Modular Automatic Complexity Analysis of Recursive Integer Programs

Nils Lommen(✉) and Jürgen Giesl

RWTH Aachen University, Aachen, Germany
{lommen,giesl}@cs.rwth-aachen.de

Abstract. In earlier work, we developed a modular approach for automatic complexity analysis of integer programs. However, these integer programs do not allow non-tail *recursive* calls or subprocedures. In this paper, we consider integer programs with function calls and present a natural extension of our modular complexity analysis approach to the recursive setting based on a new form of ranking functions. Hence, our approach combines already existing powerful techniques on the "imperative" parts of the program and our novel ranking functions on the recursive parts. The strength of this combination is demonstrated by our implementation in the complexity analysis tool KoAT.

1 Introduction

There exist numerous approaches to analyze complexity of programs automatically, e.g., [3–5, 7, 10, 11, 18, 22, 23, 28, 29, 37, 42, 45, 48], but most of them are essentially limited to non-recursive programs. There are also several techniques for complexity analysis of term rewrite systems (TRSs) [6] which can handle arbitrary recursion. However, TRSs have the drawback that they do not support built-in data types like integers. Thus, the goal of this paper is to automatically analyze the complexity of programs with built-in integers *and* arbitrary (possibly non-tail) recursion.

In previous work, we developed a *modular* technique for complexity analysis of programs with built-in integers which we implemented in the complexity analysis tool KoAT [10, 22, 35, 36, 38]. It automatically infers runtime bounds for integer transition systems (ITSs) possibly consisting of multiple loops by handling some subprograms as *twn*-loops (where there exist "complete" techniques for analyzing termination and complexity [24, 25, 35, 36, 38]) and by using multiphase-linear ranking functions [7, 8, 22] for other subprograms. By inferring bounds for one subprogram after the other, in the end we obtain a bound on the runtime of the whole program. In this paper, we extend our approach to ITSs which allow *function calls*, including non-tail recursion. In the first attempt for such an extension from [10, Sect. 5], the results of function calls were simply disregarded. In contrast, our novel approach can take the results of function calls into account which leads to a much higher precision.

```
main(x, y):
while x > 0 do
  y ← y + fac(x); x ← x − 1;
x ← 1;
while x < y do
  x ← 3 · x; y ← 2 · y;

fac(a):
if a = 0 then
  return 1;
else if a > 0 then
  return a · fac(a − 1);
```

Fig. 1: Recursive Integer Program with two Procedures

Example 1. The first **while**-loop of the procedure **main** in Fig. 1 computes $x! + \cdots + 1!$ by calling the subprocedure **fac**. We introduce a novel class of ranking functions for recursive programs to show that this loop has quadratic runtime (when every assignment has the "cost" 1). Note that y's value is bounded by $y + x \cdot x^x$, where x and y refer to the initial values of the program variables. The reason is that $x! + \cdots + 1!$ can be over-approximated by $x \cdot x^x$ since $1!, \ldots, x!$ are all bounded by x^x. This observation is crucial for the runtime of the second loop since it is executed at most $\log_2(\text{size}(y)) + 2 = \log_2(y + x \cdot x^x) + 2$ times, where "size(y)" denotes such an over-approximation of the (absolute) value of y before the second loop. Hence, as the runtime of the first loop is quadratic and the runtime of the second loop is less than quadratic, the overall program has quadratic runtime. Here, [10, Sect. 5] fails to infer a finite runtime bound, as it disregards the return value of **fac**. The runtime bound $\log_2(y) + 2$ for the second loop can be obtained by our technique based on *twn*-loops, but not by linear ranking functions. Thus, our novel approach for recursive integer programs allows us, e.g., to apply ranking functions on some (possibly recursive) parts of the program and techniques for *twn*-loops on other parts, i.e., it allows for modular proofs that use different techniques for automated complexity analysis on different subprograms in order to benefit from their individual strengths.

In this work, we extend our notions of runtime and size bounds [10, 22, 35, 36, 38] to the new setting of ITSs with function calls. On the one hand, as illustrated by Ex. 1, we need size bounds to compute runtime bounds. On the other hand, we also need runtime bounds to infer size bounds, because to this end we have to over-approximate how often loops with variable updates are iterated. Thus, our approach alternates between the computation of runtime and size bounds. All our contributions are implemented in our complexity analysis tool KoAT.

Structure: In Sect. 2 we introduce our new notion of ITSs with function calls and define runtime and size bounds for these programs. In Sect. 3, we show how to compute modular runtime bounds for our new class of programs. Analogously, we present a technique to infer size bounds in a modular way in Sect. 4. Finally, in Sect. 5 we discuss related work and our implementation in the tool KoAT, and provide an experimental evaluation demonstrating the strengths of our approach. All proofs can be found in [40, App. A].

2 Recursive Integer Transition Systems

In Sect. 2.1 we extend ITSs by function calls and recursion. Afterwards, in Sect. 2.2 we define runtime and size bounds which extend the corresponding notions for ITSs without function calls [10, 22, 35, 36, 38] in a natural way.

2.1 Syntax and Semantics of Recursive Integer Transition Systems

We fix a finite set of program variables $\mathcal{V}$. As usual, $\mathbb{Z}[\mathcal{V}]$ is the polynomial ring over the variables $\mathcal{V}$ with integer coefficients. We use polynomials for the *constraints* in the guards of transitions.

Definition 2 (Atoms and Constraints). *The set of* atoms $\mathcal{A}$ *consists of all inequations* $p_1 < p_2$ *for polynomials* $p_1, p_2 \in \mathbb{Z}[\mathcal{V}]$. *The set* $\mathcal{C}$ *of* constraints *consists of all formulas built from atoms* $\mathcal{A}$ *and* $\wedge$.

We also use "$\geq$", "$=$", "$\neq$", and negations "$\neg$", since they can be simulated by atoms and constraints (e.g., $p_1 \geq p_2$ is equivalent to $p_2 < p_1 + 1$ for integers). Disjunctions "$\vee$" are modeled by several transitions with the same start and target location.

ITSs are a widely studied formalism in automatic program verification (see, e.g., [1, 10, 12, 14, 27, 34]). An ITS consists of a set $\mathcal{L}$ of *locations* and a set $\mathcal{T}$ of *transitions*, where a transition connects two locations. Moreover, every transition has a polynomial *update* function $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$. The values of the variables are "stored" in a *state* $\sigma : \mathcal{V} \to \mathbb{Z}$, where Σ denotes the set of all states. When evaluating a transition, the values of the variables are changed according to its update function, and we move from a *configuration* $(\ell, \sigma) \in \mathcal{L} \times \Sigma$ to another configuration (ℓ', σ'). Since ITSs do not allow any non-tail recursion, Def. 3 extends them to ITSs with *function calls* (called ρ-ITSs). To this end, we introduce a set $\mathcal{F}$ of *function calls* $\ell(\zeta)$ which can now occur in the updates of variables as well. Here, ℓ is the start location of the subprogram that is called and the update $\zeta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$ sets the program variables of the subprogram to their initial values. More precisely, if $\sigma \in \Sigma$ is the state before calling the subprogram via $\ell(\zeta)$, then the subprogram starts in a configuration $(\ell, \tilde{\sigma})$, where $\tilde{\sigma}$ results from "applying" the update ζ to the state σ. For example, if $\zeta_1(a) = x$, then $f_1(\zeta_1)$ represents the function call that sets a to the current value of x and jumps to the location f_1.

In addition, we also introduce a set of *return locations* $\Omega \subseteq \mathcal{L}$, and for every return location $\ell' \in \Omega$, we let $v_{\ell'} \in \mathcal{V}$ denote its *return variable*. As soon as a called subprogram reaches a configuration (ℓ', σ') with $\ell' \in \Omega$, the value $\sigma'(v_{\ell'})$ is returned as the result of the function call $\ell(\zeta)$. (We will define the semantics of ρ-ITSs formally in Def. 5 and 6.) Thus, transitions may now have updates which map variables to polynomial combinations of variables *and* function calls. This is denoted by $\mathbb{Z}[\mathcal{V} \cup \mathcal{F}]$. In this way, the results of function calls can be combined polynomially, and they can be used by transitions when updating variables.

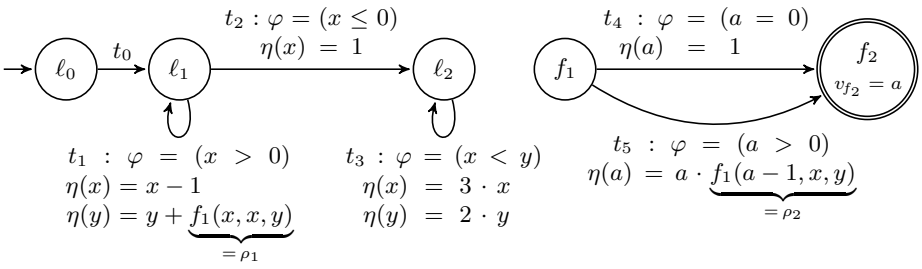

Fig. 2: An Integer Transition System with Function Calls ρ_1 and ρ_2

Definition 3 (ρ-ITS). *The tuple $(\mathcal{L}, \ell_0, \Omega, \mathcal{F}, \mathcal{T})$ is an* ITS with function calls *(ρ-ITS) where*

- *$\mathcal{L}$ is a finite set of* locations *with an initial location $\ell_0 \in \mathcal{L}$,*
- *$\mathcal{F}$ is a finite set of* function calls *$\ell(\zeta)$ with $\ell \in \mathcal{L} \setminus \{\ell_0\}$ and $\zeta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$,*
- *$\Omega \subseteq \mathcal{L}$ is a set of* return locations *and for every $\ell \in \Omega$, $v_\ell \in \mathcal{V}$ denotes its* return variable*, and*
- *$\mathcal{T}$ is a finite set of* transitions*: A transition is a 4-tuple $(\ell, \varphi, \eta, \ell')$ with start location $\ell \in \mathcal{L}$, target location $\ell' \in \mathcal{L} \setminus \{\ell_0\}$, guard $\varphi \in \mathcal{C}$, and update function $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V} \cup \mathcal{F}]$.*

We denote the set of function calls in a polynomial p, an update η, or a transition t by $\mathrm{fun}(p)$, $\mathrm{fun}(\eta)$, or $\mathrm{fun}(t)$, respectively. Similarly, for a function call $\rho \in \mathcal{F}$, $\mathrm{trans}(\rho)$ is the set of all transitions of the ITS in which ρ occurs in an update.

Note that our definition of ρ-ITSs allows non-deterministic branching since several transitions can have the same start location. Moreover, to model non-deterministic sampling, our approach can easily be extended by additional "temporary" variables which are updated arbitrarily in each evaluation step. Intuitively, these variables are set non-deterministically by an adversary trying to "sabotage" the program in order to obtain long runtimes. However, we omitted such temporary variables from the paper to simplify the presentation.

Example 4. The ρ-ITS in Fig. 2 corresponds to the program from Fig. 1. In Fig. 2, we omitted trivial guards $\varphi = \texttt{true}$ and identity updates of the form $\eta(v) = v$. The ρ-ITS has the program variables $\mathcal{V} = \{a, x, y\}$, five locations $\mathcal{L} = \{\ell_0, \ell_1, \ell_2, f_1, f_2\}$, and two function calls $\rho_1 = f_1(\zeta_1)$ and $\rho_2 = f_1(\zeta_2)$, where $\zeta_1(a) = x$ and $\zeta_2(a) = a-1$, and both ζ_1 and ζ_2 keep x and y unchanged. To ease readability, we wrote $f_1(x, x, y)$ and $f_1(a-1, x, y)$ instead of $f_1(\zeta_1)$ and $f_1(\zeta_2)$ in Fig. 2. The subprogram with the locations f_1 and f_2 computes the factorial $a!$ recursively and returns this result in the return variable a when reaching the return location f_2 (indicated by the doubled node). This subprogram is called iteratively in the loop t_1 with the argument x (i.e., with ζ_1 where $\zeta_1(a) = x$). The factorials $x!, (x-1)!, \ldots, 1$ are summed up in the variable y. Afterwards, x is set to 1 in t_2, and the second loop t_3 at location ℓ_2 is executed.

Let $[\![e]\!]_\sigma$ denote the *evaluation* of an expression e in the state $\sigma \in \Sigma$, where $[\![e]\!]_\sigma$ results from replacing every variable v in e by its value $\sigma(v)$. So, for example, evaluating $[\![3 \cdot x]\!]_\sigma$ and $[\![x > 0]\!]_\sigma$ at $\sigma(x) = 2$ results in 6 and `true`, respectively.

From now on, we fix a ρ-ITS $(\mathcal{L}, \ell_0, \Omega, \mathcal{F}, \mathcal{T})$ over the variables $\mathcal{V}$. Formally, an evaluation step of a ρ-ITS is a transformation of an *evaluation tree* $\mathbb{T}$ whose nodes are labeled with configurations from $\mathcal{L} \times (\Sigma \cup \{\bot\})$ and whose edges are labeled with transitions or function calls. We distinguish two kinds of evaluation steps: t-evaluation steps (for transitions $t \in \mathcal{T}$) and ε-evaluation steps.

If a leaf of $\mathbb{T}$ is labeled with a configuration (ℓ, σ) where a transition $t = (\ell, \varphi, \eta, \ell')$ can be applied, then a *t-evaluation step* extends $\mathbb{T}$ at the position of this leaf to a new tree $\mathbb{T}'$, denoted $\mathbb{T} \prec_t \mathbb{T}'$. If the update η does not contain any function calls, then $\mathbb{T}'$ results from $\mathbb{T}$ by adding an edge from the node labeled with (ℓ, σ) to a new node labeled with a configuration (ℓ', σ') where $\sigma'(v) = [\![\eta(v)]\!]_\sigma$ for all program variables v. This new edge is labeled with t, i.e., $(\ell, \sigma) \to_t (\ell', \sigma')$. This corresponds to ordinary evaluations of ITSs as in [22, 36, 38].

However, if t's update contains function calls $\rho_i = \ell_i(\zeta_i)$ for $1 \leq i \leq n$, then $\mathbb{T}'$ results from $\mathbb{T}$ by adding $n + 1$ children to the former leaf labeled with (ℓ, σ): The child $(\ell', \bot)$ is connected by an edge labeled with t, i.e., $(\ell, \sigma) \to_t (\ell', \bot)$. Here, $\bot$ denotes an undefined state which will be instantiated later if the function calls $\rho_1, \ldots, \rho_n$ reach return locations. Moreover, the children (ℓ_i, σ_i) are connected by edges labeled with ρ_i for all $1 \leq i \leq n$, i.e., $(\ell, \sigma) \to_{\rho_i} (\ell_i, \sigma_i)$, where $\sigma_i(v) = [\![\zeta_i(v)]\!]_\sigma$ for all program variables v.

When modeling the semantics in an alternative stack-based way, the ρ_i-edges would correspond to a push-operation where the function call ρ_i is pushed on top of the call stack. Our evaluation trees are an explicit representation of such stacks. They lift the semantics of ITSs to ρ-ITSs in a natural way by moving from evaluation *paths* to *trees*. An evaluation tree keeps track of every state that was reached during the evaluation. While our operational semantics via evaluation trees are equivalent to the stack-based semantics of recursion, the advantage over the stack representation is that evaluation trees are particularly suitable for our novel *ρ-ranking functions* for transitions with function calls in Sect. 3.1.

