Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops*

Nils Lommen, Fabian Meyer, and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract. There exist several results on deciding termination and computing runtime bounds for *triangular weakly non-linear loops* (twn-loops). We show how to use results on such subclasses of programs where complexity bounds are computable within incomplete approaches for complexity analysis of full integer programs. To this end, we present a novel modular approach which computes local runtime bounds for subprograms which can be transformed into twn-loops. These local runtime bounds are then lifted to global runtime bounds for the whole program. The power of our approach is shown by our implementation in the tool KoAT which analyzes complexity of programs where all other state-of-the-art tools fail.

1 Introduction

Most approaches for automated complexity analysis of programs are based on incomplete techniques like ranking functions (see, e.g., [1–4, 6, 11, 12, 18, 20, 21, 31]). However, there also exist numerous results on subclasses of programs where questions concerning termination or complexity are *decidable*, e.g., [5, 14, 15, 19, 22, 24, 25, 32, 34]. In this work we consider the subclass of *triangular weakly non-linear loops* (twn-loops), where there exist *complete* techniques for analyzing termination and runtime complexity (we discuss the "completeness" and decidability of these techniques below). An example for a twn-loop is:

$$\textbf{while } (x_1^2{+}x_3^5 < x_2 \wedge x_1 \neq 0) \textbf{ do } (x_1, x_2, x_3) \leftarrow (-2{\cdot}x_1,\ 3{\cdot}x_2{-}2{\cdot}x_3^3,\ x_3) \quad (1)$$

Its guard is a propositional formula over (possibly *non-linear*) polynomial inequations. The update is *weakly non-linear*, i.e., no variable x_i occurs non-linear in its own update. Furthermore, it is *triangular*, i.e., we can order the variables such that the update of any x_i does not depend on the variables $x_1, \ldots, x_{i-1}$ with smaller indices. Then, by handling one variable after the other one can compute a *closed form* which corresponds to applying the loop's update n times. Using these closed forms, termination can be reduced to an existential formula over $\mathbb{Z}$ [15] (whose validity is decidable for linear arithmetic and where SMT solvers often also prove

* funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2) and DFG Research Training Group 2236 UnRAVeL

(in)validity in the non-linear case). In this way, one can show that non-termination of twn-loops over $\mathbb{Z}$ is semi-decidable (and it is decidable over the real numbers).

While termination of twn-loops over $\mathbb{Z}$ is not decidable, by using the closed forms, [19] presented a "*complete*" complexity analysis technique. More precisely, for every twn-loop over $\mathbb{Z}$, it infers a polynomial which is an upper bound on the runtime for all those inputs where the loop terminates. So for all (possibly non-linear) terminating twn-loops over $\mathbb{Z}$, the technique of [19] *always* computes polynomial runtime bounds. In contrast, existing tools based on incomplete techniques for complexity analysis often fail for programs with non-linear arithmetic.

In [6, 18] we presented such an incomplete modular technique for complexity analysis which uses individual ranking functions for different subprograms. Based on this, we now introduce a novel approach to automatically infer runtime bounds for programs possibly consisting of multiple consecutive or nested loops by handling some subprograms as twn-loops and by using ranking functions for others. In order to compute runtime bounds, we analyze subprograms in topological order, i.e., in case of multiple consecutive loops, we start with the first loop and propagate knowledge about the resulting values of variables to subsequent loops. By inferring runtime bounds for one subprogram after the other, in the end we obtain a bound on the runtime complexity of the whole program. We first try to compute runtime bounds for subprograms by so-called multiphase linear ranking functions (MΦRFs, see [3, 4, 18, 20]). If MΦRFs do not yield a finite runtime bound for the respective subprogram, then we use our novel twn-technique on the unsolved parts of the subprogram. So for the first time, "complete" complexity analysis techniques like [19] for subclasses of programs with *non-linear* arithmetic are combined with incomplete techniques based on (linear) ranking functions like [6, 18]. Based on our approach, in future work one could integrate "complete" techniques for further subclasses (e.g., for *solvable loops* [24, 25, 30, 34] which can be transformed into twn-loops by suitable automorphisms [15]).

Structure: After introducing preliminaries in Sect. 2, in Sect. 3 we show how to lift a (local) runtime bound which is only sound for a subprogram to an overall global runtime bound. In contrast to previous techniques [6, 18], our lifting approach works for any method of bound computation (not only for ranking functions). In Sect. 4, we improve the existing results on complexity analysis of twn-loops [14, 15, 19] such that they yield concrete polynomial bounds, we refine these bounds by considering invariants, and we show how to apply these results to full programs which contain twn-loops as subprograms. Sect. 5 extends this technique to larger subprograms which can be transformed into twn-loops. In Sect. 6 we evaluate the implementation of our approach in the complexity analysis tool KoAT and show that one can now also successfully analyze the runtime of programs containing non-linear arithmetic. We refer to [26] for all proofs.

2 Preliminaries

This section recapitulates preliminaries for complexity analysis from [6, 18].

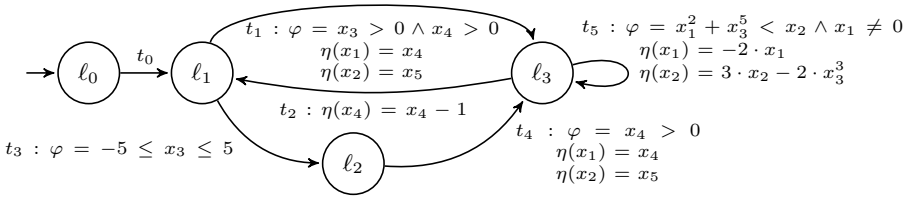

Fig. 1: An Integer Program with a Nested Self-Loop

Definition 1 (Atoms and Formulas). *We fix a set $\mathcal{V}$ of variables. The set of* atoms $\mathcal{A}(\mathcal{V})$ *consists of all inequations $p_1 < p_2$ for polynomials $p_1, p_2 \in \mathbb{Z}[\mathcal{V}]$. $\mathcal{F}(\mathcal{V})$ is the set of all propositional* formulas *built from atoms $\mathcal{A}(\mathcal{V})$, $\wedge$, and $\vee$.*

In addition to "$<$", we also use "$\geq$", "$=$", "$\neq$", etc., and negations "$\neg$" which can be simulated by formulas (e.g., $p_1 \geq p_2$ is equivalent to $p_2 < p_1 + 1$ for integers).

For integer programs, we use a formalism based on transitions, which also allows us to represent **while**-programs like (1) easily. Our programs may have *non-deterministic branching*, i.e., the guards of several applicable transitions can be satisfied. Moreover, *non-deterministic sampling* is modeled by *temporary variables* whose values are updated arbitrarily in each evaluation step.

Definition 2 (Integer Program). $(\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ *is an* integer program *where*

- $\mathcal{PV} \subseteq \mathcal{V}$ *is a finite set of* program variables*, $\mathcal{V} \setminus \mathcal{PV}$ are* temporary variables
- $\mathcal{L}$ *is a finite set of* locations *with an* initial location $\ell_0 \in \mathcal{L}$
- $\mathcal{T}$ *is a finite set of* transitions*. A transition is a 4-tuple $(\ell, \varphi, \eta, \ell')$ with a* start location $\ell \in \mathcal{L}$*,* target location $\ell' \in \mathcal{L} \setminus \{\ell_0\}$*,* guard $\varphi \in \mathcal{F}(\mathcal{V})$*, and* update function $\eta : \mathcal{PV} \to \mathbb{Z}[\mathcal{V}]$ *mapping program variables to update polynomials.*

Transitions $(\ell_0, \_, \_, \_)$ are called *initial*. Note that ℓ_0 has no incoming transitions.

Example 3. Consider the program in Fig. 1 with $\mathcal{PV} = \{x_i \mid 1 \leq i \leq 5\}$, $\mathcal{L} = \{\ell_i \mid 0 \leq i \leq 3\}$, and $\mathcal{T} = \{t_i \mid 0 \leq i \leq 5\}$, where t_5 has non-linear arithmetic in its guard and update. We omitted trivial guards, i.e., $\varphi = \texttt{true}$, and identity updates of the form $\eta(v) = v$. Thus, t_5 corresponds to the **while**-program (1).

A *state* is a mapping $\sigma : \mathcal{V} \to \mathbb{Z}$, Σ denotes the set of all states, and $\mathcal{L} \times \Sigma$ is the set of *configurations*. We also apply states to arithmetic expressions p or formulas φ, where the number $\sigma(p)$ resp. the Boolean value $\sigma(\varphi)$ results from replacing each variable v by $\sigma(v)$. So for a state with $\sigma(x_1) = -8$, $\sigma(x_2) = 55$, and $\sigma(x_3) = 1$, the expression $x_1^2 + x_3^5$ evaluates to $\sigma(x_1^2 + x_3^5) = 65$ and the formula $\varphi = (x_1^2 + x_3^5 < x_2)$ evaluates to $\sigma(\varphi) = (65 < 55) = \texttt{false}$. From now on, we fix a program $(\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$.

Definition 4 (Evaluation of Programs). *For configurations (ℓ, σ), (ℓ', σ') and $t = (\ell_t, \varphi, \eta, \ell_t') \in \mathcal{T}$, $(\ell, \sigma) \to_t (\ell', \sigma')$ is an* evaluation *step if $\ell = \ell_t$, $\ell' = \ell_t'$, $\sigma(\varphi) = \texttt{true}$, and $\sigma(\eta(v)) = \sigma'(v)$ for all $v \in \mathcal{PV}$. Let $\to_{\mathcal{T}} = \bigcup_{t \in \mathcal{T}} \to_t$, where*

we also write $\to$ *instead of* $\to_t$ *or* $\to_{\mathcal{T}}$. *Let* $(\ell_0, \sigma_0) \to^k (\ell_k, \sigma_k)$ *abbreviate* $(\ell_0, \sigma_0) \to \ldots \to (\ell_k, \sigma_k)$ *and let* $(\ell, \sigma) \to^* (\ell', \sigma')$ *if* $(\ell, \sigma) \to^k (\ell', \sigma')$ *for some* $k \geq 0$.

