Alternating Runtime and Size Complexity Analysis of Integer Programs*

M. Brockschmidt[1], F. Emmes[2], S. Falke[3], C. Fuhs[4], and J. Giesl[2]

[1] Microsoft Research, Cambridge
[2] RWTH Aachen University
[3] Karlsruhe Institute of Technology
[4] University College London

Abstract. We present a modular approach to automatic complexity analysis. Based on a novel alternation between finding symbolic time bounds for program parts and using these to infer size bounds on program variables, we can restrict each analysis step to a small part of the program while maintaining a high level of precision. Extensive experiments with the implementation of our method demonstrate its performance and power in comparison with other tools.

1 Introduction

There exist numerous methods to prove termination of imperative programs, e.g., [2,6,8,9,12,13,15–17,19,25,33–35]. In many cases, however, termination is not sufficient, but the program should terminate in reasonable (e.g., (pseudo-) polynomial) time. To prove this, it is often crucial to derive (possibly non-linear) bounds on the values of variables that are modified repeatedly in loops.

We build upon the well-known observation that rank functions for termination proofs also provide a runtime complexity bound [3,4,6,7,32]. However, this only holds for proofs using a *single* rank function. Larger programs are usually handled by a disjunctive [16,28,35] or lexicographic [6,12,13,17,19,21,23,25] combination of rank functions. Here, deriving a complexity bound is much harder.

To illustrate this, consider the program on the right and a variant where the instruction "$\mathtt{x} = \mathtt{x} + \mathtt{i}$" is removed. For both variants, the lexicographic rank function $\langle f_1, f_2 \rangle$ proves termination, where f_1 measures states by the value of $\mathtt{i}$ and f_2 is just the value of $\mathtt{x}$. 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.

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

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 by their absolute values. Our method derives runtime bounds for isolated parts of the program

* Supported by the DFG grant GI 274/6-1.

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 continues until all loops and variables have been handled.

For the example, our method proves that the first loop is executed linearly often using the rank function $\mathtt{i}$. Then, it deduces that $\mathtt{i}$ is bounded by the size of its initial value $|\mathtt{i}_0|$ in all loop iterations. Combining these bounds, it infers that $\mathtt{x}$ is incremented by a value bounded by $|\mathtt{i}_0|$ at most $|\mathtt{i}_0|$ times, 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 conclude[5] that the program's runtime is bounded by $|\mathtt{i}_0| + |\mathtt{i}_0|^2 + |\mathtt{x}_0|$. 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. Thus, our approach succeeds on many programs that are beyond the reach of previous techniques.

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. Sect. 5 discusses related work and provides an extensive experimental evaluation. Proofs for all theorems as well as several extensions of our approach can be found in [14].

2 Preliminaries

```
Input: List x
ℓ0: List y = null
ℓ1: while x ≠ null do
      y = new List(x.val, y)
      x = x.next
    done
    List z = y
ℓ2: while z ≠ null do
      List u = z.next
ℓ3:   while u ≠ null do
        z.val += u.val
        u = u.next
      done
      z = z.next
    done
```

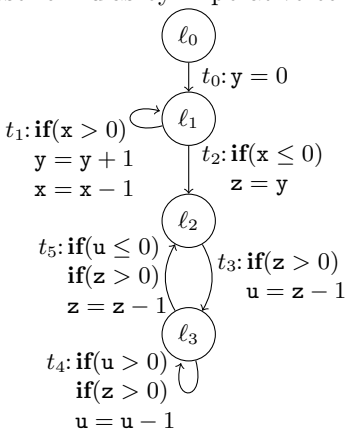
Consider the program on the right. For an input list $\mathtt{x}$, the loop at location ℓ_1 creates a list $\mathtt{y}$ by reversing the elements of $\mathtt{x}$. The loop at location ℓ_2 iterates over the list $\mathtt{y}$ and increases each element by the sum of its successors. So if $\mathtt{y}$ was $[5, 1, 3]$, it will be $[5+1+3, 1+3, 3]$ after the second loop. This example is a representative for methods using several algorithms in sequence.

We regard sequential imperative integer programs with (potentially non-linear) arithmetic and unbounded non-determinism. Our approach is compatible with methods that abstract features like heap usage to integers [2, 4, 15, 19, 29, 34]. So the above program could be abstracted automatically to the integer program below. Here, list variables are replaced by integer variables that correspond to the lengths of the lists.

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 and that all transitions $\mathcal{T}$ are reachable from ℓ_0. All transitions originating in ℓ_0 are called *initial transitions*. The

[5] 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.

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 following graph, we represented these formulas by imperative commands. For instance, t_3 is labeled by the formula $\mathtt{z} > 0 \wedge \mathtt{u}' = \mathtt{z} - 1 \wedge \mathtt{x}' = \mathtt{x} \wedge \mathtt{y}' = \mathtt{y} \wedge \mathtt{z}' = \mathtt{z}$. We used standard invariant-generation techniques (based on the Octagon domain [30]) to propagate simple integer invariants, adding the condition $\mathtt{z} > 0$ to the transitions t_4 and t_5.