For an initial state $\sigma_0 \in \Sigma$, the evaluation always starts with the initial evaluation tree $\mathbb{T}_{\sigma_0} = (\{(\ell_0, \sigma_0)\}, \varnothing)$ which consists of the single node (ℓ_0, σ_0) and does not have any edges. We now define how to extend such trees using t- and ε-steps. Then the set of *evaluation trees* is the smallest set of trees that can be obtained from such initial trees by repeated application of t- and ε-steps.

Definition 5 (Evaluation of ρ-ITSs (t-Evaluation Step)). *Let $\mathbb{T}$ be an evaluation tree. $\mathbb{T} \prec_t \mathbb{T}'$ is a* t-evaluation step *with the transition $t = (\ell, \varphi, \eta, \ell')$ iff $\mathbb{T}$ has a leaf labeled with (ℓ, σ) where $\sigma \in \Sigma$, $[\![\varphi]\!]_\sigma = \texttt{true}$, and*

- *if $\mathrm{fun}(\eta) = \varnothing$, then $\mathbb{T}'$ is the extension of $\mathbb{T}$ by an edge $(\ell, \sigma) \to_t (\ell', \sigma')$ to a new node labeled with (ℓ', σ') where $\sigma'(v) = [\![\eta(v)]\!]_\sigma$ for all $v \in \mathcal{V}$.*
- *if η contains the function calls $\rho_1 = \ell_1(\zeta_1), \ldots, \rho_n = \ell_n(\zeta_n)$, then $\mathbb{T}'$ is the extension of $\mathbb{T}$ by the edges $(\ell, \sigma) \to_t (\ell', \bot)$ and $(\ell, \sigma) \to_{\rho_i} (\ell_i, \sigma_i)$, where $\sigma_i(v) = [\![\zeta_i(v)]\!]_\sigma$ for all $v \in \mathcal{V}$ and all $1 \leq i \leq n$.*

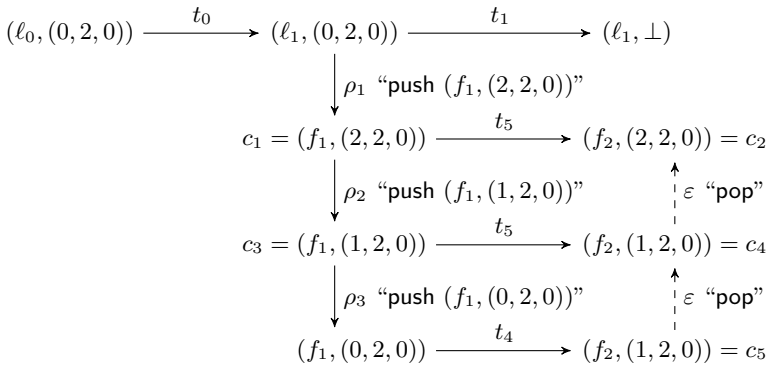

Fig. 3: Exemplary Evaluation of a ρ-ITS

To instantiate the undefined state $\bot$, we use *ε-evaluation steps*. An ε-evaluation step corresponds to a pop-operation in the stack-based semantics, i.e., it is used to return the value of a function call after it has been fully evaluated. If the tree contains $(\ell, \sigma) \to_t (\ell', \bot)$ and $(\ell, \sigma) \to_{\rho_i} (\ell_i, \sigma_i)$ for $1 \leq i \leq n$, and there are paths from the nodes (ℓ_i, σ_i) to configurations (ℓ_i', σ_i') where $\ell_i' \in \Omega$ is a return location, then the undefined state $\bot$ can be replaced by a state σ' such that $\sigma'(v) = [\![\overline{\eta}(v)]\!]_\sigma$ for all program variables v. If η is the update of the transition t, then $\overline{\eta}(v)$ results from $\eta(v)$ by replacing every function call $\ell_i(\zeta_i)$ occurring in η by the corresponding returned value $\sigma_i'(v_{\ell_i'})$ for all $1 \leq i \leq n$. We denote this by $\overline{\eta}(v) = \eta(v)\,[\ell_i(\zeta_i)/\sigma_i'(v_{\ell_i'})]$. Thus, an ε-evaluation step does not add new edges to an evaluation tree; it merely replaces $\bot$ by a state with concrete values obtained from the evaluated function calls.

Definition 6 (Evaluation of ρ-ITSs (ε-Evaluation Step)). *Let $\mathbb{T}$ be an evaluation tree. Furthermore, let there be a transition $t = (\ell, \varphi, \eta, \ell')$ such that $\mathbb{T}$ contains a node labeled with (ℓ, σ), with edges and children nodes of the form $(\ell, \sigma) \to_t (\ell', \bot)$ and $(\ell, \sigma) \to_{\rho_i} (\ell_i, \sigma_i)$ for function calls $\rho_i = \ell_i(\zeta_i) \in \mathrm{fun}(\eta)$.*

If $\mathbb{T}$ contains paths from each child labeled with (ℓ_i, σ_i) to a node labeled with $(\ell_i', \sigma_i') \in \Omega \times \Sigma$, then $\mathbb{T} \prec_\varepsilon \mathbb{T}'$ is an ε-evaluation step *iff $\mathbb{T}'$ results from $\mathbb{T}$ by replacing the label $(\ell', \bot)$ by (ℓ', σ'), where $\sigma'(v) = [\![\eta(v)\,[\ell_i(\zeta_i)/\sigma_i'(v_{\ell_i'})]]\!]_\sigma$ for all variables $v \in \mathcal{V}$.*

We write $\prec_\mathcal{T}$ for $\bigcup_{t \in \mathcal{T}} \prec_t$ and $\prec$ for $\prec_{\mathcal{T} \cup \{\varepsilon\}}$. Moreover, we denote finitely many evaluations steps $\mathbb{T} \prec \cdots \prec \mathbb{T}'$ by $\mathbb{T} \prec^* \mathbb{T}'$.

Example 7. Reconsider the ρ-ITS from Fig. 2 and let us denote states $\sigma \in \Sigma$ as tuples $(\sigma(a), \sigma(x), \sigma(y)) \in \mathbb{Z}^3$. The tree in Fig. 3 shows an evaluation starting in $\mathbb{T}_{(0,2,0)}$. Here, a dashed arrow indicates that a state which was reached via a function call was used to replace $\bot$ via an ε-evaluation step. Note that these dashed arrows are not part of the actual evaluation tree but they are just depicted

to illustrate the construction of the tree.[1] So for example, if σ_i always denotes the state of the configuration c_i, then for the configuration $c_2 = (f_2, \sigma_2)$, the value $\sigma_2(a) = 2$ is obtained from t_5's update $\eta(a) = a \cdot f_1(\zeta_2)$ and the value of the return variable in the configuration c_4, i.e., $[\![\eta(a)\,[f_1(\zeta_1)/\sigma_4(a)]]\!]_{\sigma_1} = [\![a \cdot \sigma_4(a)]\!]_{\sigma_1} = \sigma_1(a) \cdot \sigma_4(a) = 2 \cdot 1 = 2$.

In the next evaluation step, $(\ell_1, \bot)$ can be instantiated by considering the return variable in the configuration c_2. Then, $\bot$ would be replaced by $(0, 1, 2)$. Note that while in this tree, every node has at most one child connected by a ρ-edge, in general a node can have several outgoing ρ-edges if there exist transitions whose updates contain several function calls (e.g., the naive implementation of the Fibonacci numbers).

The goal of complexity analysis is to derive an upper bound on the number of t-evaluation steps starting in the initial tree $\mathbb{T}_{\sigma_0}$ with the single node (ℓ_0, σ_0). For any tree $\mathbb{T}$ and set of transitions $\mathcal{T}$, $|\mathbb{T}|_{\mathcal{T}}$ is the number of edges which are marked by a transition from $\mathcal{T}$. The *runtime complexity* measures how many transitions are evaluated in the worst case in any evaluation tree that results from $\mathbb{T}_{\sigma_0}$. In the following, let $\overline{\mathbb{N}} = \mathbb{N} \cup \{\omega\}$.

Definition 8 (Runtime Complexity). *The* runtime complexity *is* $\mathrm{rc} : \Sigma \to \overline{\mathbb{N}}$ *and* $\mathrm{rc}(\sigma_0) = \sup\{|\mathbb{T}|_{\mathcal{T}} \mid \mathbb{T}_{\sigma_0} \prec^* \mathbb{T}\}$.

Here, we count only t-edges labeled by transitions (and no ρ-edges labeled by function calls) to obtain a natural extension of our previous approach without function calls. Note that an upper bound on the number of ρ-edges can be easily derived from an upper bound on the number of t-evaluation steps by multiplying with the branching factor (i.e., by the maximal number of function calls occurring in the update of any transition).

2.2 Runtime and Size Bounds for ρ-ITSs

Now we define our notion of *bounds*. We only consider bounds which correspond to functions f that are weakly monotonically increasing in all arguments, i.e., where $x \leq y$ implies $f(\ldots x \ldots) \leq f(\ldots y \ldots)$. In this way, if f and g are both upper bounds, then $f \circ g$ is also an upper bound, i.e., bounds can be "composed" easily. For example, we used this in the introductory Ex. 1 where we inserted the size bound "size(y)" into the runtime bound $\log_2(y) + 2$ of the second loop.

In principle, every weakly monotonically increasing function could be used as a bound in our framework. However, here we restrict ourselves to bounds which are easy to represent and to compute with, and which cover the most prominent complexity classes. As in [38], bounds can also be *logarithmic*. In contrast to our

[1] This is crucial, because the edges of the evaluation trees correspond to those steps were our *ρ-ranking functions* must be (strictly or weakly) decreasing. So they have to decrease for steps with transitions like t_1 or t_5 (where variable values are changed according to the result of a function call), but not for ε- (or "pop")-steps. This is the reason for defining the evaluation of ρ-ITSs via evaluation trees instead of stacks.

earlier papers, we also consider exponential bounds with non-constant bases to represent bounds like x^x.

Definition 9 (Bounds). *The set of* bounds $\mathcal{B}$ *is the smallest set with* $\overline{\mathbb{N}} \subseteq \mathcal{B}$, $\mathcal{V} \subseteq \mathcal{B}$, *and* $\{b_1 + b_2, \max(b_1, b_2), b_1 \cdot b_2, p^{b_1}, \log_k(b_1)\} \subseteq \mathcal{B}$ *for all* $b_1, b_2 \in \mathcal{B}$, *all polynomials* $p \in \mathbb{N}[\mathcal{V}]$, *and all* $k \in \mathbb{R}_{>1}$.[2]

A *runtime bound* $\mathcal{RB}(t)$ over-approximates the number of t-evaluations that can occur in an arbitrary evaluation starting in a state $\sigma_0 \in \Sigma$, i.e., it is a bound on the number of t-edges in any evaluation tree resulting from $\mathbb{T}_{\sigma_0}$. In the following, let $|\sigma|$ denote the state with $|\sigma|(v) = |\sigma(v)|$ for all $v \in \mathcal{V}$.

Definition 10 (Runtime Bound). $\mathcal{RB} : \mathcal{T} \to \mathcal{B}$ *is a* runtime bound *if for all* $\sigma_0 \in \Sigma$, *all* $t \in \mathcal{T}$, *and all trees* $\mathbb{T}$ *with* $\mathbb{T}_{\sigma_0} \prec^* \mathbb{T}$, *we have* $|\mathbb{T}|_{\{t\}} \leq [\![\mathcal{RB}(t)]\!]_{|\sigma_0|}$.

Cor. 11 shows that to obtain an upper bound on the runtime complexity, one can compute runtime bounds for each transition separately and add them.

Corollary 11 (Over-Approximating rc**).** *Let* $\mathcal{RB}$ *be a runtime bound. Then for all states* $\sigma_0 \in \Sigma$, *we have* $\mathrm{rc}(\sigma_0) \leq [\![\sum_{t \in \mathcal{T}} \mathcal{RB}(t)]\!]_{|\sigma_0|}$.

Example 12. In Fig. 2, the transitions t_0 and t_2 executed at most once, i.e., $\mathcal{RB}(t_0) = \mathcal{RB}(t_2) = 1$. In Ex. 27, we will infer a runtime bound with $\mathcal{RB}(t_1) = \mathcal{RB}(t_4) = x$, $\mathcal{RB}(t_3) = \log_2(y + x \cdot x^x) + 2$, and $\mathcal{RB}(t_5) = x^2$. This results in a quadratic bound on the runtime complexity of the ρ-ITS.

Our approach performs a *modular* analysis, i.e., parts of the program are analyzed as standalone programs and the results are then lifted to contribute to the overall analysis. So to compute a runtime bound for a transition t, our approach considers all transitions and function calls $\vartheta \in \mathcal{T} \cup \mathcal{F}$ that can occur before t in evaluations, and it needs *size bounds* $\mathcal{SB}(\vartheta, v)$ to over-approximate the absolute values that the variables $v \in \mathcal{V}$ may have *after* these "previous" transitions and function calls ϑ. We call $\mathcal{RV} = (\mathcal{T} \cup \mathcal{F}) \times \mathcal{V}$ the set of *result variables*. Note that in contrast to runtime bounds (and to our earlier papers), we now also have to capture the effect of function calls $\mathcal{F}$ via size bounds.

Definition 13 (Size Bound). *A function* $\mathcal{SB} : \mathcal{RV} \to \mathcal{B}$ *is called a* size bound *if for all* $(\vartheta, v) \in \mathcal{RV}$, *all states* $\sigma_0 \in \Sigma$, *and all trees* $\mathbb{T}$ *with* $\mathbb{T}_{\sigma_0} \prec^* \mathbb{T}$ *containing a path* $(\ell_0, \sigma_0) \to \cdots \to_\vartheta (\_, \sigma)$ *with* $\sigma \neq \bot$, *we have* $|\sigma|(v) \leq [\![\mathcal{SB}(\vartheta, v)]\!]_{|\sigma_0|}$.

Example 14. In Fig. 2, $\mathcal{SB}(t_0, x) = x$ is a size bound, since the value of x after evaluating t_0 is bounded by the initial value of x. (Ex. 34 will show how to compute such size bounds.) Similarly, we have $\mathcal{SB}(t_2, x) = 1$ and $\mathcal{SB}(t_2, y) = y + x \cdot x^x$, see Ex. 40. The size bound $\mathcal{SB}(\rho_1, a) = x$ (see Ex. 34) expresses that the value of a after executing the function call ρ_1 is bounded by the initial value of x.

[2] More precisely, instead of $\log_k(b_1)$ we use the function $\lceil \log_k(\max\{1, b_1\}) \rceil$ to ensure that bounds are well defined, weakly monotonically increasing, and evaluate to natural numbers.

3 Modular Computation of Runtime Bounds

Now we introduce our modular approach for the computation of runtime bounds. To be precise, we infer runtime bounds for subprograms $\mathcal{T}'$ and then lift them to runtime bounds for the full program. For any non-empty $\mathcal{T}' \subseteq \mathcal{T}$, let $\mathcal{L}_{\mathcal{T}'} = \{\ell \in \mathcal{L} \mid (\ell, \_, \_, \_) \in \mathcal{T}'\}$ contain all start locations of transitions from $\mathcal{T}'$.