So when denoting states σ as tuples $(\sigma(x_1), \ldots, \sigma(x_5)) \in \mathbb{Z}^5$ for the program in Fig. 1, we have $(\ell_0, (1,5,7,1,3)) \to_{t_0} (\ell_1, (1,5,7,1,3)) \to_{t_1} (\ell_3, (1,1,3,1,3)) \to^3_{t_5} (\ell_3, (1,-8,55,1,3)) \to_{t_2} \ldots$. The runtime complexity $\mathrm{rc}(\sigma_0)$ of a program corresponds to the length of the longest evaluation starting in the initial state σ_0.

Definition 5 (Runtime Complexity). *The* runtime complexity *is* $\mathrm{rc} : \Sigma \to \overline{\mathbb{N}}$ *with* $\overline{\mathbb{N}} = \mathbb{N} \cup \{\omega\}$ *and* $\mathrm{rc}(\sigma_0) = \sup\{k \in \mathbb{N} \mid \exists (\ell', \sigma').\, (\ell_0, \sigma_0) \to^k (\ell', \sigma')\}$.

3 Computing Global Runtime Bounds

We now introduce our general approach for computing (upper) runtime bounds. We use weakly monotonically increasing functions as bounds, since they can easily be "composed" (i.e., if f and g increase monotonically, then so does $f \circ g$).

Definition 6 (Bounds [6, 18]). *The set of* bounds $\mathcal{B}$ *is the smallest set with* $\overline{\mathbb{N}} \subseteq \mathcal{B}$, $\mathcal{PV} \subseteq \mathcal{B}$, *and* $\{b_1 + b_2,\ b_1 \cdot b_2,\ k^{b_1}\} \subseteq \mathcal{B}$ *for all* $k \in \mathbb{N}$ *and* $b_1, b_2 \in \mathcal{B}$.

A bound constructed from $\mathbb{N}$, $\mathcal{PV}$, $+$, and $\cdot$ is *polynomial*. So for $\mathcal{PV} = \{x, y\}$, we have ω, x^2, $x + y$, $2^{x+y} \in \mathcal{B}$. Here, x^2 and $x + y$ are polynomial bounds.

We measure the size of variables by their absolute values. For any $\sigma \in \Sigma$, $|\sigma|$ is the state with $|\sigma|(v) = |\sigma(v)|$ for all $v \in \mathcal{V}$. So if σ_0 denotes the initial state, then $|\sigma_0|$ maps every variable to its initial "size", i.e., its initial absolute value. $\mathcal{RB}_{\mathrm{glo}} : \mathcal{T} \to \mathcal{B}$ is a *global runtime bound* if for each transition t and initial state $\sigma_0 \in \Sigma$, $\mathcal{RB}_{\mathrm{glo}}(t)$ evaluated in the state $|\sigma_0|$ over-approximates the number of evaluations of t in any run starting in the configuration (ℓ_0, σ_0). Let $\to^*_{\mathcal{T}} \circ \to_t$ denote the relation where arbitrary many evaluation steps are followed by a step with t.

Definition 7 (Global Runtime Bound [6, 18]). *The function* $\mathcal{RB}_{glo} : \mathcal{T} \to \mathcal{B}$ *is a* global runtime bound *if for all* $t \in \mathcal{T}$ *and all states* $\sigma_0 \in \Sigma$ *we have* $|\sigma_0|(\mathcal{RB}_{glo}(t)) \geq \sup\{k \in \mathbb{N} \mid \exists (\ell', \sigma').\, (\ell_0, \sigma_0)\ (\to^*_{\mathcal{T}} \circ \to_t)^k\ (\ell', \sigma')\}$.

For the program in Fig. 1, in Ex. 12 we will infer $\mathcal{RB}_{\mathrm{glo}}(t_0) = 1$, $\mathcal{RB}_{\mathrm{glo}}(t_i) = x_4$ for $1 \leq i \leq 4$, and $\mathcal{RB}_{\mathrm{glo}}(t_5) = 8 \cdot x_4 \cdot x_5 + 13006 \cdot x_4$. By adding the bounds for all transitions, a global runtime bound $\mathcal{RB}_{\mathrm{glo}}$ yields an upper bound on the program's runtime complexity. So for all $\sigma_0 \in \Sigma$ we have $|\sigma_0|(\sum_{t \in \mathcal{T}} \mathcal{RB}_{\mathrm{glo}}(t)) \geq \mathrm{rc}(\sigma_0)$.

For *local runtime bounds*, we consider the *entry transitions* of subsets $\mathcal{T}' \subseteq \mathcal{T}$.

Definition 8 (Entry Transitions [6, 18]). *Let* $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T}$. *Its* entry transitions *are* $\mathcal{E}_{\mathcal{T}'} = \{t \mid t = (\ell, \varphi, \eta, \ell') \in \mathcal{T} \setminus \mathcal{T}' \wedge$ *there is a transition* $(\ell', \_, \_, \_) \in \mathcal{T}'\}$.

So in Fig. 1, we have $\mathcal{E}_{\mathcal{T} \setminus \{t_0\}} = \{t_0\}$ and $\mathcal{E}_{\{t_5\}} = \{t_1, t_4\}$.

In contrast to global runtime bounds, a *local* runtime bound $\mathcal{RB}_{\mathrm{loc}} : \mathcal{E}_{\mathcal{T}'} \to \mathcal{B}$ only takes a subset $\mathcal{T}'$ into account. A *local run* is started by an entry transition $r \in \mathcal{E}_{\mathcal{T}'}$ followed by transitions from $\mathcal{T}'$. A *local runtime bound* considers a subset

$\mathcal{T}'_> \subseteq \mathcal{T}'$ and over-approximates the number of evaluations of any transition from $\mathcal{T}'_>$ in an arbitrary local run of the subprogram with the transitions $\mathcal{T}'$. More precisely, for every $t \in \mathcal{T}'_>$, $\mathcal{RB}_{\text{loc}}(r)$ over-approximates the number of applications of t in any run of $\mathcal{T}'$, if $\mathcal{T}'$ is entered via $r \in \mathcal{E}_{\mathcal{T}'}$. However, local runtime bounds do not consider how often an entry transition from $\mathcal{E}_{\mathcal{T}'}$ is evaluated or how large a variable is when we evaluate an entry transition. To illustrate that $\mathcal{RB}_{\text{loc}}(r)$ is a bound on the number of evaluations of transitions from $\mathcal{T}'_>$ after evaluating r, we often write $\mathcal{RB}_{\text{loc}}(\to_r \mathcal{T}'_>)$ instead of $\mathcal{RB}_{\text{loc}}(r)$.

Definition 9 (Local Runtime Bound). *Let $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$. The function $\mathcal{RB}_{loc} : \mathcal{E}_{\mathcal{T}'} \to \mathcal{B}$ is a* local runtime bound *for $\mathcal{T}'_>$ w.r.t. $\mathcal{T}'$ if for all $t \in \mathcal{T}'_>$, all $r \in \mathcal{E}_{\mathcal{T}'}$ with $r = (\ell, \_, \_, \_)$, and all $\sigma \in \Sigma$ we have $|\sigma|(\mathcal{RB}_{loc}(\to_r \mathcal{T}'_>)) \geq \sup\{k \in \mathbb{N} \mid \exists\, \sigma_0, (\ell', \sigma').\ (\ell_0, \sigma_0) \to^*_{\mathcal{T}} \circ \to_r\ (\ell, \sigma)\ (\to^*_{\mathcal{T}'} \circ \to_t)^k\ (\ell', \sigma')\}$.*

Our approach is *modular* since it computes local bounds for program parts separately. To lift local to global runtime bounds, we use *size bounds* $\mathcal{SB}(t, v)$ to over-approximate the size (i.e., absolute value) of the variable v after evaluating t in any run of the program. See [6] for the automatic computation of size bounds.

Definition 10 (Size Bound [6, 18]). *The function $\mathcal{SB} : (\mathcal{T} \times \mathcal{PV}) \to \mathcal{B}$ is a* size bound *if for all $(t, v) \in \mathcal{T} \times \mathcal{PV}$ and all states $\sigma_0 \in \Sigma$ we have $|\sigma_0|(\mathcal{SB}(t, v)) \geq \sup\{|\sigma'(v)| \mid \exists\, (\ell', \sigma').\ (\ell_0, \sigma_0)\ (\to^* \circ \to_t)\ (\ell', \sigma')\}$.*

To compute global from local runtime bounds $\mathcal{RB}_{\text{loc}}(\to_r \mathcal{T}'_>)$ and size bounds $\mathcal{SB}(r, v)$, Thm. 11 generalizes the approach of [6, 18]. Each local run is started by an entry transition r. Hence, we use an already computed global runtime bound $\mathcal{RB}_{\text{glo}}(r)$ to over-approximate the number of times that such a local run is started. To over-approximate the size of each variable v when entering the local run, we instantiate it by the size bound $\mathcal{SB}(r, v)$. So size bounds on previous transitions are needed to compute runtime bounds, and similarly, runtime bounds are needed to compute size bounds in [6]. For any bound b, "$b\,[v/\mathcal{SB}(r, v) \mid v \in \mathcal{PV}]$" results from b by replacing every program variable v by $\mathcal{SB}(r, v)$. Here, weak monotonic increase of b ensures that the over-approximation of the variables v in b by $\mathcal{SB}(r, v)$ indeed also leads to an over-approximation of b. The analysis starts with an *initial* runtime bound $\mathcal{RB}_{\text{glo}}$ and an *initial* size bound $\mathcal{SB}$ which map all transitions resp. all pairs from $\mathcal{T} \times \mathcal{PV}$ to ω, except for the transitions t which do not occur in cycles of $\mathcal{T}$, where $\mathcal{RB}_{\text{glo}}(t) = 1$. Afterwards, $\mathcal{RB}_{\text{glo}}$ and $\mathcal{SB}$ are refined repeatedly, where we alternate between computing runtime and size bounds.

