Targeting Completeness: Using Closed Forms for Size Bounds of Integer Programs*

Nils Lommen and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract. We present a new procedure to infer *size bounds* for integer programs automatically. Size bounds are important for the deduction of bounds on the runtime complexity or in general, for the resource analysis of programs. We show that our technique is *complete* (i.e., it always computes finite size bounds) for a subclass of loops, possibly with non-linear arithmetic. Moreover, we present a novel approach to combine and integrate this complete technique into an incomplete approach to infer size and runtime bounds of general integer programs. We prove completeness of our integration for an important subclass of integer programs. We implemented our new algorithm in the automated complexity analysis tool KoAT to evaluate its power, in particular on programs with non-linear arithmetic.

1 Introduction

There are numerous incomplete approaches for automatic resource analysis of programs, e.g., [1, 2, 5, 8, 10, 15, 19, 21, 29, 33]. However, also many complete techniques to decide termination, analyze runtime complexity, or study memory consumption for certain classes of programs have been developed, e.g., [3, 4, 6, 7, 16, 17, 20, 22, 27, 34, 36]. In this paper, we present a procedure to compute *size bounds* which indicate how large the absolute value of an integer variable may become. In contrast to other complete procedures for the inference of size bounds which are based on fixpoint computations [3, 6], our technique can also handle (possibly negative) constants and exponential size bounds. Similar to our earlier paper [27], we embed a procedure which is *complete* for a subclass of loops (i.e., it computes finite size bounds for all loops from this subclass) into an incomplete approach for general integer programs [8, 19]. In this way, the power of the incomplete approach is increased significantly, in particular for programs with non-linear arithmetic. However, in the current paper we tackle a completely different problem than in [27] (and thus, the actual new contributions are also completely different), because in [27] we embedded a complete technique in order to infer runtime bounds, whereas now we integrate a novel technique in order to infer size bounds. As an example, we want to determine bounds on the absolute values of the variables during (and after) the execution of the following loop.

$$\textbf{while } (x_3 > 0) \textbf{ do } (x_1, x_2, x_3, x_4) \leftarrow (3{\cdot}x_1{+}2{\cdot}x_2, -5{\cdot}x_1{-}3{\cdot}x_2, x_3{-}1, x_4{+}x_3^2) \quad (1)$$

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

We introduce a technique to compute size bounds for loops which admit a closed form, i.e., an expression which corresponds to applying the loop's update n times. Then we over-approximate the closed form to obtain a non-negative, weakly monotonically increasing function. For instance, a closed form for x_3 in our example is $x_3 - n$, since the value of x_3 is decreased by n after n iterations. The (absolute value of this) closed form can be over-approximated by $x_3 + n$, which is monotonically increasing in all variables. Finally, each occurrence of n is substituted by a runtime bound for the loop. Clearly, (1) terminates after at most x_3 iterations. So if we substitute n by the runtime bound x_3 in the over-approximated closed form x_3+n, then we infer the linear bound $2\cdot x_3$ on the size of x_3. Due to the restriction to weakly monotonically increasing over-approximations, we can plug in any over-approximation of the runtime and do not necessarily need exact bounds.

Structure We introduce our technique to compute size bounds by closed forms in Sect. 2 and show that it is complete for a subclass of loops in Sect. 3. Afterwards in Sect. 4, we incorporate our novel technique into the incomplete setting of general integer programs. In Sect. 5 we demonstrate how size bounds are used in automatic complexity analysis and study completeness for classes of general programs. In Sect. 6, we conclude with an experimental evaluation of our implementation in the tool KoAT and discuss related work. All proofs can be found in [28].