Recall that a global runtime bound $\mathcal{RB}(t)$ over-approximates how many times the transition $t \in \mathcal{T}$ may be evaluated when starting an evaluation of $\mathcal{T}$ from the initial location $\ell_0 \in \mathcal{L}$. To make this explicit, instead of $\mathcal{RB}(t)$ we could also write $\mathcal{RB}^{t,\mathcal{T}}(\ell_0)$. Now we are also interested in *local runtime bounds*, where one takes a subprogram $\mathcal{T}' \subseteq \mathcal{T}$ and a location ℓ from $\mathcal{L}_{\mathcal{T}'}$ into account. Then $\mathcal{RB}^{t,\mathcal{T}'}(\ell)$ over-approximates the number of applications of the transition $t \in \mathcal{T}'$ in any run of $\mathcal{T}'$ starting in the location $\ell \in \mathcal{L}_{\mathcal{T}'}$. To highlight that these runtime bounds may be "local" (i.e., that they can regard arbitrary subprograms $\mathcal{T}'$), we add the subscript "loc". So a *local runtime bound* is a function $\mathcal{RB}^{t,\mathcal{T}'}_{\text{loc}} : \mathcal{L}_{\mathcal{T}'} \to \mathcal{B}$. If $\mathcal{T}' = \mathcal{T}$, then $\mathcal{RB}^{t,\mathcal{T}}_{\text{loc}}(\ell_0)$ corresponds to a global runtime bound $\mathcal{RB}(t)$ of t. We define local runtime bounds as functions from locations to bounds (rather than from transitions, as with global runtime bounds) in order to simplify the presentation later on when inferring local runtime bounds from ranking functions, because ranking functions are also functions from locations to bounds, see Lemma 18.

So for any state $\sigma_0 \in \Sigma$, $[\![\mathcal{RB}^{t,\mathcal{T}'}_{\text{loc}}(\ell)]\!]_{|\sigma_0|}$ is an upper bound on the number of t-edges that can occur in any evaluation tree resulting from the initial single-node tree (ℓ, σ_0), if only transitions from $\mathcal{T}'$ and ε-steps are used, i.e., one only executes the subprogram $\mathcal{T}'$. However, local runtime bounds do not consider how often such a local evaluation of the subprogram $\mathcal{T}'$ is started or how large the variables are before starting such a local evaluation. If t and $\mathcal{T}'$ are clear from the context, we just write $\mathcal{RB}_{\text{loc}}$ instead of $\mathcal{RB}^{t,\mathcal{T}'}_{\text{loc}}$.

Definition 15 (Local Runtime Bound). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T}$ be a set of transitions and let $t \in \mathcal{T}'$. Then $\mathcal{RB}_{\text{loc}} : \mathcal{L}_{\mathcal{T}'} \to \mathcal{B}$ is a* local runtime bound *for t w.r.t. $\mathcal{T}'$ if for all $\sigma_0 \in \Sigma$, all $\ell \in \mathcal{L}_{\mathcal{T}'}$, and all trees $\mathbb{T}$ with $(\{(\ell, \sigma_0)\}, \varnothing) \prec^*_{\mathcal{T}' \cup \{\varepsilon\}} \mathbb{T}$, we have $|\mathbb{T}|_{\{t\}} \leq [\![\mathcal{RB}_{\text{loc}}(\ell)]\!]_{|\sigma_0|}$. To make this explicit, we also write $\mathcal{RB}^{t,\mathcal{T}'}_{\text{loc}}$.*

For readability, in contrast to our previous work [35, 36, 38], Def. 15 considers arbitrary initial states $\sigma_0 \in \Sigma$, but it could also be refined to only consider states $\sigma_0 \in \Sigma$ where (ℓ, σ_0) is reachable in the full program with all transitions $\mathcal{T}$. Note that the actual evaluation relation is considered both for local runtime bounds and also for local size bounds in Sect. 4, i.e., here of course one also takes the guards of transitions into account.

In Sect. 3.1 we introduce a novel form of *ρ-ranking functions* to derive local runtime bounds for ρ-ITSs with function calls automatically. Then, we show in Sect. 3.2 how global runtime bounds can be inferred from local runtime bounds.

3.1 Ranking Functions

Ranking functions are widely used to analyze termination or runtime complexity of programs (e.g., [7–9, 22, 26, 46]). The idea of ranking functions is to construct

a well-founded relation by mapping program configurations to numbers. Typically, SMT solvers are used to find such a suitable mapping automatically. We first consider programs without *recursive* function calls and recapitulate classical ranking functions in our setting of ρ-ITSs. Afterwards, we extend ranking functions to a novel form of *function call ranking functions*, which allow us to derive local runtime bounds for ρ-ITSs with arbitrary (possibly recursive) function calls.

Recall that $\mathrm{fun}(t)$ denotes the set of function calls in the transition t and $\mathrm{trans}(\rho)$ is the set of all transitions with the function call ρ. For $\mathcal{T}' \subseteq \mathcal{T}$, let $\mathrm{fun}(\mathcal{T}') = \bigcup_{t \in \mathcal{T}'} \mathrm{fun}(t)$ denote the set of all function calls $\rho \in \mathrm{fun}(t)$ for transitions $t \in \mathcal{T}'$. Moreover, $\mathrm{fun}(\mathcal{L}_{\mathcal{T}'})$ is the set of all function calls $\ell(\_)$ with $\ell \in \mathcal{L}_{\mathcal{T}'}$, i.e., all function calls that lead into the subprogram $\mathcal{T}'$. We also refer to $\mathrm{fun}(\mathcal{T}') \cap \mathrm{fun}(\mathcal{L}_{\mathcal{T}'})$ as the set of *recursive* function calls of the subprogram $\mathcal{T}'$ and to $\mathcal{T}' \cap \mathrm{trans}(\mathrm{fun}(\mathcal{L}_{\mathcal{T}'}))$ as the set of *recursive* transitions of the subprogram $\mathcal{T}'$.

Classical Ranking Function: Before we introduce our novel class of ranking functions for ρ-ITSs, we recapitulate classical ranking functions and adapt them to the setting of ρ-ITSs. As mentioned, we first consider subprograms $\mathcal{T}'$ which do not contain recursive function calls for jumps "within" $\mathcal{T}'$, i.e., where $\mathrm{fun}(\mathcal{T}') \cap \mathrm{fun}(\mathcal{L}_{\mathcal{T}'}) = \varnothing$. A function $r_{\mathrm{d}} : \mathcal{L} \to \mathbb{Z}[\mathcal{V}]$ is a classical *ranking function* for the ***d**ecreasing* transition t w.r.t. the subprogram $\mathcal{T}'$, if every evaluation step with t decreases the value of r_{d}, where r_{d}'s value is positive, and evaluation steps with the (other) transitions of $\mathcal{T}'$ do not increase the value of r_{d}. Then for any $\mathcal{T}'$-evaluation tree, $r_{\mathrm{d}}(\ell)$ is a bound on the number of edges labeled with the decreasing transition t.

We also use the relations $\to_t$ and $\to_\rho$ without referring to an actual evaluation tree. Thus, we say that $(\ell, \sigma) \to_\vartheta (\ell', \sigma')$ holds for some $\vartheta \in \mathcal{T} \cup \mathcal{F}$ if there exists an evaluation $\mathbb{T}_{\sigma_0} \prec^*_{\mathcal{T} \cup \{\varepsilon\}} \mathbb{T}$ for some initial state $\sigma_0 \in \Sigma$ such that $\mathbb{T}$ contains an edge $(\ell, \sigma) \to_\vartheta (\ell', \sigma')$. In the following definition, we extend the evaluation of an arithmetic expression e to the "undefined" state $\bot$ by defining $[\![e]\!]_\bot = 0$.

Definition 16 (Ranking Function). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T}$ with $\mathrm{fun}(\mathcal{T}') \cap \mathrm{fun}(\mathcal{L}_{\mathcal{T}'}) = \varnothing$, and let $t \in \mathcal{T}'$. Then $r_{\mathrm{d}} : \mathcal{L} \to \mathbb{Z}[\mathcal{V}]$ is a* ranking function *(RF) for the transition t w.r.t. $\mathcal{T}'$ if for all evaluation steps $(\ell, \sigma) \to_{t'} (\ell', \sigma')$, we have:*

(a) *if $t' \in \mathcal{T}'$, then* $[\![r_{\mathrm{d}}(\ell)]\!]_\sigma \geq [\![r_{\mathrm{d}}(\ell')]\!]_{\sigma'}$
(b) *if $t' = t$, then* $[\![r_{\mathrm{d}}(\ell)]\!]_\sigma > [\![r_{\mathrm{d}}(\ell')]\!]_{\sigma'}$ *and* $[\![r_{\mathrm{d}}(\ell)]\!]_\sigma > 0$

So Def. 16 does not impose any requirements on how the value of r_{d} changes when following edges labeled with function calls. The reason is that due to the requirement $\mathrm{fun}(\mathcal{T}') \cap \mathrm{fun}(\mathcal{L}_{\mathcal{T}'}) = \varnothing$, all function calls of $\mathcal{T}'$ lead outside this subprogram and thus, they are irrelevant for local runtime bounds w.r.t. $\mathcal{T}'$. Note that if t''s update η contains function calls, then in general it is not decidable whether $(\ell, \sigma) \to_{t'} (\ell', \sigma')$ holds. Thus, to over-approximate $\to_{t'}$ in our automation, we consider a modified update η' where all function calls are replaced by fresh "non-deterministic" values. To ease the automation of our approach, in practice we restrict ourselves to linear polynomial ranking functions and use the SMT solver Z3 [43] to infer ranking functions automatically. Our

modular approach allows us to consider program parts separately, such that using linear ranking functions usually suffices even if the overall program has non-linear runtime. Note that when lifting local to global runtime bounds (in Sect. 3.2), we use size bounds that indeed take the results of "previous" function calls into account (see Sect. 4 for the computation of size bounds).

Example 17. We compute a (classical) ranking function for the transition t_1 w.r.t. the subprogram $\mathcal{T}' = \{t_1\}$ from Fig. 2: We have $\mathrm{fun}(\mathcal{T}') = \{\rho_1\}$, but the location f_1 of ρ_1 is not in $\mathcal{L}_{\mathcal{T}'} = \{\ell_1\}$. Thus, $\mathrm{fun}(\mathcal{T}') \cap \mathrm{fun}(\mathcal{L}_{\mathcal{T}'}) = \{\rho_1\} \cap \varnothing = \varnothing$, i.e., $\mathcal{T}'$ does not have any recursive function calls, but all function calls of $\mathcal{T}'$ *leave* the subprogram $\mathcal{T}'$. An RF for t_1 w.r.t. $\mathcal{T}' = \{t_1\}$ is $r_{\mathrm{d}}(\ell_1) = x$. The RF can always map all remaining locations outside the subprogram $\mathcal{T}'$ to 0.

Lemma 18 shows how to obtain a local runtime bound from a (classical) ranking function. We use $[\![\cdot]\!]$ to transform a polynomial into a (weakly monotonically increasing) bound from $\mathcal{B}$ by taking the absolute values of the coefficients, e.g., for the polynomial $x - y$ we obtain the bound $[\![x - y]\!] = x + y$.

Lemma 18 (Local Runtime Bounds by RFs). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T}$ where $\mathrm{fun}(\mathcal{T}') \cap \mathrm{fun}(\mathcal{L}_{\mathcal{T}'}) = \varnothing$, and let $t \in \mathcal{T}'$. Moreover, let r_{d} be an RF for t w.r.t. $\mathcal{T}'$. Then $\mathcal{RB}_{\mathrm{loc}} : \mathcal{L}_{\mathcal{T}'} \to \mathcal{B}$ is local runtime bound for t w.r.t. $\mathcal{T}'$, where for all $\ell \in \mathcal{L}_{\mathcal{T}'}$, we define $\mathcal{RB}_{\mathrm{loc}}(\ell) = [\![r_{\mathrm{d}}(\ell)]\!]$.*

Example 19. With the ranking function of Ex. 17, Lemma 18 yields the local runtime bound $\mathcal{RB}_{\mathrm{loc}}^{t_1,\{t_1\}}(\ell_1) = [\![r_{\mathrm{d}}(\ell_1)]\!] = x$.

Function Call Ranking Functions: Now we consider arbitrary ρ-ITSs with possibly recursive function calls. To obtain local runtime bounds for such ρ-ITSs, we introduce the novel concept of "*function call ranking functions*" (ρ-RFs). In contrast to *classical* ranking functions, ρ-RFs also contain an explicit bound on the number of *function* calls. More precisely, a ρ-RF $\langle r_{\mathrm{d}}, r_{\mathrm{tf}}, r_{\mathrm{f}} \rangle$ combines a classical ranking function r_{d} with two additional ranking functions $r_{\mathrm{tf}} : \mathcal{L} \to \mathbb{Z}[\mathcal{V}]$ and $r_{\mathrm{f}} : \mathcal{L} \to \mathbb{Z}[\mathcal{V}]$. While r_{d} yields a bound for a **d**ecreasing transition t w.r.t. $\mathcal{T}'$, r_{tf} yields a bound for the **t**ransitions from $\mathcal{T}'$ with recursive **f**unction calls and r_{f} yields a bound on these recursive **f**unction calls themselves. We distinguish these three ranking functions because they have different influences on the overall bound for the possible number of applications of t in the subprogram $\mathcal{T}'$. For example, r_{f} has an exponential influence on the runtime while the other two ranking functions only have a polynomial impact, see Thm. 22.

More precisely, for a set of function calls $\mathcal{F}'$, let $\mathrm{trans}(\mathcal{F}') = \bigcup_{\rho \in \mathcal{F}'} \mathrm{trans}(\rho)$ be the set of all transitions $t \in \mathrm{trans}(\rho)$ for function calls $\rho \in \mathcal{F}'$. Recall that $\mathrm{fun}(\mathcal{L}_{\mathcal{T}'})$ is the set of all function calls $\ell(\_)$ with a location $\ell \in \mathcal{L}_{\mathcal{T}'}$. Thus, $\mathrm{trans}(\mathrm{fun}(\mathcal{L}_{\mathcal{T}'}))$ is the set of all transitions which contain a function call $\ell(\_)$ with $\ell \in \mathcal{L}_{\mathcal{T}'}$. Then for any location $\ell \in \mathcal{L}$ and any $\mathcal{T}'$-evaluation tree, $r_{\mathrm{tf}}(\ell)$ yields a bound on the number of edges labeled with transitions from $\mathcal{T}' \cap \mathrm{trans}(\mathrm{fun}(\mathcal{L}_{\mathcal{T}'}))$ on paths starting with $(\ell, \_)$, where these paths may contain both steps with

transitions and steps with function calls. So $\mathcal{T}' \cap \text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'}))$ are recursive **t**ransitions from $\mathcal{T}'$ whose **f**unction call jumps into $\mathcal{T}'$ again.

Furthermore, for any location $\ell \in \mathcal{L}$ and any $\mathcal{T}'$-evaluation tree, $r_{\mathrm{f}}(\ell)$ is a bound on the number of ρ-edges with function calls that jump into $\mathcal{T}'$ again on paths starting with $(\ell, \_)$. So there are recursive function calls from $\text{fun}(\mathcal{T}') \cap \text{fun}(\mathcal{L}_{\mathcal{T}'})$. In Ex. 7, these function calls correspond to vertical edges, whereas steps with transitions correspond to horizontal edges.

As for classical RFs, Def. 20(a) requires that r_{d}, r_{tf}, and r_{f} do not increase for edges of a $\mathcal{T}'$-evaluation tree. Furthermore, to ensure the previously mentioned properties, as for classical RFs, (b) requires that r_{d} is decreasing and positive for edges labeled with t (if the decreasing transition t is not recursive, i.e., it does not contain any recursive function calls). Condition (c) imposes these requirements for r_{tf} and edges labeled with recursive transitions from $\mathcal{T}' \cap \text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'}))$. In particular, this is also required for t if t is a recursive transition and thus, it was not already handled in (b). Finally, (d) requires these properties for r_{f} and edges labeled with recursive function calls from $\text{fun}(\mathcal{T}') \cap \text{fun}(\mathcal{L}_{\mathcal{T}'})$.

