsf{f}, \ell, \mu)$.
2. **Case 2 (Function-Call):** $\ell \in L^{\mathsf{f}}_{\mathrm{c}}$ and $(\ell, (\mathsf{g}, f), \ell')$ is the sole triple in $\to_{\mathsf{f}}$ with source label ℓ. Then one obtains that

$$T((\mathsf{f}, \ell, \nu), \mathfrak{h}) = \infty$$

whenever $T\left((\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, f(\nu, \mathfrak{h})), \mathfrak{h}\right) = \infty$ and

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

otherwise, where $\mathfrak{h}'$ is the unique heap after the execution of g. It follows from the absence of array-creation statement in our programming language that

$$T_{n+1}((\mathsf{f}, \ell, \nu), \mathfrak{h}) = 1 + T_n\left((\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, f(\nu, \mathfrak{h})), \mathfrak{h}\right) (= n + 1)$$

whenever $T\left((\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, f(\nu, \mathfrak{h})), \mathfrak{h}\right) = \infty$ and

$$T_{n+1}((\mathsf{f}, \ell, \nu), \mathfrak{h}) \leq 1 + T_n\left((\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, f(\nu, \mathfrak{h})), \mathfrak{h}\right) + T_n((\mathsf{f}, \ell', \nu), \mathfrak{h}')$$

otherwise, which implies that

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

Note that $\overline{f}(\mu) \in D_{\mathsf{g},\ell^{\mathsf{g}}_{\mathrm{in}}}$. If $\overline{T}\left(\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, \overline{f}(\mu)\right) = \infty$, then by induction hypothesis, $g\left(\mathsf{g}, \ell^{\mathsf{g}}_{\mathrm{in}}, \overline{f}(\mu)\right) \geq n$ and hence $g(\mathsf{f}, \ell, \mu) \geq n + 1$ by C3; otherwise, it holds that $\mu \in D_{\mathsf{f},\ell'}$ whenever $\ell' \neq \ell^{\mathsf{f}}_{\mathrm{out}}$, implying $\overline{T}_n(\mathsf{f}, \ell', \mu) \leq g(\mathsf{f}, \ell', \mu)$ from the induction hypothesis. In either case, by induction hypothesis and C3, one has $\overline{T}_{n+1}(\mathsf{f}, \ell, \mu) \leq g(\mathsf{f}, \ell, \mu)$.
3. **Case 3 (Conditional-Branching):** $\ell \in L^{\mathsf{f}}_{\mathrm{b}}$ and (ℓ, ϕ, ℓ_1), $(\ell, \neg\phi, \ell_2)$ are namely two triples in $\to_{\mathsf{f}}$ with source label ℓ. Then one easily obtains that

$$T((\mathsf{f}, \ell, \nu), \mathfrak{h}) = \begin{cases} 1 + T\left((\mathsf{f}, \ell_1, \nu)\right), \mathfrak{h}) & \text{if } \nu \models_{\mathfrak{h}} \phi \\ 1 + T\left((\mathsf{f}, \ell_2, \nu)\right), \mathfrak{h}) & \text{if } \nu \models_{\mathfrak{h}} \neg\phi \end{cases}.$$

It follows that

$$T_{n+1}((\mathsf{f}, \ell, \nu), \mathfrak{h}) = \begin{cases} 1 + T_n\left((\mathsf{f}, \ell_1, \nu)\right), \mathfrak{h}) & \text{if } \nu \models_{\mathfrak{h}} \phi \\ 1 + T_n\left((\mathsf{f}, \ell_2, \nu)\right), \mathfrak{h}) & \text{if } \nu \models_{\mathfrak{h}} \neg\phi \end{cases}.$$

On one hand, assume that ϕ involve no array entries. Then

$$\overline{T}_{n+1}(\mathsf{f}, \ell, \mu) \leq 1 + \mathbf{1}_{\mu \models \phi} \cdot \overline{T}_n(\mathsf{f}, \ell_1, \mu) + \mathbf{1}_{\mu \models \neg\phi} \cdot \overline{T}_n(\mathsf{f}, \ell_2, \mu)\,.$$

W.l.o.g, we assume that $\mu \models \phi$, since the case $\mu \models \neg\phi$ is completely symmetrical. Then since $(\mathsf{f}, \ell_1, \mu)$ is reachable w.r.t f^* whenever $\ell_1 \neq \ell^{\mathsf{f}}_{\mathrm{out}}$, one obtains $\overline{T}_n(\mathsf{f}, \ell_1, \mu) \leq g(\mathsf{f}, \ell_1, \mu)$ from induction hypothesis, which implies $\overline{T}_{n+1}(\mathsf{f}, \ell, \mu) \leq g(\mathsf{f}, \ell, \mu)$ from C4. On the other hand, assume that ϕ involve array entries. In the case that $\nu \models_{\mathfrak{h}} \phi$, one has $(\mathsf{f}, \ell_1, \mu)$ is reachable w.r.t f^* and hence

$$T_{n+1}((\mathsf{f}, \ell, \nu), \mathfrak{h}) = 1 + T_n((\mathsf{f}, \ell_1, \nu), \mathfrak{h}) \leq 1 + g(\mathsf{f}, \ell_1, \mu)$$

from $(\ddagger)$ and induction hypothesis; for the other case (i.e., $\nu \models_{\mathfrak{h}} \neg\phi$), one can deduce similarly that

$$T_{n+1}((\mathsf{f}, \ell, \nu), \mathfrak{h}) \leq 1 + g(\mathsf{f}, \ell_2, \mu).$$