2 Size Bounds by Closed Forms

In this section, we present our novel technique to compute size bounds for loops by closed forms in Thm. 7. We start by introducing the required preliminaries. Let $\mathcal{V} = \{x_1, \ldots, x_d\}$ be a set of variables. $\mathcal{F}(\mathcal{V})$ is the set of all *formulas* built from inequations $p > 0$ for polynomials $p \in \mathbb{Q}[\mathcal{V}]$, $\wedge$, and $\vee$. A *loop* (φ, η) consists of a guard $\varphi \in \mathcal{F}(\mathcal{V})$ and an update $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$ mapping variables to polynomials. A *closed form* $\mathtt{cl}^{x_i}$ (formally defined in Def. 1 below) is an expression in n and in the (initial values of the) variables $x_1, \ldots, x_d$ which corresponds to the value of x_i after iterating the loop n times. For our purpose we only need closed forms which hold for all $n \geq n_0$ for some fixed $n_0 \in \mathbb{N}$. Moreover, we restrict ourselves to closed forms which are so-called normalized poly-exponential expressions [16]. Nonetheless, our procedure works for any closed form expression with a finite number of arithmetic operations (i.e., the number of operations must be independent of n). We extend the application of functions like $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$ also to polynomials, vectors, and formulas, etc., by replacing each variable v in the expression by $\eta(v)$. So in particular, $(\eta_2 \circ \eta_1)(x) = \eta_2(\eta_1(x))$ stands for the polynomial $\eta_1(x)$ in which every variable v is replaced by $\eta_2(v)$. Moreover, η^n denotes the n-fold application of η.

We call a function $\sigma : \mathcal{V} \to \mathbb{Z}$ a *state*. By $\sigma(exp)$ or $\sigma(\varphi)$ we denote the number resp. Boolean value which results from replacing every variable v by the number $\sigma(v)$ in the arithmetic expression exp or the formula φ.

Definition 1 (Closed Forms). *For a loop (φ, η), an arithmetic expression $\mathtt{cl}^{x_i}$ is a* closed form *for x_i with* start value $n_0 \in \mathbb{N}$ *if* $\mathtt{cl}^{x_i} = \sum_{1 \leq j \leq \ell} \alpha_j \cdot n^{a_j} \cdot b_j^n$

with $\ell, a_j \in \mathbb{N}$, $b_j \in \mathbb{A}$,[1] $\alpha_j \in \mathbb{A}[\mathcal{V}]$, *and for all* $\sigma : \mathcal{V} \cup \{n\} \to \mathbb{Z}$ *with* $\sigma(n) \geq n_0$ *we have* $\sigma(\mathtt{cl}^{x_i}) = \sigma(\eta^n(x_i))$. *Similarly, we call* $\mathtt{cl} = (\mathtt{cl}^{x_1}, \ldots, \mathtt{cl}^{x_d})$ *a* closed form *of the update* η *(resp. for the loop* (φ, η)*) with start value* n_0 *if for all* $1 \leq i \leq d$, $\mathtt{cl}^{x_i}$ *are closed forms for* x_i *with start value* n_0.

Example 2. In Sect. 3 we will show that for the loop (1), a closed form for x_1 (with start value 0) is $\mathtt{cl}^{x_1} = \frac{1}{2} \cdot \alpha \cdot (-\mathrm{i})^n + \frac{1}{2} \cdot \overline{\alpha} \cdot \mathrm{i}^n$ where $\alpha = (1 + 3\mathrm{i}) \cdot x_1 + 2\mathrm{i} \cdot x_2$. Here, $\overline{\alpha}$ denotes the complex conjugate of α, i.e., the sign of those monomials is flipped where the coefficient is a multiple of the imaginary unit i. A closed form for x_4 (also with start value 0) is $\mathtt{cl}^{x_4} = x_4 + n \cdot (\frac{1}{6} + x_3 + x_3^2 - x_3 \cdot n - \frac{n}{2} + \frac{n^2}{3})$.

Our aim is to compute *bounds* on the sizes of variables and on the runtime. As in [8, 19], we only consider bounds which are weakly monotonically increasing in all occurring variables. Their advantage is that we can compose them easily (i.e., if f and g increase monotonically, then so does $f \circ g$).