Definition 20 (Function Call Ranking Function). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T}$ and let $t \in \mathcal{T}'$. Then $\langle r_{\mathrm{d}}, r_{\mathrm{tf}}, r_{\mathrm{f}} \rangle$ with $r_{\mathrm{d}}, r_{\mathrm{tf}}, r_{\mathrm{f}} : \mathcal{L} \to \mathbb{Z}[\mathcal{V}]$ is a* function call ranking function *(ρ-RF) for t w.r.t. $\mathcal{T}'$ if for all evaluation steps $(\ell, \sigma) \to_\vartheta (\ell', \sigma')$ with $\vartheta \in \mathcal{T}'$ or $\vartheta \in \text{fun}(\mathcal{T}') \cap \text{fun}(\mathcal{L}_{\mathcal{T}'})$, we have:*

(a)	*for all* $i \in \{\mathrm{d}, \mathrm{tf}, \mathrm{f}\}$	$[\![r_i(\ell)]\!]_\sigma \geq [\![r_i(\ell')]\!]_{\sigma'}$	
(b)	*if* $\vartheta = t$ *and* $t \notin \text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'}))$, *then*		
		$[\![r_{\mathrm{d}}(\ell)]\!]_\sigma > [\![r_{\mathrm{d}}(\ell')]\!]_{\sigma'}$ *and*	$[\![r_{\mathrm{d}}(\ell)]\!]_\sigma > 0$
(c)	*if* $\vartheta \in \mathcal{T}' \cap \text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'}))$, *then*	$[\![r_{\mathrm{tf}}(\ell)]\!]_\sigma > [\![r_{\mathrm{tf}}(\ell')]\!]_{\sigma'}$ *and*	$[\![r_{\mathrm{tf}}(\ell)]\!]_\sigma > 0$
(d)	*if* $\vartheta \in \text{fun}(\mathcal{T}') \cap \text{fun}(\mathcal{L}_{\mathcal{T}'})$, *then*	$[\![r_{\mathrm{f}}(\ell)]\!]_\sigma > [\![r_{\mathrm{f}}(\ell')]\!]_{\sigma'}$ *and*	$[\![r_{\mathrm{f}}(\ell)]\!]_\sigma > 0$

Note that r_{d} can be set to 0 for all locations if $t \in \text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'}))$.

Example 21. We now compute ρ-RFs for t_1 w.r.t. $\{t_1\}$ and for t_5 w.r.t. $\{t_4, t_5\}$ in the program from Fig. 2.

- First, we consider $\mathcal{T}' = \{t_1\}$. As in Ex. 17, we have $\text{fun}(\mathcal{T}') \cap \text{fun}(\mathcal{L}_{\mathcal{T}'}) = \varnothing$ and $\mathcal{T}' \cap \text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'})) = \varnothing$, since there is no function call with the location $\mathcal{L}_{\mathcal{T}'} = \{\ell_1\}$, i.e., $\text{fun}(\mathcal{L}_{\mathcal{T}'}) = \varnothing$. A ρ-RF for t_1 is $r_{\mathrm{tf}}(\ell_1) = r_{\mathrm{f}}(\ell_1) = 0$ and $r_{\mathrm{d}}(\ell_1) = x$. Again, locations outside $\mathcal{T}'$ can always be mapped to 0. So, here the ρ-RF corresponds to the classical ranking function from Ex. 17.
- For the subprogram $\mathcal{T}' = \{t_4, t_5\}$ we have $\text{fun}(\mathcal{T}') \cap \text{fun}(\mathcal{L}_{\mathcal{T}'}) = \{\rho_2\}$ and $\mathcal{T}' \cap \text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'})) = \{t_5\}$. Thus, a ρ-RF for t_5 is $r_{\mathrm{d}}(f_1) = r_{\mathrm{d}}(f_2) = 0$ (as $t_5 \in \text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'}))$), $r_{\mathrm{tf}}(f_1) = 1$ and $r_{\mathrm{tf}}(f_2) = 0$ (since t_5 was not already handled by r_{d}, t_5 must be decreasing for r_{tf}), and $r_{\mathrm{f}}(f_1) = a$, $r_{\mathrm{f}}(f_2) = 0$ (to make ρ_2 decreasing, since $(f_1, \sigma) \to_{\rho_2} (f_1, \sigma')$ implies $[\![r_{\mathrm{f}}(f_1)]\!]_\sigma = \sigma(a) > \sigma(a-1) = \sigma'(a) = [\![r_{\mathrm{f}}(f_1)]\!]_{\sigma'}$).

Now we show how to construct a local runtime bound from a ρ-RF. To simplify the presentation, here we restrict ourselves to transitions which have at

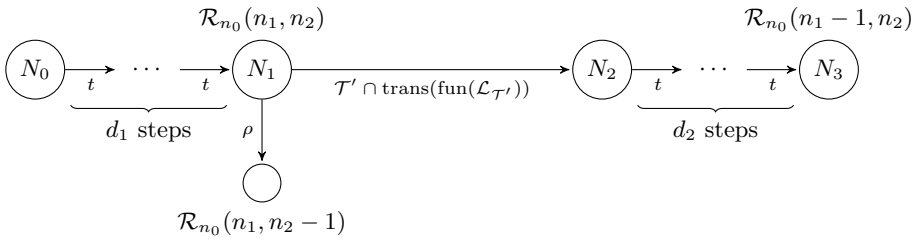

Fig. 4: Illustration of $\mathcal{R}_{n_0}(n_1, n_2)$

most one function call in their update and refer to [40, App. B] for the general case which handles transitions with arbitrary many function calls.

For a local runtime bound, we have to over-approximate how many edges labeled with t can occur in a $\mathcal{T}'$-evaluation tree starting with a configuration of the form $(\ell, \_)$. As mentioned, the ranking functions r_{d}, r_{tf}, and r_{f} influence the local runtime bound in different ways: If every path has at most n_0 edges labeled with the transition t (if t is not a recursive transition from $\text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'}))$), n_1 edges labeled with the recursive transitions from $\mathcal{T}' \cap \text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'}))$, and n_2 edges labeled with recursive function calls, then $\mathcal{R}_{n_0}(n_1, n_2)$ over-approximates the number of t-edges in any $\mathcal{T}'$-evaluation tree, where $\mathcal{R}_{n_0}(n_1, n_2)$ is defined via the following recurrence:

$$\mathcal{R}_{n_0}(n_1, n_2) = \begin{cases} n_0, & \text{if } n_1 = 0 \text{ or } n_2 = 0 \\ 1 + n_0 + \mathcal{R}_{n_0}(n_1 - 1, n_2) + \mathcal{R}_{n_0}(n_1, n_2 - 1), & \text{otherwise} \end{cases}$$

This is shown by induction on $n_1 + n_2$: If $n_1 = 0$ or $n_2 = 0$, then there is no recursive function call and thus, there can be at most n_0 edges labeled with the decreasing transition t. The induction step is illustrated in Fig. 4. First consider the case where t is not a recursive transition. Then the path from the root to the first node N_1 where a function is called recursively uses at most $d_1 \leq n_0$ edges labeled with t. Due to our restriction on the number of function calls in the update, N_1 has at most one outgoing edge labeled with a recursive function call ρ and one outgoing edge to a node N_2 labeled with the corresponding recursive transition from $\mathcal{T}' \cap \text{trans}(\text{fun}(\mathcal{L}_{\mathcal{T}'}))$ whose update contains ρ. The function call ρ leads to a subtree with at most $\mathcal{R}_{n_0}(n_1, n_2 - 1)$ many t-edges by the induction hypothesis. The path from N_2 to the next node N_3 where a function might be called uses at most d_2 edges labeled with t, where $d_1 + d_2 \leq n_0$. Finally, the subtree starting in the node N_3 has at most $\mathcal{R}_{n_0}(n_1 - 1, n_2)$ many t-edges by the induction hypothesis. Thus, the number of t-edges in the full tree is at most

$$\begin{aligned} |\mathbb{T}|_{\{t\}} &\leq d_1 + \mathcal{R}_{n_0}(n_1, n_2 - 1) + d_2 + \mathcal{R}_{n_0}(n_1 - 1, n_2) \\ &\leq n_0 + \mathcal{R}_{n_0}(n_1 - 1, n_2) + \mathcal{R}_{n_0}(n_1, n_2 - 1). \end{aligned}$$

If t is recursive, we have $d_1 = d_2 = n_0 = 0$. However, the step from N_1 to N_2 might be done with t. Thus, we obtain $|\mathbb{T}|_{\{t\}} \leq 1 + \mathcal{R}_{n_0}(n_1, n_2 - 1) + \mathcal{R}_{n_0}(n_1 - 1, n_2)$. So in both cases, we have $|\mathbb{T}|_{\{t\}} \leq 1 + n_0 + \mathcal{R}_{n_0}(n_1 - 1, n_2) + \mathcal{R}_{n_0}(n_1, n_2 - 1)$.

As shown in [40, App. A], $n_0 + n_2 \cdot (1 + 2 \cdot n_0) \cdot n_1^{n_2}$ is an over-approximating closed form solution of $\mathcal{R}_{n_0}(n_1, n_2)$. Hence, instantiating this closed form with the ranking functions yields the desired local runtime bound. As mentioned, a generalized version of this theorem for transitions with arbitrary many function calls in their updates can be found in [40, App. B].[3]

Theorem 22 (Local Runtime Bounds by ρ-RFs). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T}$ such that $|\operatorname{fun}(\eta)| \leq 1$ holds for every update η of the transitions in $\mathcal{T}'$. Moreover, let $t \in \mathcal{T}'$ and $\langle r_{\mathrm{d}}, r_{\mathrm{tf}}, r_{\mathrm{f}} \rangle$ be a ρ-RF for t w.r.t. $\mathcal{T}'$. Then $\mathcal{RB}_{\mathrm{loc}} : \mathcal{L}_{\mathcal{T}'} \to \mathcal{B}$ is local runtime bound for t w.r.t. $\mathcal{T}'$, where for all $\ell \in \mathcal{L}_{\mathcal{T}'}$, we define $\mathcal{RB}_{\mathrm{loc}}(\ell)$ as:*

$$[\![r_{\mathrm{d}}(\ell)]\!] \;+\; [\![r_{\mathrm{f}}(\ell)]\!] \;\cdot\; (1 + 2 \cdot [\![r_{\mathrm{d}}(\ell)]\!]) \;\cdot\; [\![r_{\mathrm{tf}}(\ell)]\!]^{[\![r_{\mathrm{f}}(\ell)]\!]}$$

Example 23. With the first ρ-RF of Ex. 21, Thm. 22 yields the local runtime bound $\mathcal{RB}_{\mathrm{loc}}^{t_1,\{t_1\}}(\ell_1) = [\![r_{\mathrm{d}}(\ell_1)]\!] = x$ (since $r_{\mathrm{f}}(\ell_1) = 0$). With the second ρ-RF of Ex. 21, we obtain $\mathcal{RB}_{\mathrm{loc}}^{t_5,\{t_4,t_5\}}(f_1) = [\![r_{\mathrm{f}}(f_1)]\!] \cdot [\![r_{\mathrm{tf}}(f_1)]\!]^{[\![r_{\mathrm{f}}(f_1)]\!]} = a$, since $r_{\mathrm{d}}(f_1) = 0$, $r_{\mathrm{tf}}(f_1) = 1$, and $r_{\mathrm{f}}(f_1) = a$.

3.2 Lifting Local Runtime Bounds to Global Runtime Bounds

To lift local to global runtime bounds, we consider those transitions and function calls which start an evaluation of the subprogram $\mathcal{T}'$. Remember that $\operatorname{fun}(\mathcal{L}_{\mathcal{T}'})$ denotes the set of all function calls $\ell(\_) \in \mathcal{F}$ with $\ell \in \mathcal{L}_{\mathcal{T}'}$, where $\mathcal{L}_{\mathcal{T}'}$ are the start locations of the transitions in the subprogram $\mathcal{T}'$.

Definition 24 (Entry Transitions and Function Calls). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T}$ be a non-empty set of transitions. Then $\mathcal{ET}_{\mathcal{T}'} = \{r \in \mathcal{T} \setminus \mathcal{T}' \mid \exists \ell \in \mathcal{L}_{\mathcal{T}'}. r = (\_, \_, \_, \ell)\}$ is the set of* direct entry transitions *and $\mathcal{EF}_{\mathcal{T}'} = \{r \in \mathcal{T} \setminus \mathcal{T}' \mid \operatorname{fun}(r) \cap \operatorname{fun}(\mathcal{L}_{\mathcal{T}'}) \neq \varnothing\}$ is the set of* entry (function) calling transitions *for $\mathcal{T}'$.*

Example 25. For Fig. 2 and the subprogram $\mathcal{T}' = \{t_1\}$ with the locations $\mathcal{L}_{\{t_1\}} = \{\ell_1\}$, we have the entry transitions $\mathcal{ET}_{\{t_1\}} = \{t_0\}$. Moreover, considering $\mathcal{T}' = \{t_4, t_5\}$ yields $\mathcal{L}_{\{t_4,t_5\}} = \{f_1\}$ and $\mathcal{EF}_{\{t_4,t_5\}} = \{t_1\}$.

Thm. 26 allows us to lift arbitrary local runtime bounds of a subprogram $\mathcal{T}'$ (e.g., local runtime bounds by ρ-RFs) to global runtime bounds for the full program with all transitions $\mathcal{T}$. To this end, for every entry transition $r \in \mathcal{ET}_{\mathcal{T}'} \cup \mathcal{EF}_{\mathcal{T}'}$, we consider $\mathcal{RB}(r)$ to over-approximate how often a local run of $\mathcal{T}'$ is started. In contrast to our previous work [22, 35, 38], we also have to consider *entry calling transitions* $r \in \mathcal{EF}_{\mathcal{T}'}$. Furthermore, we have to consider the size of the program variables after entering the subprogram by r or by a function call ρ in r. Hence, we replace every program variable $v \in \mathcal{V}$ by its size bound $\mathcal{SB}(\vartheta, v)$

[3] An alternative approach would be to add suitable transitions in order to transform any program into a program with at most one function call per transition. When using a ρ-RF on the transformed program, Thm. 22 would result in a similar local runtime bound as when using the ρ-RF on the original program and computing the local runtime bound via the generalized version of Thm. 22 in [40, App. B].

for $\vartheta = r$ or $\vartheta = \rho$, respectively. This is denoted by "$[v/\mathcal{SB}(\vartheta, v) \mid v \in \mathcal{V}]$". In Sect. 4, we show how to infer size bounds automatically. In the following, let $\ell_r \in \mathcal{L}$ denote the target location of a transition r and let $\ell_\rho \in \mathcal{L}$ denote the location of a function call ρ, i.e., if $\rho = \ell(\zeta)$ then $\ell_\rho = \ell$. Here, it is important to restrict ourselves to subprograms $\mathcal{T}'$ without *initial* transitions $(\ell_0, \_, \_, \_)$ that start in the initial location ℓ_0. Let $\mathcal{T}_0$ denote the set of all initial transitions. The reason is that initial transitions do not have predecessor (entry) transitions. In fact, initial transitions always have the global runtime bound 1 by construction since according to our definition of ρ-ITSs in Def. 3, the initial location ℓ_0 cannot be reached by any transition.