Definition 1 (Programs). *A* transition *is a tuple (ℓ, τ, ℓ') where $\ell, \ell' \in \mathcal{L}$ are locations and τ is a 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. We drop the index t if we do not care about the used transition and write $(\ell, \boldsymbol{v}) \to^k (\ell', \boldsymbol{v'})$ if k evaluation steps lead from configuration $(\ell, \boldsymbol{v})$ to $(\ell', \boldsymbol{v'})$.*

So for the program above, we have $(\ell_1, \boldsymbol{v}_1) \to_{t_2} (\ell_2, \boldsymbol{v}_2)$ for any valuations 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 $\boldsymbol{v}_1(\mathtt{u}) = \boldsymbol{v}_2(\mathtt{u})$.

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 method 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 $\boldsymbol{m} = (m_1, ..., m_n) \in \mathbb{N}^n$, 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 *runtime complexity* by a function rc that maps the sizes $\boldsymbol{m}$ of the program variables to the maximal number of evaluation steps that are possible from a start configuration $(\ell_0, \boldsymbol{v})$ with $\boldsymbol{v} \leq \boldsymbol{m}$. 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. In Def. 2, $\to^* \circ \to_t$ is the relation that allows to perform arbitrary many evaluation steps followed by a step with transition t.

As we generate new bounds by composing previously found bounds, we only use weakly monotonic functions $\mathcal{R}(t)$ (i.e., $m_i \geq m_i'$ 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* $\mathfrak{C}$ as the weakly monotonic functions from $\mathbb{N}^n \to \mathbb{N}$ and $\mathbf{?}$, where $\mathbf{?}(\boldsymbol{m}) = \omega$ for all $\boldsymbol{m} \in \mathbb{N}^n$. We have $\omega > n$ for all $n \in \mathbb{N}$. In our implementation, we restrict $\mathcal{R}(t)$ to functions constructed from max, min, $\mathbf{?}$, and polynomials from $\mathbb{N}[v_1, \ldots, v_n]$.

Definition 2 (Runtime Complexity and Approximation). *The* runtime complexity $\mathrm{rc} : \mathbb{N}^n \to \mathbb{N} \cup \{\omega\}$ *is defined as*[6] $\mathrm{rc}(\boldsymbol{m}) = \sup\{k \in \mathbb{N} \mid \exists \boldsymbol{v}_0, \ell, \boldsymbol{v} \,.\, \boldsymbol{v}_0 \leq$

[6] Here, $\mathrm{rc}(\boldsymbol{m}) = \omega$ means non-termination or arbitrarily long runtime. Such programs result from non-determinism, e.g., $\mathtt{i} =$ **nondet**(); **while** $\mathtt{i} > 0$ **do** $\mathtt{i} = \mathtt{i} - 1$ **done**.

$\boldsymbol{m} \wedge (\ell_0, \boldsymbol{v}_0) \to^k (\ell, \boldsymbol{v})\}$. *A function* $\mathcal{R} : \mathcal{T} \to \mathfrak{C}$ *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* $\boldsymbol{m} \in \mathbb{N}^n$. *The* initial runtime approximation $\mathcal{R}_0$ *is defined as*[7] $\mathcal{R}_0(t) = 1$ *for all initial transitions* t *and* $\mathcal{R}_0(t) = \mathbf{?}$ *otherwise.*

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.*

Definition 3 (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* $\mathcal{S} : \mathsf{RV} \to \mathfrak{C}$ *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* $\boldsymbol{m} \in \mathbb{N}^n$. *The* initial size approximation $\mathcal{S}_0$ *is defined as* $\mathcal{S}_0(t, v') = \mathbf{?}$ *for all* $|t, v'| \in \mathsf{RV}$. *A pair* $(\mathcal{R}, \mathcal{S})$ *is a* complexity approximation *if* $\mathcal{R}$ *is a runtime and* $\mathcal{S}$ *is a size approximation.*

Our approach starts with the initial approximation $(\mathcal{R}_0, \mathcal{S}_0)$ and improves it by iterative refinement. An approximation for the runtime complexity rc of the whole program $\mathcal{T}$ can be obtained by adding the runtime bounds $\mathcal{R}(t)$ for its transitions, i.e., $(\sum_{t \in \mathcal{T}} \mathcal{R}(t)) \geq \mathrm{rc}$. The overall bound $\sum_{t \in \mathcal{T}} \mathcal{R}(t) = 3{+}4{\cdot}|\mathtt{x}|{+}|\mathtt{x}|^2$ for our example was obtained in this way. Here for $f, g \in \mathfrak{C}$, the comparison, addition, multiplication, maximum, and the minimum are defined point-wise. 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 $n \in \mathbb{N} \cup \{\omega\}$.

3 Computing Runtime Bounds

To find runtime bounds automatically, we use (lexicographic combinations of) *polynomial rank functions* (PRFs). Such rank functions are widely used in termination analysis and many techniques are available to generate PRFs automatically [6, 8, 9, 12, 19–21, 33]. 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 Rank 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.

Definition 4 (PRF). *We call* $\mathcal{P}ol : \mathcal{L} \to \mathbb{Z}[v_1, \ldots, v_n]$ *a* polynomial rank function *(*PRF*) for* $\mathcal{T}$ *iff there is a non-empty* $\mathcal{T}_\succ \subseteq \mathcal{T}$ *such that the following holds:*

[7] Here, "1" denotes the constant function which maps all arguments $\boldsymbol{m} \in \mathbb{N}^n$ to 1.

- *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 constraints needed for termination proofs, allowing 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 $\mathcal{T}$.