Theorem 11 (Computing Global Runtime Bounds). *Let $\mathcal{RB}_{glo}$ be a global runtime bound, $\mathcal{SB}$ be a size bound, and $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ such that $\mathcal{T}'$ contains no initial transitions. Moreover, let $\mathcal{RB}_{loc}$ be a local runtime bound for $\mathcal{T}'_>$ w.r.t. $\mathcal{T}'$. Then $\mathcal{RB}'_{glo}$ is also a global runtime bound, where for all $t \in \mathcal{T}$ we define:*

$$\mathcal{RB}'_{glo}(t) = \begin{cases} \mathcal{RB}_{glo}(t), & \text{if } t \in \mathcal{T} \setminus \mathcal{T}'_> \\ \sum_{r \in \mathcal{E}_{\mathcal{T}'}} \mathcal{RB}_{glo}(r) \cdot (\mathcal{RB}_{loc}(\to_r \mathcal{T}'_>)\,[v/\mathcal{SB}(r, v) \mid v \in \mathcal{PV}]), & \text{if } t \in \mathcal{T}'_> \end{cases}$$

Example 12. For the example in Fig. 1, we first use $\mathcal{T}'_> = \{t_2\}$ and $\mathcal{T}' = \mathcal{T} \setminus \{t_0\}$. With the ranking function x_4 one obtains $\mathcal{RB}_{\text{loc}}(\to_{t_0} \mathcal{T}'_>) = x_4$, since t_2 decreases the value of x_4 and no transition increases it. Then we can infer the global runtime bound $\mathcal{RB}_{\text{glo}}(t_2) = \mathcal{RB}_{\text{glo}}(t_0) \cdot (x_4\,[v/\mathcal{SB}(t_0, v) \mid v \in \mathcal{PV}]) = x_4$ as $\mathcal{RB}_{\text{glo}}(t_0) = 1$ (since t_0 is evaluated at most once) and $\mathcal{SB}(t_0, x_4) = x_4$ (since t_0 does not change any variables). Similarly, we can infer $\mathcal{RB}_{\text{glo}}(t_1) = \mathcal{RB}_{\text{glo}}(t_3) = \mathcal{RB}_{\text{glo}}(t_4) = x_4$.

For $\mathcal{T}'_> = \mathcal{T}' = \{t_5\}$, our twn-approach in Sect. 4 will infer the local runtime bound $\mathcal{RB}_{\text{loc}} : \mathcal{E}_{\{t_5\}} \to \mathcal{B}$ with $\mathcal{RB}_{\text{loc}}(\to_{t_1} \{t_5\}) = 4 \cdot x_2 + 3$ and $\mathcal{RB}_{\text{loc}}(\to_{t_4} \{t_5\})$ $= 4 \cdot x_2 + 4 \cdot x_3^3 + 4 \cdot x_3^5 + 3$ in Ex. 30. By Thm. 11 we obtain the global bound

$$\begin{aligned}
\mathcal{RB}_{\text{glo}}(t_5) &= \mathcal{RB}_{\text{glo}}(t_1) \cdot (\mathcal{RB}_{\text{loc}}(\to_{t_1} \{t_5\})[v/\mathcal{SB}(t_1, v) \mid v \in \mathcal{PV}]) + \\
&\quad\ \mathcal{RB}_{\text{glo}}(t_4) \cdot (\mathcal{RB}_{\text{loc}}(\to_{t_4} \{t_5\})[v/\mathcal{SB}(t_4, v) \mid v \in \mathcal{PV}]) \\
&= x_4 \cdot (4 \cdot x_5 + 3) + x_4 \cdot (4 \cdot x_5 + 4 \cdot 5^3 + 4 \cdot 5^5 + 3) \\
&\qquad\qquad (\text{as } \mathcal{SB}(t_1, x_2) = \mathcal{SB}(t_4, x_2) = x_5 \text{ and } \mathcal{SB}(t_4, x_3) = 5) \\
&= 8 \cdot x_4 \cdot x_5 + 13006 \cdot x_4.
\end{aligned}$$

Thus, $\text{rc}(\sigma_0) \in \mathcal{O}(n^2)$ where n is the largest initial absolute value of all program variables. While the approach of [6, 18] was limited to local bounds resulting from ranking functions, here we need our Thm. 11. It allows us to use both local bounds resulting from twn-loops (for the non-linear transition t_5 where tools based on ranking functions cannot infer a bound, see Sect. 6) and local bounds resulting from ranking functions (for $t_1, \ldots, t_4$, since our twn-approach of Sect. 4 and 5 is limited to so-called simple cycles and cannot handle the full program).

In contrast to [6, 18], we allow different local bounds for different entry transitions in Def. 9 and Thm. 11. Our example demonstrates that this can indeed lead to a smaller asymptotic bound for the whole program: By distinguishing the cases where t_5 is reached via t_1 or t_4, we end up with a quadratic bound, because the local bound $\mathcal{RB}_{\text{loc}}(\to_{t_1} \{t_5\})$ is linear and while x_3 occurs with degrees 5 and 3 in $\mathcal{RB}_{\text{loc}}(\to_{t_4} \{t_5\})$, the size bound for x_3 is constant after t_3 and t_4.

To improve size and runtime bounds repeatedly, we treat the strongly connected components (SCCs)[1] of the program in topological order such that improved bounds for previous transitions are already available when handling the next SCC. We first try to infer local runtime bounds by multiphase-linear ranking functions (see [18] which also contains a heuristic for choosing $\mathcal{T}'_>$ and $\mathcal{T}'$ when using ranking functions). If ranking functions do not yield finite local bounds for all transitions of the SCC, then we apply the twn-technique from Sect. 4 and 5 on the remaining unbounded transitions (see Sect. 5 for choosing $\mathcal{T}'_>$ and $\mathcal{T}'$ in that case). Afterwards, the global runtime bound is updated according to Thm. 11.

4 Local Runtime Bounds for Twn-Self-Loops

In Sect. 4.1 we recapitulate twn-loops and their termination in our setting. Then in Sect. 4.2 we present a (complete) algorithm to infer polynomial runtime bounds

[1] As usual, a graph is *strongly connected* if there is a path from every node to every other node. A *strongly connected component* is a maximal strongly connected subgraph.

for all terminating twn-loops. Compared to [19], we increased its precision considerably by computing bounds that take the different roles of the variables into account and by using over-approximations to remove monomials. Moreover, we show how our algorithm can be used to infer local runtime bounds for twn-loops occurring in integer programs. Sect. 5 will show that our algorithm can also be applied to infer runtime bounds for larger cycles in programs instead of just self-loops.

4.1 Termination of Twn-Loops

Def. 13 extends the definition of twn-loops in [15, 19] by an initial transition and an update-invariant. Here, ψ is an *update-invariant* if $\models \psi \to \eta(\psi)$ where η is the update of the transition (i.e., invariance must hold independent of the guard).

Definition 13 (Twn-Loop). *An integer program* $(\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ *is a* triangular weakly non-linear loop (twn-loop) *if* $\mathcal{PV} = \{x_1, \ldots, x_d\}$ *for some* $d \geq 1$, $\mathcal{L} = \{\ell_0, \ell\}$, *and* $\mathcal{T} = \{t_0, t\}$ *with* $t_0 = (\ell_0, \psi, \mathrm{id}, \ell)$ *and* $t = (\ell, \varphi, \eta, \ell)$ *for some* $\psi, \varphi \in \mathcal{F}(\mathcal{PV})$ *with* $\models \psi \to \eta(\psi)$, *where* $\mathrm{id}(v) = v$ *for all* $v \in \mathcal{PV}$, *and for all* $1 \leq i \leq d$ *we have* $\eta(x_i) = c_i \cdot x_i + p_i$ *for some* $c_i \in \mathbb{Z}$ *and some polynomial* $p_i \in \mathbb{Z}[x_{i+1}, \ldots, x_d]$. *We often denote the loop by* (ψ, φ, η) *and refer to* ψ, φ, η *as its (update-) invariant, guard, and update, respectively. If* $c_i \geq 0$ *holds for all* $1 \leq i \leq d$, *then the program is a* non-negative *triangular weakly non-linear loop* (tnn-loop).

Example 14. The program consisting of the initial transition $(\ell_0, \texttt{true}, \mathrm{id}, \ell_3)$ and the self-loop t_5 in Fig. 1 is a twn-loop (corresponding to the **while**-loop (1)). This loop terminates as every iteration increases x_1^2 by a factor of 4 whereas x_2 is only tripled. Thus, $x_1^2 + x_3^5$ eventually outgrows the value of x_2.

To transform programs into twn- or tnn-form, one can combine subsequent transitions by *chaining*. Here, similar to states σ, we also apply the update η to polynomials and formulas by replacing each program variable v by $\eta(v)$.

Definition 15 (Chaining). *Let* $t_1, \ldots, t_n$ *be a sequence of transitions without temporary variables where* $t_i = (\ell_i, \varphi_i, \eta_i, \ell_{i+1})$ *for all* $1 \leq i \leq n-1$, *i.e., the target location of* t_i *is the start location of* t_{i+1}. *We may have* $t_i = t_j$ *for* $i \neq j$, *i.e., a transition may occur several times in the sequence. Then the transition* $t_1 \star \ldots \star t_n = (\ell_1, \varphi, \eta, \ell_{n+1})$ *results from* chaining $t_1, \ldots, t_n$ *where*

$$\varphi = \varphi_1 \wedge \eta_1(\varphi_2) \wedge \eta_2(\eta_1(\varphi_3)) \wedge \ldots \wedge \eta_{n-1}(\ldots \eta_1(\varphi_n) \ldots)$$
$$\eta(v) = \eta_n(\ldots \eta_1(v) \ldots) \text{ for all } v \in \mathcal{PV}, \text{ i.e., } \eta = \eta_n \circ \ldots \circ \eta_1.$$

Similar to [15, 19], we can restrict ourselves to tnn-loops, since chaining transforms any twn-loop L into a tnn-loop $L \star L$. Chaining preserves the termination behavior, and a bound on $L \star L$'s runtime can be transformed into a bound for L.

Lemma 16 (Chaining Preserves Asymptotic Runtime, see [19, Lemma 18]). *For the twn-loop* $L = (\psi, \varphi, \eta)$ *with the transitions* $t_0 = (\ell_0, \psi, \mathrm{id}, \ell)$, $t = (\ell, \varphi, \eta, \ell)$, *and runtime complexity* rc_L, *the program* $L \star L$ *with the transitions* t_0 *and* $t \star t = (\psi, \varphi \wedge \eta(\varphi), \eta \circ \eta)$ *is a tnn-loop. For its runtime complexity* $\mathrm{rc}_{L \star L}$, *we have* $2 \cdot \mathrm{rc}_{L \star L}(\sigma) \leq \mathrm{rc}_L(\sigma) \leq 2 \cdot \mathrm{rc}_{L \star L}(\sigma) + 1$ *for all* $\sigma \in \Sigma$.