Definition 3 (Bounds). *The set of* bounds $\mathcal{B}$ *is the smallest set with* $\overline{\mathbb{N}} = \mathbb{N} \cup \{\omega\} \subseteq \mathcal{B}$, $\mathcal{V} \subseteq \mathcal{B}$, *and* $\{b_1 + b_2,\ b_1 \cdot b_2,\ k^{b_1}\} \subseteq \mathcal{B}$ *for all* $k \in \mathbb{N}$ *and* $b_1, b_2 \in \mathcal{B}$.

Size bounds should be bounds on the values of variables up to the point where the loop guard is not satisfied anymore for the first time. To define size bounds, we introduce the *runtime complexity* of a loop (whereas we considered the runtime complexity of arbitrary integer programs in [8, 19, 27]). Let Σ denote the set of all states $\sigma : \mathcal{V} \to \mathbb{Z}$ and let $|\sigma|$ be the state with $|\sigma|(x) = |\sigma(x)|$ for all $x \in \mathcal{V}$.

Definition 4 (Runtime Complexity for Loops). *The* runtime complexity *of a loop* (φ, η) *is* $\mathrm{rc} : \Sigma \to \overline{\mathbb{N}}$ *with* $\mathrm{rc}(\sigma) = \inf\{n \in \mathbb{N} \mid \sigma(\eta^n(\neg\varphi))\}$, *where* $\inf \varnothing = \omega$. *An expression* $r \in \mathcal{B}$ *is a* runtime bound *if* $|\sigma|(r) \geq \mathrm{rc}(\sigma)$ *for all* $\sigma \in \Sigma$.

Example 5. The runtime complexity of the loop (1) is $\mathrm{rc}(\sigma) = \max(0, \sigma(x_3))$. For example, x_3 is a runtime bound, as $|\sigma|(x_3) \geq \max(0, \sigma(x_3))$ for all states $\sigma \in \Sigma$.

A *size bound* on a variable x is a bound on the absolute value of x after n iterations of the update η, where n is bounded by the runtime complexity. In contrast to the definition of size bounds for transitions in integer programs from [8], Def. 6 requires that size bounds also hold *before* evaluating the loop.

Definition 6 (Size Bounds for Loops). $\mathcal{SB} : \mathcal{V} \to \mathcal{B}$ *is a* size bound *for* (φ, η) *if for all* $x \in \mathcal{V}$ *and all* $\sigma \in \Sigma$, *we have* $|\sigma|(\mathcal{SB}(x)) \geq \sup\{|\sigma(\eta^n(x))| \mid n \leq \mathrm{rc}(\sigma)\}$.

For any algebraic number $c \in \mathbb{A}$, as usual $\lceil |c| \rceil$ is the smallest natural number which is greater or equal to c's absolute value. Similarly, for any poly-exponential expression $p = \sum_j (\sum_i c_{i,j} \cdot \beta_{i,j}) \cdot n^{a_j} \cdot b_j^n$ where $c_{i,j} \in \mathbb{A}$ and the $\beta_{i,j}$ are normalized monomials of the form $x_1^{e_1} \cdot \ldots \cdot x_d^{e_d}$, $\lceil |p| \rceil$ denotes $\sum_j (\sum_i \lceil |c_{i,j}| \rceil \cdot \beta_{i,j}) \cdot n^{a_j} \cdot \lceil |b_j| \rceil^n$.

We now determine size bounds by over-approximating the closed form $\mathtt{cl}^x$ by the non-negative, weakly monotonically increasing function $\lceil |\mathtt{cl}^x| \rceil$. Then we substitute n by a runtime bound r (denoted by "$[n/r]$"). Due to the monotonicity,

[1] $\mathbb{A}$ is the set of algebraic numbers, i.e., the field of all roots of polynomials in $\mathbb{Z}[x]$.

this results in a bound on the size of x not only at the end of the loop, but also during the iterations of the loop. Since the closed form is only valid for n iterations with $n \geq n_0$, we ensure that our size bound is also correct for less than n_0 iterations by symbolically evaluating the update, where we over-approximate maxima by sums. As mentioned, see [28] for the proofs of all new results.