Hence, in all cases,

$$T_{n+1}((\mathsf{f}, \ell, \nu), \mathfrak{h}) \leq 1 + \max\{g(\mathsf{f}, \ell_1, \mu), g(\mathsf{f}, \ell_2, \mu)\}.$$

By the arbitrary choice of $(\nu, \mathfrak{h})$ and C4, $\overline{T}_{n+1}(\mathsf{f}, \ell, \mu) \leq g(\mathsf{f}, \ell, \mu)$.

In all three cases above, the inductive step is proved. □

Theorem 2. If W does not involve array variables, then there exists a measure function g w.r.t f^*, ϕ^* satisfying that for all abstract valuations μ w.r.t f^* such that $\mu \models \phi^*$, $\overline{T}(\mathsf{f}^*, \ell_{\mathrm{in}}^{\mathsf{f}^*}, \mu) = g(\mathsf{f}^*, \ell_{\mathrm{in}}^{\mathsf{f}^*}, \mu)$.

Proof. Let the function g be defined by $g(\mathsf{f}, \ell, \mu) := \overline{T}(\mathsf{f}, \ell, \mu)$ for all abstract stack elements (f, ℓ, μ). Since W does not involve array entries, the function g satisfies conditions C1–C4 with equality. Thus g is a measure function. □

E. Proof for Lemma 1

Lemma 1. For each function g from

$$\left\{(\mathsf{f}, \ell, \mu) \mid \mathsf{f} \in F, \ell \in L_{\mathrm{s}}^{\mathsf{f}}, \mu \in \overline{\mathit{Val}}_{\mathsf{f}}\right\}$$

into $[0, \infty]$, $\widehat{g}$ 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 $\widehat{g}(\mathsf{f}, \ell_0, \boldsymbol{.})$ remains undefined. Then by the inductive procedure, there exists a triple (ℓ_0, α, ℓ_1) in $\rightarrow_{\mathsf{f}}$ such that $\widehat{g}(\mathsf{f}, \ell_1, \boldsymbol{.})$ remains undefined. With the same reasoning, one can inductively construct an infinite sequence $\{\ell_j\}_{j \in \mathbb{N}_0}$ such that each $\widehat{g}(\mathsf{f}, \ell_j, \boldsymbol{.})$ 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 $\ell_{j^*} \in L_{\mathrm{s}}^{\mathsf{f}}$. □

F. Proof for Proposition 1

Proposition 1. Let g be a function from

$$\left\{(\mathsf{f}, \ell, \mu) \mid \mathsf{f} \in F, \ell \in L_{\mathrm{s}}^{\mathsf{f}}, \mu \in \overline{\mathit{Val}}_{\mathsf{f}}\right\}$$

into $[0, \infty]$ and $\widehat{g}$ be defined as in Definition 13. Let I be an invariant w.r.t f^*. Assume that for all abstract stack elements (f, ℓ, μ) such that $\ell \in L_{\mathrm{s}}^{\mathsf{f}}$ and $\mu \models I(\mathsf{f}, \ell)$,

- **C2'** if $\ell \in L_{\mathrm{a}}^{\mathsf{f}}$ and (ℓ, f, ℓ') is the sole triple in $\rightarrow_{\mathsf{f}}$ with source label ℓ, then $\widehat{g}(\mathsf{f}, \ell', \overline{f}(\mu)) + 1 \leq \widehat{g}(\mathsf{f}, \ell, \mu)$;
- **C3'** if $\ell \in L_{\mathrm{c}}^{\mathsf{f}}$ and $(\ell, (\mathsf{g}, f), \ell')$ is the sole triple in $\rightarrow_{\mathsf{f}}$ with source label ℓ, then $1 + \widehat{g}(\mathsf{g}, \ell_{\mathrm{in}}^{\mathsf{g}}, \overline{f}(\mu)) + \widehat{g}(\mathsf{f}, \ell', \mu) \leq \widehat{g}(\mathsf{f}, \ell, \mu)$;
- **C4'** if $\ell \in L_{\mathrm{b}}^{\mathsf{f}}$ and $(\ell, \phi, \ell_1), (\ell, \neg\phi, \ell_2)$ are namely two triples in $\rightarrow_{\mathsf{f}}$ with source label ℓ, then (i)

  $$\mathbf{1}_{\mu \models \phi} \cdot \widehat{g}(\mathsf{f}, \ell_1, \mu) + \mathbf{1}_{\mu \models \neg\phi} \cdot \widehat{g}(\mathsf{f}, \ell_2, \mu) + 1 \leq \widehat{g}(\mathsf{f}, \ell, \mu)$$

  whenever ϕ does not involve array entries and (ii)

  $$\max\{\widehat{g}(\mathsf{f}, \ell_1, \mu), \widehat{g}(\mathsf{f}, \ell_2, \mu)\} + 1 \leq \widehat{g}(\mathsf{f}, \ell, \mu)$$

  otherwise.

