A

Analyzing Runtime and Size Complexity of Integer Programs

Marc Brockschmidt, Microsoft Research
Fabian Emmes, RWTH Aachen University
Stephan Falke, Karlsruhe Institute of Technology
Carsten Fuhs, Birkbeck, University of London
Jürgen Giesl, RWTH Aachen University

We present a modular approach to automatic complexity analysis of integer programs. Based on a novel alternation between finding symbolic time bounds for program parts and using these to infer bounds on the absolute values of program variables, we can restrict each analysis step to a small part of the program while maintaining a high level of precision. The bounds computed by our method are polynomial or exponential expressions that depend on the absolute values of input parameters.

We show how to extend our approach to arbitrary cost measures, allowing to use our technique to find upper bounds for other expended resources, such as network requests or memory consumption. Our contributions are implemented in the open source tool `KoAT`, and extensive experiments show the performance and power of our implementation in comparison with other tools.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.4 [**Software Engineering**]: Metrics; F.2.1 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity; F.3.1 [**Theory of Computation**]: Logics and Meanings of Programs

General Terms: Theory, Verification

Additional Key Words and Phrases: Runtime Complexity, Automated Complexity Analysis, Integer Programs

ACM Reference Format:
ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 48 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

There exist numerous methods to prove termination of imperative programs, e.g., [Podelski and Rybalchenko 2004; Bradley et al. 2005; Cook et al. 2006; Albert et al. 2008; Alias et al. 2010; Harris et al. 2010; Spoto et al. 2010; Falke et al. 2011; Tsitovich et al. 2011; Bagnara et al. 2012; Brockschmidt et al. 2012; Ben-Amram and Genaim 2013; Brockschmidt et al. 2013; Cook et al. 2013; Larraz et al. 2013; Heizmann et al. 2014]. In many cases, however, termination is not sufficient, but the program should also terminate in reasonable (e.g., (pseudo-)polynomial) time. To prove bounds on a program's *runtime complexity*, it is often crucial to also derive (possibly non-linear) bounds on the size of program variables, which may be modified repeatedly in loops.

Supported by the DFG grant GI 274/6-1, the Air Force Research Laboratory (AFRL), the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative, and the EPSRC.
Authors' addresses: M. Brockschmidt: Microsoft Research, Cambridge, UK, F. Emmes: LuFG Informatik 2, RWTH Aachen University, Germany, S. Falke: (Current address) aicas GmbH, Karlsruhe, Germany, C. Fuhs: Dept. of Computer Science and Information Systems, Birkbeck, University of London, UK, J. Giesl: LuFG Informatik 2, RWTH Aachen University, Germany

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

```
while i > 0 do                 while i > 0 do
  i := i − 1                     i := i − 1
                                 x := x + i
done                           done
while x > 0 do                 while x > 0 do
  x := x − 1                     x := x − 1
done                           done
```

Fig. 1. Two similar programs with different runtime

Our approach to find such bounds builds upon the well-known observation that polynomial ranking functions for termination proofs also provide a runtime complexity bound [Alias et al. 2010; Albert et al. 2011a; Albert et al. 2012; Avanzini and Moser 2013; Noschinski et al. 2013]. However, this only holds for proofs using a *single* polynomial ranking function. Larger programs are usually handled by a disjunctive [Lee et al. 2001; Cook et al. 2006; Tsitovich et al. 2011; Heizmann et al. 2014] or lexicographic [Bradley et al. 2005; Giesl et al. 2006; Fuhs et al. 2009; Alias et al. 2010; Harris et al. 2010; Falke et al. 2011; Brockschmidt et al. 2013; Cook et al. 2013; Larraz et al. 2013] combination of polynomial functions (we also refer to these components as "polynomial ranking functions"). Deriving a complexity bound in such cases is much harder.

Example 1.1. Both programs in Fig. 1 can be proven terminating using the lexicographic ranking function $\langle \mathtt{i}, \mathtt{x} \rangle$. However, the program without the instruction "$\mathtt{x} := \mathtt{x} + \mathtt{i}$" has linear runtime, while the program on the right has quadratic runtime. The crucial difference between the two programs is in the *size* of $\mathtt{x}$ after the first loop.

To handle such effects, we introduce a novel modular approach which *alternates* between finding *runtime bounds* and finding *size bounds*. In contrast to standard invariants, our size bounds express a relation to the size of the variables at the program start, where we measure the *size* of integers $m \in \mathbb{Z}$ by their absolute values $|m| \in \mathbb{N}$. Our method derives runtime bounds for isolated parts of the program and uses these to deduce (often non-linear) size bounds for program variables at certain locations. Further runtime bounds can then be inferred using size bounds for variables that were modified in preceding parts of the program. By splitting the analysis in this way, we only need to consider small program parts in each step, and the process is repeated until all loops and variables have been handled.

As an example, for the second program in Fig. 1, our method proves that the first loop is executed linearly often using the ranking function $\mathtt{i}$. Then, it deduces that $\mathtt{i}$ is bounded by the size $|\mathtt{i}_0|$ of its initial value $\mathtt{i}_0$ in all iterations of this loop. Combining these bounds, it infers that $\mathtt{x}$ is incremented by a value bounded by $|\mathtt{i}_0|$ at most $|\mathtt{i}_0|$ times in the first loop, i.e., $\mathtt{x}$ is bounded by the sum of its initial size $|\mathtt{x}_0|$ and $|\mathtt{i}_0|^2$. Finally, our method detects that the second loop is executed $\mathtt{x}$ times, and combines this with our bound $|\mathtt{x}_0| + |\mathtt{i}_0|^2$ on $\mathtt{x}$'s value when entering the second loop. In this way, we can infer the bound $|\mathtt{i}_0| + |\mathtt{x}_0| + |\mathtt{i}_0|^2$ for the program's runtime.[1] This novel combination of runtime and size bounds allows us to handle loops whose runtime depends on variables like $\mathtt{x}$ that were modified in earlier loops (where the values of these variables can also be modified in a non-linear way). Thus, our approach succeeds on many programs that are beyond the reach of previous techniques based on the use of ranking functions.

Sect. 2 introduces the basic notions for our approach. Then Sect. 3 and Sect. 4 present our techniques to compute runtime and size bounds, respectively. In Sect. 5, we extend our approach to handle possibly recursive procedure calls. Finally, we show in Sect. 6 how

[1]Since each step of our method over-approximates the runtime or size of a variable, we actually obtain the bound $2 + |\mathtt{i}_0| + \max\{|\mathtt{i}_0|, |\mathtt{x}_0|\} + |\mathtt{i}_0|^2$, cf. Sect. 4.2.

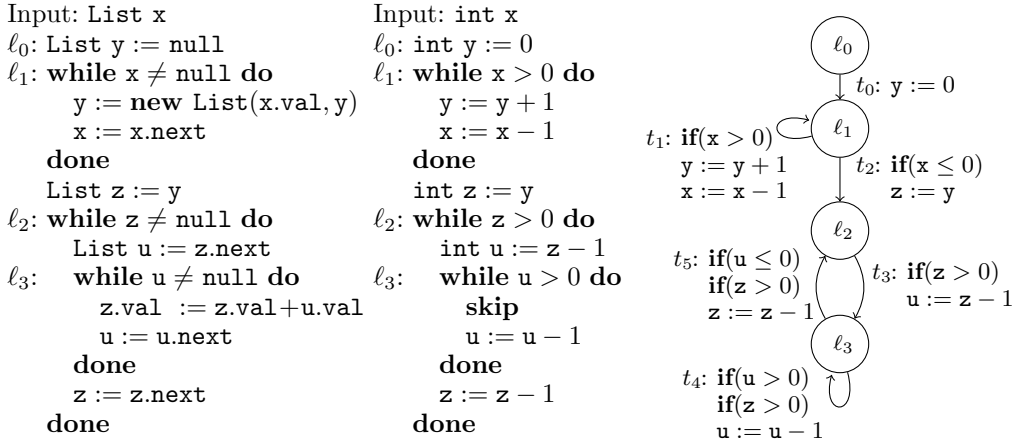


Fig. 2. List processing program, its integer abstraction, and a graph representation of the integer abstraction

a generalization to arbitrary cost measures can be used to obtain a modular analysis of procedures. Such cost measures can also express resource usage such as network requests. In Sect. 7, we compare our technique to related work and show its effectiveness in an extensive experimental evaluation. We conclude in Sect. 8, discussing limitations, possible further extensions, and applications of our method. All proofs are given in App. A.

A preliminary version of parts of this paper was published earlier [Brockschmidt et al. 2014]. It is extended substantially in the present paper:

— We present new techniques to automatically synthesize bounds for programs with an exponential growth of data sizes in Sect. 4.
— We extend our approach to programs with recursion in Sect. 5 and show how to also infer exponential runtime bounds for such programs.
— We generalize our technique to analyze complexity w.r.t. arbitrary cost measures in Sect. 6.1.
— We extend the modularity of our analysis such that program parts (e.g., library procedures) can be handled completely independently in Sect. 6.2.
— We integrated these new contributions in our prototype implementation `KoAT` and present an extensive evaluation, comparing it to recently developed competing tools in Sect. 7.3. `KoAT` is now also available as free software, allowing to easily experiment with extensions to our framework.
— We give detailed proofs for all theorems in App. A.

2. PRELIMINARIES

We regard sequential imperative integer programs with (potentially non-linear) arithmetic and unbounded non-determinism.

Example 2.1. In Fig. 2, a list processing program is shown on the left. For an input list `x`, the loop at location ℓ_1 creates a list `y` by reversing the elements of `x`. The loop at location ℓ_2 iterates over the list `y` and increases each element by the sum of its successors. So if `y` was $[5, 1, 3]$, it will be $[5 + 1 + 3, 1 + 3, 3]$ after the second loop.

In the middle of Fig. 2, an integer abstraction of our list program is shown. Here, list variables are replaced by integers that correspond to the length of the replaced list. Such integer abstractions can be obtained automatically using tools such as `COSTA` [Albert et al. 2008], `Julia` [Spoto et al. 2010], `Thor` [Magill et al. 2010], or `AProVE` [Giesl et al. 2014].

We fix a (finite) set of program variables $\mathcal{V} = \{v_1, \ldots, v_n\}$ and represent integer programs as directed graphs. Nodes are program *locations* $\mathcal{L}$ and edges are program *transitions* $\mathcal{T}$. The set $\mathcal{L}$ contains a *canonical start location* ℓ_0. W.l.o.g., we assume that no transition leads back to ℓ_0. All transitions originating in ℓ_0 are called *initial transitions*. The transitions are labeled by formulas over the variables $\mathcal{V}$ and primed post-variables $\mathcal{V}' = \{v'_1, \ldots, v'_n\}$ which represent the values of the variables after the transition. In the graph on the right of Fig. 2, we represented these formulas by sequences of instructions. For instance, t_3 is labeled by the formula $\mathtt{z} > 0 \wedge \mathtt{u}' = \mathtt{z} - 1 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{y}' = \mathtt{y} \wedge \mathtt{z}' = \mathtt{z}$. In our example, we used standard invariant-generation techniques (based on the Octagon domain [Miné 2006]) to propagate simple integer invariants, adding the condition $\mathtt{z} > 0$ to the transitions t_4 and t_5.

Definition 2.2 (*Programs*). A *transition* is a tuple (ℓ, τ, ℓ') where $\ell, \ell' \in \mathcal{L}$ are locations and τ is a quantifier-free formula relating the (pre-)variables $\mathcal{V}$ and the post-variables $\mathcal{V}'$. A *program* is a set of transitions $\mathcal{T}$. A *configuration* $(\ell, \boldsymbol{v})$ consists of a location $\ell \in \mathcal{L}$ and a *valuation* $\boldsymbol{v} : \mathcal{V} \to \mathbb{Z}$. We write $(\ell, \boldsymbol{v}) \to_t (\ell', \boldsymbol{v'})$ for an *evaluation step* with a transition $t = (\ell, \tau, \ell')$ iff the valuations $\boldsymbol{v}$, $\boldsymbol{v'}$ satisfy the formula τ of t. As usual, we say that $\boldsymbol{v}$, $\boldsymbol{v'}$ *satisfy* a quantifier-free formula τ over the variables $\mathcal{V} \cup \mathcal{V}'$ iff τ becomes *true* when every $v \in \mathcal{V}$ is instantiated by the number $\boldsymbol{v}(v)$ and every $v' \in \mathcal{V}'$ is instantiated by $\boldsymbol{v'}(v)$. We drop the index t in "$\to_t$" when that information is not important and write $(\ell, \boldsymbol{v}) \to^k (\ell', \boldsymbol{v'})$ if $(\ell', \boldsymbol{v'})$ is reached from $(\ell, \boldsymbol{v})$ in k evaluation steps.

For the program of Ex. 2.1, we have $(\ell_1, \boldsymbol{v}_1) \to_{t_2} (\ell_2, \boldsymbol{v}_2)$ for any valuations $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ where $\boldsymbol{v}_1(\mathtt{x}) = \boldsymbol{v}_2(\mathtt{x}) \leq 0$, $\boldsymbol{v}_1(\mathtt{y}) = \boldsymbol{v}_2(\mathtt{y}) = \boldsymbol{v}_2(\mathtt{z})$, and $\boldsymbol{v}_1(\mathtt{u}) = \boldsymbol{v}_2(\mathtt{u})$. Note that in our representation, every location can potentially be a "final" one (if none of its outgoing transitions is applicable for the current valuation).

Let $\mathcal{T}$ always denote the analyzed program. Our goal is to find bounds on the runtime and the sizes of program variables, where these bounds are expressed as functions in the sizes of the input variables $v_1, \ldots, v_n$. For our example, our approach will detect that its runtime is bounded by $3 + 4 \cdot |\mathtt{x}| + |\mathtt{x}|^2$ (i.e., it is quadratic in $|\mathtt{x}|$). We measure the *size* of variable values $\boldsymbol{v}(v_i)$ by their absolute values $|\boldsymbol{v}(v_i)|$. For a valuation $\boldsymbol{v}$ and a vector $\boldsymbol{m} = (m_1, \ldots, m_n) \in \mathbb{N}^n$, let $\boldsymbol{v} \leq \boldsymbol{m}$ abbreviate $|\boldsymbol{v}(v_1)| \leq m_1 \wedge \ldots \wedge |\boldsymbol{v}(v_n)| \leq m_n$. We define the *runtime complexity* of a program $\mathcal{T}$ by a function rc that maps sizes $\boldsymbol{m}$ of program variables to the maximal number of evaluation steps that are possible from an initial configuration $(\ell_0, \boldsymbol{v})$ with $\boldsymbol{v} \leq \boldsymbol{m}$.

Definition 2.3 (*Runtime Complexity*). The *runtime complexity* $\mathrm{rc} : \mathbb{N}^n \to \mathbb{N} \cup \{\omega\}$ of a program $\mathcal{T}$ is defined as $\mathrm{rc}(\boldsymbol{m}) = \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, \ell, \boldsymbol{v} \,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge (\ell_0, \boldsymbol{v}_0) \to^k (\ell, \boldsymbol{v})\}$.

Here, $\mathrm{rc}(\boldsymbol{m}) = \omega$ means non-termination or arbitrarily long runtime. Programs with arbitrarily long runtime can result from non-deterministic value assignment, e.g., $\mathtt{i} := \mathbf{nondet}()$; **while** $\mathtt{i} > 0$ **do** $\mathtt{i} := \mathtt{i} - 1$ **done**.

To analyze complexity in a modular way, we construct a *runtime approximation* $\mathcal{R}$ such that for any $t \in \mathcal{T}$, $\mathcal{R}(t)$ over-approximates the number of times that t can be used in an evaluation. As we generate new bounds by composing previously found bounds, we only use *weakly monotonic* functions $\mathcal{R}(t)$, i.e., where $m_i \geq m'_i$ implies $(\mathcal{R}(t))(m_1, \ldots, m_i, \ldots, m_n) \geq (\mathcal{R}(t))(m_1, \ldots, m'_i, \ldots, m_n))$. We define the set of *upper bounds* UB as the weakly monotonic functions from $\mathbb{N}^n \to \mathbb{N}$ and ω. Here, "ω" denotes the constant function which maps all arguments $\boldsymbol{m} \in \mathbb{N}^n$ to ω. We have $\omega > n$ for all $n \in \mathbb{N}$. In our implementation, we use a subset $\mathrm{UB}' \subsetneq \mathrm{UB}$ that is particularly suitable for the automated synthesis of bounds. We also allow an application of functions from UB' to integers instead of naturals by taking their absolute value. More precisely, UB' is the smallest set for which the following holds:

— $|v| \in \mathrm{UB}'$ for $v \in \mathcal{V}$ (variables)
— $\omega \in \mathrm{UB}'$ (unbounded function)
— $\sum_{1 \leq i \leq k} \left(a_i \cdot \prod_{1 \leq j \leq m_i} f_{i,j}\right) + a_{k+1} \in \mathrm{UB}'$ for $k, a_i, m_i \in \mathbb{N}$, $f_{i,j} \in \mathrm{UB}'$ (polynomials)

— $\max\{f_1,\ldots,f_k\} \in \mathrm{UB}'$, $\min\{f_1,\ldots,f_k\} \in \mathrm{UB}'$ for $f_1,\ldots,f_k \in \mathrm{UB}'$ (maximum and minimum)
— $k^f \in \mathrm{UB}'$ for $k \in \mathbb{N}$, $f \in \mathrm{UB}'$ (exponentials)

Thus, UB' is closed under addition, multiplication, maximum, minimum, and exponentiation (using natural numbers as bases). These closure properties will be used when combining bound approximations in the remainder of the paper.

We now formally define our notion of runtime approximations. Here, we use $\to^* \circ \to_t$ to denote the relation describing arbitrary many evaluation steps followed by a step with transition t.

Definition 2.4 (*Runtime Approximation*). A function $\mathcal{R} : \mathcal{T} \to \mathrm{UB}$ is a *runtime approximation* iff $(\mathcal{R}(t))(\boldsymbol{m}) \geq \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, \ell, \boldsymbol{v} \,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge (\ell_0, \boldsymbol{v}_0)\,(\to^* \circ \to_t)^k\,(\ell, \boldsymbol{v})\}$ holds for all transitions $t \in \mathcal{T}$ and all $\boldsymbol{m} \in \mathbb{N}^n$. We then say that $\mathcal{R}(t)$ is a *runtime bound* for the transition t. The *initial runtime approximation* $\mathcal{R}_0$ is defined as $\mathcal{R}_0(t) = 1$ for all initial transitions t and $\mathcal{R}_0(t) = \omega$ otherwise. Here, "1" denotes the constant function which maps all arguments $\boldsymbol{m} \in \mathbb{N}^n$ to 1.

We can combine the approximations for individual transitions represented by $\mathcal{R}(t)$ to obtain a bound for the runtime complexity rc of the whole program $\mathcal{T}$. Here for $f, g \in \mathrm{UB}$, the comparison, addition, multiplication, maximum, and the minimum are defined pointwise. So $f \geq g$ holds iff $f(\boldsymbol{m}) \geq g(\boldsymbol{m})$ for all $\boldsymbol{m} \in \mathbb{N}^n$ and $f + g$ is the function with $(f+g)(\boldsymbol{m}) = f(\boldsymbol{m}) + g(\boldsymbol{m})$, where $\omega + n = \omega$ for all $n \in \mathbb{N} \cup \{\omega\}$.

Remark 2.5 (*Approximating* rc). Let $\mathcal{R}$ be a runtime approximation for $\mathcal{T}$. Then $\sum_{t \in \mathcal{T}} \mathcal{R}(t) \geq \mathrm{rc}$.

The overall bound $\sum_{t \in \mathcal{T}} \mathcal{R}(t) = 3 + 4 \cdot |\mathtt{x}| + |\mathtt{x}|^2$ for the program in Ex. 2.1 was obtained in this way.

For *size complexity*, we analyze how large the value of a program variable can become. Analogous to $\mathcal{R}$, we use a *size approximation* $\mathcal{S}$, where $\mathcal{S}(t, v')$ is a bound on the size of the variable v *after* a certain transition t was used in an evaluation. For any transition $t \in \mathcal{T}$ and $v \in \mathcal{V}$, we call $|t, v'|$ a *result variable*. Later, we will build a *result variable graph* (RVG), whose nodes are result variables and whose edges represent the flow of data in our program.

Definition 2.6 (*Result Variables and Size Approximation*). Let $\mathsf{RV} = \{|t, v'| \mid t \in \mathcal{T}, v \in \mathcal{V}\}$ be the set of *result variables*. A function $\mathcal{S} : \mathsf{RV} \to \mathrm{UB}$ is a *size approximation* iff $(\mathcal{S}(t, v'))(\boldsymbol{m}) \geq \sup\{|\boldsymbol{v}(v)| \mid \exists \boldsymbol{v}_0, \ell, \boldsymbol{v} \,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge (\ell_0, \boldsymbol{v}_0)\,(\to^* \circ \to_t)\,(\ell, \boldsymbol{v})\}$ holds for all $|t, v'| \in \mathsf{RV}$ and all $\boldsymbol{m} \in \mathbb{N}^n$. We then say that $\mathcal{S}(t, v')$ is a *size bound* for the result variable $|t, v'|$. The *initial size approximation* $\mathcal{S}_0$ is defined as $\mathcal{S}_0(t, v') = \omega$ for all $|t, v'| \in \mathsf{RV}$. A pair $(\mathcal{R}, \mathcal{S})$ is a *complexity approximation* if $\mathcal{R}$ is a runtime approximation and $\mathcal{S}$ is a size approximation.

Our method performs an iterative refinement of runtime and size approximations. The general overall procedure is displayed in Fig. 3.[2] It starts with the initial approximations $\mathcal{R}_0, \mathcal{S}_0$ and then loops until time bounds for all transitions and size bounds for all result variables have been found. In each iteration, it first calls the sub-procedure TimeBounds (cf. Sect. 3) to improve the runtime bounds for those transitions $\mathcal{T}'$ for which we have no bound yet. Then, the procedure SizeBounds (cf. Sect. 4) is used

```
(R, S) := (R_0, S_0)
while there are t, v with R(t) = ω or S(t, v') = ω do
  T' := {t ∈ T | R(t) = ω}
  R  := TimeBounds(R, S, T')
  for all SCCs C of the RVG in topological order do
    S := SizeBounds(R, S, C)
  done
done
```

Fig. 3. Alternating complexity analysis procedure

[2] A refined version of this procedure will be presented in Fig. 12 of Sect. 7.2.

to obtain new size bounds, using the time bounds computed so far. It processes strongly connected components (SCCs) of the result variable graph, corresponding to sets of variables which influence each other.[3] In the next iteration of the main outer loop, TimeBounds can then use the newly obtained size bounds. In this way, the procedures for inferring runtime and size complexity alternate, allowing them to make use of each other's results. The procedure is aborted when in one iteration of the outer loop no new bounds could be found.

3. COMPUTING RUNTIME BOUNDS

To find runtime bounds automatically, we use *polynomial ranking functions* (PRFs). Such ranking functions are widely used in termination analysis and many techniques are available to generate PRFs automatically [Podelski and Rybalchenko 2004; Bradley et al. 2005; Fuhs et al. 2007; Fuhs et al. 2009; Alias et al. 2010; Falke et al. 2011; Bagnara et al. 2012; Ben-Amram and Genaim 2013; Leike and Heizmann 2014]. While most of these techniques only generate linear PRFs, the theorems of this section hold for general polynomial ranking functions as well. In our analysis framework, we repeatedly search for PRFs for different parts of the program, and the resulting combined complexity proof is analogous to the use of lexicographic combinations of ranking functions in termination analysis.

In Sect. 3.1 we recapitulate the basic approach to use PRFs for the generation of time bounds. In Sect. 3.2, we improve it to a novel modular approach which infers time bounds by combining PRFs with information about variable sizes and runtime bounds found earlier.

3.1. Runtime Bounds from Polynomial Ranking Functions

A PRF $\mathcal{P}ol : \mathcal{L} \to \mathbb{Z}[v_1, \ldots, v_n]$ assigns an integer polynomial $\mathcal{P}ol(\ell)$ over the program variables to each location ℓ. Then configurations $(\ell, \boldsymbol{v})$ are measured as the value of the polynomial $\mathcal{P}ol(\ell)$ for the numbers $\boldsymbol{v}(v_1), \ldots, \boldsymbol{v}(v_n)$. To obtain time bounds, we search for PRFs where no transition increases the measure of configurations, and at least one transition decreases it. To rule out that this decrease continues forever, we also require that the measure has a lower bound. As mentioned before Def. 2.2, here the formula τ of a transition (ℓ, τ, ℓ') may have been extended by suitable program invariants.

Definition 3.1 (*PRF*). We call $\mathcal{P}ol : \mathcal{L} \to \mathbb{Z}[v_1, \ldots, v_n]$ a *polynomial ranking function* (*PRF*) for $\mathcal{T}$ iff there is a non-empty $\mathcal{T}_\succ \subseteq \mathcal{T}$ such that the following holds:

- for all $(\ell, \tau, \ell') \in \mathcal{T}$, we have $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) \geq (\mathcal{P}ol(\ell'))(v'_1, \ldots, v'_n)$
- for all $(\ell, \tau, \ell') \in \mathcal{T}_\succ$, we have $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) > (\mathcal{P}ol(\ell'))(v'_1, \ldots, v'_n)$ and $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) \geq 1$

The constraints on a PRF $\mathcal{P}ol$ are the same as the constraints of Bradley et al. [2005] needed for finding ranking functions for termination proofs. Hence, this allows to re-use existing PRF synthesis techniques and tools. They imply that the transitions in $\mathcal{T}_\succ$ can only be used a limited number of times, as each application of a transition from $\mathcal{T}_\succ$ decreases the measure, and no transition increases it. Hence, if the program is called with input $m_1, \ldots, m_n$, no transition $t \in \mathcal{T}_\succ$ can be used more often than $(\mathcal{P}ol(\ell_0))(m_1, \ldots, m_n)$ times. Consequently, $\mathcal{P}ol(\ell_0)$ is a runtime bound for the transitions in $\mathcal{T}_\succ$. Note that no such bound is obtained for the remaining transitions in $\mathcal{T}$.

Example 3.2. To find bounds for the program in Ex. 2.1, we use $\mathcal{P}ol_1$ with $\mathcal{P}ol_1(\ell) = \mathtt{x}$ for all $\ell \in \mathcal{L}$, i.e., we measure configurations by the value of $\mathtt{x}$. No transition increases this measure and t_1 decreases it. The condition $\mathtt{x} > 0$ ensures that the measure is positive whenever t_1 is used, i.e., $\mathcal{T}_\succ = \{t_1\}$. Hence $\mathcal{P}ol_1(\ell_0)$ (i.e., the value $\mathtt{x}$ at the beginning of the program) is a bound on the number of times t_1 can be used.

[3]The result variable graph will be introduced in Sect. 4. As we will discuss in Sect. 4, proceeding in topological order avoids unnecessary computation steps.

Such PRFs lead to a basic technique for inferring time bounds. As mentioned in Sect. 2, to obtain a modular approach, we only allow weakly monotonic functions as complexity bounds. For any polynomial $p \in \mathbb{Z}[v_1, \ldots, v_n]$, let $[p]$ result from p by replacing all coefficients and variables with their absolute value (e.g., for $\mathcal{P}ol_1(\ell_0) = \mathtt{x}$ we have $[\mathcal{P}ol_1(\ell_0)] = |\mathtt{x}|$ and if $p = 2 \cdot v_1 - 3 \cdot v_2$ then $[p] = 2 \cdot |v_1| + 3 \cdot |v_2|$). As $[p](m_1, \ldots, m_n) \geq p(m_1, \ldots, m_n)$ holds for all $m_1, \ldots, m_n \in \mathbb{Z}$, this is a sound approximation, and $[p]$ is weakly monotonic. In our example, the initial runtime approximation $\mathcal{R}_0$ can now be refined to $\mathcal{R}_1$, with $\mathcal{R}_1(t_1) = [\mathcal{P}ol_1(\ell_0)] = |\mathtt{x}|$ and $\mathcal{R}_1(t) = \mathcal{R}_0(t)$ for all other transitions t. Thus, this provides a first basic method for the improvement of runtime approximations.

THEOREM 3.3 (COMPLEXITIES FROM PRFS). *Let $\mathcal{R}$ be a runtime approximation and $\mathcal{P}ol$ be a PRF for $\mathcal{T}$. Let $\mathcal{R}'(t) = [\mathcal{P}ol(\ell_0)]$ for all $t \in \mathcal{T}_{\succ}$ and $\mathcal{R}'(t) = \mathcal{R}(t)$ for all other $t \in \mathcal{T}$. Then, $\mathcal{R}'$ is also a runtime approximation.*

To ensure that $\mathcal{R}'(t)$ is at most as large as the previous bound $\mathcal{R}(t)$ in Thm. 3.3, one could also define $\mathcal{R}'(t) = \min\{[\mathcal{P}ol(\ell_0)], \mathcal{R}(t)\}$. A similar improvement is possible for all other techniques in the paper that refine the approximations $\mathcal{R}$ or $\mathcal{S}$.

3.2. Modular Runtime Bounds from PRFs and Size Bounds

Using Thm. 3.3 repeatedly to infer complexity bounds for all transitions in a program only succeeds for simple algorithms. In particular, it often fails for programs with non-linear runtime. Although corresponding SAT- and SMT-encodings exist [Fuhs et al. 2007], generating a suitable PRF $\mathcal{P}ol$ of a non-linear degree is a complex synthesis problem (and undecidable in general). This is aggravated by the need to consider all of $\mathcal{T}$ at once, which is required to check that no transition of $\mathcal{T}$ increases $\mathcal{P}ol$'s measure.

Therefore, we now present a new *modular* technique that only considers isolated program *parts* $\mathcal{T}' \subseteq \mathcal{T}$ in each PRF synthesis step. The bounds obtained from these "local" PRFs are then lifted to a bound expressed in the input values. To this end, we combine them with bounds on the size of the variables when entering the program part $\mathcal{T}'$ and with a bound on the number of times that $\mathcal{T}'$ can be reached in evaluations of the full program $\mathcal{T}$. This allows us to use existing efficient procedures for the automated generation of (often linear) PRFs for the analysis of programs with possibly non-linear runtime.

Example 3.4. We continue Ex. 3.2 and consider the *subset* $\mathcal{T}'_1 = \{t_1, \ldots, t_5\} \subseteq \mathcal{T}$. Using the constant PRF $\mathcal{P}ol_2$ with $\mathcal{P}ol_2(\ell_1) = 1$ and $\mathcal{P}ol_2(\ell_2) = \mathcal{P}ol_2(\ell_3) = 0$, we see that t_1, t_3, t_4, t_5 do not increase the measure of configurations and that t_2 decreases it. Hence, in executions that are *restricted to* $\mathcal{T}'_1$ and that start in ℓ_1, t_2 is used at most $[\mathcal{P}ol_2(\ell_1)] = 1$ times. To obtain a global result, we consider how often $\mathcal{T}'_1$ is reached in a full program run. As $\mathcal{T}'_1$ is only reached by the transition t_0, we *multiply* its runtime approximation $\mathcal{R}_1(t_0) = 1$ with the local bound $[\mathcal{P}ol_2(\ell_1)] = 1$ obtained for the sub-program $\mathcal{T}'_1$. Thus, we refine $\mathcal{R}_1$ to $\mathcal{R}_2(t_2) = \mathcal{R}_1(t_0) \cdot [\mathcal{P}ol_2(\ell_1)] = 1 \cdot 1 = 1$ and we set $\mathcal{R}_2(t) = \mathcal{R}_1(t)$ for all other t.[4]

In general, to estimate how often a sub-program $\mathcal{T}'$ is reached in an evaluation, we consider the transitions $\tilde{t} \in \mathcal{T} \setminus \mathcal{T}'$ that lead to the "entry location" ℓ of a transition from $\mathcal{T}'$. We *multiply* the runtime bound of such transitions $\tilde{t}$ (expressed in terms of the input variables) with the bound $[\mathcal{P}ol(\ell)]$ for runs starting in ℓ. This combination is an over-approximation

[4]The choice of $\mathcal{T}'_1 = \{t_1, \ldots, t_5\}$ does not quite match our procedure in Fig. 3, where we would only generate a PRF for those transitions $\mathcal{T}' = \{t \in \mathcal{T} \mid \mathcal{R}_1(t) = \omega\}$ for which we have no bound yet. So instead of $\mathcal{T}'_1$ we would use $\mathcal{T}''_1 = \{t_2, \ldots, t_5\}$. However, this would result in a worse upper bound, because $\mathcal{T}''_1$ is reached by both transitions t_0 and t_1. Thus, we would obtain $\mathcal{R}_2(t_2) = \mathcal{R}_1(t_0) \cdot [\mathcal{P}ol_2(\ell_1)] + \mathcal{R}_1(t_1) \cdot [\mathcal{P}ol_2(\ell_1)] = 1 + |\mathtt{x}|$. To obtain better bounds, our implementation uses an improved heuristic to choose $\mathcal{T}'$ in the procedure of Fig. 3: After generating a PRF $\mathcal{P}ol$ for the transitions $\mathcal{T}'$ without bounds, $\mathcal{T}'$ is extended by all further transitions $(\ell, \tau, \ell') \in \mathcal{T} \setminus \mathcal{T}'$ where $\mathcal{P}ol$ is weakly decreasing, i.e., where $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) \geq (\mathcal{P}ol(\ell'))(v'_1, \ldots, v'_n)$ holds. Thus, our implementation would extend $\mathcal{T}''_1$ to $\mathcal{T}'_1 = \mathcal{T}''_1 \cup \{t_1\}$.

of the overall runtime, as the runtime of individual runs may differ greatly. We present an improved treatment of this problem in Sect. 6.2. More generally, capturing such effects is treated in *amortized complexity analysis* (see, e.g., [Hoffmann et al. 2012; Sinn et al. 2014]). In our example, t_0 is the only transition leading to $\mathcal{T}'_1 = \{t_1, \ldots, t_5\}$ and thus, the runtime bound $\mathcal{R}_1(t_0) = 1$ is multiplied with $[\mathcal{P}ol_2(\ell_1)]$.