In the program from Sect. 2, we could 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.

Such PRFs lead to a basic technique for inferring time bounds. As mentioned in Sect. 2, to obtain a modular approach afterwards, 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.

Theorem 5 (Complexities from PRFs). *Let $\mathcal{R}$ be a runtime approximation and $\mathcal{P}ol$ be a PRF for $\mathcal{T}$. Let*[8] *$\mathcal{R}'(t) = [\mathcal{P}ol(\ell_0)]$ 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.*

3.2 Modular Runtime Bounds from PRFs and Size Bounds

The basic method from Thm. 5 only succeeds in finding complexity bounds for simple examples. In particular, it often fails for programs with non-linear runtime. Although corresponding SAT- and SMT-encodings exist [20], 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

[8] To ensure that $\mathcal{R}'(t)$ is at most as large as the previous bound $\mathcal{R}(t)$, 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 $\mathcal{S}$.

$\mathcal{T}'$ 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.

For instance, consider the *subset* $\mathcal{T}_1' = \{t_1, \ldots, t_5\}$ of the transitions in our program. 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 $\mathcal{T}_1'$ can only be 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 can refine the runtime approximation $\mathcal{R}_1$ 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.

In general, to estimate how often a sub-program $\mathcal{T}'$ is reached in an evaluation, we consider the transitions $\tilde{t} \in \mathcal{T}$ that lead to an "entry location" ℓ in $\mathcal{T}'$. We *multiply* the runtime bound of such transitions $\tilde{t}$ with the bound $[\mathcal{P}ol(\ell)]$ for runs starting in ℓ. 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 $[\mathcal{P}ol_2(\ell_1)]$.

Next, we consider the remaining transitions $\mathcal{T}_2' = \{t_3, t_4, t_5\}$ for which we have no bound yet. We use $\mathcal{P}ol_3(\ell_2) = \mathcal{P}ol_3(\ell_3) = \mathtt{z}$ where $(\mathcal{T}_2')_{\succ} = \{t_5\}$. 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, 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 $\mathcal{S}(t_2, \mathtt{z}') = |\mathtt{x}|$.

So 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 $p \in \mathbb{N}[v_1, ..., v_n]$ and $f_1, ..., f_n \in \mathfrak{C}$. Then $p(f_1, ..., f_n)$ is the function with $(p(f_1, ..., f_n))(\boldsymbol{m}) = p(f_1(\boldsymbol{m}), \ldots, f_n(\boldsymbol{m}))$ for all $\boldsymbol{m} \in \mathbb{N}^n$. Weak monotonicity of $p, f_1, ..., f_n$ also implies weak monotonicity of $p(f_1, ..., f_n)$, i.e., $p(f_1, ..., f_n) \in \mathfrak{C}$.

For example, when analyzing how often t_5 is used in the sub-program $\mathcal{T}_2' = \{t_3, t_4, t_5\}$ above, we applied the polynomial $[\mathcal{P}ol_3(\ell_2)]$ for the start 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 $[\mathcal{P}ol_3(\ell_2)](\mathcal{S}(t_2, \mathtt{x}'), \mathcal{S}(t_2, \mathtt{y}'), \mathcal{S}(t_2, \mathtt{z}'), \mathcal{S}(t_2, \mathtt{u}')) = |\mathtt{x}|$.

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 Thm. 6, our technique is represented by the procedure TimeBounds. 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 $\mathcal{R}'$.

Theorem 6 (TimeBounds). *Let $(\mathcal{R}, \mathcal{S})$ be a complexity approximation and $\mathcal{T}' \subseteq \mathcal{T}$ such that $\mathcal{T}'$ contains no initial transitions. Let $\mathcal{L}' = \{\ell \mid (\ell, \tau, \ell') \in \mathcal{T}'\}$ contain all entry locations of $\mathcal{T}'$ and let $\mathcal{P}ol$ be a PRF for $\mathcal{T}'$. For any $\ell \in \mathcal{L}'$, let $\mathcal{T}_\ell$ contain all transitions $(\tilde{\ell}, \tilde{\tau}, \ell) \in \mathcal{T} \setminus \mathcal{T}'$ leading to ℓ. 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.

By Thm. 6, we now obtain bounds for the remaining transitions in our example. 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. So 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 $\mathsf{TimeBounds}$ 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 sizes 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 the local size bounds with our runtime approximation $\mathcal{R}$ to estimate how often transitions modify a variable value.

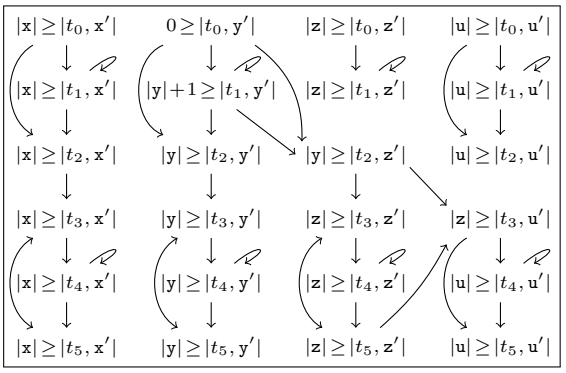
By a series of SMT queries, we find local size bounds $\mathcal{S}_l(t, v')$ that describe how the size of the post-variable v' is related to the pre-variables of a transition t. So while $\mathcal{S}(t, v')$ is a bound on the size of v after using t in a full program run, $\mathcal{S}_l(t, v')$ is a bound on v after a single use of t.