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

Proof. The proof follows directly from the fact that (i) $D_{\mathsf{f},\ell} \subseteq I(\mathsf{f}, \ell)$ for all $\mathsf{f} \in F$ and $\ell \in L^{\mathsf{f}} \backslash \{\ell_{\mathrm{out}}^{\mathsf{f}}\}$, (ii) C2'-C4' directly implies C2-C4 for $\ell \in L_{\mathrm{s}}^{\mathsf{f}}$ and (iii) C1-C4 are automatically satisfied for $\ell \notin L_{\mathrm{s}}^{\mathsf{f}}$ by Definition 13. □

G. Omitted Details for Sect. 5

In this section, we present the omitted details on our algorithm for synthesizing measure functions. As mentioned before, the synthesis algorithm is designed to synthesize one function over abstraction valuations at each function name and appropriate significant label, so that conditions C2'-C4' in Proposition 1 are fulfilled.

Below we fix an input recursive program W, with its CFG being

$$\left(F, \left\{\left(L^{\mathsf{f}}, L_{\mathrm{b}}^{\mathsf{f}}, L_{\mathrm{a}}^{\mathsf{f}}, L_{\mathrm{c}}^{\mathsf{f}}, V_{\mathrm{p}}^{\mathsf{f}}, V_{\mathrm{ar}}^{\mathsf{f}}, \ell_{\mathrm{in}}^{\mathsf{f}}, \ell_{\mathrm{out}}^{\mathsf{f}}\right)\right\}_{\mathsf{f} \in F}, \{\rightarrow_{\mathsf{f}}\}_{\mathsf{f} \in F}\right) \ ,$$

an input function name $\mathsf{f}^* \in F$, an input propositional array predicate ϕ^* and an input invariant I w.r.t f^*, ϕ^*. We demonstrate our algorithm step in step as follows.

G.1 Step 1 of SYNALGO

All details are presented in the main article.

G.2 Step 2 of SYNALGO

Computation of $\widehat{[\![\eta]\!]}$ In the computation, we use the fact that for real-valued functions $\{f_i\}_{1 \leq i \leq m}$, $\{g_j\}_{1 \leq j \leq n}$ 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)

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

for all propositional array predicates ϕ_1, ϕ_2 and abstract valuations μ, 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 array predicates without array entries are closed under substitution of expressions in $\langle \mathit{pexpr} \rangle$ for scalar variables.

G.3 Step 3 of SYNALGO

Establishment of Constraint Triples Based on $\widehat{[\![\eta]\!]}$, the algorithm generates constraint triples at each significant label of some function name, then group all generated constraint triples together in a conjunctive way.

At every significant label ℓ of some function name f, the algorithm generates constraint triples as follows. Let a disjunctive normal form of $I(\mathsf{f}, \ell)$ be $\bigvee_l \Psi_\ell^l$. The algorithm first generate constraint triples related to non-negativity of measure functions.

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

  $$\left\{\left(\mathsf{f}, \Psi_\ell^l, \eta(\mathsf{f}, \ell, \boldsymbol{.})\right)\right\}_l \ .$$

Then the algorithm generates constraint triples attached to ℓ and f through C2'-C4' as follows. Let

- **Case 1 (C2')**: $\ell \in L_{\mathrm{s}}^{\mathsf{f}} \cap L_{\mathrm{a}}^{\mathsf{f}}$ and (ℓ, f, ℓ') is the sole triple in $\rightarrow_{\mathsf{f}}$ with source label ℓ. As $\widehat{[\![\eta]\!]}(\mathsf{f}, \ell', \boldsymbol{.})$ can be represented in form (10), $\widehat{[\![\eta]\!]}(\mathsf{f}, \ell', \overline{f}(\boldsymbol{.}))$ 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 (10). Let a disjunctive normal form of each formula $I(\mathsf{f},\ell)\wedge\phi_{ij}$ be $\bigvee_l \Phi_{ij}^l$. Then the algorithm generates the collection of constraint triples

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

- **Case 2 (C3')**: $\ell\in L_{\mathsf{c}}^{\mathsf{f}}$ and $(\ell,(\mathsf{g},f),\ell')$ is the sole triple in $\rightarrow_{\mathsf{f}}$ with source label ℓ. Let $\widehat{[\![\eta]\!]}\left(\mathsf{g},\ell_{\mathrm{in}}^{\mathsf{g}},\overline{f}(\centerdot)\right)+\widehat{[\![\eta]\!]}(\mathsf{f},\ell',\centerdot)$ 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 (10). Let a disjunctive normal form of each formula $I(\mathsf{f},\ell)\wedge\phi_{ij}$ be $\bigvee_l \Phi_{ij}^l$. Then the algorithm generates the collection of constraint triples