Theorem 26 (Lifting Local Runtime Bounds). *Let $\mathcal{RB}$ be a (global) runtime bound, $\mathcal{SB}$ be a size bound, $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$, and $t \in \mathcal{T}'$. Moreover, let $\mathcal{RB}_{\text{loc}}^{t,\mathcal{T}'}$ be a local runtime bound for the transition t w.r.t. $\mathcal{T}'$. Then $\mathcal{RB}'$ is also a global runtime bound, where $\mathcal{RB}'(t') = \mathcal{RB}(t')$ for all $t' \neq t$ and*

$$\begin{aligned}\mathcal{RB}'(t) \;=\; & \sum_{r \in \mathcal{ET}_{\mathcal{T}'}} \mathcal{RB}(r) \cdot \mathcal{RB}_{\text{loc}}^{t,\mathcal{T}'}(\ell_r)\,[v/\mathcal{SB}(r, v) \mid v \in \mathcal{V}] \\ + \sum_{r \in \mathcal{EF}_{\mathcal{T}'}} & \sum_{\rho \in \text{fun}(r) \cap \text{fun}(\mathcal{L}_{\mathcal{T}'})} \mathcal{RB}(r) \cdot \mathcal{RB}_{\text{loc}}^{t,\mathcal{T}'}(\ell_\rho)\,[v/\mathcal{SB}(\rho, v) \mid v \in \mathcal{V}].\end{aligned}$$

Example 27. We now compute the remaining global runtime bounds for the ρ-ITS of Fig. 2, see Ex. 12. For the transitions t_1 and t_5, we had inferred the local runtime bounds $\mathcal{RB}_{\text{loc}}^{t_1,\{t_1\}}(\ell_1) = x$ and $\mathcal{RB}_{\text{loc}}^{t_5,\{t_4,t_5\}}(f_1) = a$ in Ex. 23.

- Thus, we obtain $\mathcal{RB}(t_1) = \mathcal{RB}(t_0) \cdot \mathcal{RB}_{\text{loc}}^{t_1,\{t_1\}}(\ell_1)\,[x/\mathcal{SB}(t_0, x)] = x$ (with $\mathcal{RB}(t_0) = 1$ and $\mathcal{SB}(t_0, x) = x$) by Thm. 26.
- Furthermore, we have $\mathcal{RB}(t_5) = \mathcal{RB}(t_1) \cdot \mathcal{RB}_{\text{loc}}^{t_5,\{t_4,t_5\}}(f_1)\,[a/\mathcal{SB}(\rho_1, a)] = x^2$ (with $\mathcal{RB}(t_1) = x$ and $\mathcal{SB}(\rho_1, a) = x$) by Thm. 26.
- Moreover, $\mathcal{RB}_{\text{loc}}^{t_4,\{t_4,t_5\}}(f_1) = 1$ is a local runtime bound since t_4 can only be executed once in the subprogram $\{t_4, t_5\}$. Here, we get $\mathcal{RB}(t_4) = \mathcal{RB}(t_1) \cdot \mathcal{RB}_{\text{loc}}^{t_4,\{t_4,t_5\}}(f_1) = x$.
- Finally, $\mathcal{RB}_{\text{loc}}^{t_3,\{t_3\}}(\ell_2) = \log_2(y) + 2$ is also a local runtime bound, which cannot be inferred by linear ranking functions but by our technique based on *twn*-loops [24, 25, 35, 36, 38] (see [40, App. C] for the detailed construction). Lifting this local bound by Thm. 26 yields the global bound $\mathcal{RB}(t_3) = \mathcal{RB}(t_2) \cdot \mathcal{RB}_{\text{loc}}^{t_3,\{t_3\}}(\ell_2)\,[y/\mathcal{SB}(t_2, y)] = \log_2(y + x \cdot x^x) + 2$ (with $\mathcal{RB}(t_2) = 1$ and $\mathcal{SB}(t_2, y) = y + x \cdot x^x$).

Thus, our modular approach allows us to consider individual subprograms separately, to use different techniques to compute their local bounds, and to combine these local bounds into a global bound afterwards.

4 Modular Computation of Size Bounds

We now introduce our modular approach to compute size bounds. To this end, we extend the technique of [10] to handle ITSs with function calls. For every result

variable $\langle \vartheta, v \rangle \in \mathcal{RV} = (\mathcal{T} \cup \mathcal{F}) \times \mathcal{V}$, we define a *local size bound* $\mathcal{SB}_{\text{loc}}(\vartheta, v) \in \mathbb{N}[\mathcal{V} \cup \mathcal{F}]$. So $\mathcal{SB}_{\text{loc}}(\vartheta, v)$ is a polynomial over the program variables and function calls (which are treated like variables). When instantiating every function call ρ in $\mathcal{SB}_{\text{loc}}(\vartheta, v)$ by the size of its result, then $\mathcal{SB}_{\text{loc}}(\vartheta, v)$ must be a bound on the size of v after a *single evaluation* step with ϑ (where "size" again refers to the absolute value). Thus, local size bounds are more restrictive than local runtime bounds, which consider arbitrary runs within the subprogram rather than being limited to a single evaluation step. Here, the result of a function call ρ can be obtained as soon as the evaluation of the call ends in a configuration (ℓ_ρ, σ_ρ) with $\ell_\rho \in \Omega$. Then the result of the call has the size $|\sigma_\rho|(v_{\ell_\rho})$.

Definition 28 (Local Size Bound). $\mathcal{SB}_{\text{loc}} : \mathcal{RV} \to \mathbb{N}[\mathcal{V} \cup \mathcal{F}]$ *is a* local size bound *if for all* $\langle \vartheta, v \rangle \in \mathcal{RV}$*, all evaluations* $(\ell', \sigma') \to_\vartheta (\ell, \sigma)$ *with* $\sigma \neq \bot$*, and all evaluations* $(\ell', \sigma') \to_\rho \circ \to^* (\ell_\rho, \sigma_\rho)$ *starting with some* $\rho \in \mathcal{F}$ *such that* $\ell_\rho \in \Omega$ *and* $\sigma_\rho \neq \bot$*, we have* $|\sigma|(v) \leq [\![\, \mathcal{SB}_{\text{loc}}(\vartheta, v)\,[\rho \,/\, |\sigma_\rho|(v_{\ell_\rho}) \mid \rho \in \mathcal{F}] \,]\!]_{|\sigma'|}$.

For every result variable $\langle t, v \rangle \in \mathcal{T} \times \mathcal{V}$ with $t = (\_, \_, \eta, \_)$, in practice we essentially use $\mathcal{SB}_{\text{loc}}(t, v) = [\![\eta(v)]\!]$, e.g., $\mathcal{SB}_{\text{loc}}(t_1, y) = y + \rho_1$ and $\mathcal{SB}_{\text{loc}}(t_1, x) = [\![x - 1]\!] = x + 1$ for the program from Fig. 2. However, due to the guard $x > 0$ of the transition t_1, here we can obtain the more precise local size bound $\mathcal{SB}_{\text{loc}}(t_1, x) = x$. Similarly, we essentially use $\mathcal{SB}_{\text{loc}}(\rho, v) = [\![\zeta(v)]\!]$ for every result variable $\langle \rho, v \rangle \in \mathcal{F} \times \mathcal{V}$ with $\rho = \ell(\zeta)$. So for Fig. 2, we would obtain $\mathcal{SB}_{\text{loc}}(\rho_2, a) = [\![a - 1]\!] = a + 1$. However, due to the guard $a > 0$ of the transition t_5 whose update contains the function call ρ_2, we can again obtain the more precise bound $\mathcal{SB}_{\text{loc}}(\rho_2, a) = a$.

Next we construct a *result variable graph* (RVG) which represents the influence of result variables on each other. Here, a node $\langle t, v \rangle$ or $\langle \rho, v \rangle$ represents the value of v after evaluating the transition t or applying the update ζ for the function call $\rho = \ell(\zeta)$, respectively. For any polynomial $p \in \mathbb{N}[\mathcal{V} \cup \mathcal{F}]$, let $\text{act}(p) \subseteq \mathcal{V} \cup \mathcal{F}$ denote the set of *active arguments* of the polynomial p, i.e., $x \in \text{act}(p)$ iff $x \in \mathcal{V} \cup \mathcal{F}$ is a program variable or a function call which occurs in p. Furthermore, for $\vartheta \in \mathcal{T} \cup \mathcal{F}$ let $\text{pre}(\vartheta)$ denote transitions or function calls that directly precede ϑ, i.e., $\vartheta' \in \text{pre}(\vartheta)$ iff there exists an evaluation tree which contains the path $\to_{\vartheta'} \circ \to_\vartheta$. Then the RVG has all result variables $\mathcal{RV}$ as its nodes, and it has an RV-edge from $\langle \vartheta', v' \rangle$ to $\langle \vartheta, v \rangle$ whenever $\vartheta' \in \text{pre}(\vartheta)$ and $v' \in \text{act}(\mathcal{SB}_{\text{loc}}(\vartheta, v))$. In practice, we use efficiently computable over-approximations for $\text{pre}(\cdot)$.

Definition 29 (Result Variable Graph (RV-Edges)). *An* RVG *has the nodes* $\mathcal{RV}$ *and the RV-edges* $\{(\langle \vartheta', v' \rangle, \langle \vartheta, v \rangle) \mid \vartheta' \in \text{pre}(\vartheta), v' \in \text{act}(\mathcal{SB}_{\text{loc}}(\vartheta, v))\}$.

Example 30. Fig. 5 depicts a part of the RVG for the program of Fig. 2. In the figure, the black non-dashed edges are RV-edges. For instance, there is an edge $\langle t_1, x \rangle \to \langle \rho_1, a \rangle$ as the value of x after transition t_1 influences the value of a after the function call ρ_1. More precisely, $t_1 \in \text{pre}(\rho_1) = \{t_0, t_1\}$ and $x \in \text{act}(\mathcal{SB}_{\text{loc}}(\rho_1, a)) = \text{act}(x) = \{x\}$. Note that we do not have an edge from $\langle \rho_1, a \rangle$

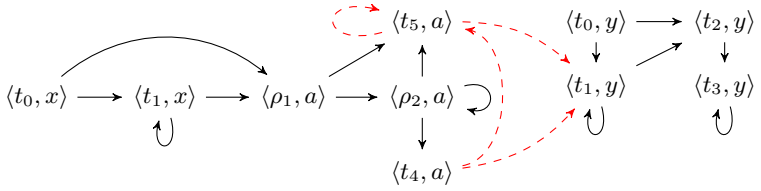

Fig. 5: Part of the RVG with RV-Edges and Ω-Edges for Fig. 2

to $\langle t_4, a\rangle$ as $\mathcal{SB}_{\text{loc}}(t_4, a) = 1$. While the full RVG has additional non-trivial SCCs,[4] we omitted them from Fig. 5 as they have no impact on the runtime.

So far, the values of function calls have been disregarded. We now address this issue by extending the RVG by a second set of edges. To this end, we say that $\langle t', v_{\ell'}\rangle$ is an *Ω-predecessor* of (t, ρ) for a transition t containing a function call ρ if the transition t' ends in a return location $\ell' \in \Omega$ which is reachable from a function call $\rho = \ell(\_) \in \text{fun}(t)$, i.e., if there is an evaluation $\to_\rho (\ell, \_) \to \cdots \to_{t'} (\ell', \_)$. This means that after executing t', the function call ρ is "finished" and its result is obtained in the return variable $v_{\ell'}$. Thus, the Ω-predecessor $\langle t', v_{\ell'}\rangle$ corresponds to the value of the function call ρ which we did not handle in the RV-edges. Let $\text{pre}^\Omega(t, \rho)$ denote the set of all Ω-predecessors of (t, ρ). Whenever $t = (\_, \_, \eta, \_)$ has the update η with a function call $\rho \in \text{fun}(\eta(v))$ and $\langle t', v_{\ell'}\rangle \in \text{pre}^\Omega(t, \rho)$, then there is an Ω-edge from $\langle t', v_{\ell'}\rangle$ to $\langle t, v\rangle$ in the RVG. As for $\text{pre}(\cdot)$, we use efficiently computable over-approximations for $\text{pre}^\Omega(\cdot, \cdot)$.

Definition 31 (Result Variable Graph (Ω-Edges)). *In addition to Def. 29, an* RVG *also has the Ω-edges* $\{(\langle t', v_{\ell'}\rangle, \langle t, v\rangle) \mid \langle t', v_{\ell'}\rangle \in \text{pre}^\Omega(t, \rho),\ t = (\_, \_, \eta, \_),\ \rho \in \text{fun}(\eta(v))\}$.

Example 32. Reconsider the RVG in Fig. 5 which depicts a part of the RVG for the program of Fig. 2. The RVG has four Ω-edges which are red-dashed in Fig. 5. We have $\text{pre}^\Omega(t_1, \rho_1) = \{\langle t_4, a\rangle, \langle t_5, a\rangle\}$, since both t_4 and t_5 end in the return location $f_2 \in \Omega$ whose return variable is $v_{f_2} = a$. Moreover, t_1's update of y contains the function call $\rho_1 = f_1(\zeta_1)$, and there are evaluations $\to_{\rho_1} (f_1, \_) \to_{t_4} (f_2, \_)$ and $\to_{\rho_1} (f_1, \_) \to_{t_5} (f_2, \_)$. Thus, there are Ω-edges from both $\langle t_4, a\rangle$ and $\langle t_5, a\rangle$ to $\langle t_1, y\rangle$ in the RVG. So Fig. 5 has the five non-trivial SCCs $\{\langle t_1, x\rangle\}$, $\{\langle \rho_2, a\rangle\}$, $\{\langle t_1, y\rangle\}$, $\{\langle t_3, y\rangle\}$, and $\{\langle t_5, a\rangle\}$, where the latter forms a cycle with an Ω-edge.

We already developed powerful techniques to lift local to global size bounds for ITSs without function calls in [10, 36, 38]. To extend these technique to ITSs *with* function calls, we now introduce a corresponding approach to obtain global

[4] As usual, a *strongly connected component* (SCC) is a maximal subgraph with a path from each node to every other node. An SCC is *trivial* if it consists of a single node without an edge to itself.

size bounds for different types of components of the RVG. More precisely, we consider the SCCs of the RVG in topological order and lift the local size bounds for the result variables of each SCC to global size bounds. First, we treat trivial SCCs of the RVG in Sect. 4.1. Here, local size bounds can be lifted by taking the global size bounds for its "predecessors" into account. Afterwards, we handle non-trivial SCCs in Sect. 4.2. To lift local to global size bounds in this case, in addition to the global size bounds of the "predecessors", one also has to consider (global) runtime bounds, since non-trivial SCCs may be traversed repeatedly.

4.1 Size Bounds for Trivial SCCs

We start with computing size bounds for *trivial* SCCs $\{\langle \vartheta, x\rangle\}$ in the RVG. To this end, we present two techniques (in Thm. 33 and 35). Thm. 33 considers the case where $\vartheta \in \mathcal{F}$ or $\vartheta \in \mathcal{T}$ with an update η such that $\mathrm{fun}(\eta(x)) = \varnothing$. The case $\mathrm{fun}(\eta(x)) \neq \varnothing$ is handled in Thm. 35. If ϑ is an initial transition, i.e., $\mathrm{pre}(\vartheta) = \varnothing$, then $\mathcal{SB}_{\mathrm{loc}}(\vartheta, x)$ is already a (global) size bound. Otherwise, if $\mathrm{pre}(\vartheta) \neq \varnothing$, then Thm. 33 over-approximates the sizes of the variables in $\mathcal{SB}_{\mathrm{loc}}(\vartheta, x)$ by the size bounds corresponding to the preceding transitions. To this end, every program variable v in $\mathcal{SB}_{\mathrm{loc}}(\alpha)$ for $\alpha = \langle \vartheta, x\rangle$ is replaced by its size bound $\mathcal{SB}(\vartheta', v)$ for a predecessor $\vartheta' \in \mathrm{pre}(\vartheta)$ in the RVG. This is again denoted by $\mathcal{SB}_{\mathrm{loc}}(\alpha)\,[v/\mathcal{SB}(\vartheta', v) \mid v \in \mathcal{V}]$. Thus, as mentioned, the SCCs of the RVG should be handled in topological order such that finite global size bounds may already be available for the predecessors of ϑ.