Example 17. The program of Ex. 14 is only a twn-loop and not a tnn-loop as x_1 occurs with a negative coefficient -2 in its own update. Hence, we chain the loop and consider $t_5 \star t_5$. The update of $t_5 \star t_5$ is $(\eta \circ \eta)(x_1) = 4 \cdot x_1$, $(\eta \circ \eta)(x_2) = 9 \cdot x_2 - 8 \cdot x_3^3$, and $(\eta \circ \eta)(x_3) = x_3$. To ease the presentation, in this example we will keep the guard φ instead of using $\varphi \wedge \eta(\varphi)$ (ignoring $\eta(\varphi)$ in the conjunction of the guard does not decrease the runtime complexity).

Our algorithm starts with computing a closed form for the loop update, which describes the values of the program variables after n iterations of the loop. Formally, a tuple of arithmetic expressions $\mathtt{cl}^n_{\vec{x}} = (\mathtt{cl}^n_{x_1}, \ldots, \mathtt{cl}^n_{x_d})$ over the variables $\vec{x} = (x_1, \ldots, x_d)$ and the distinguished variable n is a *(normalized) closed form* for the update η with *start value* $n_0 \geq 0$ if for all $1 \leq i \leq d$ and all $\sigma : \{x_1, \ldots, x_d, n\} \to \mathbb{Z}$ with $\sigma(n) \geq n_0$, we have $\sigma(\mathtt{cl}^n_{x_i}) = \sigma(\eta^n(x_i))$. As shown in [14, 15, 19], for tnn-loops such a normalized closed form and the start value n_0 can be computed by handling one variable after the other, and these normalized closed forms can be represented as so-called *normalized poly-exponential expressions*. Here, $\mathbb{N}_{\geq m}$ stands for $\{x \in \mathbb{N} \mid x \geq m\}$.

Definition 18 (Normalized Poly-Exponential Expression [14, 15, 19]). *Let $\mathcal{PV} = \{x_1, \ldots, x_d\}$. Then we define the set of all* normalized poly-exponential expressions *by* $\mathbb{NPE} = \{\sum_{j=1}^{\ell} p_j \cdot n^{a_j} \cdot b_j^n \;\big|\; \ell, a_j \in \mathbb{N},\ p_j \in \mathbb{Q}[\mathcal{PV}],\ b_j \in \mathbb{N}_{\geq 1}\}$.

Example 19. A normalized closed form (with start value $n_0 = 0$) for the tnn-loop in Ex. 17 is $\mathtt{cl}^n_{x_1} = x_1 \cdot 4^n$, $\mathtt{cl}^n_{x_2} = (x_2 - x_3^3) \cdot 9^n + x_3^3$, and $\mathtt{cl}^n_{x_3} = x_3$.

Using the normalized closed form, similar to [15] one can represent non-termination of a tnn-loop (ψ, φ, η) by the formula

$$\exists\, \vec{x} \in \mathbb{Z}^d,\ m \in \mathbb{N}.\ \forall n \in \mathbb{N}_{\geq m}.\ \psi \wedge \varphi[\vec{x}/\mathtt{cl}^n_{\vec{x}}]. \tag{2}$$

Here, $\varphi[\vec{x}/\mathtt{cl}^n_{\vec{x}}]$ means that each variable x_i in φ is replaced by $\mathtt{cl}^n_{x_i}$. Since ψ is an update-invariant, if ψ holds, then $\psi[\vec{x}/\mathtt{cl}^n_{\vec{x}}]$ holds as well for all $n \geq n_0$. Hence, whenever $\forall n \in \mathbb{N}_{\geq m}.\ \psi \wedge \varphi[\vec{x}/\mathtt{cl}^n_{\vec{x}}]$ holds, then $\mathtt{cl}^{\max\{n_0,m\}}_{\vec{x}}$ witnesses non-termination. Thus, invalidity of (2) is equivalent to termination of the loop.

Normalized poly-exponential expressions have the advantage that it is always clear which addend determines their asymptotic growth when increasing n. So as in [15], (2) can be transformed into an existential formula and we use an SMT solver to prove its invalidity in order to prove termination of the loop. As shown in [15, Thm. 42], non-termination of twn-loops over $\mathbb{Z}$ is semi-decidable and deciding termination is Co-NP-complete if the loop is linear and the eigenvalues of the update matrix are rational.

4.2 Runtime Bounds for Twn-Loops via Stabilization Thresholds

As observed in [19], since the closed forms for tnn-loops are poly-exponential expressions that are weakly monotonic in n, every tnn-loop (ψ, φ, η) *stabilizes* for each input $\vec{e} \in \mathbb{Z}^d$. So there is a number of loop iterations (a *stabilization*

threshold $\mathrm{sth}_{(\psi,\varphi,\eta)}(\vec{e})$), such that the truth value of the loop guard φ does not change anymore when performing further loop iterations. Hence, the runtime of every terminating tnn-loop is bounded by its stabilization threshold.

Definition 20 (Stabilization Threshold). *Let (ψ,φ,η) be a tnn-loop with $\mathcal{PV} = \{x_1,\ldots,x_d\}$. For each $\vec{e} = (e_1,\ldots,e_d) \in \mathbb{Z}^d$, let $\sigma_{\vec{e}} \in \Sigma$ with $\sigma_{\vec{e}}(x_i) = e_i$ for all $1 \leq i \leq d$. Let $\Psi \subseteq \mathbb{Z}^d$ such that $\vec{e} \in \Psi$ iff $\sigma_{\vec{e}}(\psi)$ holds. Then $\mathrm{sth}_{(\psi,\varphi,\eta)} : \mathbb{Z}^d \to \mathbb{N}$ is the* stabilization threshold *of (ψ,φ,η) if for all $\vec{e} \in \Psi$, $\mathrm{sth}_{(\psi,\varphi,\eta)}(\vec{e})$ is the smallest number such that $\sigma_{\vec{e}}\big(\,\eta^n(\varphi) \leftrightarrow \eta^{\mathrm{sth}_{(\psi,\varphi,\eta)}(\vec{e})}(\varphi)\,\big)$ holds for all $n \geq \mathrm{sth}_{(\psi,\varphi,\eta)}(\vec{e})$.*

For the tnn-loop from Ex. 17, it will turn out that $2 \cdot x_2 + 2 \cdot x_3^3 + 2 \cdot x_3^5 + 1$ is an upper bound on its stabilization threshold, see Ex. 28.

To compute such upper bounds on a tnn-loop's stabilization threshold (i.e., upper bounds on its runtime if the loop is terminating), we now present a construction based on *monotonicity thresholds*, which are computable [19, Lemma 12].

Definition 21 (Monotonicity Threshold [19]). *Let $(b_1,a_1),(b_2,a_2) \in \mathbb{N}^2$ such that $(b_1,a_1) >_{\mathrm{lex}} (b_2,a_2)$ (i.e., $b_1 > b_2$ or both $b_1 = b_2$ and $a_1 > a_2$). For any $k \in \mathbb{N}_{\geq 1}$, the* k-monotonicity threshold *of (b_1,a_1) and (b_2,a_2) is the smallest $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ we have $n^{a_1} \cdot b_1^n > k \cdot n^{a_2} \cdot b_2^n$.*

For example, the 1-monotonicity threshold of $(4,0)$ and $(3,1)$ is 7 as the largest root of $f(n) = 4^n - n \cdot 3^n$ is approximately 6.5139.

Our procedure again instantiates the variables of the loop guard φ by the normalized closed form $\mathtt{cl}_{\vec{x}}^n$ of the loop's update. However, in the poly-exponential expressions $\sum_{j=1}^{\ell} p_j \cdot n^{a_j} \cdot b_j^n$ resulting from $\varphi[\vec{x}/\mathtt{cl}_{\vec{x}}^n]$, the corresponding technique of [19, Lemma 21] over-approximated the polynomials p_j by a polynomial that did not distinguish the effects of the different variables $x_1,\ldots,x_d$. Such an over-approximation is only useful for a direct asymptotic bound on the runtime of the twn-loop, but it is too coarse for a useful *local* runtime bound within the complexity analysis of a larger program. For instance, in Ex. 12 it is crucial to obtain local bounds like $4 \cdot x_2 + 4 \cdot x_3^3 + 4 \cdot x_3^5 + 3$ which indicate that only the variable x_3 may influence the runtime with an exponent of 3 or 5. Thus, if the size of x_3 is bound by a constant, then the resulting global bound becomes linear.

So we now improve precision and over-approximate the polynomials p_j by the polynomial $\sqcup\{p_1,\ldots,p_\ell\}$ which contains every monomial $x_1^{e_1} \cdot \ldots \cdot x_d^{e_d}$ of $\{p_1,\ldots,p_\ell\}$, using the absolute value of the largest coefficient with which the monomial occurs in $\{p_1,\ldots,p_\ell\}$. Thus, $\sqcup\{x_3^3 - x_3^5, x_2 - x_3^3\} = x_2 + x_3^3 + x_3^5$. In the following let $\vec{x} = (x_1,\ldots,x_d)$, and for $\vec{e} = (e_1,\ldots,e_d) \in \mathbb{N}^d$, $\vec{x}^{\vec{e}}$ denotes $x_1^{e_1} \cdot \ldots \cdot x_d^{e_d}$.

Definition 22 (Over-Approximation of Polynomials). *Let $p_1,\ldots,p_\ell \in \mathbb{Z}[\vec{x}]$, and for all $1 \leq j \leq \ell$, let $\mathcal{I}_j \subseteq (\mathbb{Z}\backslash\{0\}) \times \mathbb{N}^d$ be the* index set *of the polynomial p_j where $p_j = \sum_{(c,\vec{e}) \in \mathcal{I}_j} c \cdot \vec{x}^{\vec{e}}$ and there are no $c \neq c'$ with $(c,\vec{e}),(c',\vec{e}) \in \mathcal{I}_j$. For all $\vec{e} \in \mathbb{N}^d$ we define $c_{\vec{e}} \in \mathbb{N}$ with $c_{\vec{e}} = \max\{|c| \mid (c,\vec{e}) \in \mathcal{I}_1 \cup \ldots \cup \mathcal{I}_\ell\}$, where $\max \varnothing = 0$. Then the* over-approximation *of $p_1,\ldots,p_\ell$ is $\sqcup\{p_1,\ldots,p_\ell\} = \sum_{\vec{e} \in \mathbb{N}^d} c_{\vec{e}} \cdot \vec{x}^{\vec{e}}$.*

Clearly, $\sqcup\{p_1,\ldots,p_\ell\}$ indeed over-approximates the absolute value of each p_j.