Example 3.5. We continue Ex. 3.4 and consider the transitions $\mathcal{T}'_2 = \{t_3, t_4, t_5\}$ for which we have found no bound yet. Then $\mathcal{P}ol_3(\ell_2) = \mathcal{P}ol_3(\ell_3) = \mathtt{z}$ is a PRF for $\mathcal{T}'_2$ with $(\mathcal{T}'_2)_\succ = \{t_5\}$. So *restricted to the sub-program* $\mathcal{T}'_2$, t_5 is used at most $[\mathcal{P}ol_3(\ell_2)] = |\mathtt{z}|$ times. Here, $\mathtt{z}$ refers to the value when entering $\mathcal{T}'_2$ (i.e., after transition t_2).

To translate this bound into an expression in the input values of the whole program $\mathcal{T}$, we substitute the variable $\mathtt{z}$ by its maximal size after using the transition t_2, i.e., by the *size bound* $\mathcal{S}(t_2, \mathtt{z}')$. As the runtime of the loop at ℓ_2 depends on the size of $\mathtt{z}$, our approach *alternates* between computing runtime and size bounds. Our method to compute size bounds will determine that the size of $\mathtt{z}$ after the transition t_2 is at most $|\mathtt{x}|$, cf. Sect. 4. Hence, we replace the variable $\mathtt{z}$ in $[\mathcal{P}ol_3(\ell_2)] = |\mathtt{z}|$ by $\mathcal{S}(t_2, \mathtt{z}') = |\mathtt{x}|$. To compute a global bound, we also have to examine how often $\mathcal{T}'_2$ can be executed in a full program run. As $\mathcal{T}'_2$ is only reached by t_2, we obtain $\mathcal{R}_3(t_5) = \mathcal{R}_2(t_2) \cdot |\mathtt{x}| = 1 \cdot |\mathtt{x}| = |\mathtt{x}|$. For all other transitions t, we again have $\mathcal{R}_3(t) = \mathcal{R}_2(t)$.

In general, the polynomials $[\mathcal{P}ol(\ell)]$ for the entry locations ℓ of $\mathcal{T}'$ only provide a bound in terms of the variable values at location ℓ. To find bounds expressed in the variable values at the start location ℓ_0, we use our *size approximation* $\mathcal{S}$ and replace all variables in $[\mathcal{P}ol(\ell)]$ by our approximation for their sizes at location ℓ. For this, we define the *application* of polynomials to functions. Let $p \in \mathbb{N}[v_1, \ldots, v_n]$ and $f_1, \ldots, f_n \in \mathrm{UB}$. Then $p(f_1, \ldots, f_n)$ is the function with $(p(f_1, \ldots, f_n))(\boldsymbol{m}) = p(f_1(\boldsymbol{m}), \ldots, f_n(\boldsymbol{m}))$ for all $\boldsymbol{m} \in \mathbb{N}^n$. Weak monotonicity of $p, f_1, \ldots, f_n$ also implies weak monotonicity of $p(f_1, \ldots, f_n)$, i.e., $p(f_1, \ldots, f_n) \in \mathrm{UB}$.

In Ex. 3.5, we applied the polynomial $[\mathcal{P}ol_3(\ell_2)]$ for the location ℓ_2 of $\mathcal{T}'_2$ to the size bounds $\mathcal{S}(t_2, v')$ for the variables $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$, $\mathtt{u}$ (i.e., to their sizes before entering $\mathcal{T}'_2$). As $[\mathcal{P}ol_3(\ell_2)] = |\mathtt{z}|$ and $\mathcal{S}(t_2, \mathtt{z}') = |\mathtt{x}|$, we obtained $[\mathcal{P}ol_3(\ell_2)](\mathcal{S}(t_2, \mathtt{x}'), \mathcal{S}(t_2, \mathtt{y}'), \mathcal{S}(t_2, \mathtt{z}'), \mathcal{S}(t_2, \mathtt{u}')) = |\mathtt{x}|$.

Our technique is formalized as the procedure TimeBounds in Thm. 3.6. It takes the current complexity approximation $(\mathcal{R}, \mathcal{S})$ and a sub-program $\mathcal{T}'$, and computes a PRF for $\mathcal{T}'$. Based on this, $\mathcal{R}$ is refined to the approximation $\mathcal{R}'$. In the following theorem, for any location ℓ, let $\mathcal{T}_\ell$ contain all transitions $(\tilde{\ell}, \tilde{\tau}, \ell) \in \mathcal{T} \setminus \mathcal{T}'$ leading to ℓ. Moreover, let $\mathcal{L}' = \{\ell \mid \mathcal{T}_\ell \neq \varnothing \wedge \exists \ell'.\, (\ell, \tau, \ell') \in \mathcal{T}'\}$ contain all entry locations of $\mathcal{T}'$.

THEOREM 3.6 (TimeBounds). *Let $(\mathcal{R}, \mathcal{S})$ be a complexity approximation, let $\mathcal{T}' \subseteq \mathcal{T}$ such that $\mathcal{T}'$ contains no initial transitions, and let $\mathcal{P}ol$ be a PRF for $\mathcal{T}'$. Let $\mathcal{R}'(t) = \sum_{\ell \in \mathcal{L}', \tilde{t} \in \mathcal{T}_\ell} \mathcal{R}(\tilde{t}) \cdot [\mathcal{P}ol(\ell)](\mathcal{S}(\tilde{t}, v'_1), \ldots, \mathcal{S}(\tilde{t}, v'_n))$ for $t \in \mathcal{T}'_\succ$ and $\mathcal{R}'(t) = \mathcal{R}(t)$ for all $t \in \mathcal{T} \setminus \mathcal{T}'_\succ$. Then, $\mathsf{TimeBounds}(\mathcal{R}, \mathcal{S}, \mathcal{T}') = \mathcal{R}'$ is also a runtime approximation.*

Here one can see why we require complexity bounds to be weakly monotonic. The reason is that $\mathcal{S}(\tilde{t}, v')$ over-approximates the size of v at some location ℓ. Hence, to ensure that $[\mathcal{P}ol(\ell)](\mathcal{S}(\tilde{t}, v'_1), \ldots, \mathcal{S}(\tilde{t}, v'_n))$ correctly over-approximates how often transitions of $\mathcal{T}'_\succ$ can be applied in parts of evaluations that only use transitions from $\mathcal{T}'$, $[\mathcal{P}ol(\ell)]$ must be weakly monotonic.

Example 3.7. We use Thm. 3.6 to obtain bounds for the transitions not handled in Ex. 3.5. For $\mathcal{T}'_3 = \{t_3, t_4\}$, we use $\mathcal{P}ol_4(\ell_2) = 1$, $\mathcal{P}ol_4(\ell_3) = 0$, and hence $(\mathcal{T}'_3)_\succ = \{t_3\}$. The transitions t_2 and t_5 lead to $\mathcal{T}'_3$, and thus, we obtain $\mathcal{R}_4(t_3) = \mathcal{R}_3(t_2) \cdot 1 + \mathcal{R}_3(t_5) \cdot 1 = 1 + |\mathtt{x}|$ and $\mathcal{R}_4(t) = \mathcal{R}_3(t)$ for all other transitions t.

For $\mathcal{T}'_4 = \{t_4\}$, we use $\mathcal{P}ol_5(\ell_3) = \mathtt{u}$ with $(\mathcal{T}'_4)_\succ = \mathcal{T}'_4$. The part $\mathcal{T}'_4$ is only entered by the transition t_3. To get a global bound, we substitute $\mathtt{u}$ in $[\mathcal{P}ol_5(\ell_3)] = |\mathtt{u}|$ by $\mathcal{S}(t_3, \mathtt{u}')$ (in Sect. 4,

we will determine $\mathcal{S}(t_3, \mathtt{u}') = |\mathtt{x}|$). Thus, $\mathcal{R}_5(t_4) = \mathcal{R}_4(t_3) \cdot \mathcal{S}(t_3, \mathtt{u}') = (1 + |\mathtt{x}|) \cdot |\mathtt{x}| = |\mathtt{x}| + |\mathtt{x}|^2$ and $\mathcal{R}_5(t) = \mathcal{R}_4(t)$ for all other $t \in \mathcal{T}$. So while the runtime of $\mathcal{T}'_4$ on its own is linear, the loop at location ℓ_3 is reached a linear number of times, i.e., its transition t_4 is used *quadratically* often. Thus, the overall program runtime is bounded by $\sum_{t \in \mathcal{T}} \mathcal{R}_5(t) = 3 + 4 \cdot |\mathtt{x}| + |\mathtt{x}|^2$.

4. COMPUTING SIZE BOUNDS

The procedure TimeBounds improves the runtime approximation $\mathcal{R}$, but up to now the size approximation $\mathcal{S}$ was only used as an input. To infer bounds on the size of variables, we proceed in three steps. First, we find *local size bounds* that approximate the effect of a single transition on the sizes of variables. Then, we construct a *result variable graph* that makes the flow of data between variables explicit. Finally, we analyze each strongly connected component (SCC) of this graph independently. Here, we combine our runtime approximation $\mathcal{R}$ with the local size bounds to estimate how often transitions modify a variable value.

To describe how the size of a post-variable v' is related to the pre-variables of a transition t, we use local size bounds $\mathcal{S}_l(t, v')$.[5] Thus, $\mathcal{S}(t, v')$ is a bound on the size of v after using t expressed in the sizes of the program inputs, and $\mathcal{S}_l(t, v')$ is a bound expressed in the sizes of the pre-variables of t. In most cases, such local size bounds can be inferred directly from the formula of the transition (e.g., if it contains sub-formulas such as $\mathtt{x}' = \mathtt{x} + 1$). For the remaining cases, we use SMT solving to find bounds from certain classes of templates (cf. Sect. 4.2).

Definition 4.1 (*Local Size Approximation*). We call $\mathcal{S}_l : \mathsf{RV} \to \mathrm{UB}$ a *local size approximation* iff $(\mathcal{S}_l(t, v'))(\boldsymbol{m}) \geq \sup\{|\boldsymbol{v}'(v)| \mid \exists \ell, \boldsymbol{v}, \ell', \boldsymbol{v}' . \boldsymbol{v} \leq \boldsymbol{m} \wedge (\ell, \boldsymbol{v}) \to_t (\ell', \boldsymbol{v}')\}$ for all $|t, v'| \in \mathsf{RV}$ and all $\boldsymbol{m} \in \mathbb{N}^n$.

Example 4.2. For the program of Ex. 2.1, we have $\mathcal{S}_l(t_1, \mathtt{y}') = |\mathtt{y}| + 1$, as t_1 increases $\mathtt{y}$ by 1. Similarly, $|t_1, \mathtt{x}'|$ is bounded by $|\mathtt{x}|$ as t_1 is only executed if $\mathtt{x}$ is positive and thus decreasing $\mathtt{x}$ by 1 does not increase its *absolute* value.

To track how variables influence each other, we construct a result variable graph (RVG) whose nodes are the result variables. The RVG has an edge from a result variable $|\tilde{t}, \tilde{v}'|$ to $|t, v'|$ if the transition $\tilde{t}$ can be used immediately before t and if $\tilde{v}$ occurs in the local size bound $\mathcal{S}_l(t, v')$. Such an edge means that the size of $\tilde{v}'$ in the post-location of the transition $\tilde{t}$ may influence the size of v' in t's post-location.

To state which variables may influence a function $f \in \mathrm{UB}$, we define its *active variables* as $\mathsf{actV}(f) = \{v_i \in \mathcal{V} \mid \exists m_1, \ldots, m_n, m'_i \in \mathbb{N} . f(m_1, \ldots, m_i, \ldots, m_n) \neq f(m_1, \ldots, m'_i, \ldots, m_n)\}$. To compute $\mathsf{actV}(f)$ for the upper bounds $f \in \mathrm{UB}'$ in our implementation, we simply take all variables occurring in f.

Let $\mathsf{pre}(t)$ denote the transitions that may precede t in evaluations, i.e., $\mathsf{pre}(t) = \{\tilde{t} \in \mathcal{T} \mid \exists \boldsymbol{v}_0, \ell, \boldsymbol{v} . (\ell_0, \boldsymbol{v}_0) \to^* \circ \to_{\tilde{t}} \circ \to_t (\ell, \boldsymbol{v})\}$. While $\mathsf{pre}(t)$ is undecidable in general, there exist several techniques to compute over-approximations, cf. [Fuhs et al. 2009; Falke et al. 2011].

Definition 4.3 (*RVG*). Let $\mathcal{S}_l$ be a local size approximation. An *RVG* has $\mathcal{T}$'s result variables as nodes and the edges $\{(|\tilde{t}, \tilde{v}'|, |t, v'|) \mid \tilde{t} \in \mathsf{pre}(t), \tilde{v} \in \mathsf{actV}(\mathcal{S}_l(t, v'))\}$.

Example 4.4. The RVG for the program from Ex. 2.1 is shown in Fig. 4. Here, we display local size bounds in the RVG to the left of the result variables, separated by "$\geq$" (e.g., "$|\mathtt{x}| \geq |t_1, \mathtt{x}'|$" means $\mathcal{S}_l(t_1, \mathtt{x}') = |\mathtt{x}|$). As we have $\mathcal{S}_l(t_2, \mathtt{z}') = |\mathtt{y}|$ for the transition t_2, which sets $\mathtt{z} := \mathtt{y}$, we conclude $\mathsf{actV}(\mathcal{S}_l(t_2, \mathtt{z}')) = \{\mathtt{y}\}$. The program graph implies $\mathsf{pre}(t_2) = \{t_0, t_1\}$, and thus, the RVG contains edges from $|t_0, \mathtt{y}'|$ to $|t_2, \mathtt{z}'|$ and from $|t_1, \mathtt{y}'|$ to $|t_2, \mathtt{z}'|$.

[5] So the subscript "l" in $\mathcal{S}_l$ stands for local size bounds and should not be confused with locations "ℓ".

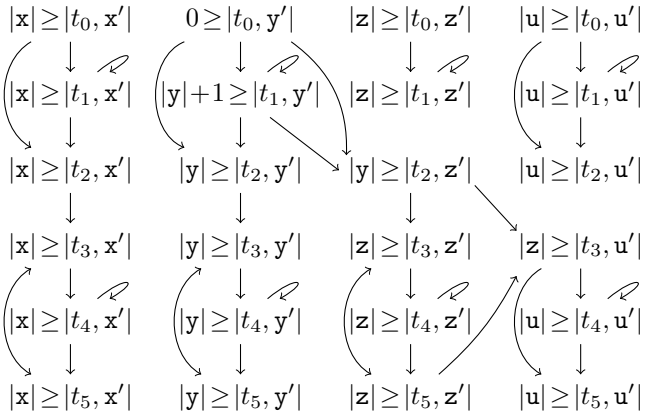


Fig. 4. Result variable graph for the program from Ex. 2.1

SCCs of the RVG represent sets of result variables that may influence each other. To lift the local approximation $\mathcal{S}_l$ to a global one, we consider each SCC on its own. We treat the SCCs in topological order, reflecting the data flow. In this way when computing size bounds for an SCC, we already have the size bounds available for those result variables that the current SCC depends on. As usual, an 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. In Sect. 4.1, we show how to deduce global bounds for trivial SCCs and in Sect. 4.2, we handle non-trivial SCCs where transitions are applied repeatedly.

4.1. Size Bounds for Trivial SCCs of the RVG

$\mathcal{S}_l(t, v')$ approximates the size of v' after the transition t w.r.t. t's pre-variables, but our goal is to obtain a *global* bound $\mathcal{S}(t, v')$ that approximates v' *w.r.t. the initial values* of the variables at the program start. For trivial SCCs that consist of a result variable $\alpha = |t, v'|$ with an initial transition t, the local bound $\mathcal{S}_l(\alpha)$ is also the global bound $\mathcal{S}(\alpha)$, as the start location ℓ_0 has no incoming transitions.

Next, we consider trivial SCCs $\alpha = |t, v'|$ with incoming edges from other SCCs. Here, $\mathcal{S}_l(\alpha)\,(\boldsymbol{m})$ is an upper bound on the size of v' after using the transition t in a configuration where the sizes of the variables are at most $\boldsymbol{m}$. To obtain a global bound, we replace $\boldsymbol{m}$ by upper bounds on t's input variables. The edges leading to α come from result variables $|\tilde{t}, v_i'|$ where $\tilde{t} \in \mathsf{pre}(t)$ and thus, a bound for the result variable $\alpha = |t, v'|$ is the maximum of all applications of $\mathcal{S}_l(\alpha)$ to $\mathcal{S}(\tilde{t}, v_1'), \ldots, \mathcal{S}(\tilde{t}, v_n')$, for all $\tilde{t} \in \mathsf{pre}(t)$.

Example 4.5. Consider the RVG of Ex. 4.4 in Fig. 4, for which we want to find size bounds. For the trivial SCC $\{|t_0, \mathtt{y}'|\}$, we have $0 \geq |t_0, \mathtt{y}'|$, and thus we set $\mathcal{S}(t_0, \mathtt{y}') = 0$. Similarly, we obtain $\mathcal{S}(t_0, \mathtt{x}') = |\mathtt{x}|$. Finally, consider the local size bound $|\mathtt{y}| \geq |t_2, \mathtt{z}'|$. To express this bound in terms of the input variables, we consider the predecessors $|t_0, \mathtt{y}'|$ and $|t_1, \mathtt{y}'|$ of $|t_2, \mathtt{z}'|$. So $\mathcal{S}_l(t_2, \mathtt{z}')$ must be applied to $\mathcal{S}(t_0, \mathtt{y}')$ and $\mathcal{S}(t_1, \mathtt{y}')$. If SCCs are handled in topological order, one already knows that $\mathcal{S}(t_0, \mathtt{y}') = 0$ and $\mathcal{S}(t_1, \mathtt{y}') = |\mathtt{x}|$. Hence, $\mathcal{S}(t_2, \mathtt{z}') = \max\{0, |\mathtt{x}|\} = |\mathtt{x}|$.

Thm. 4.6 presents the resulting procedure SizeBounds. Based on the current approximation $(\mathcal{R}, \mathcal{S})$, it improves the global size bound for the result variable in a trivial SCC of the RVG. Non-trivial SCCs will be handled in Thm. 4.15.

THEOREM 4.6 (SizeBounds FOR TRIVIAL SCCs). *Let $(\mathcal{R}, \mathcal{S})$ be a complexity approximation, let $\mathcal{S}_l$ be a local size approximation, and let $\{\alpha\} \subseteq \mathsf{RV}$ be a trivial SCC of the RVG. We define $\mathcal{S}'(\alpha') = \mathcal{S}(\alpha')$ for $\alpha' \neq \alpha$ and*

- *$\mathcal{S}'(\alpha) = \mathcal{S}_l(\alpha)$, if $\alpha = |t, v'|$ for some initial transition t*
- *$\mathcal{S}'(\alpha) = \max\{\ \mathcal{S}_l(\alpha)\,(\mathcal{S}(\tilde{t}, v'_1), \ldots, \mathcal{S}(\tilde{t}, v'_n))\ \mid \tilde{t} \in \mathsf{pre}(t)\}$, otherwise.*

Then $\mathsf{SizeBounds}(\mathcal{R}, \mathcal{S}, \{\alpha\}) = \mathcal{S}'$ is also a size approximation.

4.2. Size Bounds for Non-Trivial SCCs of the RVG

Finally, we show how to improve the size bounds for result variables in non-trivial SCCs of the RVG. Such an SCC C corresponds to the information flow in a loop and hence, each of its *local* changes can be applied several times. To approximate the overall effect of these repeated changes for a transition t of C, we combine the time bound $\mathcal{R}(t)$ with the local size bounds $\mathcal{S}_l(t, v')$. Then, to approximate the effect of the whole SCC C, we combine the bounds obtained for all transitions t of C. To simplify this approximation, we classify result variables α into three sets $\doteq$, $\dot{+}$, and $\dot{\times}$, depending on their local size bound $\mathcal{S}_l(\alpha)$:

- $\alpha \in \doteq$ (α is an "equality") if the result variable is not larger than any of its pre-variables or a constant, i.e., iff there is a number $c_\alpha \in \mathbb{N}$ with $\max\{c_\alpha, m_1, \ldots, m_n\} \geq (\mathcal{S}_l(\alpha))(m_1, \ldots, m_n)$ for all $m_1, \ldots, m_n \in \mathbb{N}$.
- $\alpha \in \dot{+}$ (α "adds a constant") if the result variable increases over the pre-variables by a constant, i.e., iff there is a number $d_\alpha \in \mathbb{N}$ with $d_\alpha + \max\{m_1, \ldots, m_n\} \geq (\mathcal{S}_l(\alpha))(m_1, \ldots, m_n)$ for all $m_1, \ldots, m_n \in \mathbb{N}$.
- $\alpha \in \dot{\times}$ (α is a "scaled sum") if the result variable is not larger than the sum of the pre-variables and a constant multiplied by a scaling factor, i.e., iff there are numbers $s_\alpha, e_\alpha \in \mathbb{N}$ with $s_\alpha \geq 1$ and $s_\alpha \cdot (e_\alpha + \sum_{i \in \{1,\ldots,n\}} m_i) \geq (\mathcal{S}_l(\alpha))(m_1, \ldots, m_n)$ for all $m_1, \ldots, m_n \in \mathbb{N}$.

Example 4.7. We continue Ex. 4.5, and obtain $\{|t_3, \mathtt{z}'|, |t_4, \mathtt{z}'|, |t_5, \mathtt{z}'|\} \subseteq \doteq$ since $\mathcal{S}_l(t_3, \mathtt{z}') = \mathcal{S}_l(t_4, \mathtt{z}') = \mathcal{S}_l(t_5, \mathtt{z}') = |\mathtt{z}|$. Similarly, we have $|t_1, \mathtt{y}'| \in \dot{+}$ as $\mathcal{S}_l(t_1, \mathtt{y}') = |\mathtt{y}| + 1$. A result variable with a local size bound like $|\mathtt{x}| + |\mathtt{y}|$ or $2 \cdot |\mathtt{x}|$ would belong to the class $\dot{\times}$.

Note that local size bounds from $\dot{\times}$ can lead to an exponential global size bound. For example, if a change bounded by $2 \cdot |\mathtt{x}|$ is applied $|\mathtt{y}|$ times to $\mathtt{x}$, the resulting value is bounded only by the exponential function $2^{|\mathtt{y}|} \cdot |\mathtt{x}|$. For each result variable α, we use a series of SMT queries in order to try to find a local size bound $\mathcal{S}_l(\alpha)$ belonging to one of our three classes above.[6]

Similar to $\mathsf{pre}(t)$ for a transition t, let $\mathsf{pre}(\alpha)$ for a result variable α be those $\tilde{\alpha} \in \mathsf{RV}$ with an edge from $\tilde{\alpha}$ to α in the RVG. To deduce a bound on the size of the result variables in an SCC C, we first consider the size of values *entering* C. Hence, we require that the resulting size bound $\mathcal{S}(\beta)$ for $\beta \in C$ should be at least as large as the sizes $\mathcal{S}(\tilde{\alpha})$ of the *inputs* $\tilde{\alpha}$, i.e., of those result variables $\tilde{\alpha}$ outside the SCC C that have an edge to some $\alpha \in C$. Moreover, if the SCC C contains result variables $\alpha = |t, v'| \in \doteq$, then the transition t either does not increase the size at all, or increases it to the constant c_α. Hence, the bound $\mathcal{S}(\beta)$ for the result variables β in C should also be at least $\max\{c_\alpha \mid \alpha \in \doteq \cap\, C\}$. As in Def. 2.4, a constant like "c_α" denotes the constant function mapping all values from $\mathbb{N}^n$ to c_α.

Example 4.8. The only predecessor of the SCC $C = \{|t_3, \mathtt{z}'|, |t_4, \mathtt{z}'|, |t_5, \mathtt{z}'|\}$ in the RVG of Fig. 4 is $|t_2, \mathtt{z}'|$ with $\mathcal{S}(t_2, \mathtt{z}') = |\mathtt{x}|$. For each $\alpha \in C$, the corresponding constant c_α is 0. Thus, for all $\beta \in C$ we obtain $\mathcal{S}(\beta) = \max\{|\mathtt{x}|, 0\} = |\mathtt{x}|$.

[6] More precisely, our implementation uses an SMT solver to check whether a result variable (i.e., a term with the variables $v_1, \ldots, v_n$) is bounded by $\max\{c, v_1, \ldots, v_n\}$, by $d + \max\{v_1, \ldots, v_n\}$, or by $s \cdot (e + \sum_{i \in \{1,\ldots,n\}} v_i)$ for large fixed numbers c, d, e, s. Afterwards, the set of variables $v_1, \ldots, v_n$ in the bound is reduced (i.e., one checks whether the bound still works without v_i) and the numbers c, d, e, s are reduced by binary search.

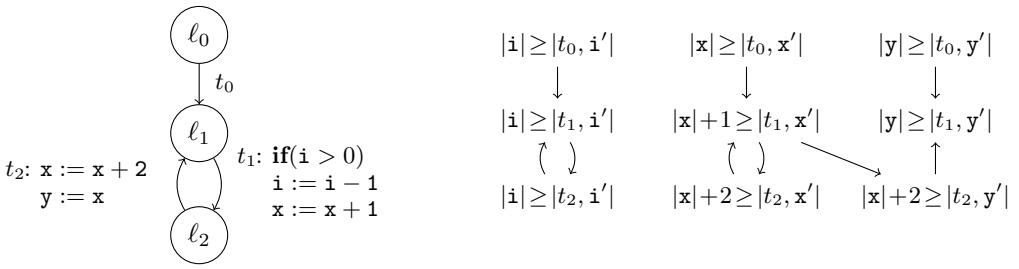


Fig. 5. Graph and RVG for the program of Ex. 4.10

To handle result variables $\alpha \in (\dot{+} \setminus \doteq) \cap C$ that add a constant d_α, we consider how often this addition is performed. Thus, while TimeBounds from Thm. 3.6 uses the size approximation $\mathcal{S}$ to improve the runtime approximation $\mathcal{R}$, SizeBounds uses $\mathcal{R}$ to improve $\mathcal{S}$. For each transition $t \in \mathcal{T}$, let $C_t = C \cap \{|t, v'| \mid v \in \mathcal{V}\}$ be all result variables from C that use the transition t. Since $\mathcal{R}(t)$ is a bound on the number of times that t is executed, the repeated traversal of t increases the overall size by at most $\mathcal{R}(t) \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \doteq) \cap C_t\}$.

Example 4.9. Consider the result variable $\alpha = |t_1, \mathtt{y}'|$ in the RVG of Fig. 4. Its local size bound is $\mathcal{S}_l(t_1, \mathtt{y}') = |\mathtt{y}| + 1$, i.e., each traversal of t_1 increases $\mathtt{y}$ by $d_\alpha = 1$. As before, we use the size bounds on the predecessors of the SCC $\{\alpha\}$ as a basis. The input value when entering this (non-trivial) SCC is $\mathcal{S}(t_0, \mathtt{y}') = 0$. Since t_1 is executed at most $\mathcal{R}(t_1) = |\mathtt{x}|$ times, we obtain the global bound $\mathcal{S}(\alpha) = \mathcal{S}(t_0, \mathtt{y}') + \mathcal{R}(\alpha) \cdot d_\alpha = 0 + |\mathtt{x}| \cdot 1 = |\mathtt{x}|$.

If the SCC C uses several different transitions from $(\dot{+} \setminus \doteq)$, then the effects resulting from these transitions have to be *added*, i.e., the repeated traversal of these transitions increases the overall size by at most $\sum_{t \in \mathcal{T}}(\mathcal{R}(t) \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \doteq) \cap C_t\})$. Note that we use the same size bound for all result variables of an SCC, since the transitions in an SCC may influence all its result variables. So for an SCC C where all result variables are in the classes $\doteq$ or $\dot{+}$, we obtain the following new size bound for all result variables of C:

$$\Big(\max(\, \{\mathcal{S}(\tilde{\alpha}) \mid \text{ there is an } \alpha \in C \text{ with } \tilde{\alpha} \in \mathsf{pre}(\alpha) \setminus C\} \,\cup\, \{c_\alpha \mid \alpha \in \doteq \cap\, C\}\,) \\ + \textstyle\sum_{t \in \mathcal{T}}(\mathcal{R}(t) \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \doteq) \cap C_t\}) \Big)$$

Example 4.10. As an example for a program with several transitions that add constants, consider the program in Fig. 5. We are interested in determining a bound for the SCC $C = \{|t_1, \mathtt{x}'|, |t_2, \mathtt{x}'|\}$, and assume that $\mathcal{R}(t_1) = \mathcal{R}(t_2) = |\mathtt{i}|$ has already been inferred. Using Thm. 4.6, we obtain $\mathcal{S}(t_0, \mathtt{x}') = |\mathtt{x}|$. Then, we can derive:

$$\begin{aligned} \mathcal{S}(t_1, \mathtt{x}') = \ \mathcal{S}(t_2, \mathtt{x}') &= \Big(\max\{\mathcal{S}(t_0, \mathtt{x}')\} + \textstyle\sum_{t \in \mathcal{T}}(\mathcal{R}(t) \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \doteq) \cap C_t\}) \Big) \\ &= \Big(\max\{|\mathtt{x}|\} + (\mathcal{R}(t_1) \cdot \max\{1\}) + (\mathcal{R}(t_2) \cdot \max\{2\}) \Big) \\ &= \Big(|\mathtt{x}| + (|\mathtt{i}| \cdot 1) + (|\mathtt{i}| \cdot 2) \Big) \\ &= |\mathtt{x}| + 3 \cdot |\mathtt{i}| \end{aligned}$$

This bound is also used for the variable $\mathtt{y}$ that is modified in the same loop of the program. The reason is that for the trivial SCC $\{|t_2, \mathtt{y}'|\}$ of the RVG, by Thm. 4.6 we have $\mathcal{S}(t_2, \mathtt{y}') = \mathcal{S}_l(t_2, \mathtt{y}')\,(\mathcal{S}(t_1, \mathtt{i}'), \mathcal{S}(t_1, \mathtt{x}'), \mathcal{S}(t_1, \mathtt{y}'))$. As $\mathcal{S}_l(t_2, \mathtt{y}') = |\mathtt{x}| + 2$ and $\mathcal{S}(t_1, \mathtt{x}') = |\mathtt{x}| + 3 \cdot |\mathtt{i}|$, we get $\mathcal{S}(t_2, \mathtt{y}') = |\mathtt{x}| + 3 \cdot |\mathtt{i}| + 2$.

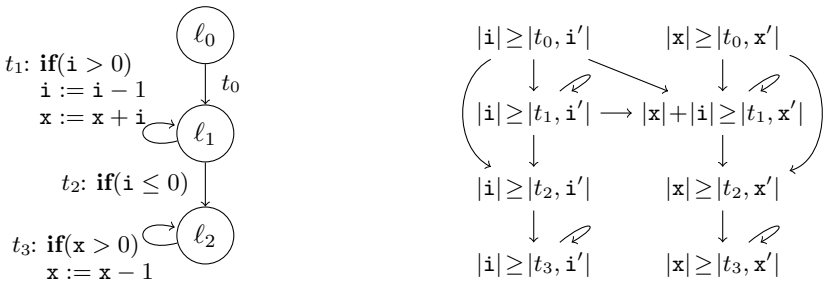


Fig. 6. Graph and RVG for the second program of Fig. 1

Finally, we discuss how to handle result variables $\alpha \in \dot{\times} \setminus \dot{+}$ that sum and scale up several program variables. For any such α in a non-trivial SCC C of the RVG, let $\mathcal{V}_\alpha = \{\tilde{v} \mid |\tilde{t}, \tilde{v}'| \in \mathsf{pre}(\alpha) \cap C\}$ be the set of those program variables $\tilde{v}$ for which there is a result variable $|\tilde{t}, \tilde{v}'|$ in C with an edge to α. For non-trivial SCCs C, we always have $\mathcal{V}_\alpha \neq \varnothing$. We first consider the case where the scaling factor is $s_\alpha = 1$ and where $|\mathcal{V}_\alpha| = 1$ holds, i.e., where no two result variables $|\hat{t}, \hat{v}'|, |\tilde{t}, \tilde{v}'|$ in α's SCC C with $\hat{v} \neq \tilde{v}$ have edges to α. However, there may be incoming edges from arbitrary result variables *outside* the SCC. Then, α may sum up several program variables, but only *one* of them comes from α's own SCC C in the RVG.