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

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

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

if ϕ does not involve array entries, and be

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

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

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

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

Example 10. *Consider our running example* mergesort *(cf. Fig. 2). On one hand, the algorithm generates the following non-trivial constraint triples from non-negativity:*

- $(\mathsf{mergesort}, i-1\geq 0\wedge j-i\geq 0, \mathfrak{t}_1(i,j))$;
- $(\mathsf{merge}, i-1\geq 0\wedge j-i\geq 0, \mathfrak{t}_2(i,j))$;
- $(\mathsf{merge}, i-1\geq 0\wedge j-i\geq 0\wedge j-l+1\geq 0, \mathfrak{t}_3(i,j,l))$;
- $(\mathsf{merge}, i-1\geq 0\wedge j-i\geq 0\wedge j-l+1\geq 0, \mathfrak{t}_4(j,l))$.

On the other hand, the algorithm generates the following constraint triples from C2'–C4' as follows.

- *(from C4' at* $(\mathsf{mergesort},1)$*) From the calculation of*

$$\mathbf{1}_{\phi}\cdot\widehat{[\![\eta]\!]}(\mathsf{mergesort},2,\centerdot)+\mathbf{1}_{\neg\phi}\cdot\widehat{[\![\eta]\!]}(\mathsf{mergesort},6,\centerdot)$$

where $\phi:=i-1\geq 0\wedge j-i-1\geq 0$*, the algorithm generates the following constraint triples:*
  - $(\mathsf{mergesort},\varphi_1,\mathfrak{t})$ *with*

$$\varphi_1:=i-1\geq 0\wedge j-i-w\geq 0\wedge w-1\geq 0\quad \textit{and}$$

$$\mathfrak{t}:=\mathfrak{t}_1(i,j)-\mathfrak{t}_2(i,j)-\mathfrak{t}_1(i+w,j)-\mathfrak{t}_1(i,i+w-1)-5;$$

  - $(\mathsf{mergesort},\varphi_2,\mathfrak{t}_1(i,j)-2)$ *with*

$$\varphi_2:=i-1\geq 0\wedge j-i\geq 0\wedge i-j\geq 0.$$

where w *represents* $\lfloor\frac{i-j+1}{2}\rfloor$.

- *(from C2' at* $(\mathsf{merge},1)$*) From the calculation of*

$$\widehat{[\![\eta]\!]}(\mathsf{merge},2,f(\centerdot))$$

where f *is the assignment function which only updates value held by* m *to be that by* i*, the algorithm generates the following constraint triple:*
  - $(\mathsf{merge}, i-1\geq 0\wedge j-i\geq 0, \mathfrak{t}_2(i,j)-t_3(i,j,i)-1)$.
- *(from C4' at* $(\mathsf{merge},4)$*) From the calculation of*

$$\mathbf{1}_{\phi}\cdot\widehat{[\![\eta]\!]}(\mathsf{merge},5,\centerdot)+\mathbf{1}_{\neg\phi}\cdot\widehat{[\![\eta]\!]}(\mathsf{merge},11,\centerdot)$$

where $\phi:=j-l\geq 0$*, the algorithm generates the following valid constraint triples:*
  - $(\mathsf{merge},\varphi_3,\mathfrak{t}_3(i,j,l)-\mathfrak{t}_3(i,j,l+1)-5)$ *with*

$$\varphi_3:=1\leq i\wedge i\leq j\wedge j-l\geq 0;$$

  - $(\mathsf{merge},\varphi_4,\mathfrak{t}_3(i,j,l)-\mathfrak{t}_4(j,i)-2)$ *with*

$$\varphi_4:=1\leq i\wedge i\leq j\wedge j+1-l\geq 0\wedge l-j-1\geq 0.$$

- *(from C4' at* $(\mathsf{merge},12)$*) From the calculation of*

$$\mathbf{1}_{\phi}\cdot\widehat{[\![\eta]\!]}(\mathsf{mergesort},13,\centerdot)+\mathbf{1}_{\neg\phi}\cdot\widehat{[\![\eta]\!]}(\mathsf{mergesort},15,\centerdot)$$

where $\phi:=j-l\geq 0$*, the algorithm generates the following valid constraint triples:*
  - $(\mathsf{merge},\varphi_5,\mathfrak{t}_4(j,l)-\mathfrak{t}_4(j,l+1)-3)$ *with*

$$\varphi_5:=i-1\geq 0\wedge j-i\geq 0\wedge j-l\geq 0;$$

  - $(\mathsf{merge},\varphi_6,\mathfrak{t}_4(j,l)-1)$ *with*

$$\varphi_6:=i-1\geq 0\wedge j-i\geq 0\wedge l-j-1\geq 0\wedge j-l+1\geq 0.$$

G.4 Step 4 of SynAlgo

The following proposition shows that Eq. (11) encompasses a simple proof system for non-negative polynomials over polyhedra.