Corollary 23 (Soundness of $\sqcup\{p_1,\dots,p_\ell\}$). *For all $\sigma : \{x_1,\dots,x_d\} \to \mathbb{Z}$ and all $1 \leq j \leq \ell$, we have $|\sigma|(\sqcup\{p_1,\dots,p_\ell\}) \geq |\sigma(p_j)|$.*

A drawback is that $\sqcup\{p_1,\dots,p_\ell\}$ considers all monomials and to obtain weakly monotonically increasing bounds from $\mathcal{B}$, it uses the absolute values of their coefficients. This can lead to polynomials of unnecessarily high degree. To improve the precision of the resulting bounds, we now allow to over-approximate the poly-exponential expressions $\sum_{j=1}^{\ell} p_j \cdot n^{a_j} \cdot b_j^n$ which result from instantiating the variables of the loop guard by the closed form. For this over-approximation, we take the invariant ψ of the tnn-loop into account. So while (2) showed that update-invariants ψ can restrict the sets of possible witnesses for non-termination and thus simplify the termination proofs of twn-loops, we now show that preconditions ψ can also be useful to improve the bounds on twn-loops.

More precisely, Def. 24 allows us to replace addends $p \cdot n^a \cdot b^n$ by $p \cdot n^i \cdot j^n$ where $(j,i) >_{\text{lex}} (b,a)$ if the monomial p is always positive (when the precondition ψ is fulfilled) and where $(b,a) >_{\text{lex}} (i,j)$ if p is always non-positive.

Definition 24 (Over-Approximation of Poly-Exponential Expressions). *Let $\psi \in \mathcal{F}(\mathcal{PV})$ and let $npe = \sum_{(p,a,b)\in\Lambda} p \cdot n^a \cdot b^n \in \mathbb{NPE}$ where Λ is a set of tuples (p,a,b) containing a monomial[2] p and two numbers $a,b \in \mathbb{N}$. Here, we may have $(p,a,b),(p',a,b) \in \Lambda$ for $p \neq p'$. Let $\Delta,\Gamma \subseteq \Lambda$ such that $\models \psi \to (p > 0)$ holds for all $(p,a,b) \in \Delta$ and $\models \psi \to (p \leq 0)$ holds for all $(p,a,b) \in \Gamma$.[3] Then*

$$\lceil npe \rceil^{\psi}_{\Delta,\Gamma} = \sum_{(p,a,b)\in\Delta\uplus\Gamma} p \cdot n^{i_{(p,a,b)}} \cdot j^n_{(p,a,b)} + \sum_{(p,a,b)\in\Lambda\setminus(\Delta\uplus\Gamma)} p \cdot n^a \cdot b^n$$

is an over-approximation *of npe if $i_{(p,a,b)}, j_{(p,a,b)} \in \mathbb{N}$ are numbers such that $(j_{(p,a,b)}, i_{(p,a,b)}) >_{\text{lex}} (b,a)$ holds if $(p,a,b) \in \Delta$ and $(b,a) >_{\text{lex}} (j_{(p,a,b)}, i_{(p,a,b)})$ holds if $(p,a,b) \in \Gamma$. Note that $i_{(p,a,b)}$ or $j_{(p,a,b)}$ can also be 0.*

Example 25. Let $npe = q_3 \cdot 16^n + q_2 \cdot 9^n + q_1 = q_3 \cdot 16^n + q_2' \cdot 9^n + q_2'' \cdot 9^n + q_1' + q_1''$, where $q_3 = -x_1^2$, $q_2 = q_2' + q_2''$, $q_2' = x_2$, $q_2'' = -x_3^3$, $q_1 = q_1' + q_1''$, $q_1' = x_3^3$, $q_1'' = -x_3^5$, and $\psi = (x_3 > 0)$. We can choose $\Delta = \{(x_3^3, 0, 1)\}$ since $\models \psi \to (x_3^3 > 0)$ and $\Gamma = \{(-x_3^5, 0, 1)\}$ since $\models \psi \to (-x_3^5 \leq 0)$. Moreover, we choose $j_{(x_3^3,0,1)} = 9$, $i_{(x_3^3,0,1)} = 0$, which is possible since $(9,0) >_{\text{lex}} (1,0)$. Similarly, we choose $j_{(-x_3^5,0,1)} = 0$, $i_{(-x_3^5,0,1)} = 0$, since $(1,0) >_{\text{lex}} (0,0)$. Thus, we replace x_3^3 and $-x_3^5$ by the larger addends $x_3^3 \cdot 9^n$ and 0. The motivation for the latter is that this removes all addends with exponent 5 from npe. The motivation for the former is that then, we have both the addends $-x_3^3 \cdot 9^n$ and $x_3^3 \cdot 9^n$ in the expression which cancel out, i.e., this removes all addends with exponent 3. Hence, we obtain $\lceil npe \rceil^{\psi}_{\Delta,\Gamma} = p_2 \cdot 16^n + p_1 \cdot 9^n$ with $p_2 = -x_1^2$ and $p_1 = x_2$. To find a suitable over-approximation which removes addends with high exponents, our implementation uses a heuristic for the choice of Δ, Γ, $i_{(p,a,b)}$, and $j_{(p,a,b)}$.

The following lemma shows the soundness of the over-approximation $\lceil npe \rceil^{\psi}_{\Delta,\Gamma}$.

[2] Here, we consider monomials of the form $p = c \cdot x_1^{e_1} \cdot \ldots \cdot x_d^{e_d}$ with coefficients $c \in \mathbb{Q}$.
[3] Δ and Γ do not have to contain *all* such tuples, but can be (possibly empty) subsets.

Lemma 26 (Soundness of $\lceil npe \rceil^{\psi}_{\Delta,\Gamma}$). *Let ψ, npe, Δ, Γ, $i_{(p,a,b)}$, $j_{(p,a,b)}$, and $\lceil npe \rceil^{\psi}_{\Delta,\Gamma}$ be as in Def. 24, and let* $D_{\lceil npe \rceil^{\psi}_{\Delta,\Gamma}} =$

$$\max(\ \{1\text{-monotonicity threshold of } (j_{(p,a,b)}, i_{(p,a,b)}) \text{ and } (b,a) \mid (p,a,b) \in \Delta\} \cup \{1\text{-monotonicity threshold of } (b,a) \text{ and } (j_{(p,a,b)}, i_{(p,a,b)}) \mid (p,a,b) \in \Gamma\}\ ).$$

Then for all $\vec{e} \in \Psi$ and all $n \geq D_{\lceil npe \rceil^{\psi}_{\Delta,\Gamma}}$, we have $\sigma_{\vec{e}}(\lceil npe \rceil^{\psi}_{\Delta,\Gamma}) \geq \sigma_{\vec{e}}(npe)$.

For any terminating tnn-loop (ψ, φ, η), Thm. 27 now uses the new concepts of Def. 22 and 24 to compute a polynomial $\text{sth}^{\sqcup}$ which is an upper bound on the loop's stabilization threshold (and hence, on its runtime). For any atom $\alpha = (s_1 < s_2)$ (resp. $s_2 - s_1 > 0$) in the loop guard φ, let $npe_\alpha \in \mathbb{NPE}$ be a poly-exponential expression which results from multiplying $(s_2 - s_1)[\vec{x}/\mathtt{cl}^n_{\vec{x}}]$ with the least common multiple of all denominators occurring in $(s_2 - s_1)[\vec{x}/\mathtt{cl}^n_{\vec{x}}]$. Since the loop is terminating, for some of these atoms this expression will become non-positive for large enough n and our goal is to compute bounds on their corresponding stabilization thresholds. First, one can replace npe_α by an over-approximation $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}$ where $\psi' = (\psi \wedge \varphi)$ considers both the invariant ψ and the guard φ. Let $\Psi' \subseteq \mathbb{Z}^d$ such that $\vec{e} \in \Psi'$ iff $\sigma_{\vec{e}}(\psi')$ holds. By Lemma 26 (i.e., $\sigma_{\vec{e}}(\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}) \geq \sigma_{\vec{e}}(npe_\alpha)$ for all $\vec{e} \in \Psi'$), it suffices to compute a bound on the stabilization threshold of $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}$ if it is always non-positive for large enough n, because if $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}$ is non-positive, then so is npe_α. We say that an over-approximation $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}$ is *eventually non-positive* iff whenever $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma} \neq npe_\alpha$, then one can show that for all $\vec{e} \in \Psi'$, $\sigma_{\vec{e}}(\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma})$ is always non-positive for large enough n.[4] Using over-approximations $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}$ can be advantageous because $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}$ may contain less monomials than npe_α and thus, the construction $\sqcup$ from Def. 22 can yield a polynomial of lower degree. So although npe_α's stabilization threshold might be smaller than the one of $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}$, our technique might compute a smaller bound on the stabilization threshold when considering $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}$ instead of npe.

Theorem 27 (Bound on Stabilization Threshold). *Let $L = (\psi, \varphi, \eta)$ be a terminating tnn-loop, let $\psi' = (\psi \wedge \varphi)$, and let $\mathtt{cl}^n_{\vec{x}}$ be a normalized closed form for η with start value n_0. For every atom $\alpha = (s_1 < s_2)$ in φ, let $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}$ be an eventually non-positive over-approximation of npe_α and let $D_\alpha = D_{\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma}}$.*

If $\lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma} = \sum_{j=1}^{\ell} p_j \cdot n^{a_j} \cdot b_j^n$ with $p_j \neq 0$ for all $1 \leq j \leq \ell$ and $(b_\ell, a_\ell) >_{\text{lex}} \ldots >_{\text{lex}} (b_1, a_1)$, then let $C_\alpha = \max\{1, N_2, M_2, \ldots, N_\ell, M_\ell\}$, where we have:

$$M_j = \begin{cases} 0, & \text{if } b_j = b_{j-1} \\ \text{1-monotonicity threshold of } (b_j, a_j) \text{ and } (b_{j-1}, a_{j-1}+1), & \text{if } b_j > b_{j-1} \end{cases} \qquad N_j = \begin{cases} 1, & \text{if } j = 2 \\ mt', & \text{if } j = 3 \\ \max\{mt, mt'\}, & \text{if } j > 3 \end{cases}$$

[4] This can be shown similar to the proof of (2) for (non-)termination of the loop. Thus, we transform $\exists \vec{x} \in \mathbb{Z}^d,\ m \in \mathbb{N}.\ \forall n \in \mathbb{N}_{\geq m}.\ \psi' \wedge \lceil npe_\alpha \rceil^{\psi'}_{\Delta,\Gamma} > 0$ into an existential formula as in [15] and try to prove its invalidity by an SMT solver.