For each variable v, let f_v^α be an upper bound on the size of those result variables $|\tilde{t}, v'| \notin C$ that have edges to α, i.e., $f_v^\alpha = \max\{\mathcal{S}(\tilde{t}, v') \mid |\tilde{t}, v'| \in \mathsf{pre}(\alpha) \setminus C\}$. The execution of α's transition t then means that the value of the variable in $\mathcal{V}_\alpha$ can be increased by adding f_v^α (for all $v \in \mathsf{actV}(\mathcal{S}_l(\alpha)) \setminus \mathcal{V}_\alpha$) plus the constant e_α. Again, this can be repeated at most $\mathcal{R}(t)$ times. The overall size is bounded by adding $\mathcal{R}(t) \cdot \max\{e_\alpha + \sum_{v \in \mathsf{actV}(\mathcal{S}_l(\alpha)) \setminus \mathcal{V}_\alpha} f_v^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\}$.

Example 4.11. Consider the second program of Fig. 1 from Ex. 1.1 again. Its program graph and RVG are depicted in Fig. 6. Our method detects the runtime bounds $\mathcal{R}(t_0) = 1$, $\mathcal{R}(t_1) = |\mathtt{i}|$, and $\mathcal{R}(t_2) = 1$. To obtain size bounds, we first infer the global size bounds $\mathcal{S}(t, \mathtt{i}') = |\mathtt{i}|$ for all $t \in \mathcal{T}$ and $\mathcal{S}(t_0, \mathtt{x}') = |\mathtt{x}|$. Next we regard the result variable $\alpha = |t_1, \mathtt{x}'|$ with the local bound $\mathcal{S}_l(\alpha) = |\mathtt{x}| + |\mathtt{i}|$. Thus, we have $\alpha \in \dot{\times} \setminus \dot{+}$.

As $\alpha = |t_1, \mathtt{x}'|$ is a predecessor of itself and its SCC contains no other result variables, we have $\mathcal{V}_\alpha = \{\mathtt{x}\}$. Of course, α also has predecessors of the form $|\tilde{t}, \mathtt{i}'|$ outside the SCC. We have $\mathsf{actV}(\mathcal{S}_l(\alpha)) = \mathsf{actV}(|\mathtt{x}| + |\mathtt{i}|) = \{\mathtt{i}, \mathtt{x}\}$, and $f_\mathtt{i}^\alpha = \max\{\mathcal{S}(t_0, \mathtt{i}'), \mathcal{S}(t_1, \mathtt{i}')\} = |\mathtt{i}|$. When entering α's SCC, the input is bounded by the preceding transitions, i.e., by $\max\{\mathcal{S}(t_0, \mathtt{i}'), \mathcal{S}(t_1, \mathtt{i}'), \mathcal{S}(t_0, \mathtt{x}')\} = \max\{|\mathtt{i}|, |\mathtt{x}|\}$. By traversing α's transition t_1 repeatedly (at most $\mathcal{R}(t_1) = |\mathtt{i}|$ times), this value may be increased by adding $\mathcal{R}(t_1) \cdot (e_\alpha + f_\mathtt{i}^\alpha) = |\mathtt{i}| \cdot (0 + |\mathtt{i}|) = |\mathtt{i}|^2$. Hence, we obtain $\mathcal{S}(\alpha) = \max\{|\mathtt{i}|, |\mathtt{x}|\} + |\mathtt{i}|^2$. Consequently, we also get $\mathcal{S}(t_2, \mathtt{x}') = \mathcal{S}(t_3, \mathtt{x}') = \max\{|\mathtt{i}|, |\mathtt{x}|\} + |\mathtt{i}|^2$.

From the inferred size bounds we can now derive a runtime bound for the last transition t_3. When we call TimeBounds on $\mathcal{T}' = \{t_3\}$, it finds the PRF $\mathcal{P}ol(\ell_2) = \mathtt{x}$, implying that $\mathcal{T}'$'s runtime is linear. When program execution reaches $\mathcal{T}'$, the size of $\mathtt{x}$ is bounded by $\mathcal{S}(t_2, \mathtt{x}')$. So $\mathcal{R}(t_3) = \mathcal{R}(t_2) \cdot [\mathcal{P}ol(\ell_2)](\mathcal{S}(t_2, \mathtt{i}'), \mathcal{S}(t_2, \mathtt{x}')) = 1 \cdot \mathcal{S}(t_2, \mathtt{x}') = \max\{|\mathtt{i}|, |\mathtt{x}|\} + |\mathtt{i}|^2$. Thus, a bound on the overall runtime is $\sum_{t \in \mathcal{T}} \mathcal{R}(t) = 2 + |\mathtt{i}| + \max\{|\mathtt{i}|, |\mathtt{x}|\} + |\mathtt{i}|^2$, i.e., it is linear in $|\mathtt{x}|$ and quadratic in $|\mathtt{i}|$.

While the global size bound for the result variable $|t_1, \mathtt{x}'| \in \dot{\times} \setminus \dot{+}$ in the above example is only quadratic, in general the resulting global size bounds for result variables α from the class $\dot{\times}$ can be exponential. This is the case when the scaling factor s_α is greater than 1 or when $|\mathcal{V}_\alpha| > 1$.

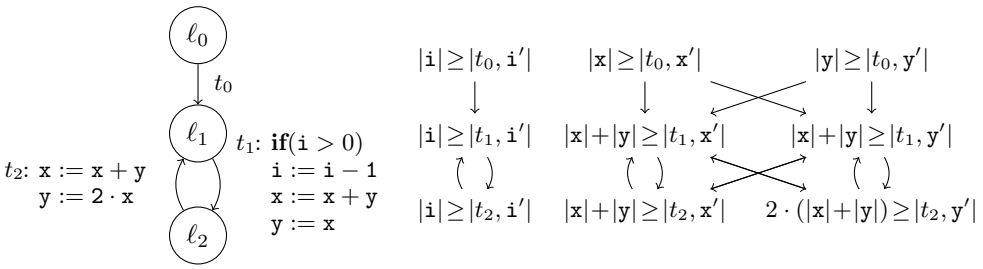


Fig. 7. Graph and RVG for the program of Ex. 4.14

Example 4.12. To illustrate that, consider the following loop.

$$\textbf{while } \mathtt{z} > 0 \textbf{ do } \mathtt{x} := \mathtt{x} + \mathtt{y};\ \mathtt{y} := \mathtt{x};\ \mathtt{z} := \mathtt{z} - 1;\ \textbf{done}$$

Here, we have a transition $t_1 = (\ell_1, \tau_1, \ell_1)$ where τ_1 is the formula $\mathtt{z} > 0 \wedge \mathtt{x}' = \mathtt{x} + \mathtt{y} \wedge \mathtt{y}' = \mathtt{x}' \wedge \mathtt{z}' = \mathtt{z} - 1$. For $\alpha_\mathtt{x} = |t_1, \mathtt{x}'|$ and $\alpha_\mathtt{y} = |t_1, \mathtt{y}'|$ we obtain $\mathcal{S}_l(\alpha_\mathtt{x}) = \mathcal{S}_l(\alpha_\mathtt{y}) = |\mathtt{x}| + |\mathtt{y}|$. Thus, $\alpha_\mathtt{x}$ and $\alpha_\mathtt{y}$ together form an SCC C where $\mathcal{V}_{\alpha_\mathtt{x}} = \mathcal{V}_{\alpha_\mathtt{y}} = \{\mathtt{x}, \mathtt{y}\}$. In other words, in each iteration of the loop, the value of x and y is not increased by a fixed value as before (that is not modified itself in the loop), but the value of x and y is doubled in each loop iteration. The reason is that here we have $|\mathcal{V}_{\alpha_\mathtt{x}}| = |\mathcal{V}_{\alpha_\mathtt{y}}| = 2$.

In general, each execution of a transition t may multiply the value of a result variable from $(\dot{\times} \setminus \dot{+}) \cap C_t$ by $\max\{|\mathcal{V}_\alpha| \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\}$. Since this multiplication may be performed $\mathcal{R}(t)$ times, the repeated traversal of t increases the overall size by at most $(\max\{|\mathcal{V}_\alpha| \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\})^{\mathcal{R}(t)}$.

If the SCC C uses several different transitions $(\dot{\times} \setminus \dot{+})$, then the effects resulting from these transitions have to be *multiplied*. Thus, the overall increase is bounded by a multiplication with the following exponential factor.

$$s_{\dot{\times}} = \prod\nolimits_{t \in \mathcal{T}} (\max\{|\mathcal{V}_\alpha| \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\})^{\mathcal{R}(t)}$$

In Ex. 4.12, $(\dot{\times} \setminus \dot{+}) \cap C_t$ is empty for all transitions t except t_1. As $|\mathcal{V}_{\alpha_\mathtt{x}}| = |\mathcal{V}_{\alpha_\mathtt{y}}| = 2$ and $\mathcal{R}(t_1) = |\mathtt{z}|$, we have $s_{\dot{\times}} = 2^{|\mathtt{z}|}$ which results in the global size bounds $\mathcal{S}(\alpha_\mathtt{x}) = \mathcal{S}(\alpha_\mathtt{y}) = 2^{|\mathtt{z}|} \cdot \max\{|\mathtt{x}|, |\mathtt{y}|\}$.

Example 4.13. In Ex. 4.12, the exponential growth of the size was due to $|\mathcal{V}_{\alpha_\mathtt{x}}| = |\mathcal{V}_{\alpha_\mathtt{y}}| > 1$. A similar effect is obtained if one has scaling factors $s_\alpha > 1$. To demonstrate this, consider the following modification of the above loop.

$$\textbf{while } \mathtt{z} > 0 \textbf{ do } \mathtt{x} := 2 \cdot \mathtt{x};\ \mathtt{z} := \mathtt{z} - 1;\ \textbf{done}$$

Now the resulting transition t_1 has the formula $\mathtt{z} > 0 \wedge \mathtt{x}' = 2 \cdot \mathtt{x} \wedge \mathtt{z}' = \mathtt{z} - 1$ and for $\alpha_\mathtt{x} = |t_1, \mathtt{x}'|$ we obtain $\mathcal{S}_l(\alpha_\mathtt{x}) = 2 \cdot |\mathtt{x}|$. In each iteration of the loop, the value of x is multiplied by the scaling factor $s_{\alpha_\mathtt{x}} = 2$.

To capture this, we revise the definition of $s_{\dot{\times}}$ as follows.

$$s_{\dot{\times}} = \prod\nolimits_{t \in \mathcal{T}} \Big(\max\{s_\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\} \ \cdot\ \max\{|\mathcal{V}_\alpha| \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\} \Big)^{\mathcal{R}(t)}$$

Then the overall increase is again bounded by a multiplication with an exponential factor. Similar to Ex. 4.12, in Ex. 4.13 we have $s_{\dot{\times}} = (s_{\alpha_\mathtt{x}} \cdot |\mathcal{V}_{\alpha_\mathtt{x}}|)^{\mathcal{R}(t_1)} = (2 \cdot 1)^{|\mathtt{z}|} = 2^{|\mathtt{z}|}$ which results in the global size bound $\mathcal{S}(\alpha_\mathtt{x}) = 2^{|\mathtt{z}|} \cdot |\mathtt{x}|$.

Example 4.14. As an example for a program with several transitions that correspond to scaled sums, consider the program in Fig. 7. Its RVG has an SCC $C = \{|t_1, \mathtt{x}'|, |t_2, \mathtt{x}'|, |t_1, \mathtt{y}'|, |t_2, \mathtt{y}'|\}$ where all results variable are in $\dot{\times} \setminus \dot{+}$. We can easily derive $\mathcal{S}(t_0, \mathtt{x}') = |\mathtt{x}|$, $\mathcal{S}(t_0, \mathtt{y}') = |\mathtt{y}|$ with Thm. 4.6, and deduce $\mathcal{R}(t_1) = \mathcal{R}(t_2) = |\mathtt{i}|$. We observe that $s_{|t_2,y'|} = 2$, and that the scaling factor is 1 for all other result variables. Moreover, we have $|V_\alpha| = 2$ for all $\alpha \in C$. For our SCC, we then compute $s_{\dot{\times}}$ as follows:

$$\begin{aligned} s_{\dot{\times}} = \; & \Big(\max\{s_{|t_1,\mathtt{x}'|}, s_{|t_1,\mathtt{y}'|}\} \cdot \max\{|V_{|t_1,\mathtt{x}'|}|, |V_{|t_1,\mathtt{y}'|}|\}\Big)^{\mathcal{R}(t_1)} \\ & \cdot \Big(\max\{s_{|t_2,\mathtt{x}'|}, s_{|t_2,\mathtt{y}'|}\} \cdot \max\{|V_{|t_2,\mathtt{x}'|}|, |V_{|t_2,\mathtt{y}'|}|\}\Big)^{\mathcal{R}(t_2)} \\ = \; & \Big(\max\{1,1\} \cdot \max\{2,2\}\Big)^{|\mathtt{i}|} \cdot \Big(\max\{1,2\} \cdot \max\{2,2\}\Big)^{|\mathtt{i}|} \\ = \; & 2^{|\mathtt{i}|} \cdot 4^{|\mathtt{i}|} = 8^{|\mathtt{i}|} \end{aligned}$$

Intuitively, this means that the inputs to the SCC C can grow at most by a factor of $8^{|\mathtt{i}|}$. As no constants are added in the SCC, we can then compute the size bound on all result variables of C as $s_{\dot{\times}} \cdot \max\{\mathcal{S}(t_0, \mathtt{x}'), \mathcal{S}(t_0, \mathtt{y}')\} = 8^{|\mathtt{i}|} \cdot \max\{|\mathtt{x}|, |\mathtt{y}|\}$.

Thm. 4.15 extends the procedure SizeBounds from Thm. 4.6 to non-trivial SCCs. Note that if an SCC contains a result variable β whose local size bound does not belong to any of our three classes, then we are not able to derive any global size bounds (i.e., then Thm. 4.15 cannot improve the current size approximation $\mathcal{S}$). This is the case when $\beta \notin \dot{\times}$ (since $\dot{=} \subseteq \dot{+} \subseteq \dot{\times}$).

THEOREM 4.15 (SizeBounds FOR NON-TRIVIAL SCCS). *Let $(\mathcal{R}, \mathcal{S})$ be a complexity approximation, $\mathcal{S}_l$ a local size approximation, and $C \subseteq \mathsf{RV}$ a non-trivial SCC of the RVG. If there is a $\beta \in C$ with $\beta \notin \dot{\times}$, then we set $\mathcal{S}' = \mathcal{S}$. Otherwise, for all $\beta \notin C$ let $\mathcal{S}'(\beta) = \mathcal{S}(\beta)$. For all $\beta \in C$, we set $\mathcal{S}'(\beta) =$*

$$\begin{aligned} s_{\dot{\times}} \cdot \Big(& \max(\,\{\mathcal{S}(\tilde{\alpha}) \mid \text{ there is an } \alpha \in C \text{ with } \tilde{\alpha} \in \mathsf{pre}(\alpha) \setminus C\} \;\cup\; \{c_\alpha \mid \alpha \in \dot{=} \cap\, C\}\,) \\ & + \textstyle\sum_{t \in \mathcal{T}} (\mathcal{R}(t) \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \dot{=}) \cap C_t\}) \\ & + \textstyle\sum_{t \in \mathcal{T}} (\mathcal{R}(t) \cdot \max\{e_\alpha + \sum_{v \in \mathsf{actV}(\mathcal{S}_l(\alpha)) \setminus \mathcal{V}_\alpha} f_v^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\}) \Big). \end{aligned}$$

Then $\mathsf{SizeBounds}(\mathcal{R}, \mathcal{S}, C) = \mathcal{S}'$ *is also a size approximation.*

Taking a different perspective, one can see our contributions on the inference of size bounds as a technique for computing potentially non-linear invariants. Our technique also takes runtime complexity analysis results into account, and its automation requires only relatively benign linear constraint solving in most cases.

5. HANDLING RECURSION

Our representation of programs as sets of simple transitions is restricted to programs without procedure calls. Of course, as long as there is no recursion (or only tail recursion), procedure calls can easily be "inlined". However, in Sect. 5.1 we present an extension of our approach to handle arbitrary (possibly non-tail recursive) procedure calls. Based on this, we show in Sect. 5.2 how to adapt PRFs in order to infer exponential runtime bounds for recursive programs.

5.1. Complexity Analysis for Recursive Programs

Example 5.1. As an example, regard the following program where the recursive procedure `fac(x)` computes `x!` and `facSum(x)` computes $\mathtt{x}! + (\mathtt{x}-1)! + \ldots + 0!$.

```
int facSum(int x)                    int fac(int x)
  ℓ0: r := 0                           ℓ5: r := 1
  ℓ1: while x ≥ 0 do                   ℓ6: if x > 0 then
  ℓ2:     r := r + fac(x)              ℓ7:     r := x · fac(x − 1)
                   ℓ3                                    ℓ8
        x := x − 1                         fi
      done                             ℓ9: return r
  ℓ4: return r
```

We extend the notion of *transitions* to the form $(\ell, \tau, \mathcal{P})$, where $\mathcal{P}$ is a non-empty multiset of locations. If $\mathcal{P}$ is a singleton set $\{\ell'\}$, we write (ℓ, τ, ℓ') instead of $(\ell, \tau, \{\ell'\})$. By using transitions with several target locations, we can express function calls, since one target location represents the evaluation of the called function, whereas the other target location represents the context which is executed when returning from the function call. Thus, the program from Ex. 5.1 can be represented by the following transitions. Note that our transitions are an over-approximation of the original program since we do not take the results of procedure calls into account (i.e., the value of the variable $\mathtt{r}'$ is arbitrary in the transitions t_3 and t_8).

$$\begin{array}{lcll}
t_0 & = & (\ell_0,\ \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = 0, & \ell_1) \\
t_1 & = & (\ell_1,\ \mathtt{x} \geq 0 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = \mathtt{r}, & \{\ell_2, \ell_3\}) \\
t_2 & = & (\ell_1,\ \mathtt{x} < 0 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = \mathtt{r}, & \ell_4) \\
t_3 & = & (\ell_2,\ \mathtt{x} \geq 0 \wedge \mathtt{x}' = \mathtt{x} - 1, & \ell_1) \\
t_4 & = & (\ell_3,\ \mathtt{x}' = \mathtt{x}, & \ell_5) \\
t_5 & = & (\ell_5,\ \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = 1, & \ell_6) \\
t_6 & = & (\ell_6,\ \mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = \mathtt{r}, & \{\ell_7, \ell_8\}) \\
t_7 & = & (\ell_6,\ \mathtt{x} \leq 0 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = \mathtt{r}, & \ell_9) \\
t_8 & = & (\ell_7,\ \mathtt{x}' = \mathtt{x}, & \ell_9) \\
t_9 & = & (\ell_8,\ \mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} - 1, & \ell_5)
\end{array}$$

A *configuration* is now a *multiset* of pairs $(\ell, \boldsymbol{v})$ of a location ℓ and a valuation $\boldsymbol{v} : \mathcal{V} \to \mathbb{Z}$. An *evaluation step* takes a pair $(\ell, \boldsymbol{v})$ in the current configuration and applies a transition to it.[7] So for a configuration F, we write $F \to_t^{(\ell,\boldsymbol{v})} F'$ for an *evaluation step* with a transition $t = (\ell, \tau, \{\ell_1, \ldots, \ell_k\}) \in \mathcal{T}$ iff there is a pair $(\ell, \boldsymbol{v}) \in F$, $F' = (F \setminus \{(\ell, \boldsymbol{v})\}) \cup \{(\ell_1, \boldsymbol{v'}), \ldots, (\ell_k, \boldsymbol{v'})\}$, and the valuations $\boldsymbol{v}$, $\boldsymbol{v'}$ satisfy the formula τ. To ease readability, sometimes we write $\to_t$ instead of $\to_t^{(\ell,\boldsymbol{v})}$. Note that in each evaluation step we only evaluate *one* pair in the configuration (i.e., this does not model concurrency where several pairs are evaluated in parallel). So in Ex. 5.1, we have

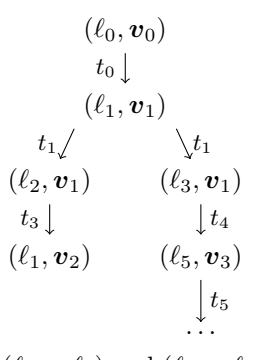
$$\begin{array}{llll}
\{(\ell_0, \boldsymbol{v}_0)\} & \to_{t_0}^{(\ell_0,\boldsymbol{v}_0)} & \{(\ell_1, \boldsymbol{v}_1)\} & \to_{t_1}^{(\ell_1,\boldsymbol{v}_1)} \\
\{(\ell_2, \boldsymbol{v}_1), (\ell_3, \boldsymbol{v}_1)\} & \to_{t_3}^{(\ell_2,\boldsymbol{v}_1)} & \{(\ell_1, \boldsymbol{v}_2), (\ell_3, \boldsymbol{v}_1)\} & \to_{t_4}^{(\ell_3,\boldsymbol{v}_1)} \\
\{(\ell_1, \boldsymbol{v}_2), (\ell_5, \boldsymbol{v}_3)\} & \to_{t_5}^{(\ell_5,\boldsymbol{v}_3)} & \ldots &
\end{array}$$

for $\boldsymbol{v}_0(\mathtt{x}) = \boldsymbol{v}_1(\mathtt{x}) = \boldsymbol{v}_3(\mathtt{x}) = 2$, $\boldsymbol{v}_2(\mathtt{x}) = 1$, $\boldsymbol{v}_0(\mathtt{r}) = 3$, $\boldsymbol{v}_1(\mathtt{r}) = 0$, $\boldsymbol{v}_2(\mathtt{r}) = -17$, and $\boldsymbol{v}_3(\mathtt{r}) = 42$.

Obviously, such an evaluation can also be represented as an *evaluation tree* whose nodes have the form $(\ell, \boldsymbol{v})$. If $F \to_t^{(\ell,\boldsymbol{v})} F'$ as above, then in the corresponding evaluation tree, the node $(\ell, \boldsymbol{v})$ has the children $(\ell_1, \boldsymbol{v'}), \ldots, (\ell_k, \boldsymbol{v'})$, and the edges to the children are labeled

[7] For a transition $(\ell, \tau, \mathcal{P})$, one can see $\mathcal{P}$ also as a *list* of locations. Then a configuration is a list of pairs $(\ell, \boldsymbol{v})$, corresponding to the *call stack*, and the topmost pair in the stack has to be evaluated first. However, for the purposes of our complexity analysis, the order in which the calls are evaluated makes no difference.

with t. So the evaluation above would be represented by the tree on the right. Albert et al. [2011a] and Debray et al. [1997] propose a similar notion of evaluation trees or computation trees for cost relations.



Note that for termination and for size complexity analysis, one could also replace transitions with multiple target locations like $(\ell_1, \tau, \{\ell_2, \ell_3\})$ by the corresponding single-target transitions (ℓ_1, τ, ℓ_2) and (ℓ_1, τ, ℓ_3). But for runtime complexity analysis, this is not possible. The reason is that the transition $(\ell_1, \tau, \{\ell_2, \ell_3\})$ means that evaluation continues both at location ℓ_2 *and* at location ℓ_3. Hence, the runtime complexities resulting from these two locations have to be *added* to obtain a bound for the runtime complexity resulting from location ℓ_1. In contrast, the two transitions (ℓ_1, τ, ℓ_2) and (ℓ_1, τ, ℓ_3) would mean that evaluation continues *either* at location ℓ_2 *or* at location ℓ_3. Hence, here it would suffice to take the *maximum* of the runtime complexities resulting from ℓ_2 and ℓ_3 to obtain a bound for the runtime complexity resulting from ℓ_1. For example, the program $\{t_0, \ldots, t_9\}$ above has quadratic runtime complexity. But if one replaced the transition t_1 by two separate transitions from ℓ_1 to ℓ_2 and from ℓ_1 to ℓ_3, then the resulting program would only have linear runtime complexity.

By adapting the required notions in a straightforward way, our approach for complexity analysis can easily be extended to programs with (possibly recursive) procedure calls. As before, the *runtime complexity* rc maps sizes $\boldsymbol{m}$ of program variables to the maximal number of evaluation steps that are possible from a start configuration $(\ell_0, \boldsymbol{v})$ with $\boldsymbol{v} \leq \boldsymbol{m}$. The only change compared to Def. 2.3 is that configurations are now multisets of pairs $(\ell, \boldsymbol{v})$:

$$\mathrm{rc}(\boldsymbol{m}) = \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F \,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_0, \boldsymbol{v}_0)\} \to^k F\}.$$

Similarly, *runtime approximations* $\mathcal{R}$ from Def. 2.4 also have to be adapted to the new notion of configurations. Now, $\mathcal{R}$ is a *runtime approximation* iff the following holds for all $t \in \mathcal{T}$:

$$(\mathcal{R}(t))(\boldsymbol{m}) \geq \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F \,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_0, \boldsymbol{v}_0)\} \,(\to^* \circ \to_t)^k \, F\}$$

As in Def. 2.6, the definition of *size approximations* $\mathcal{S}$ must ensure that $\mathcal{S}(t, v')$ is a bound on the size of the variable v after a certain transition t was used in an evaluation. However, we now regard transitions t of the form $(\ell, \tau, \mathcal{P})$ where $\mathcal{P}$ is a multiset of locations. Thus, we require that whenever there is an evaluation from an initial configuration to a configuration F such that t can be applied for the next evaluation step (i.e., such that $(\ell, \boldsymbol{v}) \in F$ and $\boldsymbol{v}, \boldsymbol{v'}$ satisfy τ for some valuations $\boldsymbol{v}$ and $\boldsymbol{v'}$), then $\mathcal{S}(t, v')$ must be a bound for $|\boldsymbol{v'}(v)|$. In other words, $\mathcal{S}$ is a *size approximation* iff for all $(\ell, \tau, \mathcal{P}) \in \mathcal{T}$ and all $v \in \mathcal{V}$, we have

$$(\mathcal{S}((\ell, \tau, \mathcal{P}), v'))(\boldsymbol{m}) \geq \sup\{|\boldsymbol{v'}(v)| \mid \exists \boldsymbol{v}_0, F, \boldsymbol{v}, \boldsymbol{v'} \,.\; \boldsymbol{v}_0 \leq \boldsymbol{m} \;\wedge\; \{(\ell_0, \boldsymbol{v}_0)\} \to^* F \;\wedge\; (\ell, \boldsymbol{v}) \in F \;\wedge\; \boldsymbol{v}, \boldsymbol{v'} \text{ satisfy } \tau\}.$$

Similarly, $\mathcal{S}_l$ is a *local size approximation* iff

$$(\mathcal{S}_l((\ell, \tau, \mathcal{P}), v'))(\boldsymbol{m}) \geq \sup\{|\boldsymbol{v'}(v)| \mid \exists \boldsymbol{v}, \boldsymbol{v'} \,.\, \boldsymbol{v} \leq \boldsymbol{m} \wedge \boldsymbol{v}, \boldsymbol{v'} \text{ satisfy } \tau\}.$$

To denote the transitions that may precede each other, we also have to adapt the definition of "pre" (which is needed for the computation of size approximations in Sect. 4). Here, we have to take into account that a transition may reach a subsequent transition multiple times. Therefore, we now define $\mathsf{pre}((\ell', \tau', \mathcal{P}'))$ to be the multiset where $(\ell, \tau, \mathcal{P}) \in \mathcal{T}$ is contained k times in $\mathsf{pre}((\ell', \tau', \mathcal{P}'))$ iff ℓ' is contained k times in $\mathcal{P}$ and there exist $\boldsymbol{v}_0, F, \boldsymbol{v}, \boldsymbol{v'}, \boldsymbol{v''}$ such that $\{(\ell_0, \boldsymbol{v}_0)\} \to^* F$, $(\ell, \boldsymbol{v}) \in F$, $\boldsymbol{v}, \boldsymbol{v'}$ satisfy τ, and $\boldsymbol{v'}, \boldsymbol{v''}$ satisfy τ'.

Finally, we also have to modify the notion of PRFs. We now measure composed configurations $\{(\ell_1, \boldsymbol{v_1}), \ldots, (\ell_k, \boldsymbol{v}_k)\}$ by *summing* up the measures of their components $(\ell_i, \boldsymbol{v}_i)$. Again, we call $\mathcal{P}ol$ a PRF for a program iff no transition increases the measure and at least

one decreases it. As before, our goal is to use a PRF to deduce a bound on the runtime complexity. To this end, we have to ensure that the measure of a configuration is at least as large as the number of decreasing transition steps that can be used when starting the program evaluation in this configuration. However, this would not be guaranteed anymore for composed configurations, since the measures of their components can also be negative. Therefore, when measuring a composed configuration of $k > 1$ components, we only sum up those components with non-negative measure. This gives rise to the following definition.

Definition 5.2 (*PRF for Recursive Programs*). We call $\mathcal{P}ol : \mathcal{L} \to \mathbb{Z}[v_1, \ldots, v_n]$ a PRF for $\mathcal{T}$ iff there is a non-empty $\mathcal{T}_\succ \subseteq \mathcal{T}$ such that the following conditions hold:

— for all $(\ell, \tau, \{\ell_1, \ldots, \ell_k\}) \in \mathcal{T}$, we have
  — $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) \geq (\mathcal{P}ol(\ell_1))(v'_1, \ldots, v'_n)$, if $k = 1$
  — $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) \geq \sum_{j \in \{1,\ldots,k\}} \max\{0, (\mathcal{P}ol(\ell_j))(v'_1, \ldots, v'_n)\}$, if $k > 1$
— for all $(\ell, \tau, \{\ell_1, \ldots, \ell_k\}) \in \mathcal{T}_\succ$, we have
  — $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) > (\mathcal{P}ol(\ell_1))(v'_1, \ldots, v'_n)$, if $k = 1$
  — $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) > \sum_{j \in \{1,\ldots,k\}} \max\{0, (\mathcal{P}ol(\ell_j))(v'_1, \ldots, v'_n)\}$, if $k > 1$
  — $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) \geq 1$

With this definition of PRFs, the procedure TimeBounds from Thm. 3.6 can also be used for the new extended notion of programs $\mathcal{T}$. For this, we now let the transitions $\mathcal{T}_\ell$ leading to the location ℓ be a multiset that contains all transitions $(\tilde{\ell}, \tilde{\tau}, \mathcal{P}) \in \mathcal{T} \setminus \mathcal{T}'$ with $\ell \in \mathcal{P}$. Here $(\tilde{\ell}, \tilde{\tau}, \mathcal{P})$ has the same multiplicity in $\mathcal{T}_\ell$ as the multiplicity of ℓ in $\mathcal{P}$ (i.e., if ℓ is contained k times in $\mathcal{P}$ then $(\tilde{\ell}, \tilde{\tau}, \mathcal{P})$ is contained k times in $\mathcal{T}_\ell$). This is needed, because the sub-program $\mathcal{T}'$ is reached k times by the transition $(\tilde{\ell}, \tilde{\tau}, \mathcal{P})$. Moreover, we now set $\mathcal{L}' = \{\ell \mid \mathcal{T}_\ell \neq \varnothing \wedge \exists \mathcal{P}'. (\ell, \tau, \mathcal{P}') \in \mathcal{T}'\}$ as entry locations. With this extension of our method, for the program of Ex. 5.1 (with the initial location ℓ_0) we obtain the runtime approximations $\mathcal{R}(t_3) = |\mathtt{x}|$ (for the loop of `facSum`) and $\mathcal{R}(t_9) = |\mathtt{x}| + |\mathtt{x}|^2$ (for the linear recursion of `fac` that is called linearly often from `facSum`). The overall runtime bound is $\sum_{t \in \mathcal{T}} \mathcal{R}(t) = 8 + 12 \cdot |\mathtt{x}| + 5 \cdot |\mathtt{x}|^2$, i.e., the program's runtime is quadratic in the size of `x`.