Theorem 7 (Size Bounds for Loops with Closed Forms). *Let* $\mathtt{cl}$ *be a closed form for the loop* (φ, η) *with start value* n_0 *and let* $r \in \mathcal{B}$ *be a runtime bound. Then the (absolute) size of* $x \in \mathcal{V}$ *is bounded by* $\mathtt{sb}^x = \lceil |\mathtt{cl}^x| \rceil [n/r] + \sum_{0 \leq i < n_0} |\eta^i(x)|$. *Hence, the function* $\mathcal{SB}$ *with* $\mathcal{SB}(x) = \mathtt{sb}^x$ *for all* $x \in \mathcal{V}$ *is a size bound for* (φ, η).

Example 8. As mentioned, for the loop (1), a closed form for x_1 with start value 0 is $\mathtt{cl}^{x_1} = \frac{1}{2} \cdot \alpha \cdot (-\mathrm{i})^n + \frac{1}{2} \cdot \overline{\alpha} \cdot \mathrm{i}^n$ where $\alpha = (1+3\mathrm{i}) \cdot x_1 + 2\mathrm{i} \cdot x_2$. Hence, $\lceil |\mathtt{cl}^{x_1}| \rceil = \left\lceil \left| \frac{1}{2} \cdot \alpha \cdot (-\mathrm{i})^n + \frac{1}{2} \cdot \overline{\alpha} \cdot \mathrm{i}^n \right| \right\rceil = (\lceil |\frac{1+3\mathrm{i}}{2}| \rceil \cdot x_1 + \lceil |\mathrm{i}| \rceil \cdot x_2) \cdot \lceil |-\mathrm{i}| \rceil^n + (\lceil |\frac{1-3\mathrm{i}}{2}| \rceil \cdot x_1 + \lceil |-\mathrm{i}| \rceil \cdot x_2) \cdot \lceil |\mathrm{i}| \rceil^n = 4 \cdot x_1 + 2 \cdot x_2$, as $\lceil |\frac{1+3\mathrm{i}}{2}| \rceil = \lceil |\frac{1-3\mathrm{i}}{2}| \rceil = \left\lceil \frac{\sqrt{10}}{2} \right\rceil = 2$ and $\lceil |\mathrm{i}| \rceil = \lceil |-\mathrm{i}| \rceil = 1$. So our approach infers *linear* size bounds for x_1 and x_2 (the similar computations for x_2 are omitted) while [8] only infers exponential size bounds.

As this over-approximation does not depend on n, it directly yields a size bound, i.e., $\mathtt{sb}^{x_1} = \lceil |\mathtt{cl}^{x_1}| \rceil$. In contrast, in the over-approximation $\lceil |\mathtt{cl}^{x_4}| \rceil = x_4 + n\left(1 + x_3 + x_3^2 + x_3 \cdot n + n + n^2\right)$, we have to replace n by a runtime bound like x_3. Thus, we obtain the overall size bound $\mathtt{sb}^{x_4} = x_4 + 3 \cdot x_3^3 + 2 \cdot x_3^2 + x_3$.

Although this section focused on closed forms which are poly-exponential expressions, our technique is applicable to all loops where we can compute over-approximating bounds for the closed form and the runtime complexity. For example, the update $\eta(x) = x^2$ has the closed form $x^{(2^n)}$, but it does not admit a poly-exponential closed form due to x's super-exponential growth. However, by instantiating n by a runtime bound, we can still compute a size bound for this update. The reason for focusing on poly-exponential expressions is that we can compute such a closed form for all so-called *solvable loops* automatically, see Sect. 3.

3 Size and Runtime Bounds for Solvable Loops

In this section, we present a class of loops where our technique of Thm. 7 is "complete". The technique relies on the computation of suitable closed forms and of runtime bounds. In Sect. 3.1, we show that poly-exponential closed forms can be computed for all *solvable loops* [17, 23, 25, 26, 32, 36]. Then we prove in Sect. 3.2 that finite runtime bounds are computable for all terminating solvable loops with only periodic rational eigenvalues.