Definition 7 (Local Size Approximation). *We call $\mathcal{S}_l : \mathsf{RV} \to \mathfrak{C}$ 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 $\boldsymbol{m} \in \mathbb{N}^n$.*

In our example, we obtain $\mathcal{S}_l(t_1, \mathtt{y}') = |\mathtt{y}| + 1$, 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, decreasing $\mathtt{x}$ by

1 does not increase its *absolute* value. The bound $\max\{0, |\mathtt{x}| - 1\}$ would also be allowed, but our approach does not compute better global size bounds from it.

To track how variables influence each other, we construct a result variable graph (RVG) whose nodes are the result variables. An RVG for our example is shown below. 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 $\mathcal{S}_l(t_1, \mathtt{x}') = |\mathtt{x}|$).



The RVG has an edge from a result variable $|\tilde{t}, \tilde{v}'|$ to $|t, v'|$ if the transition $\tilde{t}$ can be used directly 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 $f \in \mathfrak{C}$, we define its *active variables* as $\mathsf{actV}(f) = \{v_i \in \mathcal{V} \mid \exists m_1, \ldots, m_n, m_i' \in \mathbb{N}.\ f(m_1, \ldots, m_i, \ldots, m_n) \neq f(m_1, \ldots, m_i', \ldots, m_n)\}$. 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 of $\mathsf{pre}(t)$, cf. [19,21]. For example, one can disregard the formulas of the transitions and approximate $\mathsf{pre}(t)$ by all transitions that end in t's source location.

Definition 8 (RVG). *Let $\mathcal{S}_l$ be a local size approximation. An* RVG *has $\mathcal{T}$'s result variables as nodes and the edges* $\{(|\tilde{t}, \tilde{v}'|, |t, v'|) \mid \tilde{t} \in \mathsf{pre}(t), \tilde{v} \in \mathsf{actV}(\mathcal{S}_l(t, v'))\}$.

For the transition t_2 which sets $\mathtt{z} = \mathtt{y}$, we obtain $\mathcal{S}_l(t_2, \mathtt{z}') = |\mathtt{y}|$. Hence, we have $\mathsf{actV}(\mathcal{S}_l(t_2, \mathtt{z}')) = \mathtt{y}$. The program graph implies $\mathsf{pre}(t_2) = \{t_0, t_1\}$, and thus, our RVG contains edges from $|t_0, \mathtt{y}'|$ to $|t_2, \mathtt{z}'|$ and from $|t_1, \mathtt{y}'|$ to $|t_2, \mathtt{z}'|$.

Each SCC of the RVG represents a set 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. 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. For example, regard the trivial SCC with the result variable $|t_0, \mathtt{y}'|$. As $0 \geq |t_0, \mathtt{y}'|$ holds, its global size bound is also 0, and we set $\mathcal{S}(t_0, \mathtt{y}') = 0$.

Next, we consider trivial SCCs $\alpha = |t, v'|$ with incoming edges from other SCCs. Now $\mathcal{S}_l(\alpha)\,(\boldsymbol{m})$ 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 $v_i \in \mathsf{actV}(\mathcal{S}_l(\alpha))$. Thus, a bound for the result variable $\alpha = |t, v'|$ is obtained by applying $\mathcal{S}_l(\alpha)$ to $\mathcal{S}(\tilde{t}, v_1'), \ldots, \mathcal{S}(\tilde{t}, v_n')$, for all $\tilde{t} \in \mathsf{pre}(t)$.

As an example consider the result variable $|t_2, \mathtt{z}'|$. Its local size bound is $\mathcal{S}_l(t_2, \mathtt{z}') = |\mathtt{y}|$. To express this bound in terms of the input variables, we consider the predecessors $|t_0, \mathtt{y}'|$ and $|t_1, \mathtt{y}'|$ of $|t_2, \mathtt{z}'|$ in the RVG. 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 $\mathcal{S}(t_1, \mathtt{y}') = |\mathtt{x}|$. Thus, $\mathcal{S}(t_2, \mathtt{z}') = \max\{0, |\mathtt{x}|\} = |\mathtt{x}|$.

Thm. 9 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 non-trivial SCC of the RVG. Non-trivial SCCs will be handled in Thm. 10.

Theorem 9 (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 corresponds to a loop and hence, each of its *local* changes can be applied several times. By combining the time bounds $\mathcal{R}(t)$ for its transitions t with the local size bounds $\mathcal{S}_l(t, v')$, we approximate the overall effect of these repeated changes. To simplify this approximation, we use the following classification of result variables α 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 its pre-variables or a constant, i.e., iff there is a number $e_\alpha \in \mathbb{N}$ with $\max\{e_\alpha, m_1, \ldots, m_n\} \geq (\mathcal{S}_l(\alpha))(m_1, \ldots, m_n)$ for all $m_1, \ldots, m_n \in \mathbb{N}$.
- $\alpha \in \dot{+}$ (α "adds a constant") if the result variable only increases over the pre-variables by a constant, i.e., iff there is a number $e_\alpha \in \mathbb{N}$ with $e_\alpha + \max\{m_1, \ldots, m_n\} \geq (\mathcal{S}_l(\alpha))(m_1, \ldots, m_n)$ for all $m_1, \ldots, m_n \in \mathbb{N}$.
- $\alpha \in \dot{\Sigma}$ (α "adds variables") if the result variable is not larger than the sum of the pre-variables and a constant, i.e., iff there is a number $e_\alpha \in \mathbb{N}$ with $e_\alpha + \sum_{i \in \{1,\ldots,n\}} m_i \geq (\mathcal{S}_l(\alpha))(m_1, \ldots, m_n)$ for all $m_1, \ldots, m_n \in \mathbb{N}$.