Alonso-Blas et al. [2013] propose a technique to obtain runtime bounds in closed form by means of *sparse* cost relations as an approximation for the overall cost of a cost relation. Their technique allows to consider cost relations with more than one target location and, at the same time, only count the cost resulting from a single equation in each step. This is similar to Def. 5.2 where we also only count the number of evaluation steps of the transitions in $\mathcal{T}_\succ$ instead of having to count all transitions in a single proof step.

5.2. Exponential Complexities for Recursive Programs

We consider two sources for exponential runtime complexity: (1) a variable takes a value of exponential size or (2) non-linear recursion. We handled the former in Sect. 4 by deriving exponential size approximations from polynomial runtime approximations. However, up to now a *direct* application of polynomial ranking functions (that does not rely on size bounds computed for preceding loops) could only yield polynomial runtime bounds. We now adapt and extend a technique by Albert et al. [2011a] to use polynomial ranking functions in order to infer exponential bounds for recursive programs directly (Case (2)). In contrast to the approach by Albert et al. [2011a], our adaptation allows to infer exponential complexity bounds for only *some* of the transitions and polynomial bounds for others.

Example 5.3. Our technique works for recursive algorithms with multiple recursive calls, such as the (naïve) recursive computation of the Fibonacci numbers. An implementation as an imperative program and its transitions are given in Fig. 8. Since we require that there may not be any transitions back to the initial program location ℓ_0, we added a `main`

```
int main(int x)
  ℓ0: return fib(x)

int fib(int x)
  ℓ1: if x ≤ 0 then
  ℓ2:    r := 0
  ℓ3: else if x ≤ 1 then
  ℓ4:    r := 1
      else
  ℓ5:    r := fib(x − 1)          (ℓ6)
  ℓ7:    r := r + fib(x − 2)      (ℓ8)
      fi fi
  ℓ9: return r
```

$$
\begin{array}{lll}
t_0 = & (\ell_0,\ \mathtt{x}' = \mathtt{x}, & \ell_1) \\
t_1 = & (\ell_1,\ \mathtt{x} \leq 0 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = \mathtt{r},\ & \ell_2) \\
t_2 = & (\ell_2,\ \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = 0, & \ell_9) \\
t_3 = & (\ell_1,\ \mathtt{x} > 0 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = \mathtt{r},\ & \ell_3) \\
t_4 = & (\ell_3,\ \mathtt{x} \leq 1 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = \mathtt{r},\ & \ell_4) \\
t_5 = & (\ell_4,\ \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = 1, & \ell_9) \\
t_6 = & (\ell_3,\ \mathtt{x} > 1 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{r}' = \mathtt{r},\ & \{\ell_5, \ell_6\}) \\
t_7 = & (\ell_5,\ \mathtt{x} > 1 \wedge \mathtt{x}' = \mathtt{x}, & \{\ell_7, \ell_8\}) \\
t_8 = & (\ell_6,\ \mathtt{x}' = \mathtt{x} - 1, & \ell_1) \\
t_9 = & (\ell_7,\ \mathtt{x}' = \mathtt{x}, & \ell_9) \\
t_{10} = & (\ell_8,\ \mathtt{x}' = \mathtt{x} - 2, & \ell_1)
\end{array}
$$

Fig. 8. Computation of the Fibonacci numbers with non-polynomial complexity

function that calls the recursive function `fib`. As t_6 and t_7 can occur $\Omega(\mathit{fib}(|\mathtt{x}|))$ times in an evaluation of $\mathcal{T}$ (i.e., an *exponential* number of times), there does not exist any PRF with $t_6 \in \mathcal{T}_\succ$ or $t_7 \in \mathcal{T}_\succ$. The size of `x` remains bounded by its initial size throughout the execution. Thus, the super-polynomial runtime cannot be inferred using the exponential size approximations from Sect. 4.

In Sect. 5.1, we discussed why we cannot just replace transitions with multiple target locations like $(\ell_3, \tau, \{\ell_5, \ell_6\})$ by the corresponding single-target transitions (ℓ_3, τ, ℓ_5) and (ℓ_3, τ, ℓ_6) to infer polynomial bounds. However, *exponential* bounds can be inferred in this way. Recall that Def. 5.2 required that the ranking function decreases over the *sum* of all components of a configuration whenever a transition from $\mathcal{T}_\succ$ is used. To weaken this requirement, we now only require that the ranking function decreases in *each individual* component. Then, *in each path* of the evaluation tree, the number of $\mathcal{T}_\succ$-edges is bounded by $[\mathcal{P}ol(\ell_0)]$, i.e., $[\mathcal{P}ol(\ell_0)]$ is a bound for the *height* h of the evaluation tree (if one only counts $\mathcal{T}_\succ$-edges to determine this height). For Ex. 5.3, this allows to infer the following ranking function $\mathcal{P}ol$:

$$\mathcal{P}ol(\ell_0) = \mathcal{P}ol(\ell_1) = \mathcal{P}ol(\ell_2) = \mathcal{P}ol(\ell_3) = \mathcal{P}ol(\ell_4) = \mathtt{x} + 2 \qquad \mathcal{P}ol(\ell_5) = \mathcal{P}ol(\ell_6) = \mathtt{x} + 1$$
$$\mathcal{P}ol(\ell_7) = \mathcal{P}ol(\ell_8) = \mathcal{P}ol(\ell_9) = \mathtt{x}$$

In the constraints for the ranking function $\mathcal{P}ol$, we have "separated" the transition t_6 into two transitions (from ℓ_3 to ℓ_5 and from ℓ_3 to ℓ_6). Similarly, t_7 is separated into the transitions from ℓ_5 to ℓ_7 and from ℓ_5 to ℓ_8. With the above ranking function $\mathcal{P}ol$ for the "separated" version of $\mathcal{T}$, the transitions t_6 and t_7 are in $\mathcal{T}_\succ$ (i.e., their separated versions are strictly decreasing and they are bounded from below by 1). All other transitions are weakly decreasing with the ranking function $\mathcal{P}ol$.

When considering ranking functions separately for the different target locations of a transition, we speak of *Separated Ranking Functions (SRFs)*. To obtain complexity bounds using SRFs, we have to find a bound on the number of evaluation steps with transitions from $\mathcal{T}_\succ$ in the whole tree (i.e., not just in a single path of the tree). Note that each evaluation step of the program corresponds to the step from an inner node to *all of* its children. Thus, we have to approximate the number of inner nodes of the evaluation tree (and not the number of its edges), because each inner node may have *several* outgoing edges that correspond to the same transition in the evaluation sequence (i.e., to just *one* evaluation step).

Moreover, we do not have to consider all inner nodes, but we only want to find a bound on the number of those inner nodes that correspond to an evaluation step with a transition from $\mathcal{T}_\succ$. To approximate this, we consider a *merged variant* of the evaluation tree. Here, for every inner node that corresponds to an evaluation step with $\mathcal{T} \setminus \mathcal{T}_\succ$, we *merge* all its children to a single child (which is labeled by a multiset of pairs $(\ell, \boldsymbol{v})$). As we do not compute bounds for the transitions in $\mathcal{T} \setminus \mathcal{T}_\succ$, this merging is sound (i.e., the number of inner nodes that correspond to $\mathcal{T}_\succ$-steps is the same as in the original evaluation tree). Then the branching degree b of this "merged" evaluation tree is bounded by the maximal number of target locations in the transitions of $\mathcal{T}_\succ$, i.e., $b \leq \max\{|\mathcal{P}| \mid (\ell, \tau, \mathcal{P}) \in \mathcal{T}_\succ\}$. Consequently, if the evaluation tree has the height h, then $\frac{b^h-1}{b-1}$ is an upper bound for the number of inner nodes of the tree. Analogously, if one only counts $\mathcal{T}_\succ$-steps to determine the height h, then $\frac{b^h-1}{b-1}$ is an upper bound on the number of those inner nodes that correspond to $\mathcal{T}_\succ$-steps (i.e., it is a bound on the overall number of $\mathcal{T}_\succ$-steps used in an evaluation).

In Ex. 5.3 all transitions have at most two target locations, and hence the maximal branching degree is $b = 2$. The height h of the evaluation tree is at most $[\mathcal{P}ol(\ell_0)] = |\mathbf{x}| + 2$ (counting only $\mathcal{T}_\succ$-steps as contributing to the height). Our ranking function thus allows us to obtain the exponential runtime complexity approximation $\mathcal{R}(t_6) = \mathcal{R}(t_7) = \frac{2^{|\mathbf{x}|+2}-1}{2-1} = 2^{|\mathbf{x}|+2} - 1$. The following definition introduces the constraints for SRFs formally.

Definition 5.4 (*SRF*). We call $\mathcal{P}ol : \mathcal{L} \to \mathbb{Z}[v_1, \ldots, v_n]$ a *separated ranking function (SRF)* for $\mathcal{T}$ iff there is a non-empty $\mathcal{T}_\succ \subseteq \mathcal{T}$ such that the following conditions hold:

- for all $(\ell, \tau, \{\ell_1, \ldots, \ell_k\}) \in \mathcal{T} \setminus \mathcal{T}_\succ$, we have
  - $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) \geq (\mathcal{P}ol(\ell_1))(v'_1, \ldots, v'_n)$, if $k = 1$
  - $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) \geq \sum_{j \in \{1,\ldots,k\}} \max\{0, (\mathcal{P}ol(\ell_j))(v'_1, \ldots, v'_n)\}$, if $k > 1$
- for all $(\ell, \tau, \{\ell_1, \ldots, \ell_k\}) \in \mathcal{T}_\succ$, we have
  - $\boxed{\text{for all } j \in \{1, \ldots, k\}, \text{ we have } \tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) > (\mathcal{P}ol(\ell_j))(v'_1, \ldots, v'_n)}$
  - $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) \geq 1$

The difference between Def. 5.4 for SRFs and Def. 5.2 for PRFs is $\boxed{\text{highlighted}}$. For transitions in $\mathcal{T}_\succ \subseteq \mathcal{T}$, the constraints are weaker than in Def. 5.2 since now we do not require the ranking function to decrease when compared to the *sum* of the components in the target configuration, but only for each component of the target configuration separately. The price to pay for these weaker constraints is that we can only deduce exponential bounds, not polynomial bounds. The following theorem adapts Thm. 3.3 in order to use SRFs to obtain runtime complexity bounds.

Theorem 5.5 (Complexities from SRFs). *Let $\mathcal{R}$ be a runtime approximation, $\mathcal{P}ol$ be an SRF for $\mathcal{T}$, and $2 \leq b = \max\{|\mathcal{P}| \mid (\ell, \tau, \mathcal{P}) \in \mathcal{T}_\succ\}$.*

Let $\mathcal{R}'(t) = \frac{b^{[\mathcal{P}ol(\ell_0)]}-1}{b-1}$ for all $t \in \mathcal{T}_\succ$ and $\mathcal{R}'(t) = \mathcal{R}(t)$ for all other $t \in \mathcal{T}$. Then, $\mathcal{R}'$ is also a runtime approximation.

Compared to Thm. 5.5, the related technique for the inference of exponential complexities by Albert et al. [2011a] has the drawback that it requires using the same ranking function for all paths through an (outermost) loop. In contrast, we also consider the case that only *some* of the transitions are in $\mathcal{T}_\succ$. We also implemented an analogous version of the modular runtime complexity analysis of Thm. 3.6 for SRFs to obtain exponential complexities.

Example 5.6. One may wonder if we could make Thm. 5.5 more powerful by replacing the requirement in Def. 5.4 that for all $(\ell, \tau, \{\ell_1, \dots, \ell_k\}) \in \mathcal{T} \setminus \mathcal{T}_\succ$ with $k > 1$ we must have

$$\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \dots, v_n) \geq \sum\nolimits_{j \in \{1,\dots,k\}} \max\{0, (\mathcal{P}ol(\ell_j))(v'_1, \dots, v'_n)\}$$

by the following requirement for all $j \in \{1, \dots, k\}$

$$\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \dots, v_n) \geq (\mathcal{P}ol(\ell_j))(v'_1, \dots, v'_n)$$

(in analogy to the change in the requirement for $\mathcal{T}_\succ$). To see that this relaxed version of separated ranking functions would render Thm. 5.5 unsound, consider the following example.

$$\begin{array}{lll} t_0 = & (\ell_0,\ \mathtt{x}' = \mathtt{x}, & \ell_1) \\ t_1 = & (\ell_1,\ \mathtt{x}' = \mathtt{x}, & \{\ell_1, \ell_2\}) \\ t_2 = & (\ell_2,\ \mathtt{x} \geq 1 \wedge \mathtt{x}' = \mathtt{x} - 1, & \ell_1) \end{array}$$

With the above modification of Def. 5.4, the ranking function $\mathcal{P}ol$ with $\mathcal{P}ol(\ell_0) = \mathcal{P}ol(\ell_1) = \mathcal{P}ol(\ell_2) = \mathtt{x}$ would yield $\mathcal{T}_\succ = \{t_2\}$ and $\mathcal{T} \setminus \mathcal{T}_\succ = \{t_0, t_1\}$. Thus in particular, we would (wrongly) conclude that t_2 can only be used finitely often. But this is refuted by the non-terminating evaluation sequence

$$\begin{array}{llll} \{(\ell_0, \boldsymbol{v}_0)\} & \to_{t_0} & \{(\ell_1, \boldsymbol{v}_0)\} & \to_{t_1} \\ \{(\ell_1, \boldsymbol{v}_0), (\ell_2, \boldsymbol{v}_0)\} & \to_{t_2} & \{(\ell_1, \boldsymbol{v}_0), (\ell_2, \boldsymbol{v}_1)\} & \to_{t_1} \\ \{(\ell_1, \boldsymbol{v}_0), (\ell_2, \boldsymbol{v}_0), (\ell_2, \boldsymbol{v}_1)\} & \to_{t_2} & \{(\ell_1, \boldsymbol{v}_0), (\ell_2, \boldsymbol{v}_1), (\ell_2, \boldsymbol{v}_1)\} & \to_{t_1} \dots \end{array}$$

with $\boldsymbol{v}_0(\mathtt{x}) = 1$ and $\boldsymbol{v}_1(\mathtt{x}) = 0$, which uses t_2 infinitely many times.

The technique by Albert et al. [2011a] to search for *logarithmic* runtime complexity bounds could be lifted to our setting analogously. Indeed, our approach in Thm. 3.3 and Thm. 3.6 is not restricted to polynomials. Instead, we can plug in an *arbitrary* weakly monotonic function that correctly over-approximates the number of times a transition from $\mathcal{T}_\succ$ is used in $\mathcal{T}$ (or $\mathcal{T}'$). To infer such a function, we could also use a different tool for runtime complexity analysis. Even if this tool does not return a weakly monotonic function f, we could then still use a weakly monotonic over-approximation $[f]$. Thus, other approaches and tools can effectively be combined with the contributions of our paper.

6. MODULAR COMPLEXITY ANALYSIS FOR PROGRAMS USING COST MEASURES

We now show how to extend our analysis to not only compute bounds for runtime and sizes of variables, but also for other arbitrary cost measures. This enables general resource analysis, e.g., to infer bounds on the number and size of network requests. In Sect. 6.1, we discuss how to annotate each transition with a specific *cost* that depends on the current values of variables, and how to adapt our analysis to also approximate the overall cost of a program run. The general idea of allowing parametric cost measures for each transition is a fairly standard approach to lift resource analysis from runtime bounds to other cost models (see, e.g., [Albert et al. 2012; Navas et al. 2007]). The interesting contribution here is the integration into our framework. In particular, we show in Sect. 6.2 how to use costs in order to modularize our runtime complexity analysis. To this end, a program part can be summarized by a single transition whose cost corresponds to the runtime of the program part.

6.1. Programs with Cost Measures

While our analysis is focused on finding bounds for the program's runtime and for the sizes of variables, our approach can easily be extended to handle different cost measures. To this end, we consider *annotated programs* $(\mathcal{T}, \mathcal{M})$, where we use a measure function $\mathcal{M} : \mathcal{T} \to \mathrm{UB}$ to annotate each transition with a corresponding *cost*. As in Sect. 5, we consider transitions of the form $(\ell, \tau, \mathcal{P})$ where $\mathcal{P}$ is a non-empty multiset of locations. The

cost $\mathcal{M}(t)$ for a transition t may depend on the values of the variables before the transition is applied. For example, $\mathcal{M}$ can be used to approximate the number of actual arithmetic operations performed by each transition, or the number of file system or network accesses performed by each transition.

Example 6.1. For the program from Ex. 2.1 we could set

$$\mathcal{M}(t_0) = 1 \qquad \mathcal{M}(t_1) = \mathcal{M}(t_4) = 3$$
$$\mathcal{M}(t_2) = \mathcal{M}(t_3) = \mathcal{M}(t_5) = 2$$

to reflect the actual number of program statements represented by each transition (recall that $\mathtt{z} > 0$ was only introduced by invariant generation). Here, the higher cost for t_4 also reflects the statement `z.val += u.val` of the original program, which has no counterpart in the integer abstraction of the program.

Similar to the runtime complexity of a program, we define the cost complexity cc of a program as a function that maps a vector of bounds $\boldsymbol{m}$ on the input values to the maximal cost of a run that starts in an initial state where the variables have values bounded by $\boldsymbol{m}$. To make the considered program explicit, in the following definition we use $\mathrm{cc}_{(\mathcal{T},\mathcal{M})}$, but we omit the index when $\mathcal{T}$ and $\mathcal{M}$ are clear from the context. Moreover, we abbreviate $(\mathcal{M}(t_i))(\boldsymbol{v}_i(v_1), \ldots, \boldsymbol{v}_i(v_n))$ by $(\mathcal{M}(t_i))(\boldsymbol{v}_i)$.

Definition 6.2 (*Annotated Program, Cost Complexity*). Let $\mathcal{T}$ be a program and $\mathcal{M} : \mathcal{T} \to \mathrm{UB}$ a cost measure. Then, $(\mathcal{T}, \mathcal{M})$ is an *annotated program*. The cost complexity $\mathrm{cc}_{(\mathcal{T},\mathcal{M})} : \mathbb{N}^n \to \mathbb{N} \cup \{\omega\}$ of a program $(\mathcal{T}, \mathcal{M})$ is defined as

$$\mathrm{cc}_{(\mathcal{T},\mathcal{M})}(\boldsymbol{m}) = \sup\{\textstyle\sum_{0 \le i < k} (\mathcal{M}(t_i))(\boldsymbol{v}_i) \mid \text{there are } \boldsymbol{v}_0 \le \boldsymbol{m} \text{ and } k \ge 1 \text{ with } \{(\ell_0, \boldsymbol{v}_0)\} \to_{t_0}^{(\ell_0,\boldsymbol{v}_0)} F_1 \to_{t_1}^{(\ell_1,\boldsymbol{v}_1)} F_2 \to_{t_2}^{(\ell_2,\boldsymbol{v}_2)} \ldots F_k\}.$$

We can derive a *cost approximation* from our runtime approximation $\mathcal{R}$ and size approximation $\mathcal{S}$. An upper bound $\mathcal{C}(t)$ for the overall cost of a transition t in any possible evaluation is computed by combining the number of times it is used (i.e., $\mathcal{R}(t)$) with its cost $\mathcal{M}(t)$ parametrized by the maximal sizes of variables (i.e., $\mathcal{S}(\tilde{t}, v')$ for all transitions $\tilde{t}$ preceding t). To make the considered program explicit, we use $\mathcal{C}_{(\mathcal{T},\mathcal{M})}$, $\mathcal{R}_{\mathcal{T}}$, $\mathcal{S}_{\mathcal{T}}$, and $\mathsf{pre}_{\mathcal{T}}$, but we omit the indexes again when $\mathcal{T}$ resp. $\mathcal{M}$ are clear from the context.

Definition 6.3 (*Cost Approximation*). Let $(\mathcal{T}, \mathcal{M})$ be an annotated program. Given runtime and size approximations $\mathcal{R} = \mathcal{R}_{\mathcal{T}}$, $\mathcal{S} = \mathcal{S}_{\mathcal{T}}$, and using $\mathsf{pre} = \mathsf{pre}_{\mathcal{T}}$, we can derive the *cost approximation* $\mathcal{C}_{(\mathcal{T},\mathcal{M})} : \mathcal{T} \to \mathrm{UB}$ as

$$\mathcal{C}_{(\mathcal{T},\mathcal{M})}(t) = \begin{cases} \mathcal{R}(t) \cdot \mathcal{M}(t), & \text{if } t \text{ is an initial transition} \\ \mathcal{R}(t) \cdot \max\{(\mathcal{M}(t))(\mathcal{S}(\tilde{t}, v'_1), \ldots, \mathcal{S}(\tilde{t}, v'_n)) \mid \tilde{t} \in \mathsf{pre}(t)\}, & \text{otherwise.} \end{cases}$$

Example 6.4. Continuing with Ex. 6.1, we can obtain the bound $\mathcal{C}(t_1) = \mathcal{R}(t_1) \cdot \max\{\mathcal{M}(t_1)(\ldots) \mid \ldots\} = |\mathtt{x}| \cdot 3$ for t_1, reflecting that the number of evaluated statements in the first loop is bounded by $|\mathtt{x}| \cdot 3$. Similarly, we obtain $\mathcal{C}(t_0) = \mathcal{R}(t_0) \cdot \mathcal{M}(t_0) = 1$, $\mathcal{C}(t_2) = \mathcal{R}(t_2) \cdot \mathcal{M}(t_2) = 2$, $\mathcal{C}(t_3) = \mathcal{R}(t_3) \cdot \mathcal{M}(t_3) = (|\mathtt{x}| + 1) \cdot 2$, $\mathcal{C}(t_4) = \mathcal{R}(t_4) \cdot \mathcal{M}(t_4) = (|\mathtt{x}| + |\mathtt{x}|^2) \cdot 3$, and $\mathcal{C}(t_5) = \mathcal{R}(t_5) \cdot \mathcal{M}(t_5) = |\mathtt{x}| \cdot 2$. The overall costs are thus bounded by $\sum_{t \in \mathcal{T}} \mathcal{C}_{(\mathcal{T},\mathcal{M})}(t) = 3 \cdot |\mathtt{x}|^2 + 10 \cdot |\mathtt{x}| + 5$.

Note that cost complexity and cost approximation are indeed generalizations of runtime complexity and approximation. If $\mathcal{M}(t) = 1$ for all transitions $t \in \mathcal{T}$, then we obtain $\mathrm{cc}_{(\mathcal{T},\mathcal{M})} = \mathrm{rc}_{\mathcal{T}}$ and $\mathcal{C}_{(\mathcal{T},\mathcal{M})} = \mathcal{R}_{\mathcal{T}}$. The following theorem shows that our cost approximation $\mathcal{C}$ is indeed an upper bound for the cost complexity cc.

Theorem 6.5 (Soundness of $\mathcal{C}$). *Let $(\mathcal{T}, \mathcal{M})$ be an annotated program. Then $(\sum_{t \in \mathcal{T}} \mathcal{C}_{(\mathcal{T},\mathcal{M})}(t)) \ge \mathrm{cc}_{(\mathcal{T},\mathcal{M})}$.*

6.2. Separated Analysis of Program Parts

We now introduce a compositional "bottom-up" analysis in which procedures or loops are handled separately (and even in advance or in parallel). To model the runtime of program parts that were analyzed before, we can employ our cost measure.

Example 6.6. Consider the procedure `len` in Fig. 9, which computes the length of a list. In the integer abstraction of this procedure, we use the special variable `res` to model the return value of `len`. Our approach infers $\mathcal{R}(t_0^{\texttt{len}}) = 1$, $\mathcal{R}(t_1^{\texttt{len}}) = |\texttt{x}|$, $\mathcal{R}(t_2^{\texttt{len}}) = 1$, and $\mathcal{S}(t_2^{\texttt{len}}, \texttt{res}') = |\texttt{x}|$, similar to the analysis of the first loop of Ex. 2.1.

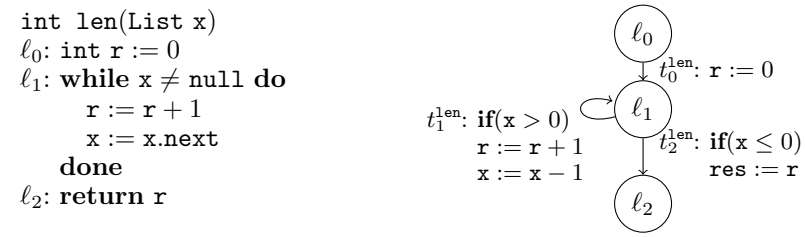


Fig. 9. Procedure to compute list length and its integer abstraction

Now, whenever a *procedure call* statement $\texttt{k} := \texttt{len}(\texttt{y})$ occurs in a program, we can use the previously computed runtime and size approximations for the procedure `len`. Instead of analyzing the procedure `len` again whenever it is called, we can represent the effect of calling `len` by introducing a new meta-transition $t^{\texttt{len}}$ that approximates its effects, i.e., it bounds the size of the return value. In our example, we label $t^{\texttt{len}}$ with the formula $|\texttt{k}'| \leq \mathcal{S}(t_2^{\texttt{len}}, \texttt{res}')(|\texttt{y}|)$.[8] To reflect the *cost* of calling `len`, we use our cost measure function and set $\mathcal{M}(t^{\texttt{len}}) = \mathcal{R}(t_0^{\texttt{len}}) + \mathcal{R}(t_1^{\texttt{len}}) + \mathcal{R}(t_2^{\texttt{len}})$.

This treatment of procedure calls differs from our technique in Sect. 5. Note that in general, our formalism of integer programs cannot represent concepts such as procedures or recursion. Hence, when translating a program into our formalism, a suitable representation of procedures needs to be chosen. The technique of Sect. 5 handles procedures and recursion by encoding procedure calls using transitions with several target locations. The drawback of this encoding is that return values of procedures are ignored (i.e., this encoding is a rather coarse abstraction of procedure calls).

In contrast, the separated "bottom-up" analysis presented in this section allows to infer information about the return value of procedure calls. However, it only works for non-recursive procedures, whereas recursive procedure need to be handled by the technique of Sect. 5.

We now extend the bottom-up approach to also allow a separated analysis for arbitrary program parts (within the *same* procedure). This bottom-up approach and the top-down approach of Fig. 3 (that is based on the sub-procedure TimeBounds from Thm. 3.6) yield similar results for programs with a simple structure, but they differ when handling *nested loops*. Here, the top-down approach usually first finds a termination (and hence complexity) argument for the *outermost* loop (since it starts by finding a PRF for all transitions $\mathcal{T}' = \mathcal{T}$), and then continues to find bounds for *inner* loops (i.e., it then only considers those transitions $\mathcal{T}' \subsetneq \mathcal{T}$ for which no bound has been computed up to now). In contrast, our bottom-up approach first finds bounds for the *innermost* loop and then replaces it by a single transition

[8] Of course, in general such formulas can also contain non-linear expressions. When generating PRFs for such transitions, one then either has to use techniques that can also deal with non-linear arithmetic or one has to "generalize" these transitions by "abstracting away" non-linear sub-expressions.

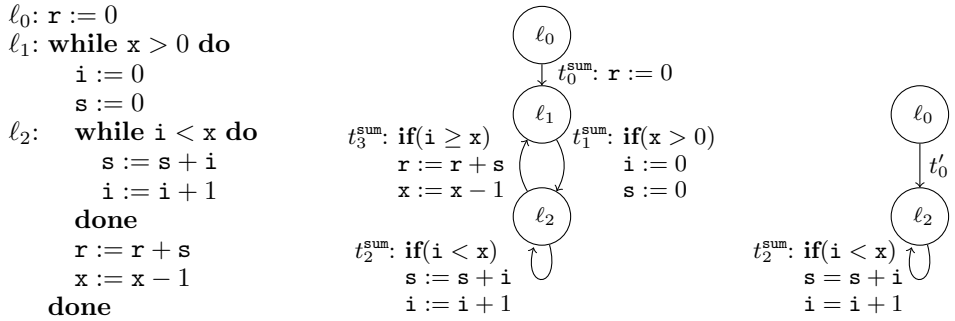


Fig. 10. Nested loop program sum, its graph representation, and isolated sub-program for inner loop

that represents its costs and its changes to program variables. It then continues on the simplified program, handling the *outer* loops in the following steps.

The top-down and the bottom-up methods are orthogonal in power and precision. When handling nested loops with a complex control flow, where inner and outer loops manipulate the *same* data, the top-down method tends to perform better, as it allows to consider all relevant program transitions at the same time.

Example 6.7. As an (albeit pathological) example, consider the following program:

```
while x > 0 do
  y := x
  while y > 0 do
    x := x − 1
    y := y − 1
  done
done
```

Here, separating the analysis of the inner loop from the analysis of the outer loop makes the complexity analysis extremely hard. The reason is that the underlying invariants $\mathtt{x} \geq 0$ and $\mathtt{y} = \mathtt{x}$ are not visible anymore when considering the transition t of the inner loop separately. We could only obtain the local size bound $\mathcal{S}_l(t, \mathtt{x}') = |\mathtt{x}| + 1$ (as decreasing x by 1 increases $|\mathtt{x}|$ for $\mathtt{x} \leq 0$). Since the inner loop is executed $|\mathtt{y}|$ times, this yields the global size bound $\mathcal{S}(t, \mathtt{x}') = |\mathtt{x}| + |\mathtt{y}|$. Using this bound afterwards for the analysis of the outer loop would fail, since x could be increased by $\mathtt{y} = \mathtt{x}$ in each loop iteration. Hence, one could not even prove termination of the outer loop anymore and would obtain the upper bound ω for the program's runtime complexity.

On the other hand, if the inner loop only modifies variables that do not influence the values in the condition of the outer loop, then the bottom-up approach may lead to less aggressive approximations and thus, to more precise size bounds.

Example 6.8. Consider the procedure sum in Fig. 10, where two nested loops are used to compute $\sum_{j=1}^{x}(\sum_{i=0}^{j-1} i)$. Using the top-down approach, the sub-procedure TimeBounds infers the runtime bound $\mathcal{R}(t_2^{\mathtt{sum}}) = |\mathtt{x}|^2$. However, when computing the size bound $\mathcal{S}(t_2^{\mathtt{sum}}, \mathtt{i}')$, this leads to a surprising effect. Using the local size bound $|\mathtt{i}| + 1$ for $|t_2^{\mathtt{sum}}, \mathtt{i}'|$, the procedure SizeBounds infers that $\mathcal{S}(t_2^{\mathtt{sum}}, \mathtt{i}') = \mathcal{R}(t_2^{\mathtt{sum}}) \cdot 1 = |\mathtt{x}|^2$. Here, the information that $t_2^{\mathtt{sum}}$ is executed at most $|\mathtt{x}|^2$ times in a full program run is combined with the fact that each application of $t_2^{\mathtt{sum}}$ increases i by 1. This is imprecise because our approximation does not take into account that i is reset to 0 after each inner loop and that $t_2^{\mathtt{sum}}$ is used at most $|\mathtt{x}|$ times in each iteration of the outer loop. For that reason, in reality $|\mathtt{i}|$ never gets larger than $|\mathtt{x}|$. Similarly, our method computes $\mathcal{S}(t_2^{\mathtt{sum}}, \mathtt{s}') = \mathcal{R}(t_2^{\mathtt{sum}}) \cdot \mathcal{S}(t_2^{\mathtt{sum}}, \mathtt{i}') = |\mathtt{x}|^2 \cdot |\mathtt{x}|^2 = |\mathtt{x}|^4$

instead of the more precise correct bound $|\mathtt{x}|^2$ and $\mathcal{S}(t_3^{\mathtt{sum}}, \mathtt{r}') = |\mathtt{x}|^5$ instead of the more precise bound $|\mathtt{x}|^3$. Such imprecisions can be avoided using a bottom-up approach which correctly determines that each time after `i` is set to 0, the transition $t_2^{\mathtt{sum}}$ is only executed $|\mathtt{x}|$ times. As will be shown, our bottom-up approach will result in $\mathcal{S}(t_3^{\mathtt{sum}}, \mathtt{r}') = |\mathtt{x}| + |\mathtt{x}|^3$, which is asymptotically a tight upper bound.

We describe our bottom-up approach in three steps. First, we show how to identify a subset $\mathcal{L}'$ of locations that should be analyzed separately (Sect. 6.2.1). In Sect. 6.2.2, we then explain how to extract the locations $\mathcal{L}'$ (and their connecting transitions $\mathcal{T}_{\mathcal{L}'}$) from a given annotated program $(\mathcal{T}, \mathcal{M})$ to create an *isolated sub-program* $(\mathcal{T}_{\mid \mathcal{L}'}, \mathcal{M}_{\mid \mathcal{L}'})$ that can be analyzed on its own. In a third step (Sect. 6.2.3), we discuss how to construct a simplified annotated program $(\mathcal{T}_{\setminus \mathcal{L}'}, \mathcal{M}_{\setminus \mathcal{L}'})$, where the extracted transitions are replaced by a summary obtained from the analysis of the sub-program. Finally, we show the soundness of our analysis by proving that $\mathrm{cc}_{(\mathcal{T}_{\setminus \mathcal{L}'}, \mathcal{M}_{\setminus \mathcal{L}'})}$ is an upper bound for $\mathrm{cc}_{(\mathcal{T},\mathcal{M})}$. For the case that we are only interested in runtime bounds, we extend the program $\mathcal{T}$ with the cost measure $\mathcal{M}(t) = 1$ for all $t \in \mathcal{T}$. Then $\mathrm{cc}_{(\mathcal{T},\mathcal{M})} = \mathrm{rc}_{\mathcal{T}}$ and hence, $\mathcal{C}_{(\mathcal{T}_{\setminus \mathcal{L}'}, \mathcal{M}_{\setminus \mathcal{L}'})}$ is also a runtime approximation for the original program $\mathcal{T}$.