A loop (φ, η) is *solvable* if η is a *solvable update* (see Def. 9 below for a formal definition), which partitions $\mathcal{V}$ into blocks $\mathcal{S}_1, \ldots, \mathcal{S}_m$ (and loop guards φ are not relevant for closed forms). Each block allows updates with *cyclic dependencies* between its variables and *non-linear* dependencies on variables in blocks with lower indices.

Definition 9 (Solvable Update [17, 23, 25, 26, 32, 36]). *An update* $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$ *is* solvable *if there exists a partition* $\mathcal{S}_1, \ldots, \mathcal{S}_m$ *of* $\{x_1, \ldots, x_d\}$ *such that for all* $1 \leq i \leq m$ *we have* $\vec{\eta}_{\mathcal{S}_i} = A_{\mathcal{S}_i} \cdot \vec{x}_{\mathcal{S}_i} + \vec{p}_{\mathcal{S}_i}$ *for an* $A_{\mathcal{S}_i} \in \mathbb{Z}^{|\mathcal{S}_i| \times |\mathcal{S}_i|}$ *and a* $\vec{p}_{\mathcal{S}_i} \in \mathbb{Z}[\bigcup_{j<i} \mathcal{S}_j]^{|\mathcal{S}_i|}$*, where* $\vec{\eta}_{\mathcal{S}_i}$ *is the vector of all* $\eta(x_j)$ *and* $\vec{x}_{\mathcal{S}_i}$ *is the vector of all* x_j *with* $j \in \mathcal{S}_i$*. The eigenvalues of a solvable loop are defined as the union of the eigenvalues of all matrices* $A_{\mathcal{S}_i}$*. The loop is* homogeneous *if* $\vec{p}_{\mathcal{S}_i} = \vec{0}$ *for all* $1 \leq i \leq m$*.*

Example 10. The loop (1) is an example for a solvable loop using the partition $\mathcal{S}_1 = \{x_1, x_2\}$, $\mathcal{S}_2 = \{x_3\}$, and $\mathcal{S}_3 = \{x_4\}$.

The crucial idea for our results in Sect. 3.1 and 3.2 is to reduce the problem of finding closed forms and runtime bounds from solvable loops to *triangular weakly non-linear* loops (*twn-loops*) [16, 17, 20]. A *twn-update* is a solvable update where each block $\mathcal{S}_j$ has cardinality one. Thus, a twn-update is *triangular*, i.e., the update of a variable does not depend on variables with higher indices. Furthermore, the update is *weakly non-linear*, i.e., a variable does not occur non-linear in its own update. We are mainly interested in loops over $\mathbb{Z}$, but to handle solvable updates, we will transform them into twn-updates with coefficients from $\mathbb{A}$.

Definition 11 (TWN-Update [16, 17, 20]). *An update* $\eta : \mathcal{V} \to \mathbb{A}[\mathcal{V}]$ *is* twn *if for all* $1 \leq i \leq d$ *we have* $\eta(x_i) = c_i \cdot x_i + p_i$ *for some* $c_i \in \mathbb{A}$ *and some polynomial* $p_i \in \mathbb{A}[x_1, \ldots, x_{i-1}]$*. A loop with a twn-update is called a* twn-loop*.*

Clearly, (1) is not a twn-loop due to the cyclic dependency between x_1 and x_2.

3.1 Closed Forms for Solvable Loops

Lemma 12 (which extends [17, Thm. 16] from solvable updates with real eigenvalues to arbitrary solvable updates) illustrates that one can transform any solvable update η_s into a twn-update η_t by an automorphism ϑ. Here, ϑ is induced by the change-of-basis matrix of the Jordan normal form of each block of η_s. Note that the Jordan normal form is always computable in polynomial time (see [9]).