So for our example, we get $\{|t_3, \mathtt{z}'|, |t_4, \mathtt{z}'|, |t_5, \mathtt{z}'|\} \subseteq \doteq$ 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 $\mathcal{S}_l(t_1, \mathtt{y}') = |\mathtt{y}| + 1$.

In the following, local size bounds like $2 \cdot |\mathtt{x}|$ are not handled because we

are currently interested only in bounds that can be expressed by *polynomials* (and max and min). If a change bounded by $2 \cdot |\mathtt{x}|$ is applied $|\mathtt{y}|$ times, the resulting value is bounded only by the exponential function $2^{|\mathtt{y}|} \cdot |\mathtt{x}|$. Of course, our approach could be extended to infer such exponential size bounds as well. In Sect. 5, we discuss the limitations and possible extensions of our approach.

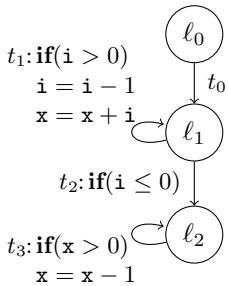
Similar to $\mathsf{pre}(t)$ for transitions t, let $\mathsf{pre}(\alpha)$ for a result variable α be those $\tilde{\alpha} \in \mathsf{RV}$ with an edge from $\tilde{\alpha}$ to α in the RVG. To deduce a bound on the size of the result variables α in an SCC C, we first consider the size of values *entering* the SCC C. Hence, we require that the resulting size bound $\mathcal{S}(\alpha)$ for $\alpha \in C$ 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 e_α. Hence, the bound $\mathcal{S}(\alpha)$ for the result variables α in C should also be at least $\max\{e_\alpha \mid \alpha \in \doteq\}$.[9]

For example, when computing the global size bounds for the result variables in the SCC $C = \{|t_3, \mathtt{z}'|, |t_4, \mathtt{z}'|, |t_5, \mathtt{z}'|\}$ in our example, the only predecessor of this SCC is $|t_2, \mathtt{z}'|$ with $\mathcal{S}(t_2, \mathtt{z}') = |\mathtt{x}|$. For each $\alpha \in C$, the corresponding constant e_α is 0. Thus, for all $\alpha \in C$ we obtain $\mathcal{S}(\alpha) = \max\{|\mathtt{x}|, 0\} = |\mathtt{x}|$.

To handle result variables $\alpha \in \dot{+} \setminus \doteq$ that add a constant e_α, we consider how often this addition is performed. Thus, while TimeBounds from Thm. 6 uses the size approximation $\mathcal{S}$ to improve the runtime approximation $\mathcal{R}$, SizeBounds uses $\mathcal{R}$ to improve $\mathcal{S}$. We define $\mathcal{R}(|t, v'|) = \mathcal{R}(t)$ for all result variables $|t, v'|$. Then, since $\mathcal{R}(\alpha)$ is a bound on the number of times that e_α is added, the repeated traversal of α's transition increases the overall size by at most $\mathcal{R}(\alpha) \cdot e_\alpha$.

For instance, consider the result variable $\alpha = |t_1, \mathtt{y}'|$ in our example. 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 $e_\alpha = 1$. As before, we use the size bounds on the predecessors of the SCC $\{\alpha\}$ as a basis. So the input value when entering the SCC is $\mathcal{S}(t_0, \mathtt{y}') = 0$. Since t_1 is executed at most $\mathcal{R}(\alpha) = \mathcal{R}(t_1) = |\mathtt{x}|$ times, we obtain the global bound $\mathcal{S}(\alpha) = \mathcal{S}(t_0, \mathtt{y}') + \mathcal{R}(\alpha) \cdot e_\alpha = 0 + |\mathtt{x}| \cdot 1 = |\mathtt{x}|$.

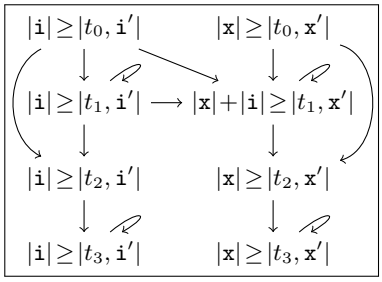
Finally, we discuss how to handle result variables $\alpha \in \dot{\Sigma} \setminus \dot{+}$. To this end, consider the program from Sect. 1 again. Its program graph is depicted on the right. Our method detects the runtime bounds $\mathcal{R}(t_0) = 1$, $\mathcal{R}(t_1) = |\mathtt{i}|$, and $\mathcal{R}(t_2) = 1$. To obtain size bounds, we first generate the RVG (see the next page). Now we can 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 $\alpha \in \dot{\Sigma} \setminus \dot{+}$.