Theorem 33 (Size Bounds for Trivial SCCs Without Function Calls). *Let $\mathcal{SB}$ be a size bound and $\{\langle \vartheta, x\rangle\}$ be a trivial SCC of the RVG such that $\vartheta \in \mathcal{F}$ or $\mathrm{fun}(\eta(x)) = \varnothing$ for the update η of $\vartheta \in \mathcal{T}$. Then $\mathcal{SB}'$ is also a size bound where $\mathcal{SB}'(\alpha) = \mathcal{SB}(\alpha)$ for all $\alpha \neq \langle \vartheta, x\rangle$, and for $\alpha = \langle \vartheta, x\rangle$ we have*

$$\mathcal{SB}'(\alpha) = \begin{cases} \mathcal{SB}_{\mathrm{loc}}(\alpha), & \text{if } \mathrm{pre}(\vartheta) = \varnothing \\ \max_{\vartheta' \in \mathrm{pre}(\vartheta)} \{\mathcal{SB}_{\mathrm{loc}}(\alpha)\,[v/\mathcal{SB}(\vartheta', v) \mid v \in \mathcal{V}]\}, & \text{otherwise.} \end{cases}$$

Note that due to the requirement on $\langle \vartheta, x\rangle$ in Thm. 33, w.l.o.g. $\mathcal{SB}_{\mathrm{loc}}(\vartheta, x) \in \mathbb{N}[\mathcal{V} \cup \mathcal{F}]$ only contains program variables $\mathcal{V}$, but no variable from $\mathcal{F}$.

Example 34. Reconsider Fig. 2 and 5. We have $\mathcal{SB}(t_0, x) = \mathcal{SB}_{\mathrm{loc}}(t_0, x) = x$ by Thm. 33 as $\mathrm{pre}(t_0) = \varnothing$. Moreover, we obtain $\mathcal{SB}(t_4, a) = 1$ since $\mathcal{SB}_{\mathrm{loc}}(t_4, a) = 1$. In Ex. 38 we will show that $\mathcal{SB}(t_1, x) = x$ is a size bound. As $\mathcal{SB}_{\mathrm{loc}}(\rho_1, a) = x$ and $\mathrm{pre}(\rho_1) = \{t_0, t_1\}$, this implies

$$\mathcal{SB}(\rho_1, a) = \max\{x[x/\mathcal{SB}(t_0, x)], x[x/\mathcal{SB}(t_1, x)]\} = x.$$

The following theorem handles trivial SCCs $\{\alpha\}$ for $\alpha = \langle t, x\rangle$ where t's update for x contains function calls ρ. Hence, in contrast to Thm. 33, we have to instantiate the function calls ρ in $\mathcal{SB}_{\mathrm{loc}}(\alpha)$ by the size bounds for the transitions of $\mathrm{pre}^{\Omega}(t, \rho)$, as they reach the corresponding return locations. If some of these function calls ρ do not reach a return location, then we can set the size bound

for $\langle t, x\rangle$ to 0, because then $\to_t$ only reaches configurations of the form $(\_, \bot)$. If $\mathrm{pre}(t) = \varnothing$ (i.e., t does not have a RV-predecessor in the RVG) but every function call $\rho_1, \dots, \rho_n \in \mathrm{fun}(\eta(x))$ has a predecessor (i.e., $\mathrm{pre}^\Omega(t, \rho_i) \neq \varnothing$ for all i), then we replace every such function call ρ_i by a size bound for the Ω-predecessor. More precisely, we replace ρ_i in $\mathcal{SB}_{\mathrm{loc}}(\alpha)$ by $\mathcal{SB}(\beta_i)$ for $\beta_i \in \mathrm{pre}^\Omega(t, \rho_i)$. In the following, this is denoted by $\mathcal{SB}_{\mathrm{loc}}(\alpha)\,[\rho_i/\mathcal{SB}(\beta_i) \mid i \in [n]]$ where $[n]$ is the set $\{1, \dots, n\}$. Otherwise, if t has RV-predecessors in the RVG, then we also have to instantiate the program variables by the size bounds corresponding to the preceding transitions, as in Thm. 33. Afterwards, the function calls ρ_i are handled as in the previous case. Note that the order of the substitutions is important here. Replacing a function call ρ_i with the size bound $\mathcal{SB}(\beta_i)$ *before* substituting program variables by size bounds from the preceding transitions is unsound in general, as variables in $\mathcal{SB}(\beta_i)$ would then also be substituted incorrectly.

Theorem 35 (Size Bounds for Trivial SCCs With Function Calls). *Let $\mathcal{SB}$ be a size bound and $\{\langle t, x\rangle\}$ be a trivial SCC of the RVG such that $t \in \mathcal{T}$ and $\mathrm{fun}(\eta(x)) \neq \varnothing$. Then $\mathcal{SB}'$ is also a size bound where $\mathcal{SB}'(\alpha) = \mathcal{SB}(\alpha)$ for all $\alpha \neq \langle t, x\rangle$, and for $\alpha = \langle t, x\rangle$ with $\mathrm{fun}(\eta(x)) = \{\rho_1, \dots, \rho_n\}$, we have*

$$\mathcal{SB}'(\alpha) = \begin{cases} 0, & \text{if } \mathrm{pre}^\Omega(t, \rho_i) = \varnothing \text{ for some } i \in [n] \\ \max\limits_{\beta_i \in \mathrm{pre}^\Omega(t,\rho_i)} \{\mathcal{SB}_{\mathrm{loc}}(\alpha)\,[\rho_i/\mathcal{SB}(\beta_i) \mid i \in [n]]\}, & \text{if all } \mathrm{pre}^\Omega(t, \rho_i) \neq \varnothing \text{ and } \mathrm{pre}(t) = \varnothing \\ \max\limits_{\substack{\beta_i \in \mathrm{pre}^\Omega(t,\rho_i) \\ \vartheta' \in \mathrm{pre}(t)}} \{\mathcal{SB}_{\mathrm{loc}}(\alpha)\,[v/\mathcal{SB}(\vartheta', v) \mid v \in \mathcal{V}]\,[\rho_i/\mathcal{SB}(\beta_i) \mid i \in [n]]\}, & \text{if all } \mathrm{pre}^\Omega(t, \rho_i) \neq \varnothing \text{ and } \mathrm{pre}(t) \neq \varnothing. \end{cases}$$

Example 36. Consider a variant of Fig. 2 where we replace the update $\eta(y)$ of t_1 by $\rho_1 = f_1(\zeta_1)$. Thus, the self-loop at $\langle t_1, y\rangle$ is removed from the RVG in Fig. 5. Then, we can apply Thm. 35 on the trivial SCC $\{\langle t_1, y\rangle\}$. Assume that we already computed $\mathcal{SB}(t_4, a) = 1$ and $\mathcal{SB}(t_5, a) = x^x$ (see Ex. 34 and 40). We have $\mathrm{pre}(t_1) = \{t_0, t_1\}$, but $\mathcal{SB}_{\mathrm{loc}}(t_1, y) = \rho_1$ does not contain variables from $\mathcal{V}$. Hence, we get $\mathcal{SB}(t_1, y) = \max\{\rho_1[\rho_1/\mathcal{SB}(t_4, a)], \rho_1[\rho_1/\mathcal{SB}(t_5, a)]\} = x^x$ as we have the Ω-predecessors $\langle t_4, a\rangle$ and $\langle t_5, a\rangle \in \mathrm{pre}^\Omega(t_1, \rho_1)$.

4.2 Size Bounds for Non-Trivial SCCs

Finally, we introduce our approach to handle non-trivial SCCs. Let $C \subseteq \mathcal{RV}$ be the nodes of such an SCC. Our approach can only be applied to SCCs where for all $\alpha \in C$, there exist $e_\alpha \in \mathbb{N}$ and $s_\alpha \in \mathbb{N}[\mathcal{V}]$ such that

$$\mathcal{SB}_{\mathrm{loc}}(\alpha) \leq s_\alpha \cdot (\, e_\alpha + \textstyle\sum_{v \in \mathrm{act}(\mathcal{SB}_{\mathrm{loc}}(\alpha)) \setminus \mathrm{act}(s_\alpha)} v \,) \tag{1}$$

where "$\leq$" is interpreted pointwise (i.e., the inequation must hold for all instantiations of the variables by natural numbers). Here, s_α captures the scaling behavior of $\mathcal{SB}_{\mathrm{loc}}(\alpha)$, e.g., it allows us to consider updates of the form $\eta(x) = 2{\cdot}x$

or $\eta(x) = a \cdot x$ for a variable $a \in \mathcal{V}$. Note that in [10], only constant factors s_α were allowed. Similarly, e_α captures the additive growth in updates like $\eta(x) = 1 + x$. For instance, we have $s_\alpha = a$, $e_\alpha = 0$, and $\mathrm{act}(\mathcal{SB}_{\mathrm{loc}}(\alpha)) \setminus \mathrm{act}(s_\alpha) = \{x\}$ for the update $\eta(x) = a \cdot x$. The restriction (1) is essential for our approach, since it allows us to apply an "accumulation" argument when lifting local to global size bounds. Note that all linear updates satisfy (1). Thus, our approach is applicable to a wide range of programs in practice. However, we cannot express all non-linear updates (e.g., the update x^x cannot be handled).

We now also define pre and pre^Ω for result variables. For $\alpha \in \mathcal{RV}$, $\mathrm{pre}(\alpha)$ ($\mathrm{pre}^\Omega(\alpha)$) is the set of all result variables α' with an RV-edge (Ω-edge) from α' to α in the RVG. Furthermore, for any result variable α in the SCC C, let $V_\alpha = \{v \in \mathcal{V} \mid \exists \vartheta.\, \langle \vartheta, v\rangle \in \mathrm{pre}(\alpha) \cap C\}$ consist of all variables v with an RV-edge to α in C, and similarly, $F_\alpha = \{v \in \mathcal{V} \mid \exists t.\, \langle t, v\rangle \in \mathrm{pre}^\Omega(\alpha) \cap C\}$ are all variables with an Ω-edge to α in C. Recall that $\mathrm{act}(p)$ is the set of active arguments of the polynomial $p \in \mathbb{N}[\mathcal{V} \cup \mathcal{F}]$. Finally, for any $p \in \mathbb{N}[\mathcal{V} \cup \mathcal{F}]$, let $\mathrm{actV}(p) = \mathrm{act}(p) \cap \mathcal{V}$ be p's *active variables* and $\mathrm{actF}(p) = \mathrm{act}(p) \cap \mathcal{F}$ be p's *active function calls*.

We first consider "additive" local size bounds only, and afterwards we generalize our method to handle "multiplicative" local size bounds as well.

Additive Local Size Bounds: We first consider local size bounds which are additive, i.e., where $s_\alpha = 1$ and $|V_\alpha| + |F_\alpha| \leq 1$. Note that both of these requirements are necessary to prevent non-additive, exponential growth.

To consider the additive growth, we over-approximate the sizes of variables on incoming edges from outside the SCC C. For example, consider the additive update $\eta(x) = x + y$ of a simple loop where y is not changed with a runtime bound rb. In this example, the value of y at the entry of the loop is repeatedly added to x in each iteration, i.e., we have $V_\alpha = \{x\}$. To capture such *initial* values of an SCC of the RVG, we introduce the expressions init_α for RV-edges and $\mathit{init}^\Omega_\alpha$ for Ω-edges which over-approximate the incoming values. Let $\mathit{init}_\alpha(v) = \max\{\mathcal{SB}(\vartheta, v) \mid \exists \vartheta \in \mathcal{T} \cup \mathcal{F}.\, \langle \vartheta, v\rangle \in \mathrm{pre}(\alpha) \setminus C\}$ be a bound on the size of v when entering C via an RV-edge to α. Analogously, $\mathit{init}^\Omega_\alpha(v) = \max\{\mathcal{SB}(t, v) \mid \exists t \in \mathcal{T}.\, \langle t, v\rangle \in \mathrm{pre}^\Omega(\alpha) \setminus C\}$ is a bound on the size of v when entering C via an Ω-edge to α. To take the effect of function calls ρ into account, we have to consider the return variables of the return locations reachable from ρ. To ease the presentation, we assume that every function call ρ always returns the same variable v_ρ, i.e., that ρ cannot reach two return locations ℓ_1, ℓ_2 with $v_{\ell_1} \neq v_{\ell_2}$.[5] The execution of α's transition or function call then means that the values of the variables in V_α or F_α can be increased by adding $\mathit{init}_\alpha(v)$ for all $v \in \mathrm{actV}(\mathcal{SB}_{\mathrm{loc}}(\alpha)) \setminus V_\alpha$ (or, respectively, by adding $\mathit{init}^\Omega_\alpha(v_\rho)$ for all $\rho \in \mathrm{actF}(\mathcal{SB}_{\mathrm{loc}}(\alpha))$ and $v_\rho \notin F_\alpha$, where v_ρ is the return variable associated to ρ) plus the constant e_α.

Reconsider our example with the update $\eta(x) = x + y$ where $V_\alpha = \{x\}$. After n iterations of the loop, the value of x is increased by $n \cdot \mathit{init}_\alpha(y)$. Using

[5] This could be generalized to function calls ρ whose set of return variables $\mathit{retvar}(\rho) = \{v_{\ell'} \mid \rho \text{ reaches } \ell' \in \Omega\}$ it not a singleton. Then in (2), instead of considering all $\rho \in \mathrm{actF}(\mathcal{SB}_{\mathrm{loc}}(\alpha))$ with $v_\rho \notin F_\alpha$, one would have to consider all $v \in \mathit{retvar}(\rho) \setminus F_\alpha$.

the runtime bound rb of this loop, our approach would over-approximate this increase by $rb \cdot init_\alpha(y)$.

In general, the increase of the variables in V_α or F_α by $init_\alpha(v)$ and $init^\Omega_\alpha(v_\rho)$ is repeated rb_α times, where $rb_\alpha = \mathcal{RB}(t)$ if $\alpha = \langle t, v\rangle$ and $rb_\alpha = \sum_{t \in \text{trans}(\rho)} \mathcal{RB}(t)$ if $\alpha = \langle \rho, v\rangle$, i.e., rb_α is a bound on how often α's transition or function call is evaluated during a program run. Thus, the following expression over-approximates the additive size-change resulting from α (ignoring the growth resulting from V_α and F_α for now):

$$add(\alpha) = rb_\alpha \cdot \Big(e_\alpha + \sum_{\substack{v \in \text{actV}(\mathcal{SB}_{\text{loc}}(\alpha)) \\ v \notin V_\alpha}} init_\alpha(v) \quad + \sum_{\substack{\rho \in \text{actF}(\mathcal{SB}_{\text{loc}}(\alpha)) \\ v_\rho \notin F_\alpha}} init^\Omega_\alpha(v_\rho) \Big) \quad (2)$$

We now take the growth resulting from the variables in V_α or F_α into account. Here, one has to consider the initial values of the variables in V_α and F_α before entering the SCC C. This leads to

$$add(\alpha) \; + \; \textstyle\sum_{v \in V_\alpha} init_\alpha(v) \; + \; \sum_{v \in F_\alpha} init^\Omega_\alpha(v).$$

Since we only have to consider the initial values of the variables in V_α and F_α, they are not multiplied with the runtime bound. So in our example with the update $\eta(x) = x + y$ and $V_\alpha = \{x\}$, $init_\alpha(x)$ is not multiplied with the runtime bound rb and we would obtain the size bound $init_\alpha(x) + rb \cdot init_\alpha(y)$. The following Thm. 37 summarizes the previous observations and yields size bounds for additive SCCs. To simplify the presentation, in contrast to [10], we do not consider transitions individually.