Lemma 12 (Transforming Solvable Updates (see [17, Thm. 16])). *Let* η_s *be a solvable update. Then* $\vartheta : \mathcal{V} \to \mathbb{A}[\mathcal{V}]$ *is an automorphism, where* ϑ *is defined by* $\vartheta(\mathcal{S}) = P \cdot \vec{x}_{\mathcal{S}}$ *for each block* $\mathcal{S}$*, where* $J(A_{\mathcal{S}}) = P \cdot A_{\mathcal{S}} \cdot P^{-1}$ *is the Jordan normal form of* $A_{\mathcal{S}}$*. Furthermore,* $\eta_t = \vartheta^{-1} \circ \eta_s \circ \vartheta$ *is a twn-update.*

Example 13. To illustrate Lemma 12, we transform the solvable update η_s of (1) into a twn-update η_t. As the blocks $\mathcal{S}_2 = \{x_3\}$ and $\mathcal{S}_3 = \{x_4\}$ have cardinality one, we only have to consider $\mathcal{S}_1 = \{x_1, x_2\}$. The restriction of η_s to $\mathcal{S}_1$ is $\binom{x_1}{x_2} \leftarrow A_{\mathcal{S}_1} \cdot \binom{x_1}{x_2}$ with $A_{\mathcal{S}_1} = \left(\begin{smallmatrix} 3 & 2 \\ -5 & -3 \end{smallmatrix}\right)$. So we get the Jordan normal form $J(A_{\mathcal{S}_1}) = P \cdot A_{\mathcal{S}_1} \cdot P^{-1} = \left(\begin{smallmatrix} -\mathrm{i} & 0 \\ 0 & \mathrm{i} \end{smallmatrix}\right)$ where $P = \begin{pmatrix} -\frac{5}{2}\mathrm{i} & \frac{1}{2}(1-3\mathrm{i}) \\ \frac{5}{2}\mathrm{i} & \frac{1}{2}(1+3\mathrm{i}) \end{pmatrix}$ and $P^{-1} = \begin{pmatrix} \frac{1}{5}(\mathrm{i}-3) & -\frac{1}{5}(\mathrm{i}+3) \\ 1 & 1 \end{pmatrix}$. Thus, we have the following automorphism ϑ and its inverse ϑ^{-1}:

$$\vartheta\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = P \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} -\frac{5}{2}\mathrm{i} \cdot x_1 + \frac{1}{2}(1-3\mathrm{i}) \cdot x_2 \\ \frac{5}{2}\mathrm{i} \cdot x_1 + \frac{1}{2}(1+3\mathrm{i}) \cdot x_2 \end{pmatrix}, \qquad \vartheta\begin{pmatrix} x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \end{pmatrix}$$

$$\vartheta^{-1}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = P^{-1} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \frac{1}{5}(\mathrm{i}-3) \cdot x_1 - \frac{1}{5}(\mathrm{i}+3) \cdot x_2 \\ x_1 + x_2 \end{pmatrix}, \qquad \vartheta^{-1}\begin{pmatrix} x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \end{pmatrix}$$

Hence, $\eta_t = \vartheta^{-1} \circ \eta_s \circ \vartheta$ is the following twn-update:

$$\eta_t(x_1) = -\mathrm{i} \cdot x_1, \quad \eta_t(x_2) = \mathrm{i} \cdot x_2, \quad \eta_t(x_3) = x_3 - 1, \quad \eta_t(x_4) = x_4 + x_3^2$$

The reason for transforming solvable updates to twn-updates is that for the latter, we can re-use our previous algorithm from [16] to compute poly-exponential closed forms. While [16] only considered updates with linear arithmetic over $\mathbb{Z}$, it can directly be extended to twn-updates over $\mathbb{A}$.

Lemma 14 (Closed Forms for TWN-Updates (see [16])). *Let η be a twn-update. Then a (poly-exponential) closed form is computable for η.*

Example 15. For η_t from Ex. 13, we obtain the following closed form (with start value 0): $\mathtt{cl}_t = ((-\mathrm{i})^n \cdot x_1, \mathrm{i}^n \cdot x_2, x_3 - n, x_4 + n(\frac{1}{6} + x_3 + x_3^2 - x_3 \cdot n - \frac{n}{2} + \frac{n^2}{3}))$.

So to obtain a closed form of a solvable update η_s, we first transform it into a twn-update η_t via Lemma 12, and then compute the closed form $\mathtt{cl}_t$ of η_t (Lemma 14). We now show how to obtain a closed form for η_s from $\mathtt{cl}_t$.