Here, mt' is the $(j-2)$-monotonicity threshold of (b_{j-1}, a_{j-1}) and (b_{j-2}, a_{j-2}) and $mt = \max\{1\text{-monotonicity threshold of } (b_{j-2}, a_{j-2}) \text{ and } (b_i, a_i) \mid 1 \leq i \leq j-3\}$. Let $Pol_\alpha = \{p_1, \ldots, p_{\ell-1}\}$, $Pol = \bigcup_{\text{atom } \alpha \text{ occurs in } \varphi} Pol_\alpha$, $C = \max\{C_\alpha \mid \text{atom } \alpha \text{ occurs in } \varphi\}$, $D = \max\{D_\alpha \mid \text{atom } \alpha \text{ occurs in } \varphi\}$, and $\mathrm{sth}^{\sqcup} \in \mathbb{Z}[\vec{x}]$ with $\mathrm{sth}^{\sqcup} = 2 \cdot \sqcup Pol + \max\{n_0, C, D\}$. Then for all $\vec{e} \in \Psi'$, we have $|\sigma_{\vec{e}}|(\mathrm{sth}^{\sqcup}) \geq \mathrm{sth}_{(\psi,\varphi,\eta)}(\vec{e})$. If the tnn-loop has the initial transition t_0 and looping transition t, then $\mathcal{RB}_{glo}(t_0) = 1$ and $\mathcal{RB}_{glo}(t) = \mathrm{sth}^{\sqcup}$ is a global runtime bound for L.

Example 28. The guard φ of the tnn-loop in Ex. 17 has the atoms $\alpha = (x_1^2 + x_3^5 < x_2)$, $\alpha' = (0 < x_1)$, and $\alpha'' = (0 < -x_1)$ (since $x_1 \neq 0$ is transformed into $\alpha' \vee \alpha''$). When instantiating the variables by the closed forms of Ex. 19 with start value $n_0 = 0$, Thm. 27 computes the bound 1 on the stabilization thresholds for α' and α''. So the only interesting atom is $\alpha = (0 < s_2 - s_1)$ for $s_1 = x_1^2 + x_3^5$ and $s_2 = x_2$. We get $npe_\alpha = (s_2 - s_1)[\vec{x}/\mathtt{cl}_{\vec{x}}^n] = q_3 \cdot 16^n + q_2 \cdot 9^n + q_1$, with q_j as in Ex. 25.

In the program of Fig. 1, the corresponding self-loop t_5 has two entry transitions t_4 and t_1 which result in two tnn-loops with the update-invariants $\psi_1 = \mathtt{true}$ resulting from transition t_4 and $\psi_2 = (x_3 > 0)$ from t_1. So ψ_2 is an update-invariant of t_5 which always holds when reaching t_5 via transition t_1.

For $\psi_1 = \mathtt{true}$, we choose $\Delta = \Gamma = \varnothing$, i.e., $\lceil npe_\alpha \rceil_{\Delta,\Gamma}^{\psi_1'} = npe_\alpha$. So we have $b_3 = 16$, $b_2 = 9$, $b_1 = 1$, and $a_j = 0$ for all $1 \leq j \leq 3$. We obtain

$$\begin{array}{r} M_2 = 0, \text{ as } 0 \text{ is the 1-monotonicity threshold of } (9,0) \text{ and } (1,1) \\ M_3 = 0, \text{ as } 0 \text{ is the 1-monotonicity threshold of } (16,0) \text{ and } (9,1) \\ N_2 = 1 \text{ and } N_3 = 1, \text{ as } 1 \text{ is the 1-monotonicity threshold of } (9,0) \text{ and } (1,0). \end{array}$$

Hence, we get $C = C_\alpha = \max\{1, N_2, M_2, N_3, M_3\} = 1$. So we obtain the runtime bound $\mathrm{sth}_{\psi_1}^{\sqcup} = 2 \cdot \sqcup\{q_1, q_2\} + \max\{n_0, C_\alpha\} = 2 \cdot x_2 + 2 \cdot x_3^3 + 2 \cdot x_3^5 + 1$ for the loop $t_5 \star t_5$ w.r.t. ψ_1. By Lemma 16, this means that $2 \cdot \mathrm{sth}_{\psi_1}^{\sqcup} + 1 = 4 \cdot x_2 + 4 \cdot x_3^3 + 4 \cdot x_3^5 + 3$ is a runtime bound for the loop at transition t_5.

For the update-invariant $\psi_2 = (x_3 > 0)$, we use the over-approximation $\lceil npe_\alpha \rceil_{\Delta,\Gamma}^{\psi_2'} = p_2 \cdot 16^n + p_1 \cdot 9^n$ with $p_2 = -x_1^2$ and $p_1 = x_2$ from Ex. 25, where $\psi_2' = (\psi_2 \wedge \varphi)$ implies that it is always non-positive for large enough n. Now we obtain $M_2 = 0$ (the 1-monotonicity threshold of $(16,0)$ and $(9,1)$) and $N_2 = 1$, where $C = C_\alpha = \max\{1, N_2, M_2\} = 1$. Moreover, we have $D_\alpha = \max\{1, 0\} = 1$, since

$$\begin{array}{l} 1 \text{ is the 1-monotonicity threshold of } (9,0) \text{ and } (1,0), \text{ and} \\ 0 \text{ is the 1-monotonicity threshold of } (1,0) \text{ and } (0,0). \end{array}$$

We now get the tighter bound $\mathrm{sth}_{\psi_2}^{\sqcup} = 2 \cdot \sqcup\{p_1\} + \max\{n_0, C_\alpha, D_\alpha\} = 2 \cdot x_2 + 1$ for $t_5 \star t_5$. So t_5's runtime bound is $2 \cdot \mathrm{sth}_{\psi_2}^{\sqcup} + 1 = 4 \cdot x_2 + 3$ when using invariant ψ_2.

Thm. 29 shows how the technique of Lemma 16 and Thm. 27 can be used to compute local runtime bounds for twn-loops whenever such loops occur within an integer program. To this end, one needs the new Thm. 11 where in contrast to [6, 18] these local bounds do not have to result from ranking functions.

To turn a self-loop t and $r \in \mathcal{E}_{\{t\}}$ from a larger program $\mathcal{P}$ into a twn-loop (ψ, φ, η), we use t's guard φ and update η. To obtain an update-invariant ψ, our

implementation uses the Apron library [23] for computing invariants on a version of the full program where we remove all entry transitions $\mathcal{E}_{\{t\}}$ except r.[5] From the invariants computed for t, we take those that are also update-invariants of t.

Theorem 29 (Local Bounds for Twn-Loops). *Let $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ be an integer program with $\mathcal{PV}' = \{x_1, \ldots, x_d\} \subseteq \mathcal{PV}$. Let $t = (\ell, \varphi, \eta, \ell) \in \mathcal{T}$ with $\varphi \in \mathcal{F}(\mathcal{PV}')$, $\eta(v) \in \mathbb{Z}[\mathcal{PV}']$ for all $v \in \mathcal{PV}'$, and $\eta(v) = v$ for all $v \in \mathcal{PV} \setminus \mathcal{PV}'$. For any entry transition $r \in \mathcal{E}_{\{t\}}$, let $\psi \in \mathcal{F}(\mathcal{PV}')$ such that $\models \psi \to \eta(\psi)$ and such that $\sigma(\psi)$ holds whenever there is a $\sigma_0 \in \Sigma$ with $(\ell_0, \sigma_0) \to_{\mathcal{T}}^* \circ \to_r (\ell, \sigma)$. If $L = (\psi, \varphi, \eta)$ is a terminating tnn-loop, then let $\mathcal{RB}_{loc}(\to_r \{t\}) = \mathrm{sth}^{\sqcup}$, where $\mathrm{sth}^{\sqcup}$ is defined as in Thm. 27. If L is a terminating twn-loop but no tnn-loop, let $\mathcal{RB}_{loc}(\to_r \{t\}) = 2 \cdot \mathrm{sth}^{\sqcup} + 1$, where $\mathrm{sth}^{\sqcup}$ is the bound of Thm. 27 computed for $L \star L$. Otherwise, let $\mathcal{RB}_{loc}(\to_r \{t\}) = \omega$. Then $\mathcal{RB}_{loc}$ is a local runtime bound for $\{t\} = \mathcal{T}'_{>} = \mathcal{T}'$ in the program $\mathcal{P}$.*

Example 30. In Fig. 1, we consider the self-loop t_5 with $\mathcal{E}_{\{t_5\}} = \{t_4, t_1\}$ and the update-invariants $\psi_1 = \texttt{true}$ resp. $\psi_2 = (x_3 > 0)$. For t_5's guard φ and update η, both (ψ_i, φ, η) are terminating twn-loops (see Ex. 14), i.e., (2) is invalid.

By Thm. 29 and Ex. 28, $\mathcal{RB}_{\mathrm{loc}}$ with $\mathcal{RB}_{\mathrm{loc}}(\to_{t_4} \{t_5\}) = 4 \cdot x_2 + 4 \cdot x_3^3 + 4 \cdot x_3^5 + 3$ and $\mathcal{RB}_{\mathrm{loc}}(\to_{t_1} \{t_5\}) = 4 \cdot x_2 + 3$ is a local runtime bound for $\{t_5\} = \mathcal{T}'_{>} = \mathcal{T}'$ in the program of Fig. 1. As shown in Ex. 12, Thm. 11 then yields the global runtime bound $\mathcal{RB}_{\mathrm{glo}}(t_5) = 8 \cdot x_4 \cdot x_5 + 13006 \cdot x_4$.

5 Local Runtime Bounds for Twn-Cycles

Sect. 4 introduced a technique to determine local runtime bounds for twn-self-loops in a program. To increase its applicability, we now extend it to larger cycles. For every entry transition of the cycle, we *chain* the transitions of the cycle, starting with the transition which follows the entry transition. In this way, we obtain loops consisting of a single transition. If the chained loop is a twn-loop, we can apply Thm. 29 to compute a local runtime bound. Any local bound on the chained transition is also a bound on each of the original transitions.[6]

By Thm. 29, we obtain a bound on the number of evaluations of the *complete cycle*. However, we also have to consider a *partial execution* which stops before traversing the full cycle. Therefore, we increase every local runtime bound by 1.