Proposition 2. *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* $\{\vdash_d\}_{d\in\mathbb{N}}$ *be generate by the following rules:*

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

$$\frac{\vdash_d f_1\geq 0,\vdash_d f_2\geq 0}{\vdash_d f_1+f_2\geq 0}\qquad\frac{\vdash_{d_1} f_1\geq 0,\vdash_{d_2} f_2\geq 0}{\vdash_{d_1+d_2} f_1\cdot f_2\geq 0}\ .$$

Then for all $d\in\mathbb{N}$ *and polynomials* $g\in\mathfrak{R}[x_1,\ldots,x_m]$*, if* $\vdash_d g\geq 0$ *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* $u_1,\ldots,u_n\in$ *Monoid*(Γ) *such that every* u_i *is a product of no more than* d *polynomials in* Γ.

Proof. By an easy induction on d. □

Solving Constraint Triples **Step 4(a): Abstraction of Logarithm, Exponentiation, and Floored Expression.** All details of Step 4(a) is in the main article.

Step 4(b): Generating Linear Constraints for Abstracted Variables.

3. *Mutual No-Greater-Than Inequalities over* $\mathcal{E}_1$. For each pair $(\mathfrak{e},\mathfrak{e}')\in\mathcal{E}_1\times\mathcal{E}_1$ 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 formula

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

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.

5. *Constraints for Exponentiation.* For each variable $v_{\mathfrak{e}}$, the algorithm adds into Γ polynomial expressions $v_{\mathfrak{e}} - t_{\mathfrak{e}}^{k-1} \cdot \tilde{\mathfrak{e}}$ and $v_{\mathfrak{e}} - t_{\mathfrak{e}}^{k}$ due to the definition of $t_{\mathfrak{e}}$. And for each variable $v'_{\mathfrak{e}}$, the algorithm adds (i) $v'_{\mathfrak{e}} - t_{\mathfrak{e}}^{k-1}$ and (ii) either $v'_{\mathfrak{e}} - t_{\mathfrak{e}}^{k-2} \cdot \tilde{\mathfrak{e}}$ when $k \geq 2$ or $\tilde{\mathfrak{e}} - t_{\mathfrak{e}}^{2-k} \cdot v'_{\mathfrak{e}}$ when $1 < k < 2$.
7. *Mutual No-Greater-Than Inequalities over $u_{\mathfrak{e}}$'s.* For each pair $(\mathfrak{e}, \mathfrak{e}') \in \mathcal{E}_1 \times \mathcal{E}_1$ 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

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

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 $\tilde{\mathfrak{e}}$ (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

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

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

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

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 $\mathfrak{e} \neq \mathfrak{e}'$, $r^*_{\mathfrak{e},\mathfrak{e}'}, b^*_{\mathfrak{e},\mathfrak{e}'}$ are successfully found and $r^*_{\mathfrak{e},\mathfrak{e}'} > 0, b^*_{\mathfrak{e},\mathfrak{e}'} \geq 0$, the algorithm adds

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

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

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

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

Step 4(c): Solving Unknown Coefficients in the Template. All details are presented in the main article.

Theorem 6. Our algorithm, SYNALGO, is a polynomial-time algorithm and a sound approach for the RECTERMBOU problem, i.e., if SYNALGO succeeds to synthesize a function g on $\left\{(\mathsf{f}, \ell, \mu) \mid \mathsf{f} \in F, \ell \in L^{\mathsf{f}}_{\mathsf{s}}, \mu \in \overline{Val}_{\mathsf{f}}\right\}$, then $\hat{g}$ is a measure function and an upper bound on the abstract termination time.

Proof. The proof follows from the facts that (i) once the constraint triples are fulfilled, then $\hat{g}$ satisfies the conditions specified in Proposition 1, (ii) the abstraction is an over approximation of the floored expressions, logarithm terms and exponentiation terms, and (iii) Handelman's Theorem provides a sound form for positive polynomials over polyhedra. □

H. Experimental Details

In this part, we present the details for our experimental results. To implement Closest-Pair algorithm, Karatsuba's Algorithm and Strassen's Algorithm, we need to augment our syntax with array-creation statement $\langle arvar \rangle := \mathbf{newar}[\langle aexpr \rangle]$ where $\langle arvar \rangle$ is restricted to local array variables of function calls and $\langle aexpr \rangle$ specifies the length of the new array to be created. The behaviour of array-creation statement is similar to assignment statement. The semantics (right before Section 3) is then augmented by the following:

4. *array-creation*: if ℓ refers to an array-creation statement involving a local array variable ar and an $\langle aexpr \rangle$ expression $\mathfrak{e}$, then

$$(i)(w_{j+1}, \mathfrak{h}_{j+1}) := ((\mathsf{f}, \ell', \nu') \centerdot w', \mathfrak{h}')$$

whenever $\ell' \neq \ell^{\mathsf{f}}_{\text{out}}$ and otherwise

$$(i)(w_{j+1}, \mathfrak{h}_{j+1}) := (w', \mathfrak{h}')$$

where ℓ' is the unique successor of ℓ, $\mathfrak{h}'$ is obtained from $\mathfrak{h}$ by first finding the least d such that $\mathfrak{h}(d) \neq \perp$ and then changing $\mathfrak{h}(d)$ to be the array of length $\max\{1, \mathfrak{e}_{\mathfrak{h}}(\nu)\}$ with every entry being 0, and ν' is obtained by changing $\nu(ar)$ to d.