For result variables α that sum up several program variables, we require that only *one* comes from α's own SCC in the RVG. Otherwise, we would also consider loops like **while** $\mathtt{z} > 0$ **do** $\mathtt{x} = \mathtt{x} + \mathtt{y}$; $\mathtt{y} = \mathtt{x}$; $\mathtt{z} = \mathtt{z} - 1$; **done** that increase the size of $\mathtt{x}$ exponentially. To express our requirement formally, let $\mathcal{V}_\alpha = \{v \mid |t, v'| \in \mathsf{pre}(\alpha) \cap C\}$ be those variables whose result variables in C have an edge to

[9] Again, "e_α" denotes the constant function mapping all values from $\mathbb{N}^n$ to e_α.

α. We require $|\mathcal{V}_\alpha| = 1$, i.e., no two result variables $|t, v'|, |\tilde{t}, \tilde{v}'|$ in α's SCC C with $v \neq \tilde{v}$ may have edges to α. But we allow incoming edges from arbitrary result variables *outside* the SCC. The requirement is satisfied in our RVG, as $\alpha = |t_1, \mathtt{x}'|$ is a predecessor of itself and its SCC contains no other result variables. Thus, $\mathcal{V}_\alpha = \{\mathtt{x}\}$. Of course, α also has predecessors of the form $|t, \mathtt{i}'|$ outside the SCC.



For each variable v, let f_v^α be an upper bound on the size of those result variables $|t, v'| \notin C$ that have edges to α, i.e., $f_v^\alpha = \max\{\mathcal{S}(t, v') \mid |t, v'| \in \mathsf{pre}(\alpha) \setminus C\}$. The execution of α's transition 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 $\mathcal{R}(\alpha)$ times. So the overall size is bounded by adding $\mathcal{R}(\alpha) \cdot (e_\alpha + \sum_{v \in \mathsf{actV}(\mathcal{S}_l(\alpha)) \setminus \mathcal{V}_\alpha} f_v^\alpha)$.

In our example with $\alpha = |t_1, \mathtt{x}'|$, we have $\mathcal{V}_\alpha = \{\mathtt{x}\}$, $\mathsf{actV}(\mathcal{S}_l(\alpha)) = \mathsf{actV}(|\mathtt{x}| + |\mathtt{i}|) = \{\mathtt{i}, \mathtt{x}\}$, 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 $\mathcal{R}(\alpha) = \mathcal{R}(t_1) = |\mathtt{i}|$ times), this value may be increased by adding $\mathcal{R}(\alpha) \cdot (e_\alpha + f_\mathtt{i}^\alpha) = |\mathtt{i}| \cdot (0 + |\mathtt{i}|) = |\mathtt{i}|^2$. Hence, we obtain $\mathcal{S}(\alpha) = \max\{|\mathtt{i}|, |\mathtt{x}|\} + |\mathtt{i}|^2$. Consequently, we also get $\mathcal{S}(t_2, \mathtt{x}') = \mathcal{S}(t_3, \mathtt{x}') = \max\{|\mathtt{i}|, |\mathtt{x}|\} + |\mathtt{i}|^2$. Thm. 10 extends the procedure SizeBounds from Thm. 9 to non-trivial SCCs.

Theorem 10 (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 an $\alpha \in C$ with $\alpha \notin \dot{\Sigma}$ or both $\alpha \in \dot{\Sigma} \setminus \dot{+}$ and $|\mathcal{V}_\alpha| > 1$, then we set $\mathcal{S}' = \mathcal{S}$. Otherwise, for all $\alpha \notin C$ let $\mathcal{S}'(\alpha) = \mathcal{S}(\alpha)$. For all $\alpha \in C$, we set*

$$\begin{aligned}\mathcal{S}'(\alpha) = \max(\ &\{\mathcal{S}(\tilde{\alpha}) \mid \textit{ there is an } \alpha \in C \textit{ with } \tilde{\alpha} \in \mathsf{pre}(\alpha) \setminus C\} \cup \{e_\alpha \mid \alpha \in \dot{=}\}\ ) \\ &+ \textstyle\sum_{\alpha \in \dot{+} \setminus \dot{=}} \mathcal{R}(\alpha) \cdot e_\alpha \\ &+ \textstyle\sum_{\alpha \in \dot{\Sigma} \setminus \dot{+}} \mathcal{R}(\alpha) \cdot (e_\alpha + \sum_{v \in \mathsf{actV}(\mathcal{S}_l(\alpha)) \setminus \mathcal{V}_\alpha} f_v^\alpha)\end{aligned}$$

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

In our example, by the inferred size bounds we can derive a runtime bound for the last transition t_3. When calling 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 reaching $\mathcal{T}'$, the size of $\mathtt{x}$ is bounded by $\mathcal{S}(t_2, \mathtt{x}')$. So $\mathcal{R}(t_3) = \mathcal{R}(t_2) \cdot [\mathcal{P}ol(\ell_2)](\mathcal{S}(t_2, \mathtt{i}'), \mathcal{S}(t_2, \mathtt{x}')) = 1 \cdot \mathcal{S}(t_2, \mathtt{x}') = \max\{|\mathtt{i}|, |\mathtt{x}|\} + |\mathtt{i}|^2$. So 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 $|\mathtt{i}|$.

5 Implementation and Related Work

We presented a new alternating modular approach for runtime and size complexity analysis of integer programs. Each step only considers a small part of the program, and runtime bounds help to infer size bounds and vice versa.