Theorem 16 (Closed Forms for Solvable Updates). *Let η_s be a solvable update and ϑ be an automorphism as in Lemma 12 such that $\eta_t = \vartheta^{-1} \circ \eta_s \circ \vartheta$ is a twn-update. If $\mathtt{cl}_t$ is a closed form of η_t with start value n_0, then $\mathtt{cl}_s = \vartheta \circ \mathtt{cl}_t \circ \vartheta^{-1}$ is a closed form of η_s with start value n_0.*

Example 17. In Ex. 13 we transformed η_s into the twn-update η_t via an automorphism ϑ and in Ex. 15, we gave a closed form $\mathtt{cl}_t$ of η_t. Thus, by Thm. 16, we can infer a closed form $\mathtt{cl}_s = \vartheta \circ \mathtt{cl}_t \circ \vartheta^{-1}$ of η_s. For example, we compute a closed form for x_1 with start value 0 ($\mathtt{cl}_s^{x_2}$ can be inferred in a similar way):

$$\begin{aligned}
\mathtt{cl}_s^{x_1} &= \left(\tfrac{1}{5}(\mathrm{i}-3) \cdot x_1 - \tfrac{1}{5}(\mathrm{i}+3) \cdot x_2\right) \ [v/\mathtt{cl}_t^v \mid v \in \mathcal{V}] \ [v/\vartheta(v) \mid v \in \mathcal{V}] \\
&= \left(\tfrac{1}{5}(\mathrm{i}-3) \cdot (-\mathrm{i})^n \cdot x_1 - \tfrac{1}{5}(\mathrm{i}+3) \cdot \mathrm{i}^n \cdot x_2\right) \ [v/\vartheta(v) \mid v \in \mathcal{V}] \\
&= \tfrac{1}{2}(\underbrace{(1+3\mathrm{i}) \cdot x_1 + 2\mathrm{i} \cdot x_2}_{\alpha}) \cdot (-\mathrm{i})^n + \tfrac{1}{2}(\underbrace{(1-3\mathrm{i}) \cdot x_1 - 2\mathrm{i} \cdot x_2}_{\overline{\alpha}}) \cdot \mathrm{i}^n.
\end{aligned}$$

3.2 Periodic Rational Solvable Loops

In Sect. 3.1, we discussed how to compute closed forms for solvable updates (by transforming them to twn-updates). However to compute size bounds, we have to instantiate the variable n in the closed forms by runtime bounds (Thm. 7). In [20], it was shown that (polynomial) runtime bounds can always be computed for terminating twn-loops over the integers. However, in general, transforming solvable loops via Lemma 12 yields twn-updates which may contain algebraic (complex) numbers. We now show that for the subclass of terminating *periodic rational* solvable loops, our approach is "complete" (i.e., finite runtime bounds and thus, also finite size bounds are always computable).

Definition 18 (Periodic Rational [25]). *A number $\lambda \in \mathbb{A}$ is* periodic rational *if $\lambda^p \in \mathbb{Q}$ for some $p \in \mathbb{N}$ with $p > 0$. The* period *of λ is the smallest such p with $\lambda^p \in \mathbb{Q}$. A solvable loop is* periodic rational *(i.e., it is a* prs loop*) with period p if all its eigenvalues λ are periodic rational and p is the least common multiple of all their periods. A prs loop is a* unit *prs loop if $|\lambda| \leq 1$ for all its eigenvalues λ.*

So i, $-$i, and $\sqrt{2}\cdot$i are periodic rational with period 2, while $\sqrt{2}+$i is not periodic rational. The following lemma from [25] gives a bound on the period of prs loops and thus yields an algorithm to detect prs loops and to compute their period.

Lemma 19 (Bound on the Period [25]). *Let $A \in \mathbb{Z}^{n\times n}$. If λ is a periodic rational eigenvalue of A with period p, then $p \leq n^3$.*

Now we show that by *chaining* (i.e., by performing p iterations of a prs loop with period p in a single step), one can transform any prs loop into a solvable loop with only integer eigenvalues. Then, our previous results on twn-loops [17, 20] can be used to infer runtime bounds for these loops.