Array-creation statement is treated much the same as **skip** in the setting of measure functions (cf. Definition 11, Theorem 1, Definition 12, Definition 13 and Proposition 1) with an exception of change of array length, since we deem execution time of every array-creation statement to be 1. Note that we stipulate every newly-created array entry to be 0 simply to ensure that our semantics is deterministic; in the following implementations of algorithms we do not rely on this setting.

Pseudo-codes for Our Examples. Figure 7 and Figure 8 together account for Karatsuba's Algorithm. Figure 9 – Figure 12 demonstrate the divide-and-conquer algorithm for Closest-Pair problem. Figure 13 – Figure 15 show the Strassen's algorithm. Invariants are bracketed (i.e., $[\dots]$) at significant labels in the programs.

Approximation constants. We use approximation of constants upto 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 $2^{0.9}$.

```
// Initialize all array entries in A to be zero.
initialize(A, len) {
  [len ≥ 1]
  i := 1;
  [len ≥ 1 ∧ i ≤ len + 1]
  while i ≤ len do
    A[i] := 0;
    i := i + 1
  od
}

// Copy first len entries from beginA in A into B from beginB.
copy(A, beginA, B, beginB, len) {
  [len ≥ 1]
  i := 1;
  [len ≥ 1 ∧ i ≤ len + 1]
  while i ≤ len do
    j := i + beginA − 1;
    k := i + beginB − 1;
    B[k] := A[j];
    i := i + 1;
  od
}

// Add two arrays entrywise.
add(A, beginA, B, beginB, len) {
  [len ≥ 1]
  i := 1;
  [len ≥ 1 ∧ i ≤ len + 1]
  while i ≤ len do
    j := i + beginA − 1;
    k := i + beginB − 1;
    A[j] := A[j] + B[k];
    i := i + 1
  od
}

// Subtract two arrays entrywise.
subtract(A, beginA, B, beginB, len) {
  [len ≥ 1]
  i := 1;
  [len ≥ 1 ∧ i ≤ len + 1]
  while i ≤ len do
    j := i + beginA − 1;
    k := i + beginB − 1;
    A[j] := A[j] − B[k];
    i := i + 1
  od
}
```

Figure 7. Auxiliary Function Calls for Karatsuba's Algorithm

```
// The program calculates the product of polynomials A, B.
// The degree should be arranged in increasing order in both A, B.
// The result is stored in Ans.
// n is the length of both A and B and should be a power of 2.

karatsuba(A, B, Ans, 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-divide A (resp. B) into P1, P2 (resp. Q1, Q2)
      P1 := newar[t]; copy(A, 1, P1, 1, t);
      P2 := newar[t]; copy(A, t+1, P2, 1, t);
      Q1 := newar[t]; copy(B, 1, Q1, 1, t);
      Q2 := newar[t]; copy(B, t+1, Q2, 1, t);

      // T1 = P1 + P2 and T2 = Q1 + Q2
      T1 := newar[t]; T2 := newar[t];
      copy(P1, 1, T1, 1, t); add(T1, 1, P2, 1, t);
      copy(Q1, 1, T2, 1, t); add(T2, 1, Q2, 1, t);

      // recursive calls
      Z1 := newar[n-1]; karatsuba(P1, Q1, Z1, t);
      Z2 := newar[n-1]; karatsuba(P2, Q2, Z2, t);
      Z3 := newar[n-1]; karatsuba(T1, T2, Z3, t);

      // adjusting Z3
      subtract(Z3, 1, Z1, 1, n-1);
      subtract(Z3, 1, Z2, 1, n-1);

      // combining step
      initialize(Ans, 2*n-1);
      add(Ans, 1, Z1, 1, n-1);
      add(Ans, 1, Z2, n, n-1);
      add(Ans, t+1, Z3, 1, n-1)
    else skip // If n is not even, simply fail.
    fi
  else Ans[1] := A[1]*B[1] fi
}
```

Figure 8. Main Function Call for Karatsuba's Algorithm

```
// Copy first len entries from beginA in A into B from beginB.
copy(A, beginA, B, beginB, len) {
  [len ≥ 1]
  i := 1;
  [len ≥ 1 ∧ i ≤ len + 1]
  while i ≤ len do
    j := i + beginA - 1;
    k := i + beginB - 1;
    B[k] := A[j];
    i := i + 1
  od
}

// sorting A while adjusting B accordingly

mergesort(A, B, i, j, TA, TB) {
  [1 ≤ i ∧ i ≤ j]
  if 1 ≤ i and i ≤ j - 1 then
    k := i + ⌊(j-i+1)/2⌋ - 1;
    mergesort(A, B, i, k, TA, TB);
    mergesort(A, B, k+1, j, TA, TB);
    merge(i, j, k, A, B, TA, TB)
  else
    skip
  fi
}

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

Figure 9. Merge-Sort and Copy for Closest-Pair