Our overall procedure to compute the runtime and size approximations $\mathcal{R}$ and

$\mathcal{S}$ is displayed on the right. After starting with the initial approximations $\mathcal{R}_0, \mathcal{S}_0$, the procedure TimeBounds (Thm. 6) is used to improve the runtime bounds for those transitions $\mathcal{T}'$ for which we have no bound yet.[10] Afterwards, the procedure SizeBounds (Thm. 9 and 10) considers the SCCs of the result variable graph in topological order to update the size approximation.

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

When all bounds have been determined, $\mathcal{R}$ and $\mathcal{S}$ are returned. Of course, we do not always succeed in finding bounds for all transitions and variables. Thus, while the procedure keeps on improving the bounds, at any point during its run, $\mathcal{R}$ and $\mathcal{S}$ are over-approximations of the actual runtimes and sizes. Hence, the procedure can be interrupted at any time and it always returns correct bounds.

Several methods to determine symbolic complexity bounds for programs have been developed in recent years. The approaches of [3, 4] (implemented in `COSTA` and its backend `PUBS`) and [37] (implemented in `Loopus`) also use an iterative procedure based on termination proving techniques to find runtime bounds for isolated loops, which are then combined to an overall result. However, [3, 4] handles all loop transitions at once and [37] is restricted to termination proofs via the size-change principle [28]. The approach of [6] (implemented in `Rank`) first proves termination by a lexicographic combination of linear rank functions, similar to our Thm. 6. However, while Thm. 6 combines these rank functions with size bounds, [6] approximates the reachable state space using Ehrhart polynomials. The tool `SPEED` [24] instruments programs by counters and employs an invariant generation tool to obtain bounds on these counters. The `ABC` system [11] 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 [10] which defines syntactic criteria for programs to have polynomial complexity.

The work on determining the *worst-case execution time* (WCET) for real-time systems [36] 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` [27].

There is also a wealth of work on complexity for declarative paradigms. For instance, *resource aware* `ML` [26] analyzes amortized complexity for recursive functional programs with inductive data types, but it does not handle programs whose complexity depends on integers. There are also numerous techniques for complexity analysis of term rewriting and logic programming [7, 18, 22, 31, 32].

[10] After generating a PRF $\mathcal{P}ol$ for $\mathcal{T}'$, it is advantageous to extend $\mathcal{T}'$ by all remaining transitions (ℓ, τ, ℓ') from $\mathcal{T} \setminus \mathcal{T}'$ where the measure $\mathcal{P}ol$ is also (weakly) decreasing, i.e., where $\tau \Rightarrow (\mathcal{P}ol(\ell))(v_1, \ldots, v_n) \geq (\mathcal{P}ol(\ell'))(v'_1, \ldots, v'_n)$. Calling the procedure TimeBounds with this extended set $\mathcal{T}'$ yields better results and may also improve previously found runtime bounds. We also used this strategy for the example in Sect. 3.

Our approach builds upon well-known basic concepts (like lexicographic rank 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.

To evaluate our approach, we implemented a prototype KoAT and compared it with PUBS [3, 4] and Rank [6]. We also contacted the authors of SPEED [24] and Loopus [37], but were not able to obtain these tools. We did not compare KoAT to ABC [11], RAML [26], or r-TuBound [27], as their input or analysis goals differ considerably from ours. As benchmarks, we collected 682 programs from the literature on termination and complexity of integer programs. These include all 36 examples from the evaluation of Rank, all but one of the 53 examples used to evaluate PUBS,[11] all 27 examples from the evaluations of SPEED, and the examples from the current paper (which can be handled by KoAT, but not by PUBS or Rank). Where examples were available as C programs, we used the tool KITTeL [19] to transform them into integer programs automatically. The collection contains 48 recursive examples, which cannot be analyzed with Rank, and 20 examples with non-linear arithmetic, which can be handled by neither Rank nor PUBS. The remaining examples are compatible with all tested tools. All examples, the results of the three tools, and a binary of KoAT are available at [1].

The table illustrates 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 the column "Time", we give the average runtime on those examples where the respective tool was successful. The average runtime on those 65 examples where *all* tools succeeded were 0.5 s for KoAT, 0.2 s for PUBS, and 0.6 s for Rank. 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.

	1	$\log n$	n	$n \log n$	n^2	n^3	$n^{>3}$	EXP	Time
KoAT	121	0	145	0	59	3	3	0	1.1 s
PUBS	116	5	131	5	22	7	0	6	0.8 s
Rank	56	0	19	0	8	1	0	0	0.5 s

On this collection, our approach was more powerful than the two other tools and still efficient. In fact, KoAT is only a simple prototype whose efficiency could still be improved considerably by fine-tuning its implementation. As shown in [1], there are 77 examples where KoAT infers a bound of a lower asymptotic class than PUBS, 548 examples where the bounds are in the same class, and 57 examples where the bound of PUBS is (asymptotically) more precise than KoAT's. Similarly, there are 259 examples where KoAT is asymptotically more precise than Rank, 410 examples where they are equal, and 13 examples where Rank is more precise. While KoAT is the only of the three tools that can also handle non-linear arithmetic, even when disregarding the 20 examples with non-linear arithmetic, KoAT can detect runtime bounds for 325 examples, whereas PUBS succeeds only for 292 programs and Rank only finds bounds for 84 examples.