6.2.1. Identifying Candidates for Separated Complexity Analysis. We have developed a heuristic SepHeuristic to identify suitable candidate sets $\mathcal{L}'$ for a separated analysis. To achieve good results, the set $\mathcal{L}'$ should correspond to a semantic unit of the program. Such a unit can be a procedure (or a group of mutually recursive procedures), or a sub-loop of the program, such that the separated part does not "interfere" with the remaining program. So this is a special case of a loop extraction technique, in which we introduce additional constraints on the data flow. In future work, it could be interesting to investigate whether other loop extraction techniques (e.g., [Wei et al. 2007]) could be used to further improve precision of our analysis.

Formally, we require (i) $\ell_0 \notin \mathcal{L}'$ and (ii) that for all transitions $(\ell, \tau, \mathcal{P})$ that lead from $\mathcal{L}'$ to *several* locations (i.e., where $\ell \in \mathcal{L}'$ and $|\mathcal{P}| > 1$), the locations in $\mathcal{P}$ may only reach locations in $\mathcal{L}'$ again. Here, we define $\ell_1 \to_{reach} \ell_2$ iff there exists a transition $(\ell_1, \tilde{\tau}, \tilde{\mathcal{P}}) \in \mathcal{T}$ with $\ell_2 \in \tilde{\mathcal{P}}$, and we say that ℓ_1 *reaches* ℓ_2 iff $\ell_1 \to^*_{reach} \ell_2$, where $\to^*_{reach}$ denotes the transitive and reflexive closure of $\to_{reach}$. For example, if there is a transition $(\ell, \tau, \{\ell, \ell'\})$ with $\ell \in \mathcal{L}'$, then ℓ' may not reach any locations ℓ'' outside of $\mathcal{L}'$. The reason for requirement (ii) is that otherwise, in every iteration of the sub-loop from ℓ to itself, one would also reach ℓ''. Hence, the number of evaluation steps inside $\mathcal{L}'$ would influence how often another part of the program is called. This would complicate a separated treatment of $\mathcal{L}'$ substantially.

The procedure SepHeuristic$(\mathcal{T}, \mathcal{R})$ returns a set of location sets $\mathcal{L}'$ that satisfy the requirements (i) and (ii) above and in addition are good candidates for separation. For this, we define the sets $G_{\mathcal{T}'}, U_{\mathcal{T}'} \subseteq \mathcal{V}$. The *guard variables* $G_{\mathcal{T}'}$ are those variables from $\mathcal{V}$ that are "restricted" by conditions in the formulas of the transitions of $\mathcal{T}'$. Formally, we define $v \in G_{\mathcal{T}'}$ iff $v \in \mathcal{V}$ and there exists a transition $(\ell, \tau, \mathcal{P}) \in \mathcal{T}'$ and some $m \in \mathbb{Z}$ such that $\forall v_1, \ldots, v_n. \exists v_1', \ldots, v_n'. \tau[v/m]$ is unsatisfiable. Here, $\tau[v/m]$ denotes the result of instantiating the variable v in τ by the number m. For example, if the only transition of $\mathcal{T}'$ has the formula $\mathtt{x} > \mathtt{y} \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{y}' = \mathtt{y} \wedge \mathtt{z}' = \mathtt{z}$, then $G_{\mathcal{T}'} = \{\mathtt{x}, \mathtt{y}\}$.

The *update variables* $U_{\mathcal{T}'}$ are all variables that may be changed by a transition from $\mathcal{T}'$ (i.e., variables $v \in \mathcal{V}$ where $v' = v$ cannot be inferred from the transition's formula). Now the heuristic SepHeuristic$(\mathcal{T}, \mathcal{R})$ works as follows. First, we identify those location sets $\mathcal{L}_1$, $\mathcal{L}_2$, …, $\mathcal{L}_n$ that are strongly connected components (SCCs) of the program graph and satisfy the requirement (i). Then, we return those proper sub-cycles $\mathcal{L}' \subsetneq \mathcal{L}_i$ that satisfy requirement (ii) and where for the transitions $\mathcal{T}' = \{(\ell, \tau, \mathcal{P}) \in \mathcal{T} \mid \ell \in \mathcal{L}'\}$ that originate in $\mathcal{L}'$, there exists a $t \in \mathcal{T}'$ with $\mathcal{R}(t) = \omega$ and $G_{\mathcal{T} \setminus \mathcal{T}'} \cap U_{\mathcal{T}'} = \varnothing$ holds. The latter requirement

ensures that variables that are updated in the separated part (i.e., in the inner cycle) do not influence the guards in the enclosing cycle.

6.2.2. Creating Isolated Sub-Programs for Separated Complexity Analysis. Note that our requirements on $\mathcal{L}'$ from Sect. 6.2.1 ensure that for every transition $(\ell, \tau, \mathcal{P})$ with $\ell \in \mathcal{L}'$, we either have $\mathcal{P} \cap \mathcal{L}' = \varnothing$ or $\mathcal{P} \subseteq \mathcal{L}'$. Thus, the *inner transitions* $\mathcal{T}_{\mathcal{L}'} \subseteq \mathcal{T}$ of $\mathcal{L}'$ are the transitions $(\ell, \tau, \mathcal{P})$ with $\ell \in \mathcal{L}'$ and $\mathcal{P} \subseteq \mathcal{L}'$. The *isolated sub-program* $\mathcal{T}_{|\,\mathcal{L}'}$ corresponding to $\mathcal{L}'$ then consists of $\mathcal{T}_{\mathcal{L}'}$ and a set of extra transitions that connect the start location ℓ_0 to all "entry locations" of $\mathcal{L}'$.

Definition 6.9 (*Isolated Sub-Program*). Let $(\mathcal{T}, \mathcal{M})$ be an annotated program and $\mathcal{L}' \subseteq \mathcal{L}$ where $\ell_0 \notin \mathcal{L}'$. Moreover, for all $(\ell, \tau, \mathcal{P})$ with $\ell \in \mathcal{L}'$ and $|\mathcal{P}| > 1$, there may not be any $\ell' \in \mathcal{P}$ and $\ell'' \in \mathcal{L} \setminus \mathcal{L}'$ with $\ell' \to^*_{reach} \ell''$. We define the *inner transitions* as $\mathcal{T}_{\mathcal{L}'} = \{(\ell, \tau, \mathcal{P}) \mid \ell \in \mathcal{L}', \mathcal{P} \subseteq \mathcal{L}'\}$. Then, the *isolated sub-program* $(\mathcal{T}_{|\,\mathcal{L}'}, \mathcal{M}_{|\,\mathcal{L}'})$ is defined by $\mathcal{T}_{|\,\mathcal{L}'} = \mathcal{T}_{\mathcal{L}'} \cup \{(\ell_0, \bigwedge_{v \in \mathcal{V}} v' = v, \mathcal{P}) \mid (\tilde{\ell}, \tau, \mathcal{P}) \in \mathcal{T}, \mathcal{P} \subseteq \mathcal{L}', \tilde{\ell} \notin \mathcal{L}'\}$ and $\mathcal{M}_{|\,\mathcal{L}'}(t) = \mathcal{M}(t)$ for all $t \in \mathcal{T}_{\mathcal{L}'}$ and $\mathcal{M}_{|\,\mathcal{L}'}(t) = 0$ for all $t \in \mathcal{T}_{|\,\mathcal{L}'} \setminus \mathcal{T}_{\mathcal{L}'}$.

Example 6.10. We continue with `sum` from Ex. 6.8, where $\mathcal{M}_{|\,\mathcal{L}'}(t_i^{\mathtt{sum}}) = 1$ for all $0 \leq i \leq 3$, i.e., our aim is just to analyze its runtime complexity. Let $\mathcal{L}' = \{\ell_2\}$ be the only self-loop of the example. For the transitions $\mathcal{T}' = \{t_2^{\mathtt{sum}}, t_3^{\mathtt{sum}}\}$ originating in $\mathcal{L}'$, we indeed have $G_{\mathcal{T} \setminus \mathcal{T}'} \cap U_{\mathcal{T}'} = \{\mathtt{x}\} \cap \{\mathtt{s}, \mathtt{i}\} = \varnothing$, i.e., $\mathcal{L}'$ would be suggested by `SepHeuristic`. We have the inner transition $\mathcal{T}_{\mathcal{L}'} = \{t_2^{\mathtt{sum}}\}$. The isolated sub-program $(\mathcal{T}_{|\,\mathcal{L}'}, \mathcal{M}_{|\,\mathcal{L}'})$ is displayed on the right of Fig. 10. Its costs are $\mathcal{M}_{|\,\mathcal{L}'}(t_0') = 0$ and $\mathcal{M}_{|\,\mathcal{L}'}(t_2^{\mathtt{sum}}) = 1$, reflecting that t_0' is a fresh transition introduced only for technical reasons.

The isolated sub-program $\mathcal{T}_{|\,\mathcal{L}'}$ can now be analyzed independently to obtain a runtime approximation $\mathcal{R}_{\mathcal{T}_{|\,\mathcal{L}'}}$ and a size approximation $\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}$. This can either be done by splitting it up into further isolated sub-programs, or by applying the top-down approach with the sub-procedure TimeBounds directly. In our example, only the latter makes sense, as there is only one loop remaining. We obtain $\mathcal{R}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_0') = 1$, $\mathcal{R}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_2^{\mathtt{sum}}) = |\mathtt{x}| + |\mathtt{i}|$ (as $\mathtt{x} - \mathtt{i}$ is a PRF for $\{t_2^{\mathtt{sum}}\}$), $\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_2^{\mathtt{sum}}, \mathtt{r}') = |\mathtt{r}|$, $\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_2^{\mathtt{sum}}, \mathtt{x}') = |\mathtt{x}|$, $\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_2^{\mathtt{sum}}, \mathtt{i}') = |\mathtt{i}| + (|\mathtt{x}| + |\mathtt{i}|) \cdot 1 = 2 \cdot |\mathtt{i}| + |\mathtt{x}|$, and $\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_2^{\mathtt{sum}}, \mathtt{s}') = \max\{|\mathtt{i}|, |\mathtt{s}|\} + (|\mathtt{x}| + |\mathtt{i}|) \cdot (2 \cdot |\mathtt{i}| + |\mathtt{x}|) = \max\{|\mathtt{i}|, |\mathtt{s}|\} + 3 \cdot |\mathtt{x}| \cdot |\mathtt{i}| + 2 \cdot |\mathtt{i}|^2 + |\mathtt{x}|^2$. Finally, using the costs from above, we obtain $\mathcal{C}_{(\mathcal{T}_{|\,\mathcal{L}'}, \mathcal{M}_{|\,\mathcal{L}'})}(t_0') = 0$ and $\mathcal{C}_{(\mathcal{T}_{|\,\mathcal{L}'}, \mathcal{M}_{|\,\mathcal{L}'})}(t_2^{\mathtt{sum}}) = |\mathtt{x}| + |\mathtt{i}|$.

6.2.3. Replacing Sub-Programs by the Results of Their Complexity Analysis. Note that the runtime approximation $\mathcal{R}_{\mathcal{T}_{|\,\mathcal{L}'}}$, size approximation $\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}$, and cost approximation $\mathcal{C}_{(\mathcal{T}_{|\,\mathcal{L}'}, \mathcal{M}_{|\,\mathcal{L}'})}$ describe how the complexity of the loop $\mathcal{T}_{\mathcal{L}'}$ depends on the values *before* entering $\mathcal{T}_{\mathcal{L}'}$. Thus, the next step is to include the information obtained for the isolated sub-program $\mathcal{T}_{\mathcal{L}'}$ in the original program $\mathcal{T}$, which yields a so-called $\mathcal{L}'$*-reduced program* $(\mathcal{T}_{\setminus\,\mathcal{L}'}, \mathcal{M}_{\setminus\,\mathcal{L}'})$ with a complexity approximation $(\mathcal{R}_{\setminus\,\mathcal{L}'}, \mathcal{S}_{\setminus\,\mathcal{L}'})$. To define $\mathcal{T}_{\setminus\,\mathcal{L}'}$, let $\mathcal{T}_{\to\mathcal{L}'}$ be all transitions that reach $\mathcal{L}'$ and let $\mathcal{T}_{\mathcal{L}'\to}$ be all transitions that leave $\mathcal{L}'$:

$$\mathcal{T}_{\to\mathcal{L}'} = \{(\ell, \tau, \mathcal{P}) \in \mathcal{T} \mid \ell \notin \mathcal{L}' \wedge\ \mathcal{P} \cap \mathcal{L}' \neq \varnothing\}$$
$$\mathcal{T}_{\mathcal{L}'\to} = \{(\ell, \tau, \mathcal{P}) \in \mathcal{T} \mid \ell \in \mathcal{L}' \wedge\ \mathcal{P} \cap \mathcal{L}' = \varnothing\}$$

Now we remove all locations $\mathcal{L}'$ from $\mathcal{T}$ and replace them by a fresh entry and exit location $\ell_{\to\mathcal{L}'}$ and $\ell_{\mathcal{L}'\to}$. To handle computations that use inner transitions of $\mathcal{L}'$, we add a new "meta-transition" $t_{\mathcal{T}_{\mathcal{L}'}}$ that approximates the effect of the inner transitions $\mathcal{T}_{\mathcal{L}'}$ in one transition step. To handle computations that just "skip" the inner transitions, we introduce an additional transition t_{skip} between the entry and exit location $\ell_{\to\mathcal{L}'}$ and $\ell_{\mathcal{L}'\to}$. In the example of Fig. 10 such a "skip" transition would not be required, since the inner loop of the sub-program $\mathcal{T}_{\mathcal{L}'} = \{\ell_2\}$ is always executed at least once. But if one applies the bottom-up

approach to the program of Fig. 2, a "skip" transition would be needed when removing the isolated sub-program with the location ℓ_3, in order to model computations where t_5 directly follows t_3, without using t_4 in between. So we obtain

$$\begin{aligned}\mathcal{T}_{\setminus \mathcal{L}'} = &\mathcal{T} \setminus (\mathcal{T}_{\to\mathcal{L}'} \cup \mathcal{T}_{\mathcal{L}'} \cup \mathcal{T}_{\mathcal{L}'\to}) \\ &\cup \{(\ell, \tau, (\mathcal{P} \setminus \mathcal{L}') \cup \{\ell_{\to\mathcal{L}'}\}) \mid (\ell, \tau, \mathcal{P}) \in \mathcal{T}_{\to\mathcal{L}'}\} \\ &\cup \{(\ell_{\mathcal{L}'\to}, \tau, \mathcal{P}) \mid (\ell, \tau, \mathcal{P}) \in \mathcal{T}_{\mathcal{L}'\to}\} \\ &\cup \{t_{\mathcal{T}_{\mathcal{L}'}}, t_{skip}\}\end{aligned}$$

To reflect the changes of values in the sub-program $\mathcal{T}_{\mathcal{L}'}$, we make use of the size approximation $\mathcal{S}_{\mathcal{T}_{|\mathcal{L}'}}$ obtained for our isolated sub-program. It provides upper bounds on the sizes of variables expressed in the input values $v_1, \ldots, v_n$ before entering the sub-program $\mathcal{T}_{\mathcal{L}'}$. Thus, they can be used for the formula of our new meta-transition $t_{\mathcal{T}_{\mathcal{L}'}}$. More precisely, we only have to consider $\mathcal{S}_{\mathcal{T}_{|\mathcal{L}'}}$ for result variables $|\tilde{t}, v_i'|$ where $\tilde{t} \in \mathcal{T}_{\mathcal{L}'}$ might be the last transition used before leaving the sub-program $\mathcal{T}_{\mathcal{L}'}$. In contrast, the "skip" transition t_{skip} does not change the values of the program variables.

$$\begin{aligned}&t_{\mathcal{T}_{\mathcal{L}'}} = (\ell_{\to\mathcal{L}'}, \tau_{\mathcal{T}_{\mathcal{L}'}}, \ell_{\mathcal{L}'\to}) \text{ with} \\ &\tau_{\mathcal{T}_{\mathcal{L}'}} = \bigwedge\nolimits_{v\in\mathcal{V}} |v'| \leq \max\{\mathcal{S}_{\mathcal{T}_{|\mathcal{L}'}}(\tilde{t}, v') \mid \tilde{t} = (\ell, \tau, \mathcal{P}) \in \mathcal{T}_{\mathcal{L}'} \wedge \\ &\qquad\qquad \text{there exist } \ell' \in \mathcal{P} \text{ and } (\ell', \tau', \mathcal{P}') \in \mathcal{T} \setminus \mathcal{T}_{\mathcal{L}'}\} \\ &t_{skip} = (\ell_{\to\mathcal{L}'}, \bigwedge\nolimits_{v\in\mathcal{V}} v' = v, \ \ell_{\mathcal{L}'\to})\end{aligned}$$

In the formula for $\tau_{\mathcal{T}_{\mathcal{L}'}}$, as usual the empty conjunction is considered to be *true*, i.e., if there is no $\ell \in \mathcal{L}'$ with $\ell \to_{reach} \ell'$ for some $\ell' \notin \mathcal{L}'$, then $\tau_{\mathcal{T}_{\mathcal{L}'}} = true$.

We define the cost for the new meta-transition $t_{\mathcal{T}_{\mathcal{L}'}}$ to be $\sum_{\tilde{t}\in\mathcal{T}_{\mathcal{L}'}} \mathcal{C}_{(\mathcal{T}_{|\mathcal{L}'}, \mathcal{M}_{|\mathcal{L}'})}(\tilde{t})$, since this is a bound on the cost of any evaluation that only uses the removed inner transitions $\mathcal{T}_{\mathcal{L}'}$. Note that here we implicitly use the runtime approximation $\mathcal{R}_{\mathcal{T}_{|\mathcal{L}'}}$, which is used to compute $\mathcal{C}_{(\mathcal{T}_{|\mathcal{L}'}, \mathcal{M}_{|\mathcal{L}'})}$. The cost for the "skip" transition t_{skip} is 0. Hence, we obtain the following cost measure for the reduced program:

$$\mathcal{M}_{\setminus\mathcal{L}'}(t) = \begin{cases} \sum_{\tilde{t}\in\mathcal{T}_{\mathcal{L}'}} \mathcal{C}_{(\mathcal{T}_{|\mathcal{L}'}, \mathcal{M}_{|\mathcal{L}'})}(\tilde{t}) & \text{if } t = t_{\mathcal{T}_{\mathcal{L}'}} \\ 0 & \text{if } t = t_{skip} \\ \mathcal{M}(t) & \text{if } t \in \mathcal{T} \setminus (\mathcal{T}_{\to\mathcal{L}'} \cup \mathcal{T}_{\mathcal{L}'} \cup \mathcal{T}_{\mathcal{L}'\to}) \\ \max\{\mathcal{M}(t') \mid \exists\, \mathcal{P}'.\, t' = (\ell, \tau, \mathcal{P} \cup \mathcal{P}') \in \mathcal{T}_{\to\mathcal{L}'}\} & \text{if } t = (\ell, \tau, \mathcal{P} \uplus \{\ell_{\to\mathcal{L}'}\}) \\ \max\{\mathcal{M}(t') \mid \exists\, \ell.\, t' = (\ell, \tau, \mathcal{P}) \in \mathcal{T}_{\mathcal{L}'\to}\} & \text{if } t = (\ell_{\mathcal{L}'\to}, \tau, \mathcal{P}) \end{cases}$$

Finally, the runtime and size approximations for the new transitions are set to ω:

$$\mathcal{R}_{\setminus\mathcal{L}'}(t) = \begin{cases} \mathcal{R}(t) & \text{if } t \in \mathcal{T} \setminus (\mathcal{T}_{\to\mathcal{L}'} \cup \mathcal{T}_{\mathcal{L}'} \cup \mathcal{T}_{\mathcal{L}'\to}) \\ \omega & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\setminus\mathcal{L}'}(t, v') = \begin{cases} \mathcal{S}(t, v') & \text{if } t \in \mathcal{T} \setminus (\mathcal{T}_{\to\mathcal{L}'} \cup \mathcal{T}_{\mathcal{L}'} \cup \mathcal{T}_{\mathcal{L}'\to}) \\ \omega & \text{otherwise} \end{cases}$$

Definition 6.11 (*Reduced Program*). Let $(\mathcal{T}, \mathcal{M})$ be an annotated program and $(\mathcal{R}, \mathcal{S})$ be a complexity approximation for $\mathcal{T}$, let $\mathcal{L}' \subseteq \mathcal{L}$ satisfy the requirements in Def. 6.9, and let $\mathcal{R}_{\mathcal{T}_{|\mathcal{L}'}}$, $\mathcal{S}_{\mathcal{T}_{|\mathcal{L}'}}$, resp. $\mathcal{M}_{\mathcal{T}_{|\mathcal{L}'}}$ be the runtime resp. size approximation resp. cost measure for the isolated sub-program $(\mathcal{T}_{|\mathcal{L}'}, \mathcal{M}_{|\mathcal{L}'})$. Then the $\mathcal{L}'$*-reduced program* is defined as $\texttt{Reduce}(\mathcal{T}, \mathcal{M}, \mathcal{L}', \mathcal{R}_{\mathcal{T}_{|\mathcal{L}'}}, \mathcal{S}_{\mathcal{T}_{|\mathcal{L}'}}, \mathcal{M}_{\mathcal{T}_{|\mathcal{L}'}}) = (\mathcal{T}_{\setminus\mathcal{L}'}, \mathcal{M}_{\setminus\mathcal{L}'}, \mathcal{R}_{\setminus\mathcal{L}'}, \mathcal{S}_{\setminus\mathcal{L}'})$.

Example 6.12. We continue Ex. 6.10 with $\mathcal{L}' = \{\ell_2\}$. The $\mathcal{L}'$-reduced program is displayed in Fig. 11. The transition $t_{\{t_2^{\texttt{sum}}\}}$ is labeled with the formula $|\mathtt{r}'| \leq |\mathtt{r}| \wedge |\mathtt{x}'| \leq |\mathtt{x}| \wedge |\mathtt{i}'| \leq 2 \cdot |\mathtt{i}| + |\mathtt{x}| \wedge |\mathtt{s}'| \leq \max\{|\mathtt{i}|, |\mathtt{x}|\} + 3 \cdot |\mathtt{x}| \cdot |\mathtt{i}| + 2 \cdot |\mathtt{i}|^2 + |\mathtt{x}|^2$, using the results obtained from the isolated sub-program. Similarly, the cost for $t_{\{t_2^{\texttt{sum}}\}}$ is $\mathcal{C}_{(\mathcal{T}_{|\{\ell_2\}}, \mathcal{M}_{|\{\ell_2\}})}(t_2^{\texttt{sum}}) = |\mathtt{x}| + |\mathtt{i}|$. We can now apply the procedures TimeBounds and SizeBounds on the simplified program. For example, we can use the PRF $\mathcal{P}$ with $\mathcal{P}(\ell_1) = \mathcal{P}(\ell_{\to\mathcal{L}'}) = \mathcal{P}(\ell_{\mathcal{L}'\to}) = \mathtt{x}$ to infer $\mathcal{R}(t_3^{\texttt{sum}}) = |\mathtt{x}|$. In a similar way, we also obtain $\mathcal{R}(t_1^{\texttt{sum}}) = \mathcal{R}(t_{\{t_2^{\texttt{sum}}\}}) = \mathcal{R}(t_{skip}) = |\mathtt{x}|$. We can also trivially infer $\mathcal{S}(t_1^{\texttt{sum}}, \mathtt{i}') = \mathcal{S}(t_1^{\texttt{sum}}, \mathtt{s}') = 0$ from the formula of $t_1^{\texttt{sum}}$. This allows to deduce $\mathcal{S}(t_{\{t_2^{\texttt{sum}}\}}, \mathtt{i}') = |\mathtt{x}|$ and $\mathcal{S}(t_{\{t_2^{\texttt{sum}}\}}, \mathtt{s}') = |\mathtt{x}|^2$. In turn, we also obtain a size bound for $\mathtt{r}$, resulting in the (asymptotically) precise bound $\mathcal{S}(t_3^{\texttt{sum}}, \mathtt{r}') = \max\{\mathcal{S}(t_0^{\texttt{sum}}, \mathtt{r}'), \mathcal{S}(t_{\{t_2^{\texttt{sum}}\}}, \mathtt{s}'), \mathcal{S}(t_{skip}, \mathtt{s}')\} + \mathcal{R}(t_3^{\texttt{sum}}) \cdot \mathcal{S}(t_{\{t_2^{\texttt{sum}}\}}, \mathtt{s}') = |\mathtt{x}| + |\mathtt{x}| \cdot |\mathtt{x}|^2 = |\mathtt{x}| + |\mathtt{x}|^3$.

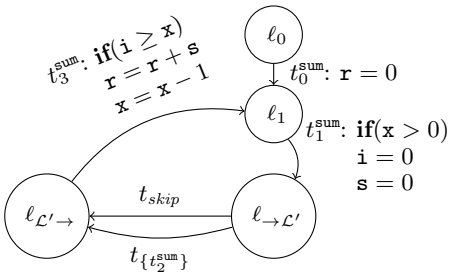


Fig. 11. Reduced program

The following theorem states the correctness of our bottom-up approach, justifying Def. 6.11.

Theorem 6.13 (Soundness of Separated Modular Complexity Analysis). *Let $(\mathcal{T}, \mathcal{M})$ be an annotated program, let $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{L}' \subseteq \mathcal{L}$ satisfy the requirements in Def. 6.9, and let $(\mathcal{T}_{\setminus \mathcal{L}'}, \mathcal{M}_{\setminus \mathcal{L}'}, \mathcal{R}_{\setminus \mathcal{L}'}, \mathcal{S}_{\setminus \mathcal{L}'})$ be the $\mathcal{L}'$-reduced program. Then $(\mathcal{R}_{\setminus \mathcal{L}'}, \mathcal{S}_{\setminus \mathcal{L}'})$ is a complexity approximation for $\mathcal{T}_{\setminus \mathcal{L}'}$ and $\mathrm{cc}_{(\mathcal{T}_{\setminus \mathcal{L}'}, \mathcal{M}_{\setminus \mathcal{L}'})} \geq \mathrm{cc}_{(\mathcal{T}, \mathcal{M})}$.*

In Fig. 12 of Sect. 7.2, we will present an improved general procedure which combines the top-down approach of Fig. 3 with the bottom-up approach of Thm. 6.13.

7. RELATED WORK, IMPLEMENTATION, AND EVALUATION

Several methods to determine symbolic runtime complexity bounds for programs have been developed in recent years. We give an overview on related work in Sect. 7.1, describe the implementation of our approach in Sect. 7.2, and compare our implementation empirically with related tools in Sect. 7.3.

7.1. Related Work

Our approach builds upon well-known basic concepts (like lexicographic ranking functions), but uses them in a novel way to obtain a more powerful technique than previous approaches. In particular, in contrast to previous work, our approach deals with non-linear information flow between different program parts, by benefiting from intermediate analysis results for runtime complexity bounds in order to deduce size bounds.

Similar to our technique, the approaches of Albert et al. [2011a] and Albert et al. [2012] (implemented in `COSTA` and its backend `PUBS`), of Flores-Montoya and Hähnle [2014] (implemented in `CoFloCo`), and of Zuleger et al. [2011] and Sinn et al. [2014] (implemented in `Loopus`) also use an iterative procedure based on termination proving techniques to find runtime bounds for single loops. The techniques differ in the decomposition of the program into smaller (and thus, easier to analyze) parts and in the combination of partial results into an overall bound. However, they do not take the interaction between runtime and data sizes into account. In contrast, in our approach this enables a flexible decomposition of the considered program during the analysis, making use of intermediate results.

In the approach by Albert et al. [2011a] and Albert et al. [2012], all transitions of an SCC in the program graph need to be handled at once. Pre-processing techniques are employed to decompose nested loops such as `while` B_1 `do while` B_2 `do` P`; done; done`

into two loops `while` B_2 `do` P`; done` and `while` B_1 `do` P_i`; done`, where P_i has a cost corresponding to the bound computed for the inner loop. A related idea is also used in our technique from Sect. 6.2. Flores-Montoya and Hähnle [2014] extend this idea to a new complexity analysis framework based on control-flow refinement. The program is first decomposed into different "chains", where each chain corresponds to program runs under a certain condition (i.e., the program can be analyzed separately for the cases $\mathtt{x} > 0$ and $\mathtt{x} \leq 0$). Experiments show that this allows to find precise bounds in programs with complex control flow.

The approach of Zuleger et al. [2011] is restricted to termination proofs via the size-change principle [Lee et al. 2001]. Finally, Sinn et al. [2014] introduce an amortized complexity analysis based on lexicographic termination proofs, in which the interaction between different components of the lexicographic termination argument is handled explicitly.

The approach of Alias et al. [2010] (implemented in `Rank`) first proves termination by a lexicographic combination of linear ranking functions, similar to our Thm. 3.6. However, while Thm. 3.6 combines these ranking functions with size bounds, Alias et al. [2010] approximate the reachable state space using Ehrhart polynomials. The tool `SPEED` [Gulwani et al. 2009] instruments programs by counters and employs an invariant generation tool to obtain bounds on these counters. The `ABC` system [Blanc et al. 2010] also determines symbolic bounds for nested loops, but does not treat sequences of loops.

Finally, our technique in Sect. 4.2 to infer size bounds by estimating the effect of repeated local changes has some similarities to the approach of Ben-Amram et al. [2008]. They introduce a Turing-incomplete imperative programming language and propose a sound and complete automated technique to decide whether variables in that language have linear, polynomial, or exponential size bounds. They also classify the effects of each transition (more generally, of a sub-program) on a variable w.r.t. the respective pre-variables, similar to our classification of local size bounds into $\doteq$, $\dot{+}$, and $\dot{\times}$. Based on a *dataflow relation*, which is similar to our result variable graph, they introduce a calculus to compose such bounds. However, their analysis does not aim for *concrete* bounds, but considers the more coarse-grained question whether a dependency of a result variable on an input variable is linear, polynomial, or exponential.

The work on determining the *worst-case execution time* (WCET) for real-time systems [Wilhelm et al. 2008] is largely orthogonal to symbolic loop bounds. It distinguishes processor instructions according to their complexity, but requires loop bounds to be provided by the user. Recently, recurrence solving has been used as an automatic pre-processing step for WCET analysis in the tool `r-TuBound` [Knoop et al. 2012].

There is also a wealth of work on complexity analysis for declarative paradigms. For instance, *resource aware* `ML` [Hoffmann et al. 2012; Hoffmann and Shao 2014] analyzes amortized complexity for recursive functional programs with inductive data types, natural numbers, and arrays. `RAML` uses a type-based approach to obtain (possibly non-linear) complexity bounds. The sizes of data structures are used as parameters, and the approach is automated by linear constraint solving. `RAML` computes bounds on runtime and size simultaneously using templates for the considered classes of resource polynomials. However, `RAML` appears to have only limited support for programs whose complexity depends on (possibly negative) integers: On a manually translated variant of the integer program from Ex. 2.1 where the variables are annotated with natural numbers as their type, `RAML`'s prototype web interface at `http://raml.tcs.ifi.lmu.de/prototype` infers a quadratic bound for the runtime complexity, similar to our method. However, on the integer version of the example from Fig. 2, `RAML` does not find any bound.

Complexity analysis is also an active topic of research in the area of logic programming. In particular, already Debray and Lin [1993] propose a technique to compute bounds on runtime using a dedicated analysis for size bounds. They use a graph structure similar to our RVGs to track local size bounds between input and output variables. Among other

techniques, they apply computer algebra systems to determine non-linear size bounds. While our approach in Sect. 5 over-approximates the results of recursive calls via a fresh variable, the analysis by Debray and Lin [1993] takes them into account. The work of Debray and Lin [1993] is extended by Navas et al. [2007] and by Serrano et al. [2014] to support also user-defined complexity metrics, similar to our Sect. 6.1, and to benefit from sized types. However, the idea of using analysis results for runtime bounds to compute size bounds is not present in these works.

There is a recent trend in program analysis to use Horn clauses (the core formalism behind logic programs) as an intermediate format for program verification [Grebenshchikov et al. 2012; Hojjat et al. 2012; Bjørner et al. 2014]. Thus, via a suitable complexity-preserving translation to Horn clauses, work on complexity analysis for logic programs can become applicable also for imperative programs.