Theorem 37 (Size Bounds for Non-Trivial Additive SCCs). *Let $\mathcal{SB}$ be a size bound and C be a non-trivial SCC of the RVG, where for all $\alpha \in C$, $\mathcal{SB}_{\text{loc}}(\alpha)$ satisfies (1) for a suitable e_α and $s_\alpha = 1$, and moreover $|V_\alpha| + |F_\alpha| \leq 1$. Then $\mathcal{SB}'$ is also a size bound where $\mathcal{SB}'(\alpha) = \mathcal{SB}(\alpha)$ for all $\alpha \in \mathcal{RV} \setminus C$, and $\mathcal{SB}'(\alpha) = \mathcal{SB}'(C)$ for all $\alpha \in C$, where*

$$\mathcal{SB}'(C) = \textstyle\sum_{\alpha \in C} (add(\alpha) + \sum_{v \in V_\alpha} init_\alpha(v) + \sum_{v \in F_\alpha} init^\Omega_\alpha(v))$$

Example 38. Reconsider Fig. 2 and 5. We now infer size bounds for the non-trivial SCCs $\{\langle t_1, x\rangle\}$ and $\{\langle \rho_2, a\rangle\}$.

- For $\alpha = \langle t_1, x\rangle$ with $\mathcal{SB}_{\text{loc}}(t_1, x) = x$, we have $s_\alpha = 1$, $e_\alpha = 0$, $V_\alpha = \{x\}$, $F_\alpha = \varnothing$, $\text{actV}(\mathcal{SB}_{\text{loc}}(t_1, x)) = \{x\}$, and $\text{actF}(\mathcal{SB}_{\text{loc}}(t_1, x)) = \varnothing$. This implies $add(\alpha) = 0$ and $init_\alpha(x) = \mathcal{SB}(t_0, x) = x$. Thus, we obtain the size bound $\mathcal{SB}(t_1, x) = x$.
- Similarly, for $\alpha = \langle \rho_2, a\rangle$ with $\mathcal{SB}_{\text{loc}}(\rho_2, a) = a$, we obtain $s_\alpha = 1$, $e_\alpha = 0$, $V_\alpha = \{a\}$, $F_\alpha = \varnothing$, $\text{actV}(\mathcal{SB}_{\text{loc}}(\rho_2, a)) = \{a\}$, and $\text{actF}(\mathcal{SB}_{\text{loc}}(\rho_2, a)) = \varnothing$. Thus, we have $add(\alpha) = 0$ and $init_\alpha(a) = \mathcal{SB}(\rho_1, a) = x$ by Ex. 34. This yields the size bound $\mathcal{SB}(\rho_2, a) = x$.

Multiplicative Local Size Bounds: Now, we consider local size bounds where $s_\alpha \neq 1$ or $|V_\alpha| + |F_\alpha| > 1$. Again, let us introduce the essentially ideas for the size bound computations with a simple loop with the runtime bound rb. In our example, we consider the update $\eta(x) = \eta(y) = x + y$ where $x, y \in V_\alpha$. Then, both x and y grow with a factor of two. A similar effect would be obtained for scaling factors $s_\alpha > 1$. If $|V_\alpha| + |F_\alpha| > 1$, then each execution of α's transition or function call may multiply the value of a variable by $|V_\alpha| + |F_\alpha|$. As for the additive growth, this multiplication must be performed rb_α times. Thus, we obtain $(|V_\alpha| + |F_\alpha|)^{rb_\alpha}$ as the scaling factor caused by V_α and F_α. The scaling factor s_α can be handled similarly. However, here we need to be careful as s_α might contain variables. To this end, we over-approximate every variable in s_α by the size bound of the predecessors and ensure that the scaling factor is at least 1. This is captured by $scale(\alpha)$ with

$$scale(\alpha) = (\max_{\langle \vartheta, \_ \rangle \in \mathrm{pre}(\alpha)} \{1, s_\alpha[v/\mathcal{SB}(\vartheta, v) \mid v \in \mathcal{V}]\} \cdot (|V_\alpha| + |F_\alpha|))^{rb_\alpha} \quad (3)$$

So for our example $\eta(x) = \eta(y) = x+y$, we infer the scaling factor $scale(\alpha) = 2^{rb}$. Since $V_\alpha = \{x, y\}$, here we obtain the size bound $2^{rb} \cdot (init_\alpha(x) + init_\alpha(y))$ by multiplying the scaling factor with the initial values $init_\alpha(x) + init_\alpha(y)$.

The following theorem shows how to compute size bounds for non-trivial SCCs C in general, by accumulating $scale(\alpha)$ and $add(\alpha)$ for all $\alpha \in C$. To this end, we have to multiply $scale(\alpha)$ with the initial size bounds for V_α and F_α as in the example. (The other initial size bounds are already covered in $add(\alpha)$.) So, the scaling effect of several result variables in C has to be multiplied whereas the additive size change for several result variables in C has to be added.

Theorem 39 (Size Bounds for Non-Trivial SCCs). *Let $\mathcal{SB}$ be a size bound and C be a non-trivial SCC of the RVG, where for all $\alpha \in C$, $\mathcal{SB}_{\mathrm{loc}}(\alpha)$ satisfies (1) for suitable e_α and s_α. Then $\mathcal{SB}'$ is also a size bound where $\mathcal{SB}'(\alpha) = \mathcal{SB}(\alpha)$ for all $\alpha \in \mathcal{RV} \setminus C$, and $\mathcal{SB}'(\alpha) = \mathcal{SB}'(C)$ for all $\alpha \in C$, where*

$$\mathcal{SB}'(C) = \prod_{\alpha \in C} scale(\alpha) \cdot (\sum_{\alpha \in C}(add(\alpha) + \sum_{v \in V_\alpha} init_\alpha(v) + \sum_{v \in F_\alpha} init^{\Omega}_\alpha(v)))$$

Example 40. We consider Fig. 2 and 5 again and infer size bounds for the missing SCCs $\{\langle t_5, a\rangle\}$ and $\{\langle t_1, y\rangle\}$.

- For $\alpha = \langle t_5, a\rangle$ with $\mathcal{SB}_{\mathrm{loc}}(t_5, a) = a \cdot \rho_2$, we have $s_\alpha = a$, $e_\alpha = 0$, $V_\alpha = \varnothing$, $F_\alpha = \{a\}$, $\mathrm{actV}(\mathcal{SB}_{\mathrm{loc}}(t_5, a)) = \{a\}$, and $\mathrm{actF}(\mathcal{SB}_{\mathrm{loc}}(t_5, a)) = \{\rho_2\}$. As $\mathrm{pre}(t_5) = \{\rho_1, \rho_2\}$, we have

  $$scale(\alpha) = (\max\{1, a[a/\mathcal{SB}(\rho_1, a)], a[a/\mathcal{SB}(\rho_2, a)]\})^{\mathcal{RB}(t_5)} = x^{x^2}$$

  (with $\mathcal{RB}(t_5) = x^2$ by Ex. 27 and $\mathcal{SB}(\rho_1, a) = \mathcal{SB}(\rho_2, a) = x$ when using the invariant $x > 0$ and thus, $\max\{1, x\} = x$), $add(\alpha) = x^2 \cdot 0 = 0$, and $init^{\Omega}_\alpha(a) = \mathcal{SB}(t_4, a) = 1$ by Ex. 34. Hence, we obtain the size bound $\mathcal{SB}(t_5, a) = x^{x^2}$. Note that (3) could be improved by considering a local runtime bound instead of the global runtime bound rb_α. A similar improvement is used in [36, Thm. 34]. Thus, to obtain simpler bounds for readability, we use the size bound $\mathcal{SB}(t_5, a) = x^x$.

- Finally, for $\alpha = \langle t_1, y \rangle$ with $\mathcal{SB}_{\text{loc}}(t_1, y) = y + \rho_1$, we have $s_\alpha = 1$, $e_\alpha = 0$, $V_\alpha = \{y\}$, $F_\alpha = \varnothing$, $\text{actV}(\mathcal{SB}_{\text{loc}}(t_1, y)) = \{y\}$, and $\text{actF}(\mathcal{SB}_{\text{loc}}(t_1, y)) = \{\rho_1\}$. Thus, we obtain $scale(\alpha) = 1$, $add(\alpha) = \mathcal{RB}(t_1) \cdot init^{\Omega}_{\alpha}(a) = x \cdot \max\{\mathcal{SB}(t_4, a), \mathcal{SB}(t_5, a)\} = x \cdot x^x$ (with $\mathcal{RB}(t_1) = x$ by Ex. 27, $\mathcal{SB}(t_4, a) = 1$, and $\mathcal{SB}(t_5, a) = x^x$), and $init_\alpha(y) = \mathcal{SB}(t_0, y) = y$. Hence, we get the size bound $\mathcal{SB}(t_1, y) = y + x \cdot x^x$. This also implies $\mathcal{SB}(t_2, y) = y + x \cdot x^x$.

5 Conclusion, Evaluation, and Related Work

In this paper we presented a novel framework for complexity analysis of integer programs with (possibly non-tail recursive) function calls. We introduced a unified approach that alternates between inferring runtime and size bounds.

The approach is modular, since it handles subprograms separately and allows us to use different techniques to generate bounds for the respective subprograms. To infer runtime bounds, in addition to techniques based on multiphase-linear ranking functions [7, 8, 22] and "complete" techniques for *twn*-loops [10, 24, 25, 35, 36, 38], we introduced the new class of ρ-ranking functions to handle subprograms with function calls in Sect. 3. To infer size bounds, we showed in Sect. 4 how to generalize the respective technique from [10] to ITSs with function calls. So in particular, our novel approach can compute size bounds for the return values of function calls, i.e., the results of function calls are taken into account when analyzing complexity. We implemented our new approach in the open-source tool KoAT such that it can now also analyze ITSs with (possibly recursive) function calls.

Of course, our approach and its implementation have several limitations. Currently, KoAT can only successfully analyze programs where the arguments in recursive calls decrease w.r.t. a linear polynomial ranking function. Moreover, our approach via ρ-RFs can only infer polynomial or exponential bounds. So for example, we cannot obtain precise bounds for recursive algorithms with logarithmic runtime. (Our implementation only computes logarithmic bounds for subprograms that are *twn*-loops [38].) Another restriction is that we can only lift local size bounds of the form (1) to global size bounds. In addition, KoAT may fail to infer finite bounds for several other reasons, e.g., some examples would need stronger invariants to make our approach succeed. A detailed discussion and a corresponding list of examples to demonstrate the limitations of our approach and its implementation can be found on our webpage [41].

In the following, we conclude by discussing related work and by providing an experimental evaluation of our approach using the implementation in KoAT.

5.1 Related Work

As mentioned in the introduction, there exist many approaches to analyze complexity of programs automatically, e.g., [3–5, 7, 10, 11, 18, 22, 23, 28, 29, 37, 42, 45, 48]. However, only few of them focus on programs with recursion or function

calls. While we already discussed an extension to recursive ITSs in [10], here the return values of function calls were ignored.

Techniques for complexity analysis of *term rewrite systems* [6] can handle (possibly non-tail) recursion, but standard TRSs do not support built-in types like integers. However in [44], *recursive natural transition systems* with potential non-tail recursion were introduced in order to study the connection between complexity analysis for TRSs and our approach from [10] for complexity analysis of ITSs. Here, the idea is to summarize (and subsequently eliminate) subprocedures by approximating their runtime and size. Thus, this approach does not benefit from techniques such as our new class of ranking functions which allows us to handle subprograms with function calls directly.

Instead of representing integer programs as ITSs, there are also techniques based on *cost equation systems* which can express non-tail recursive integer programs as well, e.g., [17, 18], implemented in the tool CoFloCo. This approach analyzes program parts independently and uses linear invariants to compose the results, i.e., it differs significantly from our approach which can also infer non-linear size bounds. Similarly, in the tool PUBS [2, 4], *cost relations* are analyzed which are a system of recursive equations that capture the cost of the program. There are also numerous approaches for automatic resource analysis of functional programs, often based on amortized analysis (see [30] for an overview). For example, an approach for automatic complexity analysis of OCaml programs is presented in [28, 29], which however has limitations w.r.t. modularity, see [44], and is restricted to polynomial bounds. There are also several approaches based on types, e.g., the resource consumption of Liquid Haskell programs is encoded in a type system in [23], but here bounds are not inferred automatically. Another line of work automatically infers bounds from recursive programs by generating and solving recurrence relations, e.g., [31, 47]. In future work, it might be interesting to integrate such techniques within our framework, as they can infer bounds that go beyond polynomials and exponentials.

There also exist tools which analyze the runtime complexity of C-code, e.g., Loopus [48] or MaxCore [5] with CoFloCo or PUBS in the backend. For KoAT, we used Clang [13] and llvm2kittel [16] to transform pointer-free C programs into ITSs, and to handle more general C programs, we developed the framework AProVE (KoAT + LoAT) [39], which also participates in the annual *Software Verification Competition* (*SV-COMP*) [49]. AProVE can also prove termination of recursive C programs. But when analyzing complexity, all these approaches are limited to C programs without recursion. In the future, it would be interesting to use our novel approach for ρ-ITSs to extend complexity analysis to recursive C programs, in particular in our framework AProVE (KoAT + LoAT).

5.2 Evaluation and Implementation

We implemented our novel results and integrated them into our open-source tool KoAT which also features powerful techniques for subprograms without function calls [10, 22, 35–38]. In the beginning, KoAT preprocesses the program, e.g., by

	$\mathcal{O}(1)$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$< \omega$	AVG$^+$(s)	AVG(s)
KoAT	128	11	281 (18)	121 (8)	28 (4)	583 (34)	3.45	21.78
KoAT1	132	0	231 (15)	108 (4)	16 (2)	499 (24)	0.64	8.06
CoFloCo	125	0	234 (1)	95	10 (1)	464 (2)	3.19	16.38

Table 1: Evaluation on the Collection of ITSs and ρ-ITSs

extending the guards of transitions with invariants inferred by Apron [32]. For all SMT problems (including the generation of ρ-RFs), KoAT calls Z3 [43].

For our evaluation, we use the set of 838 ITSs for complexity analysis of integer programs from the *Termination Problems Data Base* (*TPDB*) [50] which is used in the annual *Termination and Complexity Competition* (*TermComp*) [21]. While the *TPDB* contains a large collection of ITSs without function calls, up to now there does not exist any such standard benchmark set for ITSs with function calls. Thus, to obtain a representative collection of typical recursive programs, we extended our set by 45 new examples. To this end, we transformed 20 recursive C programs of the *TPDB* and of the collection used at *SV-COMP* into our novel formalism of ρ-ITSs. Moreover, we included our leading example (Fig. 2) along with several variants. Our benchmarks also contain ρ-ITSs with nested function calls and algorithms with multiple recursive function calls (e.g., the naive implementation of the Fibonacci numbers). Moreover, our set of examples includes recursive versions of insertion sort and selection sort (where lists were abstracted to their lengths), for which we can infer quadratic runtime bounds. In particular, there are also benchmarks that depend on non-linear size bounds, e.g., an algorithm which computes the product of two numbers recursively and then uses this result in a subsequent loop. The average size of our benchmarks is 26 lines, whereas the largest one has 1615 lines. In contrast, the average size of those benchmarks where KoAT infers a finite bound is 17 lines and the largest of them has 246 lines.