Note that this replacement of a cycle by a self-loop which results from chaining its transitions is only sound for *simple* cycles. A cycle is simple if each iteration through the cycle can only be done in a unique way. So the cycle must not have any subcycles and there also must not be any indeterminisms concerning the

[5] Regarding invariants for the full program in the computation of local bounds for t is possible since in contrast to [6, 18] our definition of local bounds from Def. 9 is restricted to states that are reachable from an initial configuration (ℓ_0, σ_0).

[6] This is sufficient for our improved definition of local bounds in Def. 9 where in contrast to [6, 18] we do not require a bound on the *sum* but only on *each* transition in the considered set $\mathcal{T}'$. Moreover, here we again benefit from our extension to compute individual local bounds for different entry transitions.

Algorithm 1: Algorithm to Compute Local Runtime Bounds for Cycles

input : A program $(\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ and a simple cycle $\mathcal{C} = \{t_1, \ldots, t_n\} \subset \mathcal{T}$
output : A local runtime bound $\mathcal{RB}_{\text{loc}}$ for $\mathcal{C} = \mathcal{T}'_> = \mathcal{T}'$

1 Initialize $\mathcal{RB}_{\text{loc}}$: $\mathcal{RB}_{\text{loc}}(\to_r \mathcal{C}) = \omega$ for all $r \in \mathcal{E}_\mathcal{C}$.
2 **forall** $r \in \mathcal{E}_\mathcal{C}$ **do**
3 Let $i \in \{1, \ldots, n\}$ such that r's target location is the start location ℓ_i of t_i.
4 Let $t = t_i \star \ldots \star t_n \star t_1 \star \ldots \star t_{i-1}$.
5 **if** *there exists a renaming π of $\mathcal{PV}$ such that $\pi(t)$ results in a twn-loop* **then**
6 Set $\mathcal{RB}_{\text{loc}}(\to_r \mathcal{C}) \leftarrow \pi^{-1}(1 + \text{result of Thm. 29 on } \pi(t) \text{ and } \pi(r))$.
7 **return** local runtime bound $\mathcal{RB}_{\text{loc}}$.

next transition to be taken. Formally, $\mathcal{C} = \{t_1, \ldots, t_n\} \subset \mathcal{T}$ is a simple cycle if $\mathcal{C}$ does not contain temporary variables and there are pairwise different locations $\ell_1, \ldots, \ell_n$ such that $t_i = (\ell_i, \_, \_, \ell_{i+1})$ for $1 \leq i \leq n-1$ and $t_n = (\ell_n, \_, \_, \ell_1)$. This ensures that if there is an evaluation with $\to_{t_i} \circ \to^*_{\mathcal{C}\setminus\{t_i\}} \circ \to_{t_i}$, then the steps with $\to^*_{\mathcal{C}\setminus\{t_i\}}$ have the form $\to_{t_{i+1}} \circ \ldots \circ \to_{t_n} \circ \to_{t_1} \circ \ldots \circ \to_{t_{i-1}}$.

Alg. 1 describes how to compute a local runtime bound for a simple cycle $\mathcal{C} = \{t_1, \ldots, t_n\}$ as above. In the loop of Line 2, we iterate over all entry transitions r of $\mathcal{C}$. If r reaches the transition t_i, then in Lines 3 and 4 we chain $t_i \star \ldots \star t_n \star t_1 \star \ldots \star t_{i-1}$ which corresponds to one iteration of the cycle starting in t_i. If a suitable renaming (and thus also reordering) of the variables turns the chained transition into a twn-loop, then we use Thm. 29 to compute a local runtime bound $\mathcal{RB}_{\text{loc}}(\to_r \mathcal{C})$ in Lines 5 and 6. If the chained transition does not give rise to a twn-loop, then $\mathcal{RB}_{\text{loc}}(\to_r \mathcal{C})$ is ω (Line 1). In practice, to use the twn-technique for a transition t in a program, our tool KoAT searches for those simple cycles that contain t and where the chained cycle is a twn-loop. Among those cycles it chooses the one with the smallest runtime bounds for its entry transitions.

Theorem 31 (Correctness of Alg. 1). *Let $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ be an integer program and let $\mathcal{C} \subset \mathcal{T}$ be a simple cycle in $\mathcal{P}$. Then the result $\mathcal{RB}_{loc} : \mathcal{E}_\mathcal{C} \to \mathcal{B}$ of Alg. 1 is a local runtime bound for $\mathcal{C} = \mathcal{T}'_> = \mathcal{T}'$.*

Example 32. We apply Alg. 1 on the cycle $\mathcal{C} = \{t_{5a}, t_{5b}\}$ of the program in Fig. 2. $\mathcal{C}$'s entry transitions t_1 and t_4 both end in ℓ_3. Chaining t_{5a} and t_{5b} yields the transition t_5 of Fig. 1, i.e., $t_5 = t_{5a} \star t_{5b}$. Thus, Alg. 1 essentially transforms the program of Fig. 2 into Fig. 1. As in Ex. 28 and 30, we obtain $\mathcal{RB}_{\text{loc}}(\to_{t_4} \mathcal{C}) = 1 + (2 \cdot \text{sth}^{\sqcup}_{\texttt{true}} + 1) = 4 \cdot x_2 + 4 \cdot x_3^3 + 4 \cdot x_3^5 + 4$ and $\mathcal{RB}_{\text{loc}}(\to_{t_1} \mathcal{C}) = 1 + (2 \cdot \text{sth}^{\sqcup}_{x_3 > 0} + 1) = 4 \cdot x_2 + 4$, resulting in the global runtime bound $\mathcal{RB}_{\text{glo}}(t_{5a}) = \mathcal{RB}_{\text{glo}}(t_{5b}) = 8 \cdot x_4 \cdot x_5 + 13008 \cdot x_4$, which again yields $\text{rc}(\sigma_0) \in \mathcal{O}(n^2)$.

6 Conclusion and Evaluation

We showed that results on subclasses of programs with computable complexity bounds like [19] are not only theoretically interesting, but they have an important practical value. To our knowledge, our paper is the first to integrate such results into an incomplete approach for automated complexity analysis like [6, 18]. For

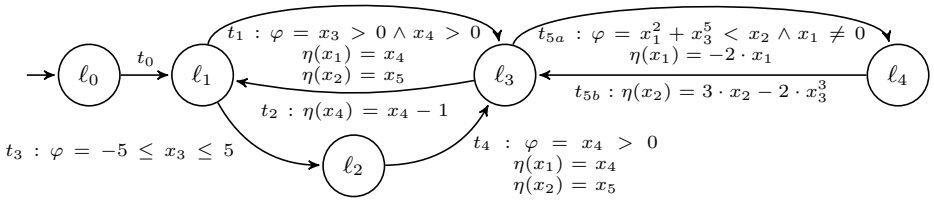

Fig. 2: An Integer Program with a Nested Non-Self-Loop

this integration, we developed several novel contributions which extend and improve the previous approaches in [6, 18, 19] substantially:

(a) We extended the concept of local runtime bounds such that they can now depend on entry transitions (Def. 9).
(b) We generalized the computation of global runtime bounds such that one can now lift arbitrary local bounds to global bounds (Thm. 11). In particular, the local bounds might be due to either ranking functions or twn-loops.
(c) We improved the technique for the computation of bounds on twn-loops such that these bounds now take the roles of the different variables into account (Def. 22, Cor. 23, and Thm. 27).
(d) We extended the notion of twn-loops by update-invariants and developed a new over-approximation of their closed forms which takes invariants into account (Def. 13 and 24, Lemma 26, and Thm. 27).
(e) We extended the handling of twn-loops to twn-cycles (Thm. 31).

The need for these improvements is demonstrated by our leading example in Fig. 1 (where the contributions (a) - (d) are needed to infer quadratic runtime complexity) and by the example in Fig. 2 (which illustrates (e)). In this way, the power of automated complexity analysis is increased substantially, because now one can also infer runtime bounds for programs containing non-linear arithmetic.

To demonstrate the power of our approach, we evaluated the integration of our new technique to infer local runtime bounds for twn-cycles in our re-implementation of the tool KoAT (written in OCaml) and compare the results to other state-of-the-art tools. To distinguish our re-implementation of KoAT from the original version of the tool from [6], let KoAT1 refer to the tool from [6] and let KoAT2 refer to our new re-implementation. KoAT2 applies a local control-flow refinement technique [18] (using the tool iRankFinder [8]) and preprocesses the program in the beginning, e.g., by extending the guards of transitions by invariants inferred using the Apron library [23]. For all occurring SMT problems, KoAT2 uses Z3 [28]. We tested the following configurations of KoAT2, which differ in the techniques used for the computation of local runtime bounds:

- KoAT2+RF only uses linear ranking functions to compute local runtime bounds
- KoAT2+MΦRF5 uses multiphase-linear ranking functions of depth ≤ 5
- KoAT2+TWN only uses twn-cycles to compute local runtime bounds (Alg. 1)
- KoAT2+TWN+RF uses Alg. 1 for twn-cycles and linear ranking functions
- KoAT2+TWN+MΦRF5 uses Alg. 1 for twn-cycles and MΦRFs of depth ≤ 5

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(EXP)$	$< \infty$	AVG$^+$(s)	AVG(s)
KoAT2 + TWN + MΦRF5	26	231 (5)	73 (5)	13 (4)	1 (1)	344 (15)	8.72	23.93
KoAT2 + TWN + RF	27	227 (5)	73 (5)	13 (4)	1 (1)	341 (15)	8.11	19.77
KoAT2 + MΦRF5	24	226 (1)	68	10	0	328 (1)	8.23	21.63
KoAT2 + RF	25	214 (1)	68	10	1	318 (1)	8.49	16.56
MaxCore	23	216 (2)	66	7	0	312 (2)	2.02	5.31
CoFloCo	22	196 (1)	66	5	0	289 (1)	0.62	2.66
KoAT1	25	169 (1)	74	12	6	286 (1)	1.77	2.77
Loopus	17	170 (1)	49	5 (1)	0	241 (2)	0.42	0.43
KoAT2 + TWN	20 (1)	111 (4)	3 (2)	2 (2)	0	136 (9)	2.54	26.59

Fig. 3: Evaluation on the Collection CINT$^+$

Existing approaches for automated complexity analysis are already very powerful on programs that only use linear arithmetic in their guards and updates. The corresponding benchmarks for *Complexity of Integer Transitions Systems* (CITS) and *Complexity of* C *Integer Programs* (CINT) from the *Termination Problems Data Base* [33] which is used in the annual *Termination and Complexity Competition (TermComp)* [17] contain almost only examples with linear arithmetic. Here, the existing tools already infer finite runtimes for more than 89 % of those examples in the collections CITS and CINT where this *might*[7] be possible.