Complexity analysis has also become a topic of intense study for term rewrite systems (TRSs), see, e.g., [Avanzini and Moser 2013; Noschinski et al. 2013]. To deduce bounds on the runtime complexity of TRSs, most techniques adapt suitable techniques for termination proving (e.g., polynomial interpretations [Lankford 1979; Fuhs et al. 2007], which are very similar to our PRFs). In fact, this line of research forms one of the inspirations for the use of PRFs in the present paper. A restriction of these works is that in contrast to us, they do not use a dedicated initial location ℓ_0, but allow a rewrite sequence to start at a term with an arbitrary function symbol at the root. Moreover, the inferred complexity bounds are only asymptotic and univariate (i.e., the size of the initial data enters the analysis only as a whole and not argument-dependent, as in our case).

The potential of this approach is harnessed by Giesl et al. [2012] for complexity analysis of logic programs. Giesl et al. [2012] propose a translation of logic programs to TRSs in such a way that a complexity bound for the TRS also induces a runtime bound for the logic program. Based on a tool for complexity analysis of TRSs, the experiments by Giesl et al. [2012] indicate that the resulting implementation is competitive to complexity analysis with dedicated logic programming tools like `CASLOG` [Debray and Lin 1993] and `CiaoPP` [Hermenegildo et al. 2012].

7.2. Implementation

We have implemented our contributions in the prototype tool `KoAT`, an open-source `OCaml` project available online at `https://github.com/s-falke/kittel-koat`.

The overall analysis implemented by `KoAT` is shown in Fig. 12, which combines the top-down approach of Fig. 3 with the modular bottom-up approach of Sect. 6.2. Of course, we do not always succeed in finding bounds for all transitions and variables. In our implementation, we give up after a certain time, and report the values of $\mathcal{R}$ and $\mathcal{S}$ at that point. As these bounds are correct over-approximations of runtime and size, even a partial result may still provide useful bounds for a part of the input program.

Procedure `KoAT`
Input Transitions $\mathcal{T}$, cost measure $\mathcal{M}$
$(\mathcal{R}, \mathcal{S}) := (\mathcal{R}_0, \mathcal{S}_0)$
while there are t, v with $\mathcal{R}(t) = \omega$ or $\mathcal{S}(t, v') = \omega$ **do**
  $(\mathcal{T}, \mathcal{M}, \mathcal{R}, \mathcal{S}) := \mathsf{InformationPropagate}(\mathcal{T}, \mathcal{M}, \mathcal{R}, \mathcal{S})$
  if exists $\mathcal{L}' \in \mathsf{SepHeuristic}(\mathcal{T}, \mathcal{R})$ **then**
    $(\mathcal{T}_{\restriction \mathcal{L}'}, \mathcal{M}_{\restriction \mathcal{L}'}, \mathcal{R}_{\restriction \mathcal{L}'}, \mathcal{S}_{\restriction \mathcal{L}'}) := \mathtt{KoAT}(\mathcal{T}_{\restriction \mathcal{L}'}, \mathcal{M}_{\restriction \mathcal{L}'})$
    $(\mathcal{T}, \mathcal{M}, \mathcal{R}, \mathcal{S}) := \mathsf{Reduce}(\mathcal{T}, \mathcal{M}, \mathcal{L}', \mathcal{R}_{\restriction \mathcal{L}'}, \mathcal{S}_{\restriction \mathcal{L}'}, \mathcal{M}_{\restriction \mathcal{L}'})$
  else
    $\mathcal{T}' := \{t \in \mathcal{T} \mid \mathcal{R}(t) = \omega\}$
    $\mathcal{R} := \mathsf{TimeBounds}(\mathcal{R}, \mathcal{S}, \mathcal{T}')$
  fi
  for all SCCs C of the RVG in topological order **do**
    $\mathcal{S} := \mathsf{SizeBounds}(\mathcal{R}, \mathcal{S}, C)$
  done
done
return $(\mathcal{T}, \mathcal{M}, \mathcal{R}, \mathcal{S})$

Fig. 12. Alternating modular complexity analysis

Here, the sub-procedures TimeBounds, SizeBounds, and

Reduce are as defined in Thm. 3.6, Thm. 4.6 and Thm. 4.15, and Def. 6.11. When applying TimeBounds to transitions with multiple target locations, we also apply the extension of Sect. 5.2 which uses PRFs to infer exponential runtime bounds. As some of our sub-procedures change the program $\mathcal{T}$ and its cost measure $\mathcal{M}$, `KoAT` does not only return runtime and size bounds $\mathcal{R}$ and $\mathcal{S}$, but also the updated versions of $\mathcal{T}$ and $\mathcal{M}$. With these results, one can then compute a cost approximation $\mathcal{C}_{(\mathcal{T},\mathcal{M})}$ as in Def. 6.3. This cost approximation is an upper bound for the cost complexity of the original program.

To find polynomial ranking functions in TimeBounds, we employ a standard synthesis procedure based on Farkas' Lemma [Podelski and Rybalchenko 2004; Alias et al. 2010]. To handle SMT queries, we use the `Z3` SMT solver [Moura and Bjørner 2008]. The procedure `SepHeuristic` is used to identify a sub-program that can be separated, and is implemented as described in Sect. 6.2. In our implementation, we allow to backtrack from the bottom-up approach if `KoAT` fails to find time bounds for all transitions in the separated program (i.e., if there is a t with $\mathcal{R}_{\mathcal{T}_{|\mathcal{L}'}}(t) = \omega$), and proceed to the top-down approach in the else-case with the standard TimeBounds procedure instead.

Finally, we use the sub-procedure InformationPropagate to perform simplifications of the problem. More precisely, this sub-procedure uses four techniques:

— **Removing unreachable transitions**. In the very first iteration of the **while**-loop, we remove transitions t that cannot be reached from the initial location, i.e., transitions t where there are no $\boldsymbol{v}_0, F$ with $\{(\ell_0, \boldsymbol{v}_0)\} \to^* \circ \to_t F$. As in the computation of the RVG where we over-approximate pre, this is approximated by decidable sufficient criteria.

— **Knowledge propagation**. We propagate runtime information to successor transitions in simple sequences of code. For a location ℓ with only one outgoing transition $t = (\ell, \tau, \mathcal{P})$, we use that t cannot be used more often than all of its predecessors together, and set $\mathcal{R}(t) = \sum_{\tilde{t} \in \mathsf{pre}(t)} \mathcal{R}(\tilde{t})$ whenever $\mathcal{R}(\tilde{t}) \neq \omega$ for all $\tilde{t} \in \mathsf{pre}(t)$ and $\ell \notin \mathcal{P}$. So while we used a (trivial) constant ranking function to infer $\mathcal{R}_4(t_3) = 1 + |\mathtt{x}|$ in Ex. 3.7, `KoAT` computes this directly as $\mathcal{R}_3(t_2) + \mathcal{R}_3(t_5)$, without synthesizing a ranking function.
Similarly, if all transitions only have a single target location, we assign the trivial runtime bound 1 for transitions that do not occur in cycles of the program graph. For example, in Fig. 2, this procedure directly infers $\mathcal{R}(t_2) = 1$.

— **Invariant generation**. To infer valuable program invariants relating variables with each other, we use the `Apron` library [Jeannet and Miné 2009]. `Apron` is an abstract interpretation [Cousot and Cousot 1977] framework implementing a number of numerical domains. In our experience, the Octagon [Miné 2006] domain with an aggressive widening strategy yields the best tradeoff between precision and performance.

— **Transition chaining**. Heuristically (more precisely, when other techniques do not yield new results anymore), we use a complexity-preserving variation of the chaining technique by Falke et al. [2011] to "unroll" loops.

Note that the procedure `KoAT` calls itself recursively on an isolated sub-program when using the bottom-up approach in the **then**-case of Fig. 12. In these cases, a simple memoization technique can be used to avoid recomputing results for reappearing sub-programs (e.g., code copies or inlined procedures).

7.3. Evaluation

To evaluate our approach, we compare our implementation `KoAT` with `PUBS` [Albert et al. 2011a; Albert et al. 2012], `Rank` [Alias et al. 2010], `Loopus` [Sinn et al. 2014], and `CoFloCo` [Flores-Montoya and Hähnle 2014]. We also compare our new version with `KoAT-TACAS`, which implements only the techniques described in the preliminary version [Brockschmidt et al. 2014] of our paper (i.e., it does not feature exponential size and time

Tool	1	$\log n$	n	$n \log n$	n^2	n^3	$n^{>3}$	EXP	No res.	Time
KoAT	131	0	167	0	78	7	3	18	285	0.7 s
CoFloCo	117	0	153	0	66	9	2	0	342	1.3 s
Loopus	117	0	130	0	49	5	5	0	383	0.2 s
KoAT-TACAS	118	0	127	0	50	0	3	0	391	1.1 s
PUBS	109	4	127	6	24	8	0	7	404	0.8 s
Rank	56	0	16	0	8	1	0	0	608	0.1 s
Loopus (authors)	128	0	161	0	69	7	7	0	286	n/a

Fig. 13. Experimental results, grouped by asymptotic complexity classes

bounds (Sect. 4 and Sect. 5), the handling of recursion (Sect. 5), or the bottom-up approach (Sect. 6)). We contacted the authors of SPEED [Gulwani et al. 2009], but were not able to obtain their tool. We decided not to compare KoAT to ABC [Blanc et al. 2010], RAML [Hoffmann et al. 2012; Hoffmann and Shao 2014], or r-TuBound [Knoop et al. 2012], as their input or analysis goals differ considerably from ours.

As benchmarks, we collected 689 programs from the literature on termination and complexity of integer programs. These include all 36 examples from the evaluation of Rank, all 53 programs used to evaluate PUBS except one program with undefined semantics, all 27 examples from the evaluations of SPEED, and all examples from this paper. The large majority of our example collection has also been used in the evaluations of Loopus and CoFloCo. The collection contains 50 recursive examples, which cannot be analyzed with KoAT-TACAS, Rank, and Loopus, and 20 examples with non-linear arithmetic, which can be handled by neither Rank nor PUBS.

Where examples were available as C programs, we used the tool KITTeL [Falke et al. 2011] to transform them into integer programs automatically. To compare KoAT, PUBS, Rank, and CoFloCo, we used automatic translators between the different integer program formats provided by the authors of the respective tools (in the case of PUBS and CoFloCo) or by us (in the case of Rank). Loopus only accepts C programs as input. We used a translation from our integer programs to C/goto programs to compare with Loopus, even for examples which were originally available as C programs. The reason for this is that we wanted to avoid differences that were due to different integer abstractions of C programs. However, as the results differ greatly from the results reported by the authors of Loopus in [Sinn et al. 2014], we have also included their results in our evaluation, in the row "Loopus (authors)". Note that these results include 81 cases where Loopus reports a bound depending on a value non-deterministically chosen in the program, whereas we consider the runtime of such programs as unbounded in the row for Loopus since it does not depend on the values of the input parameters (cf. Def. 2.3). Also note that the numbers in this row refer to a run on a subset of 658 examples.

The source repository for KoAT also contains all the scripts created by us for this evaluation and for the generation of the detailed reports, together with a document explaining how to reproduce the experiments.

The evaluation results are summarized in Fig. 13, showing how often each tool could infer a specific runtime bound for the example set. Here, 1, $\log n$, n, $n \log n$, n^2, n^3, and $n^{>3}$ represent their corresponding asymptotic classes and EXP is the class of exponential functions. In order to simplify the comparison, we did not perform a more fine-grained analysis (e.g., we represent all linear bounds by the same asymptotic class n, although one linear bound might be tighter or incomparable with another linear bound). The column "No res." lists the number of examples for which the tool could not find any bound (i.e., where it either failed or ran into a timeout). In the column "Time", we give the average runtime on those examples where the respective tool was successful. The benchmarks were executed on

a computer with 6GB of RAM and an Intel i7 CPU clocked at 3.07 GHz, using a timeout of 60 seconds for each example. A longer timeout did not yield additional results.

On this collection, our approach was more powerful than the competing tools. The experiments also clearly show the usefulness of the new contributions compared to the preliminary version of our paper [Brockschmidt et al. 2014] (since KoAT now succeeds on $689-285 = 404$ examples, whereas KoAT-TACAS only succeeded on $689-391 = 298$ programs). However, our experiments show that all the different tools have their own strengths. To illustrate this, consider the results in Fig. 14, which show for how many results the bounds reported by a tool were (asymptotically) tighter (i.e., more precise) than those reported by KoAT and vice versa. For details, we refer to our website, which allows to view all examples, results, and proof outputs, and also offers dynamic controls to compare the results of different tools:

Compared tool	more precise	less precise
CoFloCo	31	80
KoAT-TACAS	0	118
PUBS	46	134
Loopus	16	117
Rank	5	327

Fig. 14. Comparison of results relative to KoAT

http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity-Journal/

8. CONCLUSION

We presented an alternating modular approach for runtime and size complexity analysis of integer programs, shown in Fig. 12. Each step only considers a small part of the program, and runtime bounds help to infer (possibly non-linear and even non-polynomial) size bounds and vice versa, cf. Sect. 3 and 4.

In Sect. 5, we provided an extension to handle (possibly recursive) procedure calls in a modular fashion and to use polynomial ranking functions for the inference of exponential runtime bounds as well. Moreover, in Sect. 6 we showed how to treat other forms of bounds (e.g., on the number of sent network requests) and how to compute bounds for separate program parts in advance or in parallel. The power of our approach was demonstrated by an implementation in the tool KoAT and an extensive experimental comparison with related tools in Sect. 7.

In future work, we intend to investigate how to use our method for high-level programming languages such as C and Java. To this end, we plan to combine the framework presented in this paper with earlier work on complexity analysis of term rewriting systems [Noschinski et al. 2013], and use terms to represent inductive heap data structures (such as lists or trees).

A limitation of our implementation is that it only generates PRFs to detect polynomial and exponential bounds. In contrast, PUBS uses PRFs to find logarithmic complexity bounds as well [Albert et al. 2011a]. As mentioned in Sect. 5.2, such an extension could also be directly integrated into our method. Moreover, we are restricted to weakly monotonic bounds in order to allow their modular composition. Thus, for a loop like

$$\textbf{while } \mathtt{x} < \mathtt{y} \textbf{ do } \mathtt{x} := \mathtt{x} + 1$$

we can only infer the runtime bound $|\mathtt{y}| + |\mathtt{x}|$ instead of $|\mathtt{y} - \mathtt{x}|$. Another limitation is that our size analysis only handles certain forms of local size bounds in non-trivial SCCs of the result variable graph. For that reason, it often over-approximates the sizes of variables that are both incremented and decremented in the same loop. Due to all these imprecisions, our approach sometimes infers bounds that are asymptotically larger than the actual asymptotic costs.

Thus, another interesting line of research for future work is to improve our core complexity analysis. For this, our procedure TimeBounds could be strengthened by combining it with recent advances in finding complexity bounds [Sinn et al. 2014] and by using recurrence

solving. Furthermore, the precision of our analysis could be improved by using ideas from control-flow refinement (as those by Flores-Montoya and Hähnle [2014]). Moreover, the techniques in the tool ABC [Blanc et al. 2010] could be invoked whenever our separated analysis from Sect. 6.2 proposes a sub-program whose shape is suitable for the specialized analysis of Blanc et al. [2010]. We are also interested in experimenting with Max-SMT solving to infer more precise PRFs, as this strategy has shown great promise for termination proving [Larraz et al. 2013]. Finally, similar to the coupling of COSTA with the tool KeY [Albert et al. 2011b], we want to automatically *certify* the complexity bounds found by our implementation KoAT.

Acknowledgments. We thank the anonymous reviewers for their in-depth reviews and comments – their work contributed greatly to the readability of this paper. Furthermore, we thank Amir Ben-Amram, Byron Cook, Christian von Essen, and Carsten Otto for valuable discussions and Christophe Alias, Antonio Flores-Montoya, Samir Genaim, and Moritz Sinn for help with the experiments.

REFERENCES

Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2008. Termination Analysis of Java Bytecode. In *FMOODS '08*. 2–18.

Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2011a. Closed-Form Upper Bounds in Static Cost Analysis. *JAR* 46, 2 (2011), 161–203.

Elvira Albert, Richard Bubel, Samir Genaim, Reiner Hähnle, Germán Puebla, and Guillermo Román-Díez. 2011b. Verified Resource Guarantees using COSTA and KeY. In *PEPM '11*. 73–76.

Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2012. Cost Analysis of Object-Oriented Bytecode Programs. *TCS* 413, 1 (2012), 142–159.

Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-Dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS '10*. 117–133.

Diego Esteban Alonso-Blas, Puri Arenas, and Samir Genaim. 2013. Precise Cost Analysis via Local Reasoning. In *ATVA '13*. 319–333.

Martin Avanzini and Georg Moser. 2013. A Combination Framework for Complexity. In *RTA '13*. 55–70.

Roberto Bagnara, Fred Mesnard, Andrea Pescetti, and Enea Zaffanella. 2012. A New Look at the Automatic Synthesis of Linear Ranking Functions. *IC* 215 (2012), 47–67.

Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. 2008. Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time. In *CiE '08*. 67–76.

Amir M. Ben-Amram and Samir Genaim. 2013. On the Linear Ranking Problem for Integer Linear-Constraint Loops. In *POPL '13*. 51–62.

Nikolaj Bjørner, Fabio Fioravanti, Andrey Rybalchenko, and Valerio Senni (Eds.). 2014. *First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014*. EPTCS, Vol. 169.

Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. 2010. ABC: Algebraic Bound Computation for Loops. In *LPAR '10*. 103–118.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *CAV '05*. 491–504.

Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. 2012. Automated Termination Proofs for Java Programs with Cyclic Data. In *CAV '12*. 105–122.

Marc Brockschmidt, Byron Cook, and Carsten Fuhs. 2013. Better Termination Proving Through Cooperation. In *CAV '13*. 413–429.

Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *TACAS '14*. 140–155.

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination Proofs for Systems Code. In *PLDI '06*. 415–426.

Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *TACAS '13*. 47–61.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL '77*. 238–252.

Saumya Debray and Nai-Wei Lin. 1993. Cost Analysis of Logic Programs. *TOPLAS* 15 (1993), 826–875.

Saumya Debray, Pedro López-García, Manuel V. Hermenegildo, and Nai-Wei Lin. 1997. Lower Bound Cost Estimation for Logic Programs. In *ILPS '97*. 291–305.

Stephan Falke, Deepak Kapur, and Carsten Sinz. 2011. Termination Analysis of `C` Programs Using Compiler Intermediate Languages. In *RTA '11*. 41–50.

Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *APLAS '14*. 275–295.

Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. 2007. SAT Solving for Termination Analysis with Polynomial Interpretations. In *SAT '07*. 340–354.

Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. 2009. Proving Termination of Integer Term Rewriting. In *RTA '09*. 32–47.

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. 2006. Mechanizing and Improving Dependency Pairs. *JAR* 37, 3 (2006), 155–203.

Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs. 2012. Symbolic Evaluation Graphs and Term Rewriting: A General Methodology for Analyzing Logic Programs. In *PPDP '12*. 1–12.

Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2014. Proving Termination of Programs Automatically with `AProVE`. In *IJCAR '14*. 184–191.

Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing Software Verifiers from Proof Rules. In *PLDI '12*. 405–416.

Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. `SPEED`: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL '09*. 127–139.

William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. 2010. Alternation for Termination. In *SAS '10*. 304–319.

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *CAV '14*. 797–813.

Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla. 2012. An Overview of Ciao and its Design Philosophy. *TPLP* 12, 1-2 (2012), 219–252.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate Amortized Resource Analysis. *TOPLAS* 34, 3 (2012).

Jan Hoffmann and Zhong Shao. 2014. Type-Based Amortized Resource Analysis with Integers and Arrays. In *FLOPS '14*. 152–168.

Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. 2012. A Verification Toolkit for Numerical Transition Systems – Tool Paper. In *FM '12*. 247–251.

Bertrand Jeannet and Antoine Miné. 2009. `Apron`: A Library of Numerical Abstract Domains for Static Analysis. In *CAV '09*. 661–667.

Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. 2012. `r-TuBound`: Loop Bounds for WCET Analysis. In *LPAR '12*. 435–444.

Dallas Lankford. 1979. *On Proving Term Rewriting Systems are Noetherian*. Technical Report MTP-3. Louisiana Technical University, Ruston, LA, USA.

Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2013. Proving Termination of Imperative Programs Using Max-SMT. In *FMCAD '13*. 218–225.

Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The Size-Change Principle for Program Termination. In *POPL '01*. 81–92.

Jan Leike and Matthias Heizmann. 2014. Ranking Templates for Linear Loops. In *TACAS '14*. 172–186.

Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. 2010. Automatic Numeric Abstractions for Heap-Manipulating Programs. In *POPL '10*. 211–222.

Antoine Miné. 2006. The Octagon Abstract Domain. *HOSC* 19, 1 (2006), 31–100.

Leonardo M. de Moura and Nikolaj Bjørner. 2008. `Z3:` An Efficient SMT Solver. In *TACAS '08*. 337–340.

Jorge A. Navas, Edison Mera, Pedro López-García, and Manuel V. Hermenegildo. 2007. User-Definable Resource Bounds Analysis for Logic Programs. In *ICLP '07*. 348–363.

Lars Noschinski, Fabian Emmes, and Jürgen Giesl. 2013. Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs. *JAR* 51, 1 (2013), 27–56.

Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI '04*. 239–251.

Alejandro Serrano, Pedro López-García, and Manuel V. Hermenegildo. 2014. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *TPLP* 14, 4-5 (2014), 739–754.

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *CAV '14*. 745–761.

Fausto Spoto, Fred Mesnard, and Étienne Payet. 2010. A Termination Analyser for `Java Bytecode` Based on Path-Length. *TOPLAS* 32, 3 (2010).

Aliaksei Tsitovich, Natasha Sharygina, Christoph M. Wintersteiger, and Daniel Kroening. 2011. Loop Summarization and Termination Analysis. In *TACAS '11*. 81–95.

Tao Wei, Jian Mao, Wei Zou, and Yu Chen. 2007. A New Algorithm for Identifying Loops in Decompilation. In *SAS '07*. 170–183.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. 2008. The Worst-Case Execution-Time Problem: Overview of Methods and Survey of Tools. *TECS* 7, 3 (2008), 36:1–36:53.

Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-Change Abstraction. In *SAS '11*. 280–297.

A. PROOFS

In this section, we provide the proofs for all theorems of our paper.

Remark 2.5 (*Approximating* rc). Let $\mathcal{R}$ be a runtime approximation for $\mathcal{T}$. Then $\sum_{t\in\mathcal{T}} \mathcal{R}(t) \geq \mathrm{rc}$.

Proof. We have to show that $(\sum_{t\in\mathcal{T}}(\mathcal{R}(t))(\boldsymbol{m})) \geq \mathrm{rc}(\boldsymbol{m})$ holds for all $\boldsymbol{m} \in \mathbb{N}^n$. We first regard the case where there is a valuation $\boldsymbol{v}_0$ with $\boldsymbol{v}_0 \leq \boldsymbol{m}$ such that $(\ell_0, \boldsymbol{v}_0)$ starts an infinite evaluation (thus, $\mathrm{rc}(\boldsymbol{m}) = \omega$). Hence, there is some transition $t \in \mathcal{T}$ which is used infinitely often in this evaluation. Thus, we have $(\mathcal{R}(t))(\boldsymbol{m}) = \omega$ and therefore $(\sum_{t\in\mathcal{T}}(\mathcal{R}(t))(\boldsymbol{m})) = \omega$.

Otherwise, it suffices to prove that for any $\boldsymbol{v}_0$ with $\boldsymbol{v}_0 \leq \boldsymbol{m}$ and $(\ell_0, \boldsymbol{v}_0) \to^k (\ell, \boldsymbol{v})$ for some configuration $(\ell, \boldsymbol{v})$, we have $(\sum_{t\in\mathcal{T}}(\mathcal{R}(t))(\boldsymbol{m})) \geq k$. Let $\mathcal{T} = \{t_1, \ldots, t_d\}$. Moreover, for each $i \in \{1, \ldots, d\}$, let k_i be the number of times that the transition t_i was used in the evaluation $(\ell_0, \boldsymbol{v}_0) \to^k (\ell, \boldsymbol{v})$. Then we obviously have $k = k_1 + \ldots + k_d$ and for each $i \in \{1, \ldots, d\}$, there is an evaluation w.r.t. $(\to^* \circ \to_{t_i})$ of length k_i that starts in the configuration $(\ell_0, \boldsymbol{v}_0)$. Hence, $(\mathcal{R}(t_i))(\boldsymbol{m}) \geq k_i$. This implies $(\sum_{t\in\mathcal{T}}(\mathcal{R}(t))(\boldsymbol{m})) \geq k_1 + \ldots + k_d = k$, as desired. □

In Sect. 5 we generalized transitions to allow several target locations. Sect. 5.1 adapted the notions of complexity to such transitions and Def. 5.2 correspondingly generalized the notion of PRFs from Def. 3.1. We now provide the proofs of soundness for Thm. 3.3 and Thm. 3.6 for these generalized notions of transitions and PRFs.

Theorem 3.3 (Complexities from PRFs). *Let $\mathcal{R}$ be a runtime approximation and $\mathcal{P}ol$ be a PRF for $\mathcal{T}$. Let $\mathcal{R}'(t) = [\mathcal{P}ol(\ell_0)]$ for all $t \in \mathcal{T}_\succ$ and $\mathcal{R}'(t) = \mathcal{R}(t)$ for all other $t \in \mathcal{T}$. Then, $\mathcal{R}'$ is also a runtime approximation.*

Proof. To prove

$$(\mathcal{R}'(t))(\boldsymbol{m}) \;\geq\; \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F \,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_0, \boldsymbol{v}_0)\}\,(\to^* \circ \to_t)^k\, F\}$$

for all $t \in \mathcal{T}$ and $\boldsymbol{m} \in \mathbb{N}^n$, it obviously suffices to show that

$$[\mathcal{P}ol(\ell_0)](\boldsymbol{m}) \;\geq\; \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F \,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_0, \boldsymbol{v}_0)\}\,(\to^* \circ \to_t)^k\, F\}$$

holds for all $t \in \mathcal{T}_\succ$ and $\boldsymbol{m} \in \mathbb{N}^n$. To this end, let $\boldsymbol{v}_0$ be a valuation with $\boldsymbol{v}_0 \leq \boldsymbol{m}$ and consider a (finite or infinite) evaluation of the form

$$\{(\ell_0, \boldsymbol{v}_0)\} \to_{t_0} \{(\ell_{1,1}, \boldsymbol{v}_{1,1}), \ldots, (\ell_{1,p_1}, \boldsymbol{v}_{1,p_1})\} \to_{t_1} \{(\ell_{2,1}, \boldsymbol{v}_{2,1}), \ldots, (\ell_{2,p_2}, \boldsymbol{v}_{2,p_2})\} \to_{t_2} \ldots$$

where $k \in \mathbb{N} \cup \{\omega\}$ steps are performed with the transition t. In the following, we also write $\ell_{0,1}$ for ℓ_0, $p_0 = 1$, and $\boldsymbol{v}_{0,1} = \boldsymbol{v}_0$.

Our goal is to show that $[\mathcal{P}ol(\ell_0)](\boldsymbol{m}) \geq k$ holds. This is trivial for the case $k = 0$, as $[\mathcal{P}ol(\ell_0)]$ only has non-negative coefficients. Thus, we now consider the case where $k > 0$.

Since $\mathcal{P}ol$ is a PRF for $\mathcal{T}$, we have

$$\begin{array}{c} \sum_{j\in\{1,\ldots,p_i\}} \max\{0, (\mathcal{P}ol(\ell_{i,j}))(\boldsymbol{v}_{i,j}(v_1), \ldots, \boldsymbol{v}_{i,j}(v_n))\} \\ \sum_{j\in\{1,\ldots,p_{i+1}\}} \max\{0, (\mathcal{P}ol(\ell_{i+1,j}))(\boldsymbol{v}_{i+1,j}(v_1), \ldots, \boldsymbol{v}_{i+1,j}(v_n))\} \end{array} \quad \geq$$

for all i.

Let $i_1 < i_2 < \ldots$ be the k indices where $t_i = t$. Then for all $i \in \{i_1, i_2, \ldots\}$, $t \in \mathcal{T}_\succ$ implies

$$\begin{array}{c} \sum_{j\in\{1,\ldots,p_i\}} \max\{0, (\mathcal{P}ol(\ell_{i,j}))(\boldsymbol{v}_{i,j}(v_1), \ldots, \boldsymbol{v}_{i,j}(v_n))\} \\ \sum_{j\in\{1,\ldots,p_{i+1}\}} \max\{0, (\mathcal{P}ol(\ell_{i+1,j}))(\boldsymbol{v}_{i+1,j}(v_1), \ldots, \boldsymbol{v}_{i+1,j}(v_n))\} \end{array} \quad >$$

and

$$\sum_{j\in\{1,\ldots,p_i\}} \max\{0, (\mathcal{P}ol(\ell_{i,j}))(\boldsymbol{v}_{i,j}(v_1), \ldots, \boldsymbol{v}_{i,j}(v_n))\} \geq 1.$$

Thus, we obtain

$$\begin{array}{ll}
\max\{0, (\mathcal{P}ol(\ell_0))(\boldsymbol{v}_0(v_1), \ldots, \boldsymbol{v}_0(v_n))\} & \geq \\
\sum_{j \in \{1,\ldots,p_{i_1}\}} \max\{0, (\mathcal{P}ol(\ell_{i_1,j}))(\boldsymbol{v}_{i_1,j}(v_1), \ldots, \boldsymbol{v}_{i_1,j}(v_n))\} & > \\
\sum_{j \in \{1,\ldots,p_{i_2}\}} \max\{0, (\mathcal{P}ol(\ell_{i_2,j}))(\boldsymbol{v}_{i_2,j}(v_1), \ldots, \boldsymbol{v}_{i_2,j}(v_n))\} & > \\
 & \vdots \\
\sum_{j \in \{1,\ldots,p_{i_{k-1}}\}} \max\{0, (\mathcal{P}ol(\ell_{i_{k-1},j}))(\boldsymbol{v}_{i_{k-1},j}(v_1), \ldots, \boldsymbol{v}_{i_{k-1},j}(v_n))\} & > \\
\sum_{j \in \{1,\ldots,p_{i_k}\}} \max\{0, (\mathcal{P}ol(\ell_{i_k,j}))(\boldsymbol{v}_{i_k,j}(v_1), \ldots, \boldsymbol{v}_{i_k,j}(v_n))\} & \geq 1.
\end{array}$$

From $\max\{0, (\mathcal{P}ol(\ell_0))(\boldsymbol{v}_0(v_1), \ldots, \boldsymbol{v}_0(v_n))\} \geq 1$ we obtain in particular that $0 \neq \max\{0, (\mathcal{P}ol(\ell_0))(\boldsymbol{v}_0(v_1), \ldots, \boldsymbol{v}_0(v_n))\} = (\mathcal{P}ol(\ell_0))(\boldsymbol{v}_0(v_1), \ldots, \boldsymbol{v}_0(v_n))$. So we have $k \neq \omega$ and $\mathcal{P}ol(\ell_0)\,(\boldsymbol{v}_0(v_1), \ldots, \boldsymbol{v}_0(v_n)) \geq k$, i.e., $\mathcal{P}ol(\ell_0)\,(\boldsymbol{v}_0(v_1), \ldots, \boldsymbol{v}_0(v_n))$ is an upper bound on the number of evaluation steps with the transition t. This implies

$$\begin{array}{ll}
[\mathcal{P}ol(\ell_0)]\,(\boldsymbol{m}) & \geq \\
[\mathcal{P}ol(\ell_0)]\,(|\boldsymbol{v}_0(v_1)|, \ldots, |\boldsymbol{v}_0(v_n)|) & \geq \\
\mathcal{P}ol(\ell_0)\,(\boldsymbol{v}_0(v_1), \ldots, \boldsymbol{v}_0(v_n)) & \geq k.
\end{array}$$

□

In the following theorem, for any location ℓ, let $\mathcal{T}_\ell$ be the multiset of all transitions $(\tilde{\ell}, \tilde{\tau}, \mathcal{P}) \in \mathcal{T} \setminus \mathcal{T}'$ with $\ell \in \mathcal{P}$. Here, $(\tilde{\ell}, \tilde{\tau}, \mathcal{P})$ is contained k times in $\mathcal{T}_\ell$ iff ℓ is contained k times in $\mathcal{P}$. Moreover, let $\mathcal{L}' = \{\ell \mid \mathcal{T}_\ell \neq \varnothing \wedge \exists\, \mathcal{P}'.\, (\ell, \tau, \mathcal{P}') \in \mathcal{T}'\}$ contain all entry locations of $\mathcal{T}'$.