Our benchmark collection, a detailed description of every new benchmark, and also the detailed results of our evaluation can be found on our webpage [41].

To distinguish the original KoAT implementation of [10] from our re-implementation, we refer to the tool of [10] as KoAT1 in the following. We evaluated our novel version of KoAT with the approach of the current paper against the tools KoAT1 and CoFloCo, which can also handle certain forms of recursion. To this end, we transformed our novel benchmark ρ-ITSs manually into their formalism. We do not compare with PUBS, because as stated in [15] by one of the authors of PUBS, CoFloCo is strictly stronger than PUBS. Table 1 shows the results of our evaluation, where as in *TermComp*, we used a timeout of 5 minutes per example. All tools were run inside an Ubuntu Docker container on a machine with an AMD Ryzen 7 3700X octa-core CPU and 8 GB of RAM. The first entry in every cell denotes the number of benchmarks for which the tool inferred the respective bound, where we consider both the ITSs from the *TPDB* and our

new ρ-ITS benchmarks. The number in brackets only considers the 45 new ρ-ITS benchmarks. The runtime bounds inferred by the tools are compared asymptotically as functions which depend on the largest initial absolute value n of all program variables. So for example, KoAT proved an (at most) linear runtime bound for $128 + 11 + 281 = 420$ benchmarks, i.e., for these examples KoAT can show that $\mathrm{rc}(\sigma_0) \in \mathcal{O}(n)$ for all initial states $\sigma_0 \in \Sigma$ where $|\sigma_0(v)| \leq n$ for all $v \in \mathcal{V}$. Overall, this configuration succeeds on 583 examples, i.e., "$< \omega$" is the number of examples where a finite bound on the runtime complexity could be computed by the tool within the time limit. Moreover, our tool KoAT is able to prove termination for 626 benchmarks. So the termination proof succeeds for 43 additional examples since one does not have to construct actual runtime bounds and does not have to consider size bounds. "AVG$^+$(s)" denotes the average runtime of successful runs in seconds, whereas "AVG(s)" is the average runtime of all runs.

In our experiments, KoAT was the most powerful tool for runtime complexity analysis on both classical ITSs and – due to the novel contributions of this paper – also on ITSs with recursive function calls. Note that KoAT1 heuristically applies loop-unrolling which might eliminate loops with constant runtime. For this reason, KoAT1 infers constant runtime bounds for slightly more benchmarks than KoAT. For all other complexity classes in Table 1, KoAT finds more examples for the respective class than the two other tools. Moreover, KoAT is the only of the three tools which can also infer logarithmic bounds (due to the integration of dedicated analysis techniques for subprograms that are *twn*-loops). Nevertheless, the three tools are "orthogonal", i.e., for each tool there are examples where the tool provides a finite bound and the other two tools fail.

Note that the tool LoAT [19, 20] is able to prove absence of finite runtime bounds for 231 of the 883 benchmarks. Thus, KoAT is able to infer finite complexity bounds for 89% of all $883 - 231 = 652$ benchmarks where this is potentially possible. Our experiments demonstrate that handling return values directly yields significantly more precise bounds than prior approaches that simply ignore them. In particular, to our knowledge KoAT is currently the only tool which can infer a finite runtime bound for the recursive ITS from our leading example (Fig. 2). So the new contributions of the paper are crucial in order to extend automated complexity analysis to the setting of recursive programs. Moreover, verification frameworks for other programming languages can now be extended to analyze complexity of recursive programs by using KoAT as a backend.

KoAT's source code, a binary, a Docker image, and details on our new benchmarks and our evaluation are available at our webpage [41]:

https://koat.verify.rwth-aachen.de/function-calls

This website also contains details on our input format for ρ-ITSs and a *web interface* to run different configurations of KoAT directly online. In addition, we also provide an artifact [33] with KoAT's binary and Docker images in order to reproduce our experiments.

Data Availability Statement

An artifact including our implementation in the tool KoAT is available at [33]:

https://doi.org/10.5281/zenodo.15586347

This artifact allows to reproduce the results of our evaluation.

References

[1] P. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. "General Decidability Theorems for Infinite-State Systems". In: *Proc. LICS '96*. 1996, pp. 313–321. DOI: 10.1109/LICS.1996.561359.

[2] E. Albert, P. Arenas, S. Genaim, and G. Puebla. "Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis". In: *Proc. SAS '08*. LNCS 5079. 2008, pp. 221–237. DOI: 10.1007/978-3-540-69166-2_15.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. "Cost Analysis of Object-Oriented Bytecode Programs". In: *Theoretical Computer Science* 413.1 (2012), pp. 142–159. DOI: 10.1016/j.tcs.2011.07.009.

[4] E. Albert, S. Genaim, and A. N. Masud. "On the Inference of Resource Usage Upper and Lower Bounds". In: *ACM Transactions on Computational Logic* 14.3 (2013). DOI: 10.1145/2499937.2499943.

[5] E. Albert, M. Bofill, C. Borralleras, E. Martín-Martín, and A. Rubio. "Resource Analysis Driven by (Conditional) Termination Proofs". In: *Theory and Practice of Logic Programming* 19.5-6 (2019), pp. 722–739. DOI: 10.1017/S1471068419000152.

[6] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. DOI: 10.1017/CBO9781139172752.

[7] A. M. Ben-Amram and S. Genaim. "On Multiphase-Linear Ranking Functions". In: *Proc. CAV '17*. LNCS 10427. 2017, pp. 601–620. DOI: 10.1007/978-3-319-63390-9_32.

[8] A. M. Ben-Amram, J. J. Doménech, and S. Genaim. "Multiphase-Linear Ranking Functions and Their Relation to Recurrent Sets". In: *Proc. SAS '19*. LNCS 11822. 2019, pp. 459–480. DOI: 10.1007/978-3-030-32304-2_22.

[9] A. R. Bradley, Z. Manna, and H. B. Sipma. "The Polyranking Principle". In: *Proc. ICALP '05*. LNCS 3580. 2005, pp. 1349–1361. DOI: 10.1007/11523468_109.

[10] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. "Analyzing Runtime and Size Complexity of Integer Programs". In: *ACM Transactions on Programming Languages and Systems* 38 (2016), pp. 1–50. DOI: 10.1145/2866575.

[11] Q. Carbonneaux, J. Hoffmann, and Z. Shao. "Compositional Certified Resource Bounds". In: *Proc. PLDI '15*. 2015, pp. 467–478. DOI: 10.1145/2737924.2737955.

[12] K. Chatterjee, A. Goharshady, E. Goharshady, M. Karrabi, and D. Zikelic. "Sound and Complete Witnesses for Template-Based Verification of LTL Properties on Polynomial Programs". In: *Proc. FM '25*. LNCS 14933. 2024, pp. 600–619. DOI: 10.1007/978-3-031-71162-6_31.

[13] Clang Compiler. URL: https://clang.llvm.org/.

[14] J. Dingel and T. Filkorn. "Model Checking for Infinite State Systems Using Data Abstraction, Assumption-Commitment Style Reasoning and Theorem Proving". In: *In Proc. CAV '95*. LNCS 939. 1995, pp. 54–69. DOI: 10.1007/3-540-60045-0_40.

[15] J. J. Doménech, J. P. Gallagher, and S. Genaim. "Control-Flow Refinement by Partial Evaluation, and Its Application to Termination and Cost Analysis". In: *Theory and Practice of Logic Programming* 19.5-6 (2019), pp. 990–1005. DOI: 10.1017/S1471068419000310.

[16] S. Falke, D. Kapur, and C. Sinz. "Termination Analysis of C Programs Using Compiler Intermediate Languages". In: *Proc. RTA '11*. LIPIcs 10. 2011, pp. 41–50. DOI: 10.4230/LIPIcs.RTA.2011.41.

[17] A. Flores-Montoya. "Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations". In: *Proc. FM '16*. LNCS 9995. 2016, pp. 254–273. DOI: 10.1007/978-3-319-48989-6_16.

[18] A. Flores-Montoya and R. Hähnle. "Resource Analysis of Complex Programs with Cost Equations". In: *Proc. APLAS '14*. LNCS 8858. 2014, pp. 275–295. DOI: 10.1007/978-3-319-12736-1_15.

[19] F. Frohn and J. Giesl. "Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description)". In: *Proc. IJCAR '22*. LNCS 13385. 2022, pp. 712–722. DOI: 10.1007/978-3-031-10769-6_41.

[20] F. Frohn and J. Giesl. "Proving Non-Termination by Acceleration Driven Clause Learning (Short Paper)". In: *Proc. CADE '23*. LNCS 14132. 2023, pp. 220–233. DOI: 10.1007/978-3-031-38499-8_13.

[21] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. "The Termination and Complexity Competition". In: *Proc. TACAS '19*. LNCS 11429. Website of *TermComp*: https://termination-portal.org/wiki/Termination_Competition. 2019, pp. 156–166. DOI: 10.1007/978-3-030-17502-3_10.

[22] J. Giesl, N. Lommen, M. Hark, and F. Meyer. "Improving Automatic Complexity Analysis of Integer Programs". In: *The Logic of Software. A Tasting Menu of Formal Methods*. LNCS 13360. 2022, pp. 193–228. DOI: 10.1007/978-3-031-08166-8_10.

[23] M. A. T. Handley, N. Vazou, and G. Hutton. "Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2020). DOI: 10.1145/3371092.

[24] M. Hark, F. Frohn, and J. Giesl. "Polynomial Loops: Beyond Termination". In: *Proc. LPAR '20*. EPiC 73. 2020, pp. 279–297. DOI: 10.29007/nxv1.

[25] M. Hark, F. Frohn, and J. Giesl. "Termination of Triangular Polynomial Loops". In: *Formal Methods in System Design* 65.1 (2025), pp. 70–132. DOI: 10.1007/s10703-023-00440-z.

[26] M. Heizmann and J. Leike. "Ranking Templates for Linear Loops". In: *Logical Methods in Computer Science* 11.1 (2015). DOI: 10.2168/LMCS-11(1:16)2015.

[27] M. Henzinger, T. Henzinger, and P. Kopke. "Computing Simulations on Finite and Infinite Graphs". In: *Proc. FoCS '95*. 1995, pp. 453–462. DOI: 10.1109/SFCS.1995.492576.

[28] J. Hoffmann, K. Aehlig, and M. Hofmann. "Multivariate Amortized Resource Analysis". In: *ACM Transactions on Programming Languages and Systems* 34.3 (2012). DOI: 10.1145/2362389.2362393.

[29] J. Hoffmann, A. Das, and S.-C. Weng. "Towards Automatic Resource Bound Analysis for OCaml". In: *Proc. POPL '17*. 2017, pp. 359–373. DOI: 10.1145/3009837.3009842.

[30] J. Hoffmann and S. Jost. "Two Decades of Automatic Amortized Resource Analysis". In: *Mathematical Structures in Computer Science* 32.6 (2022), pp. 729–759. DOI: 10.1017/S0960129521000487.

[31] D. Ishimwe, K. Nguyen, and T. Nguyen. "Dynaplex: Analyzing Program Complexity using Dynamically Inferred Recurrence Relations". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021). DOI: 10.1145/3485515.

[32] B. Jeannet and A. Miné. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In: *Proc. CAV '09*. LNCS 5643. 2009, pp. 661–667. DOI: 10.1007/978-3-642-02658-4_52.

[33] KoAT *with Function Calls: Artifact, Evaluation, and Benchmarks*. Zenodo. 2025. DOI: 10.5281/zenodo.15586347.

[34] O. Kupferman and M. Y. Vardi. "An Automata-Theoretic Approach to Reasoning about Infinite-State Systems". In: *Proc. CAV '00*. LNCS 1855. 2000, pp. 36–52. DOI: 10.1007/10722167_7.

[35] N. Lommen, F. Meyer, and J. Giesl. "Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops". In: *Proc. IJCAR '22*. LNCS 13385. 2022, pp. 734–754. DOI: 10.1007/978-3-031-10769-6_43.

[36] N. Lommen and J. Giesl. "Targeting Completeness: Using Closed Forms for Size Bounds of Integer Programs". In: *Proc. FroCoS '23*. LNCS 14279. 2023, pp. 3–22. DOI: 10.1007/978-3-031-43369-6_1.

[37] N. Lommen, É. Meyer, and J. Giesl. "Control-Flow Refinement for Complexity Analysis of Probabilistic Programs in KoAT (Short Paper)". In: *Proc. IJCAR '24*. LNCS 14739. 2024, pp. 233–243. DOI: 10.1007/978-3-031-63498-7_14.

[38] N. Lommen, É. Meyer, and J. Giesl. "Targeting Completeness: Automated Complexity Analysis of Integer Programs". In: *CoRR* abs/2412.01832 (2024). DOI: 10.48550/arXiv.2412.01832.

[39] N. Lommen and J. Giesl. "AProVE (KoAT + LoAT)". In: *Proc. TACAS '25*. LNCS 15698. 2025, pp. 205–211. DOI: 10.1007/978-3-031-90660-2_13.

[40] N. Lommen and J. Giesl. "Modular Automatic Complexity Analysis of Recursive Integer Programs". In: *CoRR* abs/2512.18851 (2025). DOI: 10.48550/arXiv.2512.18851.

[41] N. Lommen and J. Giesl. *Experiments and Details for "Modular Automatic Complexity Analysis of Recursive Integer Programs"*. 2025. URL: https://koat.verify.rwth-aachen.de/function-calls.

[42] P. López-García, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo. "Interval-Based Resource Usage Verification by Translation into Horn Clauses and an Application to Energy Consumption". In: *Theory and Practice of Logic Programming* 18.2 (2018), pp. 167–223. DOI: 10.1017/S1471068418000042.

[43] L. M. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Proc. TACAS '08*. LNCS 4963. 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[44] M. Naaf, F. Frohn, M. Brockschmidt, C. Fuhs, and J. Giesl. "Complexity Analysis for Term Rewriting by Integer Transition Systems". In: *Proc. FroCoS '17*. LNCS 10483. 2017, pp. 132–150. DOI: 10.1007/978-3-319-66167-4_8.

[45] L. Pham, F. A. Saad, and J. Hoffmann. "Robust Resource Bounds with Static Analysis and Bayesian Inference". In: *Proceedings of the ACM on Programming Languages* 8.PLDI (2024). DOI: 10.1145/3656380.

[46] A. Podelski and A. Rybalchenko. "A Complete Method for the Synthesis of Linear Ranking Functions". In: *Proc. VMCAI '04*. LNCS 2937. 2004, pp. 239–251. DOI: 10.1007/978-3-540-24622-0_20.

[47] L. Rustenholz, M. Klemen, M. Á. Carreira-Perpiñán, and P. López-García. "A Machine Learning-Based Approach for Solving Recurrence Relations and its use in Cost Analysis of Logic Programs". In: *Theory and Practice of Logic Programming* 24.6 (2024), pp. 1163–1207. DOI: 10.1017/S1471068424000413.

[48] M. Sinn, F. Zuleger, and H. Veith. "Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints". In: *Journal of Automated Reasoning* 59.1 (2017), pp. 3–45. DOI: 10.1007/s10817-016-9402-4.

[49] *Software Verification Competition (*SV-COMP*)*. URL: https://sv-comp.sosy-lab.org/.

[50] *Termination Problems Data Base (*TPDB*)*. URL: https://github.com/TermCOMP/TPDB-ARI.