The main benefit of our new integration of the twn-technique is that in this way one can also infer finite runtime bounds for programs that contain non-linear guards or updates. To demonstrate this, we extended both collections CITS and CINT by 20 examples that represent typical such programs, including several benchmarks from the literature [3, 14, 15, 18, 20, 34], as well as our programs from Fig. 1 and 2. See [27] for a detailed list and description of these examples.

Fig. 3 presents our evaluation on the collection CINT$^+$, consisting of the 484 examples from CINT and our 20 additional examples for non-linear arithmetic. We refer to [27] for the (similar) results on the corresponding collection CITS$^+$.

In the C programs of CINT$^+$, all variables are interpreted as integers over $\mathbb{Z}$ (i.e., without overflows). For KoAT2 and KoAT1, we used Clang [7] and llvm2kittel [10] to transform C programs into integer transitions systems as in Def. 2. We compare KoAT2 with KoAT1 [6] and the tools CoFloCo [11, 12], MaxCore [2] with CoFloCo in the backend, and Loopus [31]. We do not compare with RaML [21], as it does not support programs whose complexity depends on (possibly negative) integers (see [29]). We also do not compare with PUBS [1], because as stated in [9] by one of its authors, CoFloCo is stronger than PUBS. For the same reason, we only consider MaxCore with the backend CoFloCo instead of PUBS.

All tools were run inside an Ubuntu Docker container on a machine with an AMD Ryzen 7 3700X octa-core CPU and 48 GB of RAM. As in *TermComp*, we applied a timeout of 5 minutes for every program.

In Fig. 3, the first entry in every cell denotes the number of benchmarks from CINT$^+$ where the respective tool inferred the corresponding bound. The number

[7] The tool LoAT [13, 16] proves unbounded runtime for 217 of the 781 examples from CITS and iRankFinder [4, 8] proves non-termination for 118 of 484 programs of CINT.

in brackets is the corresponding number of benchmarks when only regarding our 20 new examples for non-linear arithmetic. The runtime bounds computed by the tools are compared asymptotically as functions which depend on the largest initial absolute value n of all program variables. So for instance, there are $26 + 231 = 257$ programs in CINT$^+$ (and 5 of them come from our new examples) where KoAT2+TWN+MΦRF5 can show that $\mathrm{rc}(\sigma_0) \in \mathcal{O}(n)$ holds for all initial states σ_0 where $|\sigma_0(v)| \leq n$ for all $v \in \mathcal{PV}$. For 26 of these programs, KoAT2+TWN+MΦRF5 can even show that $\mathrm{rc}(\sigma_0) \in \mathcal{O}(1)$, i.e., their runtime complexity is constant. Overall, this configuration succeeds on 344 examples, i.e., "$< \infty$" is the number of examples where a finite bound on the runtime complexity could be computed by the respective tool within the time limit. "AVG$^+$(s)" is the average runtime of the tool on successful runs in seconds, i.e., where the tool inferred a finite time bound before reaching the timeout, whereas "AVG(s)" is the average runtime of the tool on all runs including timeouts.

On the original benchmarks CINT where very few examples contain non-linear arithmetic, integrating TWN into a configuration that already uses multiphase-linear ranking functions does not increase power much: KoAT2+TWN+MΦRF5 succeeds on $344 - 15 = 329$ such programs and KoAT2+MΦRF5 solves $328 - 1 = 327$ examples. On the other hand, if one only has linear ranking functions, then an improvement via our twn-technique has similar effects as an improvement with multiphase-linear ranking functions (here, the success rate of KoAT2+MΦRF5 is similar to KoAT2+TWN+RF which solves $341 - 15 = 326$ such programs).

But the main benefit of our technique is that it also allows to successfully handle examples with non-linear arithmetic. Here, our new technique is significantly more powerful than previous ones. Other tools and configurations without TWN in Fig. 3 solve at most 2 of the 20 new examples. In contrast, KoAT2+TWN+RF and KoAT2+TWN+MΦRF5 both succeed on 15 of them.[8] In particular, our running examples from Fig. 1 and 2 and even isolated twn-loops like t_5 or $t_5 \star t_5$ from Ex. 14 and 17 can *only* be solved by KoAT2 with our twn-technique.

To summarize, our evaluations show that KoAT2 with the added twn-technique outperforms all other configurations and tools for automated complexity analysis on all considered benchmark sets (i.e., CINT$^+$, CINT, CITS$^+$, and CITS) and it is the only tool which is also powerful on examples with non-linear arithmetic.

KoAT's source code, a binary, and a Docker image are available at https://aprove-developers.github.io/KoAT_TWN/. The website also has details on our experiments and *web interfaces* to run KoAT's configurations directly online.

Acknowledgments We are indebted to M. Hark for many fruitful discussions about complexity, twn-loops, and KoAT. We are grateful to S. Genaim and J. J. Doménech for a suitable version of iRankFinder which we could use for control-flow refinement in KoAT's backend. Moreover, we thank A. Rubio and E. Martín-Martín for a static binary of MaxCore, A. Flores-Montoya and F. Zuleger for help in running CoFloCo and Loopus, F. Frohn for help and advice, and the reviewers for their feedback to improve the paper.

[8] One is the non-terminating leading example of [15], so at most 19 *might* terminate.

References

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

[2] E. Albert, M. Bofill, C. Borralleras, E. Martín-Martín, and A. Rubio. "Resource Analysis Driven by (Conditional) Termination Proofs". In: *Theory and Practice of Logic Programming* 19 (2019), pp. 722–739. DOI: 10.1017/S1471068419000152.

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

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

[5] M. Braverman. "Termination of Integer Linear Programs". In: *Proc. CAV*. LNCS 4144. 2006, pp. 372–385. DOI: 10.1007/11817963_34.

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

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

[8] J. J. Doménech and S. Genaim. "iRankFinder". In: *Proc. WST*. 2018, p. 83. URL: http://wst2018.webs.upv.es/wst2018proceedings.pdf.

[9] J. J. Doménech, J. P. Gallagher, and S. Genaim. "Control-Flow Refinement by Partial Evaluation, and Its Application to Termination and Cost Analysis". In: *Theory and Practice of Logic Programming* 19 (2019), pp. 990–1005. DOI: 10.1017/S1471068419000310.

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

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

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

[13] F. Frohn and J. Giesl. "Proving Non-Termination via Loop Acceleration". In: *Proc. FMCAD*. 2019, pp. 221–230. DOI: 10.23919/FMCAD.2019.8894271.

[14] F. Frohn and J. Giesl. "Termination of Triangular Integer Loops Is Decidable". In: *Proc. CAV*. LNCS 11562. 2019, pp. 426–444. DOI: 10.1007/978-3-030-25543-5_24.

[15] F. Frohn, M. Hark, and J. Giesl. "Termination of Polynomial Loops". In: *Proc. SAS*. LNCS 12389. Full version available at https://arxiv.org/abs/1910.11588. 2020, pp. 89–112. DOI: 10.1007/978-3-030-65474-0_5.

[16] F. Frohn, M. Naaf, M. Brockschmidt, and J. Giesl. "Inferring Lower Runtime Bounds for Integer Programs". In: *ACM Transactions on Programming Languages and Systems* 42 (2020). DOI: 10.1145/3410331.

[17] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. "The Termination and Complexity Competition". In: *Proc. TACAS*. LNCS 11429. 2019, pp. 156–166. DOI: 10.1007/978-3-030-17502-3_10.

[18] J. Giesl, N. Lommen, M. Hark, and F. Meyer. "Improving Automatic Complexity Analysis of Integer Programs". In: *The Logic of Software: A Tasting Menu of Formal Methods*. LNCS 13360. To appear. Also appeared in *CoRR*, abs/2202.01769. URL: https://arxiv.org/abs/2202.01769. 2022.

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

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

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

[22] M. Hosseini, J. Ouaknine, and J. Worrell. "Termination of Linear Loops Over the Integers". In: *Proc. ICALP*. LIPIcs 132. 2019. DOI: 10.4230/LIPIcs.ICALP.2019.118.

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

[24] Z. Kincaid, J. Breck, J. Cyphert, and T. W. Reps. "Closed Forms for Numerical Loops". In: *Proc. ACM Program. Lang.* 3.POPL (2019). DOI: 10.1145/3290368.

[25] L. Kovács. "Reasoning Algebraically About p-Solvable Loops". In: *Proc. TACAS*. LNCS 4963. 2008, pp. 249–264. DOI: 10.1007/978-3-540-78800-3_18.

[26] N. Lommen, F. Meyer, and J. Giesl. "Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops". In: *CoRR* abs/2205.08869 (2022). URL: https://arxiv.org/abs/2205.08869.

[27] N. Lommen, F. Meyer, and J. Giesl. Empirical Evaluation of: "Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops". URL: https://aprove-developers.github.io/KoAT_TWN/.

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

[29] RaML (Resource Aware ML). URL: https://www.raml.co/interface/.

[30] E. Rodríguez-Carbonell and D. Kapur. "Automatic Generation of Polynomial Loop Invariants: Algebraic Foundation". In: *Proc. ISSAC*. 2004, pp. 266–273. DOI: 10.1145/1005285.1005324.

[31] M. Sinn, F. Zuleger, and H. Veith. "Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints". In: *Journal of Automated Reasoning* 59 (2017), pp. 3–45. DOI: 10.1007/s10817-016-9402-4.

[32] A. Tiwari. "Termination of Linear Programs". In: *Proc. CAV*. LNCS 3114. 2004, pp. 70–82. DOI: 10.1007/978-3-540-27813-9_6.

[33] TPDB (Termination Problems Data Base). URL: https://github.com/TermCOMP/TPDB.

[34] M. Xu and Z.-B. Li. "Symbolic Termination Analysis of Solvable Loops". In: *Journal of Symbolic Computation* 50 (2013), pp. 28–49. DOI: 10.1016/j.jsc.2012.05.005.