THEOREM 3.6 (TimeBounds). *Let $(\mathcal{R}, \mathcal{S})$ be a complexity approximation, let $\mathcal{T}' \subseteq \mathcal{T}$ such that $\mathcal{T}'$ contains no initial transitions, and let $\mathcal{P}ol$ be a PRF for $\mathcal{T}'$. Let $\mathcal{R}'(t) = \sum_{\ell \in \mathcal{L}', \tilde{t} \in \mathcal{T}_\ell} \mathcal{R}(\tilde{t}) \cdot [\mathcal{P}ol(\ell)](\mathcal{S}(\tilde{t}, v'_1), \ldots, \mathcal{S}(\tilde{t}, v'_n))$ for $t \in \mathcal{T}'_\succ$ and $\mathcal{R}'(t) = \mathcal{R}(t)$ for all $t \in \mathcal{T} \setminus \mathcal{T}'_\succ$. Then, $\mathsf{TimeBounds}(\mathcal{R}, \mathcal{S}, \mathcal{T}') = \mathcal{R}'$ is also a runtime approximation.*

PROOF. To prove

$$(\mathcal{R}'(t))(\boldsymbol{m}) \;\geq\; \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F\,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_0, \boldsymbol{v}_0)\}\,(\to^* \circ \to_t)^k\, F\}$$

for all $t \in \mathcal{T}$ and $\boldsymbol{m} \in \mathbb{N}^n$, similar to the proof of Thm. 3.3, it suffices to show that for all $t \in \mathcal{T}'_\succ$ and $\boldsymbol{m} \in \mathbb{N}^n$, we have

$$\begin{array}{l}
\sum_{\ell \in \mathcal{L}', \tilde{t} \in \mathcal{T}_\ell} (\mathcal{R}(\tilde{t}))\,(\boldsymbol{m}) \cdot [\mathcal{P}ol(\ell)]((\mathcal{S}(\tilde{t}, v'_1))(\boldsymbol{m}), \ldots, (\mathcal{S}(\tilde{t}, v'_n))(\boldsymbol{m})) \quad \geq \\
\sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F\,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_0, \boldsymbol{v}_0)\}\,(\to^* \circ \to_t)^k\, F\}.
\end{array}$$

To this end, let $\boldsymbol{v}_0$ again be a valuation with $\boldsymbol{v}_0 \leq \boldsymbol{m}$ and consider a (finite or infinite) evaluation starting with the configuration $\{(\ell_0, \boldsymbol{v}_0)\}$ where $k \in \mathbb{N} \cup \{\omega\}$ steps are performed with the transition t. The goal now is to show

$$\sum_{\ell \in \mathcal{L}', \tilde{t} \in \mathcal{T}_\ell} (\mathcal{R}(\tilde{t}))\,(\boldsymbol{m}) \cdot [\mathcal{P}ol(\ell)]((\mathcal{S}(\tilde{t}, v'_1))(\boldsymbol{m}), \ldots, (\mathcal{S}(\tilde{t}, v'_n))(\boldsymbol{m})) \;\geq\; k.$$

As in the proof of Thm. 3.3, this is trivial for $k = 0$. Thus, we now consider the case where $k > 0$.

For any set of transitions $\mathcal{T}$, let $\to_{\mathcal{T}} = \bigcup_{t \in \mathcal{T}} \to_t$ and let $\to^+_{\mathcal{T}}$ denote the transitive closure of $\to_{\mathcal{T}}$. Then we can represent the considered evaluation as

$$\begin{array}{lll}
\{(\ell_0, \boldsymbol{v}_0)\} & \to^+_{\mathcal{T} \setminus \mathcal{T}'} & \{(\hat{\ell}_{1,1}, \hat{\boldsymbol{v}}_{1,1}), \ldots, (\hat{\ell}_{1,\hat{p}_1}, \hat{\boldsymbol{v}}_{1,\hat{p}_1})\} \to^+_{\mathcal{T}'} \\
\{(\ell_{1,1}, \boldsymbol{v}_{1,1}), \ldots, (\ell_{1,p_1}, \boldsymbol{v}_{1,p_1})\} & \to^+_{\mathcal{T} \setminus \mathcal{T}'} & \{(\hat{\ell}_{2,1}, \hat{\boldsymbol{v}}_{2,1}), \ldots, (\hat{\ell}_{2,\hat{p}_2}, \hat{\boldsymbol{v}}_{2,\hat{p}_2})\} \to^+_{\mathcal{T}'} \\
\{(\ell_{2,1}, \boldsymbol{v}_{2,1}), \ldots, (\ell_{2,p_2}, \boldsymbol{v}_{2,p_2})\} & \to^+_{\mathcal{T} \setminus \mathcal{T}'} & \ldots,
\end{array} \tag{1}$$

where in the corresponding evaluation tree, for all i, j the outgoing edges of $(\hat{\ell}_{i,j}, \hat{\boldsymbol{v}}_{i,j})$ are labeled by transitions from $\mathcal{T}'$ and the outgoing edges of $(\ell_{i,j}, \boldsymbol{v}_{i,j})$ are labeled by transitions from $\mathcal{T} \setminus \mathcal{T}'$.

Now we have to investigate how often the transition t is used in the evaluation (1). Since $t \in \mathcal{T}'$, it can only be used in sequences of the form

$$\{(\hat{\ell}_{i,1}, \hat{\boldsymbol{v}}_{i,1}), \ldots, (\hat{\ell}_{i,\hat{p}_i}, \hat{\boldsymbol{v}}_{i,\hat{p}_i})\} \to_{\mathcal{T}'}^{+} \{(\ell_{i,1}, \boldsymbol{v}_{i,1}), \ldots, (\ell_{i,p_i}, \boldsymbol{v}_{i,p_i})\}. \tag{2}$$

As in the proof of Thm. 3.3 one can show that

$$\sum\nolimits_{j \in \{1,\ldots,\hat{p}_i\}} [\mathcal{P}ol(\hat{\ell}_{i,j})](|\hat{\boldsymbol{v}}_{i,j}(v_1)|, \ldots, |\hat{\boldsymbol{v}}_{i,j}(v_n)|)$$

is an upper bound on the number of times the transition t is used in the sequence (2).

For all $j \in \{1, \ldots, \hat{p}_i\}$, let the edge in the evaluation tree reaching $(\hat{\ell}_{i,j}, \hat{\boldsymbol{v}}_{i,j})$ be labeled by $\tilde{t}_{i,j}$. Thus, we have $\hat{\ell}_{i,j} \in \mathcal{L}'$ and $\tilde{t}_{i,j} \in \mathcal{T}_{\hat{\ell}_{i,j}}$. As $(\ell_0, \boldsymbol{v}_0) \to_{\mathcal{T}}^{*} \circ \to_{\tilde{t}_{i,j}} \{(\hat{\ell}_{i,1}, \hat{\boldsymbol{v}}_{i,1}), \ldots, (\hat{\ell}_{i,\hat{p}_i}, \hat{\boldsymbol{v}}_{i,\hat{p}_i})\}$ and $\boldsymbol{v}_0 \leq \boldsymbol{m}$, by the definition of size approximations we have $(\mathcal{S}(\tilde{t}_{i,j}, v'))(\boldsymbol{m}) \geq |\hat{\boldsymbol{v}}_{i,j}(v)|$. By the weak monotonicity of $[\mathcal{P}ol(\hat{\ell}_{i,j})]$, we obtain

$$\begin{array}{l}[\mathcal{P}ol(\hat{\ell}_{i,j})]((\mathcal{S}(\tilde{t}_{i,j}, v'_1))(\boldsymbol{m}), \ldots, (\mathcal{S}(\tilde{t}_{i,j}, v'_n))(\boldsymbol{m})) \;\geq \\ [\mathcal{P}ol(\hat{\ell}_{i,j})](|\hat{\boldsymbol{v}}_{i,j}(v_1)|, \ldots, |\hat{\boldsymbol{v}}_{i,j}(v_n)|).\end{array}$$

Thus, $\sum_{j \in \{1,\ldots,\hat{p}_i\}} [\mathcal{P}ol(\hat{\ell}_{i,j})]((\mathcal{S}(\tilde{t}_{i,j}, v'_1))(\boldsymbol{m}), \ldots, (\mathcal{S}(\tilde{t}_{i,j}, v'_n))(\boldsymbol{m}))$ is an upper bound on the number of times the transition t is used in the sequence (2).

It remains to examine how often a sequence like (2) can occur in the full evaluation (1). As observed above, the edge reaching $(\hat{\ell}_{i,j}, \hat{\boldsymbol{v}}_{i,j})$ in the evaluation tree is always labeled by some $\tilde{t}_{i,j} \in \mathcal{T}_{\hat{\ell}_{i,j}}$. Note that by defining $\mathcal{T}_\ell$ as a multiset, we take into account that the same transition $\tilde{t}_{i,j}$ might give rise to multiple sub-configurations with the same location $\hat{\ell}_{i,j}$. Thus, a sequence like (2) cannot occur more often than the transitions in $\mathcal{T}_{\hat{\ell}_{i,j}}$. Note that each $\tilde{t}_{i,j}$ can occur at most $(\mathcal{R}(\tilde{t}_{i,j}))(\boldsymbol{m})$ times in evaluations. As discussed above, in every $\mathcal{T}'$-sequence (2), the transition t can be applied at most $\sum_{j \in \{1,\ldots,\hat{p}_i\}} [\mathcal{P}ol(\hat{\ell}_{i,j})](|\hat{\boldsymbol{v}}_{i,j}(v_1)|, \ldots, |\hat{\boldsymbol{v}}_{i,j}(v_n)|)$ times. Thus, an upper bound for the number k of applications of t in the overall sequence (1) is

$$\sum\nolimits_{\ell \in \mathcal{L}', \tilde{t} \in \mathcal{T}_\ell} \mathcal{R}(\tilde{t})(\boldsymbol{m}) \cdot [\mathcal{P}ol(\ell)]((\mathcal{S}(\tilde{t}, v'_1))(\boldsymbol{m}), \ldots, (\mathcal{S}(\tilde{t}, v'_n))(\boldsymbol{m})).$$

□

To ease readability, we present the proofs for Thm. 4.6 and Thm. 4.15 only for transitions with single target locations. Their extension to transitions with multiple target locations is straightforward: instead of evaluations one simply has to regard paths of the evaluation tree.

THEOREM 4.6 (SizeBounds FOR TRIVIAL SCCs). *Let $(\mathcal{R}, \mathcal{S})$ be a complexity approximation, let $\mathcal{S}_l$ be a local size approximation, and let $\{\alpha\} \subseteq \mathsf{RV}$ be a trivial SCC of the RVG. We define $\mathcal{S}'(\alpha') = \mathcal{S}(\alpha')$ for $\alpha' \neq \alpha$ and*

- *$\mathcal{S}'(\alpha) = \mathcal{S}_l(\alpha)$, if $\alpha = |t, v'|$ for some initial transition t*
- *$\mathcal{S}'(\alpha) = \max\{\, \mathcal{S}_l(\alpha)(\mathcal{S}(\tilde{t}, v'_1), \ldots, \mathcal{S}(\tilde{t}, v'_n)) \mid \tilde{t} \in \mathsf{pre}(t)\}$, otherwise.*

Then $\mathsf{SizeBounds}(\mathcal{R}, \mathcal{S}, \{\alpha\}) = \mathcal{S}'$ is also a size approximation.

PROOF. Let $\alpha = |t, v'| \in \mathsf{RV}$ be the trivial SCC for which the processor was applied. We have to show that

$$(\mathcal{S}'(t, v'))(\boldsymbol{m}) \;\geq\; \sup\{|\boldsymbol{v}(v)| \mid \exists \boldsymbol{v}_0, \ell, \boldsymbol{v} \,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge (\ell_0, \boldsymbol{v}_0)(\to^* \circ \to_t)(\ell, \boldsymbol{v})\}$$

holds for all $\boldsymbol{m} \in \mathbb{N}^n$. To this end, we consider a valuation $\boldsymbol{v}_0 \leq \boldsymbol{m}$ and an evaluation

$$(\ell_0, \boldsymbol{v}_0)\ (\rightarrow^* \circ \rightarrow_t)\ (\ell, \boldsymbol{v}). \tag{3}$$

Now the goal is to show that $(\mathcal{S}'(t, v'))\,(\boldsymbol{m}) \geq |\boldsymbol{v}(v)|$ holds.

If t is an initial transition, then the evaluation (3) has the form $(\ell_0, \boldsymbol{v}_0) \rightarrow_t (\ell, \boldsymbol{v})$, since by definition there are no transitions leading back to the initial location ℓ_0. Thus, $(\mathcal{S}_l(t, v'))\,(\boldsymbol{m}) \geq |\boldsymbol{v}(v)|$. Since $\mathcal{S}(t, v')$ is defined as $\mathcal{S}_l(t, v')$ for initial transitions t, we obtain

$$(\mathcal{S}'(t, v'))\,(\boldsymbol{m}) \;=\; (\mathcal{S}_l(t, v'))\,(\boldsymbol{m}) \;\geq\; |\boldsymbol{v}(v)|.$$

In the case where t is not an initial transition, the evaluation (3) has the form

$$(\ell_0, \boldsymbol{v}_0)\ (\rightarrow^* \circ \rightarrow_{\tilde{t}})\ (\tilde{\ell}, \tilde{\boldsymbol{v}}) \;\rightarrow_t\; (\ell, \boldsymbol{v})$$

for some transition $\tilde{t} \in \mathsf{pre}(t)$. As $\boldsymbol{v}_0 \leq \boldsymbol{m}$, we have $(\mathcal{S}(\tilde{t}, v_i'))\,(\boldsymbol{m}) \geq |\tilde{\boldsymbol{v}}(v_i)|$ for all $i \in \{1, \ldots, n\}$, i.e., $\mathcal{S}(\tilde{t}, v_i')$ is a bound for the size of the variable v_i before the transition t is applied.

The local size change resulting from the transition t is approximated by the function $\mathcal{S}_l(t, v')$. Thus, we have

$$(\mathcal{S}_l(t, v'))\,((\mathcal{S}(\tilde{t}, v_1'))\,(\boldsymbol{m}), \ldots, (\mathcal{S}(\tilde{t}, v_n'))\,(\boldsymbol{m})) \;\geq\; |\boldsymbol{v}(v)|,$$

and hence $(\mathcal{S}(t, v'))\,(\boldsymbol{m}) \;\geq\; |\boldsymbol{v}(v)|$, as desired. □

Theorem 4.15 (SizeBounds for Non-Trivial SCCs). *Let $(\mathcal{R}, \mathcal{S})$ be a complexity approximation, $\mathcal{S}_l$ a local size approximation, and $C \subseteq \mathsf{RV}$ a non-trivial SCC of the RVG. If there is a $\beta \in C$ with $\beta \notin \dot{\times}$, then we set $\mathcal{S}' = \mathcal{S}$. Otherwise, for all $\beta \notin C$ let $\mathcal{S}'(\beta) = \mathcal{S}(\beta)$. For all $\beta \in C$, we set $\mathcal{S}'(\beta) =$*

$$\begin{aligned} s_{\dot{\times}} \;\cdot\; \Big(& \max(\,\{\mathcal{S}(\tilde{\alpha}) \mid \text{ there is an } \alpha \in C \text{ with } \tilde{\alpha} \in \mathsf{pre}(\alpha) \setminus C\} \;\cup\; \{c_\alpha \mid \alpha \in \doteq \cap\, C\}\,) \\ & + \textstyle\sum_{t \in \mathcal{T}} (\mathcal{R}(t) \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \doteq) \cap C_t\}) \\ & + \textstyle\sum_{t \in \mathcal{T}} (\mathcal{R}(t) \cdot \max\{e_\alpha + \sum_{v \in \mathsf{actV}(\mathcal{S}_l(\alpha)) \setminus \mathcal{V}_\alpha} f_v^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\}) \Big). \end{aligned}$$

Then $\mathsf{SizeBounds}(\mathcal{R}, \mathcal{S}, C) = \mathcal{S}'$ is also a size approximation.

Proof. We only regard the case where $\alpha \in \dot{\times}$ holds for all $\alpha \in C$. Then for all $|r, w'| \in \mathsf{RV}$ we have to show that

$$(\mathcal{S}'(r, w'))\,(\boldsymbol{m}) \;\geq\; \sup\{|\boldsymbol{v}(w)| \mid \exists \boldsymbol{v}_0, \ell, \boldsymbol{v}\,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge (\ell_0, \boldsymbol{v}_0)\,(\rightarrow^* \circ \rightarrow_r)\,(\ell, \boldsymbol{v})\}$$

holds for all $\boldsymbol{m} \in \mathbb{N}^n$. To this end, we now fix $\boldsymbol{m}$ to an arbitrary value and consider a fixed valuation $\boldsymbol{v}_0 \leq \boldsymbol{m}$ and a fixed evaluation

$$(\ell_0, \boldsymbol{v}_0)\ (\rightarrow^* \circ \rightarrow_r)\ (\ell, \boldsymbol{v}). \tag{4}$$

Our goal is to show that

$$(\mathcal{S}'(r, w'))\,(\boldsymbol{m}) \;\geq\; |\boldsymbol{v}(w)|.$$

For any transition $t \in \mathcal{T}$, let k_t be the number of times that the transition t was used in the evaluation (4). To simplify the remaining proof, we define the following values:

$$\begin{aligned} e \;&=\; \max(\,\{\mathcal{S}(\tilde{\alpha})\,(\boldsymbol{m}) \mid \text{ there is an } \alpha \in C \text{ with } \tilde{\alpha} \in \mathsf{pre}(\alpha) \setminus C\} \cup \{c_\alpha \mid \alpha \in \doteq \cap\, C\}\,) \\ f^\alpha \;&=\; \textstyle\sum_{v \in \mathsf{actV}(\mathcal{S}_l(\alpha)) \setminus \mathcal{V}_\alpha} f_v^\alpha(\boldsymbol{m}) \\ s_t \;&=\; \max\{s_\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\} \;\cdot\; \max\{|\mathcal{V}_\alpha| \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\} \\ d \;&=\; \Big(\textstyle\prod_{t \in \mathcal{T}} s_t^{k_t} \Big) \cdot \Big(e \;+ \textstyle\sum_{t \in \mathcal{T}} (k_t \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \doteq) \cap C_t\}) \\ &\qquad\qquad + \textstyle\sum_{t \in \mathcal{T}} (k_t \cdot \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\}) \Big) \end{aligned}$$

Then we prove the following claim:

$$d \geq |\boldsymbol{v}(w)|. \tag{5}$$

Note that Thm. 4.15 follows from Claim (5), as $(\mathcal{R}(t))(\boldsymbol{m}) \geq k_t$ holds by definition of k_t. Let $\gamma = |r, w'|$. The only interesting case is if $\gamma \in C$. Then we have

$$\begin{aligned}
(\mathcal{S}'(\gamma))(\boldsymbol{m}) &= \Big(\textstyle\prod_{t\in\mathcal{T}} s_t^{\mathcal{R}(t)}\Big) \cdot \Big(e + \textstyle\sum_{t\in\mathcal{T}}(\mathcal{R}(t)(\boldsymbol{m}) \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \doteq) \cap C_t\}) \\
&\qquad + \textstyle\sum_{t\in\mathcal{T}}(\mathcal{R}(t)(\boldsymbol{m}) \cdot \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\})\Big) \\
&\geq \Big(\textstyle\prod_{t\in\mathcal{T}} s_t^{k_t}\Big) \quad \cdot \Big(e + \textstyle\sum_{t\in\mathcal{T}}(k_t \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \doteq) \cap C_t\}) \\
&\qquad + \textstyle\sum_{t\in\mathcal{T}}(k_t \cdot \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\})\Big) \\
&= d \\
&\geq |\boldsymbol{v}(w)| \qquad \text{by (5).}
\end{aligned}$$

We prove the claim (5) by induction on the length of the evaluation (4). Intuitively, we show that we correctly approximate the effect of the *last* transition step r on the size of the value obtained so far (which in turn is captured by the induction hypothesis).

Note that r cannot be an initial transition, since there are no transitions leading back to the initial location ℓ_0 (i.e., then $\gamma = |r, w'|$ would not be contained in a non-trivial SCC C of the result variable graph). Thus, the reduction (4) has the form

$$(\ell_0, \boldsymbol{v}_0)\ (\to^* \circ \to_{\tilde{r}})\ (\tilde{\ell}, \tilde{\boldsymbol{v}})\ \to_r\ (\ell, \boldsymbol{v})$$

for some transition $\tilde{r} \in \mathsf{pre}(r)$. In the induction step, we perform a case analysis depending on the class of the local size bound $\mathcal{S}_l(\gamma)$. As induction hypothesis, we assume that (5) holds for the evaluation leading up to $(\tilde{\ell}, \tilde{\boldsymbol{v}})$.

Case 1: $\gamma \in \doteq$

Then for all $u_1, \ldots, u_n \in \mathbb{N}$, we have

$$\max\{c_\gamma, u_1, \ldots, u_n\} \geq (\mathcal{S}_l(\gamma))\,(u_1, \ldots, u_n). \tag{6}$$

While (6) describes the local effect of the transition r (i.e., of the last step in the evaluation (4)), we now have to estimate the sizes of the input variables $u_1, \ldots, u_n$ of r. So for each $v_i \in \mathcal{V}$, we have to find a bound on its size after the application of the transition $\tilde{r}$ that precedes r.

If $|\tilde{r}, v_i'|$ is also a result variable of the SCC C (i.e., $|\tilde{r}, v_i'| \in C$), then the induction hypothesis implies that the claim (5) also holds for the shorter reduction from $(\ell_0, \boldsymbol{v}_0)$ to $(\tilde{\ell}, \tilde{\boldsymbol{v}})$. In other words, we have $d \geq |\tilde{\boldsymbol{v}}(v_i)|$. Of course, one might even obtain a more precise bound than d, because one could replace k_r by $k_r - 1$ now. However, the bound $d \geq |\tilde{\boldsymbol{v}}(v_i)|$ is already sufficient for our purpose.

If $|\tilde{r}, v_i'| \notin C$, then $\boldsymbol{v}_0 \leq \boldsymbol{m}$ implies $(\mathcal{S}(\tilde{r}, v_i'))\,(\boldsymbol{m}) \geq |\tilde{\boldsymbol{v}}(v_i)|$. As $|\tilde{r}, v_i'| \in \mathsf{pre}(\gamma) \setminus C$, we have $e \geq (\mathcal{S}(\tilde{r}, v_i'))\,(\boldsymbol{m})$, which implies $d \geq e \geq (\mathcal{S}(\tilde{r}, v_i'))\,(\boldsymbol{m}) \geq |\tilde{\boldsymbol{v}}(v_i)|$. So irrespective of whether $|\tilde{r}, v_i'|$ is in the same SCC C or not, we always obtain $d \geq |\tilde{\boldsymbol{v}}(v_i)|$.

As $d \geq e \geq c_\alpha$ for all $\alpha \in \doteq \cap\, C$, we also have $d \geq c_\gamma$. Hence,

$$\begin{aligned}
d &\geq \max\{c_\gamma, |\tilde{\boldsymbol{v}}(v_1)|, \ldots, |\tilde{\boldsymbol{v}}(v_n)|\} \\
&\geq (\mathcal{S}_l(\gamma))\,(|\tilde{\boldsymbol{v}}(v_1)|, \ldots, |\tilde{\boldsymbol{v}}(v_n)|) && \text{by (6)} \\
&\geq |\boldsymbol{v}(w)| && \text{by definition of local size approximations.}
\end{aligned}$$

Case 2: $\gamma \in \dot{+} \setminus \doteq$

Now for all $u_1, \ldots, u_n \in \mathbb{N}$, we have

$$d_\gamma + \max\{u_1, \ldots, u_n\} \geq (\mathcal{S}_l(\gamma))\,(u_1, \ldots, u_n). \tag{7}$$

Again, we have to estimate the sizes of the input variables $u_1, \ldots, u_n$, i.e., we have to find a bound on the size of each $v_i \in \mathcal{V}$ after the application of the transition $\tilde{r}$ that precedes r.

If $|\tilde{r}, v_i'| \in C$, then the induction hypothesis implies that (5) also holds for the reduction from $(\ell_0, \boldsymbol{v}_0)$ to $(\tilde{\ell}, \tilde{\boldsymbol{v}})$. Let

$$d' \;=\; \Big(\prod_{t\in\mathcal{T}} s_t^{k_t}\Big) \cdot \Big(e \;+\sum_{t\in\mathcal{T}\setminus\{r\}}(k_t \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \dot{=}) \cap C_t\}) \\ +(k_r - 1) \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \dot{=}) \cap C_r\} \\ +\sum_{t\in\mathcal{T}}(k_t \cdot \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\})\Big).$$

The reason for using "$k_r - 1$" in d' is that the last application of the transition r in the evaluation (4) is missing in the evaluation from $(\ell_0, \boldsymbol{v}_0)$ to $(\tilde{\ell}, \tilde{\boldsymbol{v}})$. Then the induction hypothesis implies $d' \geq |\tilde{\boldsymbol{v}}(v_i)|$. Again, one might even obtain a more precise bound than d' by replacing all occurrences of k_r by $k_r - 1$. However, the bound $d' \geq |\tilde{\boldsymbol{v}}(v_i)|$ is already sufficient for our purpose.

If $|\tilde{r}, v_i'| \notin C$, then as in Case 1 we obtain $d' \geq e \geq (\mathcal{S}(\tilde{r}, v_i'))\,(\boldsymbol{m}) \geq |\tilde{\boldsymbol{v}}(v_i)|$. So irrespective of whether $|\tilde{r}, v_i'|$ is in the same SCC C or not, we always get $d' \geq |\tilde{\boldsymbol{v}}(v_i)|$. Thus, we also have $d' \geq \max\{|\tilde{\boldsymbol{v}}(v_1)|, \ldots, |\tilde{\boldsymbol{v}}(v_n)|\}$. Hence,

$$\begin{aligned} d &\geq \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \dot{=}) \cap C_r\} + d' && \text{as } \textstyle\prod_{t\in\mathcal{T}} s_t^{k_t} \geq 1 \\ &\geq d_\gamma + d' \\ &\geq d_\gamma + \max\{|\tilde{\boldsymbol{v}}(v_1)|, \ldots, |\tilde{\boldsymbol{v}}(v_n)|\} \\ &\geq \mathcal{S}_l(\gamma)\,(|\tilde{\boldsymbol{v}}(v_1)|, \ldots, |\tilde{\boldsymbol{v}}(v_n)|) && \text{by (7)} \\ &\geq |\boldsymbol{v}(w)|. \end{aligned}$$

Case 3: $\gamma \in \dot{\times} \setminus \dot{+}$

Now for all $u_1, \ldots, u_n \in \mathbb{N}$, we have

$$s_\gamma \cdot (e_\gamma + u_1 + \ldots + u_n) \geq \mathcal{S}_l(\gamma)\,(u_1, \ldots, u_n).$$

Since $\mathcal{S}_l(\gamma)$ only depends on the active variables in $\mathsf{actV}(\mathcal{S}_l(\gamma))$, let $\hat{u}_i = u_i$ if $v_i \in \mathsf{actV}(\mathcal{S}_l(\gamma))$ and $\hat{u}_i = 0$ otherwise. Then

$$\begin{aligned} s_\gamma \cdot (e_\gamma + \sum\nolimits_{v_i \in \mathsf{actV}(\mathcal{S}_l(\gamma))} u_i) \;&=\; s_\gamma \cdot (e_\gamma + \hat{u}_1 + \ldots + \hat{u}_n) \\ &\geq\; \mathcal{S}_l(\gamma)\,(\hat{u}_1, \ldots, \hat{u}_n) \\ &=\; \mathcal{S}_l(\gamma)\,(u_1, \ldots, u_n). \end{aligned} \tag{8}$$

Again, we have to estimate the sizes of the input variables $u_1, \ldots, u_n$, i.e., we have to find a bound on the size of each $v_i \in \mathcal{V}$ after the application of the transition $\tilde{r}$ that precedes r.

If $|\tilde{r}, v_i'| \in C$, then the induction hypothesis again implies that (5) also holds for the reduction from $(\ell_0, \boldsymbol{v}_0)$ to $(\tilde{\ell}, \tilde{\boldsymbol{v}})$. Let

$$d'' \;=\; \Big(\prod_{t\in\mathcal{T}\setminus\{r\}} s_t^{k_t}\Big) \cdot s_r^{k_r-1} \cdot \Big(e \;+\sum_{t\in\mathcal{T}}(k_t \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \dot{=}) \cap C_t\}) \\ +\sum_{t\in\mathcal{T}\setminus\{r\}}(k_t \cdot \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\}) \\ +(k_r - 1) \cdot \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_r\})\Big).$$

As in Case 2, the reason for using "$k_r - 1$" in d'' is that the last application of the transition r in the evaluation (4) is missing in the evaluation from $(\ell_0, \boldsymbol{v}_0)$ to $(\tilde{\ell}, \tilde{\boldsymbol{v}})$. Then the induction hypothesis implies $d'' \geq |\tilde{\boldsymbol{v}}(v_i)|$.

For $v_i \in \mathsf{actV}(\mathcal{S}_l(\gamma))$ with $v_i \notin \mathcal{V}_\gamma$, we have $|\tilde{r}, v_i'| \notin C$ (by definition of $\mathcal{V}_\gamma$). For such v_i, we can deduce the following:

$$f^\gamma_{v_i}(\boldsymbol{m}) \geq \mathcal{S}(\tilde{r}, v_i')\,(\boldsymbol{m}) \geq |\tilde{\boldsymbol{v}}(v_i)|. \tag{9}$$

We now combine these bounds to prove that d is indeed a bound for $|\boldsymbol{v}(w)|$:

$$\begin{aligned}
d \;=\; & \Big(\textstyle\prod_{t\in\mathcal{T}} s_t^{k_t}\Big) \cdot \Big(e \;+\textstyle\sum_{t\in\mathcal{T}}(k_t \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \dot{=}) \cap C_t\}) \\
& \qquad + \textstyle\sum_{t\in\mathcal{T}}(k_t \cdot \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\})\Big) \\
=\; & s_r \cdot \Big(\textstyle\prod_{t\in\mathcal{T}\setminus\{r\}} s_t^{k_t}\Big) \cdot s_r^{k_r-1} \cdot \Big(e \;+\textstyle\sum_{t\in\mathcal{T}}(k_t \cdot \max\{d_\alpha \mid \alpha \in (\dot{+} \setminus \dot{=}) \cap C_t\}) \\
& \qquad + \textstyle\sum_{t\in\mathcal{T}\setminus\{r\}}(k_t \cdot \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_t\}) \\
& \qquad + (k_r - 1) \cdot \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_r\}) \\
& \qquad + \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_r\})\Big) \\
=\; & s_r \cdot d'' \;+\; s_r \cdot \Big(\textstyle\prod_{t\in\mathcal{T}\setminus\{r\}} s_t^{k_t}\Big) \cdot s_r^{k_r-1} \cdot \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_r\} \\
\geq\; & s_r \cdot \Big(d'' \;+\; \max\{e_\alpha + f^\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_r\}\Big) \\
\geq\; & s_r \cdot \Big(d'' \;+\; e_\gamma \;+\; f^\gamma\Big) \\
=\; & \max\{s_\alpha \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_r\} \;\cdot\; \max\{|\mathcal{V}_\alpha| \mid \alpha \in (\dot{\times} \setminus \dot{+}) \cap C_r\} \;\cdot\; \Big(d'' \;+\; e_\gamma \;+\; f^\gamma\Big) \\
\geq\; & s_\gamma \;\cdot\; |\mathcal{V}_\gamma| \;\cdot\; \Big(d'' \;+\; e_\gamma \;+\; f^\gamma\Big) \\
\geq\; & s_\gamma \;\cdot\; \Big(|\mathcal{V}_\gamma| \cdot d'' \;+\; e_\gamma \;+\; f^\gamma\Big) \\
=\; & s_\gamma \;\cdot\; \Big(\textstyle\sum_{v_i\in\mathcal{V}_\gamma} d'' \;+\; e_\gamma \;+\; f^\gamma\Big) \\
\geq\; & s_\gamma \;\cdot\; \Big(\textstyle\sum_{v_i\in\mathcal{V}_\gamma} |\tilde{\boldsymbol{v}}(v_i)| \;+\; e_\gamma \;+\; f^\gamma\Big) \qquad \text{by the induction hypothesis} \\
=\; & s_\gamma \;\cdot\; \Big(\textstyle\sum_{v_i\in\mathcal{V}_\gamma} |\tilde{\boldsymbol{v}}(v_i)| \;+\; e_\gamma \;+\; \textstyle\sum_{v_i\in\mathsf{actV}(\mathcal{S}_l(\gamma))\setminus\mathcal{V}_\gamma} f^\gamma_{v_i}(\boldsymbol{m})\Big) \\
\geq\; & s_\gamma \;\cdot\; \Big(\textstyle\sum_{v_i\in\mathcal{V}_\gamma} |\tilde{\boldsymbol{v}}(v_i)| \;+\; e_\gamma \;+\; \textstyle\sum_{v_i\in\mathsf{actV}(\mathcal{S}_l(\gamma))\setminus\mathcal{V}_\gamma} |\tilde{\boldsymbol{v}}(v_i)|\Big) \qquad \text{by (9)} \\
=\; & s_\gamma \;\cdot\; \Big(e_\gamma \;+\; \textstyle\sum_{v_i\in\mathsf{actV}(\mathcal{S}_l(\gamma))} |\tilde{\boldsymbol{v}}(v_i)|\Big) \qquad \text{since } \mathcal{V}_\gamma \subseteq \mathsf{actV}(\mathcal{S}_l(\gamma)) \\
\geq\; & \mathcal{S}_l(\gamma)(|\tilde{\boldsymbol{v}}(v_1)|, \ldots, |\tilde{\boldsymbol{v}}(v_n)|) \qquad \text{by (8)} \\
\geq\; & |\boldsymbol{v}(w)|
\end{aligned}$$

□

Theorem 5.5 (Complexities from SRFs). *Let $\mathcal{R}$ be a runtime approximation, $\mathcal{P}ol$ be an SRF for $\mathcal{T}$, and $2 \leq b = \max\{|\mathcal{P}| \mid (\ell, \tau, \mathcal{P}) \in \mathcal{T}_\succ\}$.*

Let $\mathcal{R}'(t) = \frac{b^{[\mathcal{P}ol(\ell_0)]}-1}{b-1}$ for all $t \in \mathcal{T}_\succ$ and $\mathcal{R}'(t) = \mathcal{R}(t)$ for all other $t \in \mathcal{T}$. Then, $\mathcal{R}'$ is also a runtime approximation.