```
// Caculates the shortest distance of a finite set of points.
// A stores x-coordinates and B stores y-coordinates.
// The return value is -1 for single-point case.
// Otherwise, the return value is the square of the shortest distance.
// The return value is stored in Res[1].

clst_pair_main(A, B, i, j, Res) {
  [1 ≤ i ∧ i ≤ j]

  // copying A, B into C, D
  C := newar[j − i + 1]; D := newar[j − i + 1];
  copy(A, i, C, i, j − i + 1); copy(B, i, D, i, j − i + 1);

  // arrays for sorting and mid-line use
  T_A := newar[j − i + 1]; T_B := newar[j − i + 1];

  // sorting A, B in x-coordinate
  mergesort(A, B, i, j, T_A, T_B);

  // sorting C, D in y-coordinate
  mergesort(D, C, i, j, T_A, T_B);

  // solving the result
  clst_pair(A, B, i, j, C, D, T_A, T_B, Res)
}
```

Figure 10. Main Function Call for Closest-Pair

```
// principle recursive function call for solving Closest-Pair
// The return value is stored in Res[1].

clst_pair(A, B, i, j, C, D, T_A, T_B, Res) {
  [1 ≤ i ∧ i ≤ j]
  if 1 ≤ i and i ≤ j − 3 then
    // recursive case where there are at least 4 points

    k := i + ⌊(j−i+1)/2⌋ − 1;
    R_1 := newar[1]; R_2 := newar[1];
    clst_pair(A, B, i, k, C, D, T_A, T_B, R_1);
    clst_pair(A, B, k + 1, j, C, D, T_A, T_B, R_2);

    // taking the minimum
    p := 1;
    if R_1[p] ≥ R_2[p] then
      Res[p] := R_1[p]
    else
      Res[p] := R_2[p]
    fi;

    // fetch and scan the mid-line
    fetch&scan(A, B, i, j, C, D, T_A, T_B, Res)
  else
    // base case (fewer than 4 points)
    base_clstpair(A, B, i, j, Res)
  fi
}
```

Figure 11. Principle Recursive Function Call for Closest-Pair

```
compare(A, B, i, j, Res) {
  [1 ≤ i ∧ i ≤ j]
  if Res[1] ≥ (A[i] − A[j])^2 + (B[i] − B[j])^2
     or Res[1] ≤ −1
  then Res[1] := (A[i] − A[j])^2 + (B[i] − B[j])^2
  else skip fi
}

// base case (fewer than 4 points)
base_clstpair(A, B, i, j, Res) {
  [1 ≤ i ∧ i ≤ j]
  Res[1] := −1;
  if i ≤ j and j ≤ i then
    // single-point case
    skip
  else
    if i + 1 ≤ j and j ≤ i + 1 then
      // double-point case
      compare(A, B, i, j, Res);
    else
      // triple-point case
      k := i + 1;
      compare(A, B, i, k, Res);
      compare(A, B, i, j, Res);
      compare(A, B, k, j, Res)
    fi
  fi;
}

// fetch and scan the mid-line
fetch&scan(A, B, i, j, C, D, T_A, T_B, Res) {
  // fetching the points on the mid-line
  [1 ≤ i ∧ i ≤ j − 3]
  l := i; k := i + ⌊(j−i+1)/2⌋ − 1; n := k + 1; p := i
  [1 ≤ i ∧ i ≤ j − 3 ∧ p ≤ j + 1 ∧ l ≤ j + 1]
  while p ≤ j do
    if A[k] + A[n] − 2 ∗ Res[1] ≤ 2 ∗ C[l]
       and 2 ∗ C[l] ≤ A[k] + A[n] + 2 ∗ Res[1]
    then
      T_A[l] := C[l]; T_B[l] := D[l]; l := l + 1
    else skip fi;
    p := p + 1
  od

  if l ≥ i + 1 then
    p := i;

    // scanning the points on the mid-line
    [1 ≤ i ∧ i ≤ j − 3 ∧ l ≤ j + 1 ∧ p ≤ l]
    while p ≤ l − 1 do
      m := p + 1;

      // checking 7 points ahead on the mid-line
      [1 ≤ i ∧ i ≤ j ∧ l ≤ j + 1 ∧ m ≤ p + 8]
      while m − p ≤ 7 and m ≤ l − 1 do
        compare(T_A, T_B, p, m, Res);
        m := m + 1
      od;
      p := p + 1
    od
  else skip fi
}
```

Figure 12. Other Function Calls for Closest-Pair

```
// The program calculates the product A · B of square matrices A, B.
// The result is stored in Ans.
// n is the row/column size of both A and B and should be a power of 2.
// Each of A, B is stored in an array of length n^2.