A limitation of our implementation is that it only generates (possibly non-linear) PRFs to detect *polynomial* bounds. In contrast, PUBS uses PRFs to find *logarithmic* and *exponential* complexity bounds as well [3]. Such an extension

[11] We removed one example with undefined semantics.

could also be directly integrated into our method. Moreover, we are restricted to weakly monotonic bounds in order to allow their modular composition. 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.

Our method is easily *extended*. In [14], we provide an extension to handle (possibly recursive) procedure calls in a modular fashion. Moreover, we show 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.

Future work will be concerned with refining the precision of the inferred runtime and size approximations and with improving our implementation (e.g., by extending it to infer also non-polynomial complexities). Moreover, instead of abstracting heap operations to integers, we intend to investigate an extension of our approach to apply it directly to programs operating on the heap. Finally, similar to the coupling of `COSTA` with the tool `KeY` in [5], we want to automatically *certify* the complexity bounds found by our implementation `KoAT`.

Acknowledgments. We thank A. Ben-Amram, B. Cook, C. von Essen, C. Otto for valuable discussions and C. Alias and S. Genaim for help with the experiments.

References

1. `http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity/`
2. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination analysis of Java Bytecode. In: FMOODS '08. pp. 2–18 (2008)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. JAR 46(2), 161–203 (2011)
4. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. TCS 413(1), 142–159 (2012)
5. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: Verified resource guarantees using `COSTA` and `KeY`. In: PEPM '11. pp. 73–76 (2011)
6. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: SAS '10. pp. 117–133 (2010)
7. Avanzini, M., Moser, G.: A combination framework for complexity. In: RTA '13. pp. 55–70 (2013)
8. Bagnara, R., Mesnard, F., Pescetti, A., Zaffanella, E.: A new look at the automatic synthesis of linear ranking functions. IC 215, 47–67 (2012)
9. Ben-Amram, A.M., Genaim, S.: On the linear ranking problem for integer linear-constraint loops. In: POPL '13. pp. 51–62 (2013)
10. Ben-Amram, A.M., Jones, N.D., Kristiansen, L.: Linear, polynomial or exponential? Complexity inference in polynomial time. In: CiE '08. pp. 67–76 (2008)
11. Blanc, R., Henzinger, T.A., Hottelier, T., Kovács, L.: `ABC`: Algebraic bound computation for loops. In: LPAR-16. pp. 103–118 (2010)
12. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: CAV '05. pp. 491–504 (2005)
13. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. In: CAV '13. pp. 413–429 (2013)

14. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Alternating runtime and size complexity analysis of integer programs. Tech. Rep. AIB 2013-12, RWTH Aachen (2013), available from [1] and from `aib.informatik.rwth-aachen.de`
15. Brockschmidt, M., Musiol, R., Otto, C., Giesl, J.: Automated termination proofs for Java programs with cyclic data. In: CAV '12. pp. 105–122 (2012)
16. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI '06. pp. 415–426 (2006)
17. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: TACAS '13. pp. 47–61 (2013)
18. Debray, S., Lin, N.: Cost analysis of logic programs. TOPLAS 15, 826–875 (1993)
19. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: RTA '11. pp. 41–50 (2011)
20. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: SAT '07. pp. 340–354 (2007)
21. Fuhs, C., Giesl, J., Plücker, M., Schneider-Kamp, P., Falke, S.: Proving termination of integer term rewriting. In: RTA '09. pp. 32–47 (2009)
22. Giesl, J., Ströder, T., Schneider-Kamp, P., Emmes, F., Fuhs, C.: Symbolic evaluation graphs and term rewriting: A general methodology for analyzing logic programs. In: PPDP '12. pp. 1–12 (2012)
23. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. JAR 37(3), 155–203 (2006)
24. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: `SPEED`: Precise and efficient static estimation of program computational complexity. In: POPL '09. pp. 127–139 (2009)
25. Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K.: Alternation for termination. In: SAS '10. pp. 304–319 (2010)
26. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. TOPLAS 34(3) (2012)
27. Knoop, J., Kovács, L., Zwirchmayr, J.: `r-TuBound`: Loop bounds for WCET analysis. In: LPAR-18. pp. 435–444 (2012)
28. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL '01. pp. 81–92 (2001)
29. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL '10. pp. 211–222 (2010)
30. Miné, A.: The Octagon abstract domain. HOSC 19(1), 31–100 (2006)
31. Navas, J.A., Mera, E., López-García, P., Hermenegildo, M.V.: User-definable resource bounds analysis for logic programs. In: ICLP '07. pp. 348–363 (2007)
32. Noschinski, L., Emmes, F., Giesl, J.: Analyzing innermost runtime complexity of term rewriting by dependency pairs. JAR 51(1), 27–56 (2013)
33. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: VMCAI '04. pp. 239–251 (2004)
34. Spoto, F., Mesnard, F., Payet, É.: A termination analyser for Java Bytecode based on path-length. TOPLAS 32(3) (2010)
35. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop summarization and termination analysis. In: TACAS '11. pp. 81–95 (2011)
36. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D.B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P.P., Staschulat, J., Stenström, P.: The worst-case execution-time problem: overview of methods and survey of tools. TECS 7(3), 36:1–36:53 (2008)
37. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound analysis of imperative programs with the size-change abstraction. In: SAS '11. pp. 280–297 (2011)