Proof. To prove

$$(\mathcal{R}'(t))(\boldsymbol{m}) \;\geq\; \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F\,.\,\boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_0, \boldsymbol{v}_0)\}\,(\to^* \circ \to_t)^k\,F\}$$

for all $t \in \mathcal{T}$ and $\boldsymbol{m} \in \mathbb{N}^n$, it obviously suffices to show that

$$\frac{b^{[\mathcal{P}ol(\ell_0)](\boldsymbol{m})}-1}{b-1} \;\geq\; \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F\,.\,\boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_0, \boldsymbol{v}_0)\}\,(\to^* \circ \to_t)^k\,F\}$$

holds for all $t \in \mathcal{T}_\succ$ and $\boldsymbol{m} \in \mathbb{N}^n$. To this end, let $\boldsymbol{v}_0$ be a valuation with $\boldsymbol{v}_0 \leq \boldsymbol{m}$ and consider a (finite or infinite) evaluation of the form

$$\{(\ell_0, \boldsymbol{v}_0)\} \to_{t_0} \{(\ell_{1,1}, \boldsymbol{v}_{1,1}), \ldots, (\ell_{1,p_1}, \boldsymbol{v}_{1,p_1})\} \to_{t_1} \{(\ell_{2,1}, \boldsymbol{v}_{2,1}), \ldots, (\ell_{2,p_2}, \boldsymbol{v}_{2,p_2})\} \to_{t_2} \ldots$$

where $k \in \mathbb{N} \cup \{\omega\}$ steps are performed with the transition t.

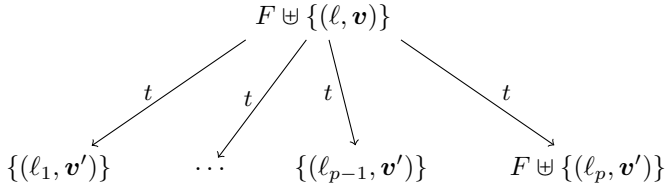
Our goal is to show that $\frac{b^{[\mathcal{P}ol(\ell_0)](\boldsymbol{m})}-1}{b-1} \geq k$ holds. This is trivial for the case $k = 0$, as $\frac{b^{[\mathcal{P}ol(\ell_0)](\boldsymbol{m})}-1}{b-1} \geq 0$. Thus, we now consider the case where $k > 0$.

For a composed configuration $\{(\ell_1, \boldsymbol{v}_1), \ldots, (\ell_p, \boldsymbol{v}_p)\}$, we have

$$\sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F \,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_1, \boldsymbol{v}_1), \ldots, (\ell_p, \boldsymbol{v}_p)\} \, (\rightarrow^* \circ \rightarrow_t)^k \, F\} =$$
$$\sum\nolimits_{j \in \{1,\ldots,p\}} \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F \,.\, \boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_j, \boldsymbol{v}_j)\} \, (\rightarrow^* \circ \rightarrow_t)^k \, F\}.$$

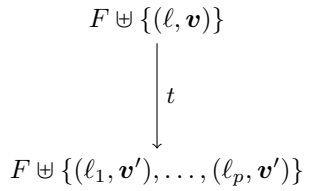
Based on this observation, for a given SRF we represent a (finite or infinite) evaluation $\{(\ell_0, \boldsymbol{v}_0)\} \rightarrow F_1 \rightarrow F_2 \rightarrow \ldots$ as a tree structure. This tree is similar to the evaluation tree, but it does not branch for steps with transitions from $\mathcal{T} \setminus \mathcal{T}_{\succ}$. To this end, an evaluation step $F_1 = F \uplus \{(\ell, \boldsymbol{v})\} \rightarrow_t F \uplus \{(\ell_1, \boldsymbol{v'}), \ldots, (\ell_p, \boldsymbol{v'})\} = F_2$ is represented as follows. Let $t = (\ell, \tau, \{\ell_1, \ldots, \ell_p\})$.

— If $t \in \mathcal{T}_{\succ}$, we write the following:



So here a transition is "split up", corresponding to the constraints imposed on the SRF. In the last child $F \uplus \{(\ell_p, \boldsymbol{v'})\}$, we also keep the pairs F that were not altered by the transition.

— If $t \in \mathcal{T} \setminus \mathcal{T}_{\succ}$, we write:



So here a transition is just taken as such and not split up.

Now we consider each (maximal) *path* from the root of this tree individually. Such a path has the shape

$$\{(\ell_0, \boldsymbol{v}_0)\} \rightarrow_{t_0} \{(\ell_{1,1}, \boldsymbol{v}_{1,1}), \ldots, (\ell_{1,p_1}, \boldsymbol{v}_{1,p_1})\} \rightarrow_{t_1} \{(\ell_{2,1}, \boldsymbol{v}_{2,1}), \ldots, (\ell_{2,p_2}, \boldsymbol{v}_{2,p_2})\} \rightarrow_{t_2} \ldots$$

where $h \in \mathbb{N} \cup \{\omega\}$ steps are performed with the transition t. In the following, we also write $\ell_{0,1}$ for ℓ_0, $p_0 = 1$, and $\boldsymbol{v}_{0,1} = \boldsymbol{v}_0$.

Since $\mathcal{P}ol$ is a SRF for $\mathcal{T}$, we have

$$\begin{array}{c} \sum_{j \in \{1,\ldots,p_i\}} \max\{0, (\mathcal{P}ol(\ell_{i,j}))(\boldsymbol{v}_{i,j}(v_1), \ldots, \boldsymbol{v}_{i,j}(v_n))\} \\ \sum_{j \in \{1,\ldots,p_{i+1}\}} \max\{0, (\mathcal{P}ol(\ell_{i+1,j}))(\boldsymbol{v}_{i+1,j}(v_1), \ldots, \boldsymbol{v}_{i+1,j}(v_n))\} \end{array} \quad \geq$$

for all i.

Let $i_1 < i_2 < \ldots$ be the h indices where $t_i = t$. Then for all $i \in \{i_1, i_2, \ldots\}$, $t \in \mathcal{T}_{\succ}$ implies

$$\begin{array}{c} \sum_{j \in \{1,\ldots,p_i\}} \max\{0, (\mathcal{P}ol(\ell_{i,j}))(\boldsymbol{v}_{i,j}(v_1), \ldots, \boldsymbol{v}_{i,j}(v_n))\} \\ \sum_{j \in \{1,\ldots,p_{i+1}\}} \max\{0, (\mathcal{P}ol(\ell_{i+1,j}))(\boldsymbol{v}_{i+1,j}(v_1), \ldots, \boldsymbol{v}_{i+1,j}(v_n))\} \end{array} \quad >$$

and

$$\sum\nolimits_{j\in\{1,\dots,p_i\}} \max\{0, (\mathcal{P}ol(\ell_{i,j}))(\boldsymbol{v}_{i,j}(v_1),\dots,\boldsymbol{v}_{i,j}(v_n))\} \geq 1.$$

Recall that we are looking at the paths of the specially constructed tree here, not at the original evaluation. There, this reasoning would not work.

Thus, we obtain

$$\begin{array}{ll}
\max\{0, (\mathcal{P}ol(\ell_0))(\boldsymbol{v}_0(v_1),\dots,\boldsymbol{v}_0(v_n))\} & \geq \\
\sum_{j\in\{1,\dots,p_{i_1}\}} \max\{0, (\mathcal{P}ol(\ell_{i_1,j}))(\boldsymbol{v}_{i_1,j}(v_1),\dots,\boldsymbol{v}_{i_1,j}(v_n))\} & > \\
\sum_{j\in\{1,\dots,p_{i_2}\}} \max\{0, (\mathcal{P}ol(\ell_{i_2,j}))(\boldsymbol{v}_{i_2,j}(v_1),\dots,\boldsymbol{v}_{i_2,j}(v_n))\} & > \\
 & \vdots \\
\sum_{j\in\{1,\dots,p_{i_{h-1}}\}} \max\{0, (\mathcal{P}ol(\ell_{i_{h-1},j}))(\boldsymbol{v}_{i_{h-1},j}(v_1),\dots,\boldsymbol{v}_{i_{h-1},j}(v_n))\} & > \\
\sum_{j\in\{1,\dots,p_{i_h}\}} \max\{0, (\mathcal{P}ol(\ell_{i_h,j}))(\boldsymbol{v}_{i_h,j}(v_1),\dots,\boldsymbol{v}_{i_h,j}(v_n))\} & \geq 1.
\end{array}$$

From $\max\{0, (\mathcal{P}ol(\ell_0))(\boldsymbol{v}_0(v_1),\dots,\boldsymbol{v}_0(v_n))\} \geq 1$ we obtain in particular that $0 \neq \max\{0, (\mathcal{P}ol(\ell_0))(\boldsymbol{v}_0(v_1),\dots,\boldsymbol{v}_0(v_n))\} = (\mathcal{P}ol(\ell_0))(\boldsymbol{v}_0(v_1),\dots,\boldsymbol{v}_0(v_n))$. So we have $h \neq \omega$ and $\mathcal{P}ol(\ell_0)\,(\boldsymbol{v}_0(v_1),\dots,\boldsymbol{v}_0(v_n)) \geq h$, i.e., $\mathcal{P}ol(\ell_0)\,(\boldsymbol{v}_0(v_1),\dots,\boldsymbol{v}_0(v_n))$ is an upper bound on the number of evaluation steps with the transition t. This implies

$$\begin{array}{ll}
[\mathcal{P}ol(\ell_0)]\,(\boldsymbol{m}) & \geq \\
[\mathcal{P}ol(\ell_0)]\,(|\boldsymbol{v}_0(v_1)|,\dots,|\boldsymbol{v}_0(v_n)|) & \geq \\
\mathcal{P}ol(\ell_0)\,(\boldsymbol{v}_0(v_1),\dots,\boldsymbol{v}_0(v_n)) & \geq h.
\end{array}$$

By construction of the tree, the tree can only branch with degree ≥ 2 if a transition $t \in \mathcal{T}_\succ$ is used. For all those (finite or infinite) paths $F_1 \to F_2 \to \dots$ of the tree where all edges are labeled by transitions from $\mathcal{T} \setminus \mathcal{T}_\succ$, we collapse the path to the single node F_1. Then we get a finite tree whose inner nodes are the nodes that have an outgoing edge labeled by a transition from $\mathcal{T}_\succ$. We want to count these inner nodes since their number corresponds to the number of steps with transitions from $\mathcal{T}_\succ$ in the original evaluation.

Let b be the maximum arity of $\mathcal{P}$ for the transitions $(\ell, \tau, \mathcal{P}) \in \mathcal{T}_\succ$. Note that b is an upper bound on the branching factor in the collapsed tree. As the height of the collapsed tree is bounded by h, it has at most $\frac{b^h-1}{b-1}$ inner nodes.

As $[\mathcal{P}ol(\ell_0)](\boldsymbol{m}) \geq h$, this implies the desired statement that for all $t \in \mathcal{T}_\succ$, we have

$$\frac{b^{[\mathcal{P}ol(\ell_0)](\boldsymbol{m})} - 1}{b-1} \;\geq\; \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, F\,.\,\boldsymbol{v}_0 \leq \boldsymbol{m} \wedge \{(\ell_0, \boldsymbol{v}_0)\}\,(\to^* \circ \to_t)^k\,F\}.$$

□

Theorem 6.5 (Soundness of $\mathcal{C}$). *Let $(\mathcal{T}, \mathcal{M})$ be an annotated program. Then $(\sum_{t\in\mathcal{T}} \mathcal{C}_{(\mathcal{T},\mathcal{M})}(t)) \geq \mathrm{cc}_{(\mathcal{T},\mathcal{M})}$.*

Proof. Let $\boldsymbol{m}$ be arbitrary, but fixed. We consider an evaluation

$$\{(\ell_0, \boldsymbol{v}_0)\} \to_{t_0}^{(\ell_0,\boldsymbol{v}_0)} F_1 \to_{t_1}^{(\ell_1,\boldsymbol{v}_1)} F_2 \to_{t_2}^{(\ell_2,\boldsymbol{v}_2)} \dots \tag{10}$$

for some $\boldsymbol{v}_0 \leq \boldsymbol{m}$. To prove the theorem, it suffices to show that

$$\sum\nolimits_{t\in\mathcal{T}} (\mathcal{C}_{(\mathcal{T},\mathcal{M})}(t))(\boldsymbol{m}) \;\geq\; \sum\nolimits_{i\geq 0} (\mathcal{M}(t_i))(\boldsymbol{v}_i(v_1),\dots,\boldsymbol{v}_i(v_n)).$$

Let $\mathcal{T}_0$ consist of all initial transitions from $\mathcal{T}$ and for any transition $t \in \mathcal{T}$, let $k_t \in \mathbb{N} \cup \{\omega\}$ be the number of $\to_t$-steps in our evaluation. For any $i > 0$, let $\tilde{i} < i$ be the step where the sub-configuration $(\ell_i, \boldsymbol{v}_i)$ was introduced, i.e., the edge reaching $(\ell_i, \boldsymbol{v}_i)$ is labeled by $t_{\tilde{i}}$ in the evaluation tree corresponding to (10). In other words, for every $i > 0$, there is an

$\tilde{i} < i$ such that $t_{\tilde{i}} \in \mathsf{pre}(t_i)$ and $\boldsymbol{v}_{\tilde{i}}, \boldsymbol{v}_i$ satisfies the formula of $t_{\tilde{i}}$. Thus, $\{(\ell_0, \boldsymbol{v}_0)\} \to^* F_{\tilde{i}}$ and $(\ell_{\tilde{i}}, \boldsymbol{v}_{\tilde{i}}) \in F_{\tilde{i}}$ implies $(\mathcal{S}(t_{\tilde{i}}, v'))(\boldsymbol{m}) \geq \boldsymbol{v}_i(v)$ for all $v \in \mathcal{V}$. Thus, we obtain

$$\begin{array}{ll} & \sum_{t\in\mathcal{T}}(\mathcal{C}_{(\mathcal{T},\mathcal{M})}(t))(\boldsymbol{m}) \\ = & \sum_{t\in\mathcal{T}_0}(\mathcal{R}(t))(\boldsymbol{m}) \cdot (\mathcal{M}(t))(\boldsymbol{m}) + \\ & \sum_{t\in\mathcal{T}\setminus\mathcal{T}_0}(\mathcal{R}(t))(\boldsymbol{m}) \cdot \max\{(\mathcal{M}(t))((\mathcal{S}(\tilde{t}, v'_1))(\boldsymbol{m}), \ldots, (\mathcal{S}(\tilde{t}, v'_n))(\boldsymbol{m})) \mid \tilde{t} \in \mathsf{pre}(t)\} \\ \geq & (\mathcal{M}(t_0))(\boldsymbol{m}) + \sum_{t\in\mathcal{T}\setminus\mathcal{T}_0} k_t \cdot \max\{(\mathcal{M}(t))((\mathcal{S}(\tilde{t}, v'_1))(\boldsymbol{m}), \ldots, (\mathcal{S}(\tilde{t}, v'_n))(\boldsymbol{m})) \mid \tilde{t} \in \mathsf{pre}(t)\} \\ \geq & (\mathcal{M}(t_0))(\boldsymbol{m}) + \sum_{i>0} \max\{(\mathcal{M}(t_i))((\mathcal{S}(\tilde{t}, v'_1))(\boldsymbol{m}), \ldots, (\mathcal{S}(\tilde{t}, v'_n))(\boldsymbol{m})) \mid \tilde{t} \in \mathsf{pre}(t_i)\} \\ \geq & (\mathcal{M}(t_0))(\boldsymbol{m}) + \sum_{i>0}(\mathcal{M}(t_i))((\mathcal{S}(t_{\tilde{i}}, v'_1))(\boldsymbol{m}), \ldots, (\mathcal{S}(t_{\tilde{i}}, v'_n))(\boldsymbol{m})) \\ \geq & \sum_{i\geq 0}(\mathcal{M}(t_i))(\boldsymbol{v}_i(v_1), \ldots, \boldsymbol{v}_i(v_n)). \end{array}$$

□

Theorem 6.13 (Soundness of Separated Modular Complexity Analysis). *Let $(\mathcal{T}, \mathcal{M})$ be an annotated program, let $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{L}' \subseteq \mathcal{L}$ satisfy the requirements in Def. 6.9, and let $(\mathcal{T}_{\setminus\mathcal{L}'}, \mathcal{M}_{\setminus\mathcal{L}'}, \mathcal{R}_{\setminus\mathcal{L}'}, \mathcal{S}_{\setminus\mathcal{L}'})$ be the $\mathcal{L}'$-reduced program. Then $(\mathcal{R}_{\setminus\mathcal{L}'}, \mathcal{S}_{\setminus\mathcal{L}'})$ is a complexity approximation for $\mathcal{T}_{\setminus\mathcal{L}'}$ and $\mathrm{cc}_{(\mathcal{T}_{\setminus\mathcal{L}'}, \mathcal{M}_{\setminus\mathcal{L}'})} \geq \mathrm{cc}_{(\mathcal{T},\mathcal{M})}$.*

Proof. We only prove $\mathrm{cc}_{(\mathcal{T}_{\setminus\mathcal{L}'}, \mathcal{M}_{\setminus\mathcal{L}'})} \geq \mathrm{cc}_{(\mathcal{T},\mathcal{M})}$ (the claim that $(\mathcal{R}_{\setminus\mathcal{L}'}, \mathcal{S}_{\setminus\mathcal{L}'})$ is a complexity approximation for $\mathcal{T}_{\setminus\mathcal{L}'}$ is straightforward). Consider an evaluation in the program $\mathcal{T}$. For the proof, we transform this evaluation to a corresponding evaluation in the $\mathcal{L}'$-reduced program $\mathcal{T}_{\setminus\mathcal{L}'}$. Here, we replace parts of the evaluation that make use of $\mathcal{T}_{\mathcal{L}'}$ by corresponding evaluation steps with $\mathcal{T}_{\setminus\mathcal{L}'}$. We then show that the cost of the replaced evaluation steps with $\mathcal{T}_{\mathcal{L}'}$ is bounded by the cost of the newly introduced evaluation steps with $\mathcal{T}_{\setminus\mathcal{L}'}$. Of course, this observation is trivial if the original evaluation in $\mathcal{T}$ never reaches locations from $\mathcal{L}'$, because then all transitions used in the original evaluation are also contained in $\mathcal{T}_{\setminus\mathcal{L}'}$.

Otherwise, we regard the corresponding evaluation tree and consider a maximal subtree whose edges are labeled with transitions from $\mathcal{T}_{\mathcal{L}'}$. Let $(\ell_1, \boldsymbol{v}_1)$ be the root of this subtree where $\ell_1 \in \mathcal{L}'$. As the start location is not contained in $\mathcal{L}'$, $(\ell_1, \boldsymbol{v}_1)$ has a parent node $(\ell, \boldsymbol{v})$ with $\ell \notin \mathcal{L}'$, and the edge from $(\ell, \boldsymbol{v})$ to $(\ell_1, \boldsymbol{v}_1)$ is labeled by a transition $(\ell, \tau, \mathcal{P}) \in \mathcal{T}_{\to\mathcal{L}'}$. To transform the current evaluation tree w.r.t. $\mathcal{T}$ to an evaluation tree w.r.t. $\mathcal{T}_{\setminus\mathcal{L}'}$, we replace the label $(\ell, \tau, \mathcal{P})$ on the edges from $(\ell, \boldsymbol{v})$ to its children by the transition $(\ell, \tau, (\mathcal{P} \setminus \mathcal{L}') \cup \{\ell_{\to\mathcal{L}'}\})$. Note that the cost for this transition is at least as high as the cost of the original transition $(\ell, \tau, \mathcal{P})$. Moreover, whenever $(\ell, \boldsymbol{v})$ has a child node $(\ell', \boldsymbol{v}')$ with $\ell' \in \mathcal{L}'$, then we replace that child node by $(\ell_{\to\mathcal{L}'}, \boldsymbol{v}')$. The child nodes $(\ell', \boldsymbol{v}')$ with $\ell' \notin \mathcal{L}'$ are not modified. So in particular, $(\ell_1, \boldsymbol{v}_1)$ is replaced by $(\ell_{\to\mathcal{L}'}, \boldsymbol{v}_1)$.

Case 1: There is a path from $(\ell_1, \boldsymbol{v}_1)$ to a node $(\ell_{k+1}, \boldsymbol{v}_{k+1})$ with $\ell_{k+1} \notin \mathcal{L}'$.

Let the path have the form $(\ell_1, \boldsymbol{v}_1), (\ell_2, \boldsymbol{v}_2), \ldots, (\ell_k, \boldsymbol{v}_k), (\ell_{k+1}, \boldsymbol{v}_{k+1})$, where ℓ_{k+1} is the first location from $\mathcal{L} \setminus \mathcal{L}'$ on the path (i.e., $\ell_1, \ldots, \ell_k \in \mathcal{L}'$). Since we have $\ell_i \to^*_{reach} \ell_{k+1}$ for all $1 \leq i \leq k$, the transitions t_i used on this path all have the form $(\ell_i, \tau_i, \ell_{i+1})$ for $1 \leq i \leq k$ (i.e., these transitions only have singleton multisets of locations in their third components). Thus, we have the evaluation steps

$$(\ell_1, \boldsymbol{v}_1) \to_{t_1} (\ell_2, \boldsymbol{v}_2) \to_{t_2} \ldots \to_{t_{k-1}} (\ell_k, \boldsymbol{v}_k) \to_{t_k} (\ell_{k+1}, \boldsymbol{v}_{k+1})$$

where $t_1, \ldots, t_{k-1} \in \mathcal{T}_{\mathcal{L}'}$ and $t_k \in \mathcal{T}_{\mathcal{L}'\to}$.

So in this case, the subtree that has to be replaced only consists of a "list" of evaluations with $\mathcal{T}_{\mathcal{L}'}$. Recall that we already replaced $(\ell_1, \boldsymbol{v}_1)$ by $(\ell_{\to\mathcal{L}'}, \boldsymbol{v}_1)$. Similarly, we now replace $(\ell_k, \boldsymbol{v}_k)$ by $(\ell_{\mathcal{L}'\to}, \boldsymbol{v}_k)$ and on its edge to $(\ell_{k+1}, \boldsymbol{v}_{k+1})$, we replace the current label $t_k = (\ell_k, \tau_k, \ell_{k+1})$ by $t'_k = (\ell_{\mathcal{L}'\to}, \tau_k, \ell_{k+1})$. Again, the cost for the transition t'_k is at least as high as the cost of the original transition t_k.

It remains to replace the path from $(\ell_1, \boldsymbol{v}_1)$ to $(\ell_k, \boldsymbol{v}_k)$, where $(\ell_1, \boldsymbol{v}_1)$ has already been modified to $(\ell_{\to\mathcal{L}'}, \boldsymbol{v}_1)$ and $(\ell_k, \boldsymbol{v}_k)$ has been modified to $(\ell_{\mathcal{L}'\to}, \boldsymbol{v}_k)$.

If $k = 1$, then no transition of $\mathcal{T}_{\mathcal{L}'}$ was used in the path above, and thus, the original edge from $(\ell_1, \boldsymbol{v}_1) = (\ell_k, \boldsymbol{v}_k)$ to $(\ell_{k+1}, \boldsymbol{v}_{k+1})$ labeled by t_k is replaced by a path with two edges. The first edge is from $(\ell_{\to\mathcal{L}'}, \boldsymbol{v}_1)$ to $(\ell_{\mathcal{L}'\to}, \boldsymbol{v}_k)$ and it is labeled with t_{skip}. The second edge from $(\ell_{\mathcal{L}'\to}, \boldsymbol{v}_k)$ to $(\ell_{k+1}, \boldsymbol{v}_{k+1})$ is labeled with t'_k.

If $k > 1$, then the path from $(\ell_1, \boldsymbol{v}_1)$ to $(\ell_k, \boldsymbol{v}_k)$ and further to $(\ell_{k+1}, \boldsymbol{v}_{k+1})$ labeled with $t_1, \ldots, t_{k-1}, t_k$ is also replaced by a path with two edges. The first edge is again from $(\ell_{\to\mathcal{L}'}, \boldsymbol{v}_1)$ to $(\ell_{\mathcal{L}'\to}, \boldsymbol{v}_k)$, but now it is labeled with $t_{\mathcal{T}_{\mathcal{L}'}}$. The second edge from $(\ell_{\mathcal{L}'\to}, \boldsymbol{v}_k)$ to $(\ell_{k+1}, \boldsymbol{v}_{k+1})$ is again labeled with t'_k. It remains to show that the evaluation step using $t_{\mathcal{T}_{\mathcal{L}'}}$ can indeed lead to the valuation $\boldsymbol{v}_k$ (**S**), and that the cost of $t_{\mathcal{T}_{\mathcal{L}'}}$ is at least as high as the sum of the costs of the original transitions $t_1, \ldots, t_{k-1}$ (**C**).

To prove the latter observation (**C**), let $|\boldsymbol{v}_1| = (|\boldsymbol{v}_1(v_1)|, \ldots, |\boldsymbol{v}_1(v_n)|)$. Then we have

$$
\begin{aligned}
&(\mathcal{M}_{\setminus \mathcal{L}'}(t_{\mathcal{T}_{\mathcal{L}'}}))(|\boldsymbol{v}_1|)\\
&= \sum\nolimits_{t \in \mathcal{T}_{\mathcal{L}'}} (\mathcal{C}_{(\mathcal{T}_{|\,\mathcal{L}'}, \mathcal{M}_{|\,\mathcal{L}'})}(t))(|\boldsymbol{v}_1|)\\
&\overset{(\dagger)}{=} \sum\nolimits_{t \in \mathcal{T}_{\mathcal{L}'}} (\mathcal{R}_{\mathcal{T}_{|\,\mathcal{L}'}}(t))(|\boldsymbol{v}_1|) \cdot \max\{(\mathcal{M}_{\mathcal{T}_{|\,\mathcal{L}'}}(t))((\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(\tilde{t}, v'_1))(|\boldsymbol{v}_1|), \ldots,\\
&\qquad\qquad (\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(\tilde{t}, v'_n))(|\boldsymbol{v}_1|)) \mid \tilde{t} \in \mathsf{pre}_{\mathcal{T}_{|\,\mathcal{L}'}}(t)\}\\
&\overset{(\ddagger)}{\geq} \sum\nolimits_{1 \leq i < k} \max\{(\mathcal{M}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_i))((\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(\tilde{t}, v'_1))(|\boldsymbol{v}_1|), \ldots, (\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(\tilde{t}, v'_n))(|\boldsymbol{v}_1|)) \mid\\
&\qquad\qquad \tilde{t} \in \mathsf{pre}_{\mathrm{sep}\,\mathcal{T}_{|\,\mathcal{L}'}}(t_i)\}\\
&\overset{(\ddagger\ddagger)}{\geq} \sum\nolimits_{1 \leq i < k} (\mathcal{M}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_i))((\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_{i-1}, v'_1))(|\boldsymbol{v}_1|), \ldots, (\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_{i-1}, v'_n))(|\boldsymbol{v}_1|))\\
&\overset{(\ddagger\ddagger\ddagger)}{\geq} \sum\nolimits_{1 \leq i < k} (\mathcal{M}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_i))(|\boldsymbol{v}_i(v_1)|, \ldots, |\boldsymbol{v}_i(v_n)|).
\end{aligned}
$$

(†): By construction, as $\mathcal{T}_{\mathcal{L}'}$ contains no initial transitions.
(‡): By soundness of $\mathcal{R}_{\mathcal{T}_{|\,\mathcal{L}'}}$, as otherwise, there would be some $t \in \mathcal{T}_{\mathcal{L}'}$ that is used more often in an evaluation with $\mathcal{T}_{|\,\mathcal{L}'}$ than $\mathcal{R}_{\mathcal{T}_{|\,\mathcal{L}'}}$ allows.
(‡‡): Each path of the evaluation tree corresponds to an evaluation with a variant of $\mathcal{T}$ where each transition only contains a single target location. Hence, $t_{i-1} \in \mathsf{pre}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_i)$ holds for all $2 \leq i < k$. Moreover, the isolated sub-program $\mathcal{T}_{|\,\mathcal{L}'}$ contains an extra transition from ℓ_0 to the set of locations $\mathcal{P}$, where $\ell_1 \in \mathcal{P}$. Thus, there is a transition $t_0 = (\ell_0, \bigwedge_{v \in \mathcal{V}} v' = v, \ell_1) \in \mathsf{pre}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_1)$ and an evaluation $(\ell_0, \boldsymbol{v}_1) \to_{t_0} (\ell_1, \boldsymbol{v}_1)$.
(‡ ‡ ‡): By soundness of $\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}$, as for all result variables $|t_{i-1}, v'_j|$ with $1 \leq i < k$ and $1 \leq j \leq n$ we have $\mathcal{S}_{\mathcal{T}_{|\,\mathcal{L}'}}(t_{i-1}, v'_j) \geq \boldsymbol{v}_i(v_j)$.

The proposition (**S**) can be proven analogously to the step (‡ ‡ ‡).

Case 2: There is no path from $(\ell_1, \boldsymbol{v}_1)$ to a node $(\ell_{k+1}, \boldsymbol{v}_{k+1})$ with $\ell_{k+1} \notin \mathcal{L}'$.

This means that all edges in the subtree rooted by $(\ell_1, \boldsymbol{v}_1)$ are labeled by transitions from $\mathcal{T}_{\mathcal{L}'}$. Recall that we already replaced $(\ell_1, \boldsymbol{v}_1)$ by $(\ell_{\to\mathcal{L}'}, \boldsymbol{v}_1)$. To finish the transformation of this subtree, the whole subtree rooted by $(\ell_1, \boldsymbol{v}_1)$ is replaced by $(\ell_{\to\mathcal{L}'}, \boldsymbol{v}_1)$ and an edge to an additional node $(\ell_{\mathcal{L}'\to}, \boldsymbol{v}_2)$, where the edge is labeled with $t_{\mathcal{T}_{\mathcal{L}'}}$. Here, we choose a valuation $\boldsymbol{v}_2$ such that $\boldsymbol{v}_1, \boldsymbol{v}'_2$ satisfy the formula $\tau_{\mathcal{T}_{\mathcal{L}'}}$. It remains to show that the cost of $t_{\mathcal{T}_{\mathcal{L}'}}$ is at least as high as the cost of the evaluation of $(\ell_1, \boldsymbol{v}_1)$ represented by the original subtree. Note that this part of the evaluation corresponds to an evaluation w.r.t. the isolated sub-program $\mathcal{T}_{|\,\mathcal{L}'}$. As in the proof of Thm. 6.5, one can show that

$(\mathcal{M}_{\setminus \mathcal{L}'}(t_{\mathcal{T}_{\mathcal{L}'}}))(|\boldsymbol{v}_1|) = \sum_{t \in \mathcal{T}_{\mathcal{L}'}} (\mathcal{C}_{(\mathcal{T}_{|\mathcal{L}'}, \mathcal{M}_{|\mathcal{L}'})}(t))(|\boldsymbol{v}_1|)$ is an upper bound for the costs of this evaluation w.r.t. $(\mathcal{T}_{|\mathcal{L}'}, \mathcal{M}_{|\mathcal{L}'})$. $\square$