strassen(A, B, Ans, 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-divide A (resp. B) into A_ij, B_ij's)
      A_{1,1} := newar [t*t]; A_{1,2} := newar [t*t];
      A_{2,1} := newar [t*t]; A_{2,2} := newar [t*t];
      matrixtoblocks(A, A_{1,1}, A_{1,2}, A_{2,1}, A_{2,2}, n, t);
      B_{1,1} := newar [t*t]; B_{1,2} := newar [t*t];
      B_{2,1} := newar [t*t]; B_{2,2} := newar [t*t];
      matrixtoblocks(B, B_{1,1}, B_{1,2}, B_{2,1}, B_{2,2}, n, t);

      // sums of matrices
      T_1 := newar [t*t]; T_2 := newar [t*t];
      copy(T_1, A_{1,1}, t); add(T_1, A_{2,1}, t);
      copy(T_2, B_{1,1}, t); add(T_2, B_{2,2}, t);

      T_3 := newar [t*t]; T_4 := newar [t*t];
      copy(T_3, A_{2,1}, t); add(T_3, A_{2,2}, t);
      copy(T_4, B_{1,2}, t); subtract(T_4, B_{2,2}, t);

      T_5 := newar [t*t]; T_6 := newar [t*t];
      copy(T_5, B_{2,1}, t); add(T_5, B_{1,1}, t);
      copy(T_6, A_{1,1}, t); add(T_6, A_{1,2}, t);

      T_7 := newar [t*t]; T_8 := newar [t*t];
      copy(T_7, A_{2,1}, t); subtract(T_7, A_{1,1}, t);
      copy(T_8, B_{1,1}, t); add(T_8, B_{1,2}, t);

      T_9 := newar [t*t]; T_10 := newar [t*t];
      copy(T_9, A_{1,2}, t); subtract(T_9, A_{2,2}, t);
      copy(T_10, B_{2,1}, t); add(T_10, B_{2,2}, t);

      // recursive calls
      D_1 := newar [t*t]; strassen(T_1, T_2, D_1, t);
      D_2 := newar [t*t]; strassen(T_3, B_{1,1}, D_2, t);
      D_3 := newar [t*t]; strassen(A_{1,1}, T_4, D_3, t);
      D_4 := newar [t*t]; strassen(A_{2,2}, T_5, D_4, t);
      D_5 := newar [t*t]; strassen(T_6, B_{2,2}, D_5, t);
      D_6 := newar [t*t]; strassen(T_7, T_8, D_6, t);
      D_7 := newar [t*t]; strassen(T_9, T_10, D_7, t);

      // combining stage
      copy(A_{1,1}, D_1, t); add(A_{1,1}, D_4, t);
      subtract(A_{1,1}, D_5, t); add(A_{1,1}, D_7, t);

      copy(A_{1,2}, D_3, t); add(A_{1,2}, D_5, t);
      copy(A_{2,1}, D_2, t); add(A_{2,1}, D_4, t);

      copy(A_{2,2}, D_1, t); add(A_{2,2}, D_3, t);
      subtract(A_{2,2}, D_2, t); add(A_{2,2}, D_6, t);

      blockstomatrix(Ans, A_{1,1}, A_{1,2}, A_{2,1}, A_{2,2}, n, t)
    else skip // If n is not even, simply fail.
    fi
  else Ans[1] := A[1] * B[1] fi
}
```

Figure 13. Main Function Call for Strassen's Algorithm

```
// Partition matrix A into block matrices.
matrixtoblocks(A, A_{1,1}, A_{1,2}, A_{2,1}, A_{2,2}, 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
      l := (t − 1) * i + j;
      k := (n − 1) * i + j; A_{1,1}[l] := A[k];
      k := (n − 1) * i + j + t; A_{1,2}[l] := A[k];
      k := (n − 1) * (i + t) + j; A_{2,1}[l] := A[k];
      k := (n − 1) * (i + t) + j + t; A_{2,1}[l] := A[k];
      j := j + 1
    od;
    i := i + 1
  od
}

// Construct matrix A from block matrices.
blockstomatrix(A, A_{1,1}, A_{1,2}, A_{2,1}, A_{2,2}, 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
      l := (t − 1) * i + j;
      k := (n − 1) * i + j; A[k] := A_{1,1}[l];
      k := (n − 1) * i + j + t; A[k] := A_{1,2}[l];
      k := (n − 1) * (i + t) + j; A[k] := A_{2,1}[l];
      k := (n − 1) * (i + t) + j + t; A[k] := A_{2,1}[l];
      j := j + 1
    od;
    i := i + 1
  od
}

// Copy square matrix B to A.
copy(A, B, 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
      k := (n − 1) * i + j; A[k] := B[k];
      j := j + 1
    od;
    i := i + 1
  od
}
```

Figure 14. Auxiliary Function Calls for Strassen's Algorithm

```
// Add two matrices (entrywise).
add(A, B, 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
      k := (n − 1) ∗ i + j;  A[k] := A[k] + B[k];
      j := j + 1
    od;
    i := i + 1
  od
}

// Subtract two matrices (entrywise).
subtract(A, B, 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
      k := (n − 1) ∗ i + j;  A[k] := A[k] − B[k];
      j := j + 1
    od;
    i := i + 1
  od
}
```

Figure 15. Matrix Addition and Subtraction for Strassen's Algorithm