Definition 20 (Chaining Loops). *Let $L = (\varphi, \eta)$ be a loop and $p \in \mathbb{N} \setminus \{0\}$. Then $L_p = (\varphi_p, \eta_p)$ results from iterating L p times, i.e., $\varphi_p = \varphi \wedge \eta(\varphi) \wedge \eta(\eta(\varphi)) \wedge \ldots \wedge \eta^{p-1}(\varphi)$ and $\eta_p(v) = \eta^p(v)$ for all $v \in \mathcal{V}$.*

Example 21. The eigenvalues $\pm$i of (1) have period 2. Chaining yields $(\varphi \wedge \eta(\varphi), \eta^2)$:

$$\textbf{while } (x_3 > 0 \wedge x_3 > 1) \textbf{ do } (x_1, x_2, x_3, x_4) \leftarrow (-x_1, -x_2, x_3 - 2, x_4 + (x_3 - 1)^2 + x_3^2) \quad (2)$$

Due to Lemma 12 we can transform every solvable update into a twn-update by a (linear) automorphism ϑ. For prs loops, ϑ's range can be restricted to $\mathbb{Q}[\mathcal{V}]$, i.e., one does not need algebraic numbers. So, we first chain the prs loop L and then compute a $\mathbb{Q}$-automorphism ϑ transforming the chained loop L_p into a twn-loop L_t via Lemma 12. Then we can infer a runtime bound for L_t as in [20]. The reason is that all factors c_i in the update of L_t are integers and thus, we can compute a closed form $\sum_j \alpha_j \cdot n^{a_j} \cdot b_j^n$ such that $\alpha_j \in \mathbb{Q}[\mathcal{V}]$ and $b_j \in \mathbb{Z}$. Afterwards, the runtime bound for L_t can be lifted to a runtime bound for the original loop by reconsidering the automorphism ϑ. Similarly, in order to prove termination of the prs loop L, we analyze termination of L_t on $\vartheta(\mathbb{Z}^d) = \{\vartheta(\vec{x}) \mid \vec{x} \in \mathbb{Z}^d\}$.[2]

Lemma 22 (Runtime Bounds for PRS Loops). *Let L be a prs loop with period p and let $L_p = (\varphi_p, \eta_p)$ result from chaining as in Def. 20. From η_p, one can compute a linear automorphism $\vartheta : \mathcal{V} \to \mathbb{Q}[\mathcal{V}]$ as in Lemma 12, such that:*

(a) L_p is solvable and only has integer eigenvalues.
(b) $(\vartheta^{-1} \circ \eta_p \circ \vartheta) : \mathcal{V} \to \mathbb{Q}[\mathcal{V}]$ is a twn-update as in Def. 11 such that all $c_i \in \mathbb{Z}$.
(c) $L_t = (\varphi_t, \eta_t)$ with $\varphi_t = \vartheta^{-1}(\varphi_p)$ and $\eta_t = \vartheta^{-1} \circ \eta_p \circ \vartheta$ is a twn-loop. Moreover, the following holds:
- *L terminates on $\mathbb{Z}^d$ iff*
- *L_p terminates on $\mathbb{Z}^d$ iff*
- *L_t terminates on $\vartheta(\mathbb{Z}^d) = \{\vartheta(\vec{x}) \mid \vec{x} \in \mathbb{Z}^d\}$.*

(d) If r is a runtime bound[3] for L_t, then $p \cdot \lceil |\vartheta(r)| \rceil + p - 1$ is a runtime bound for L.

[2] By [17], termination of L_t on $\vartheta(\mathbb{Z}^d)$ is reducible to invalidity of a formula $\exists \vec{x} \in \mathbb{Q}^d. \psi_{\vartheta(\mathbb{Z}^d)} \wedge \xi_{L_t}$. Here, $\psi_{\vartheta(\mathbb{Z}^d)}$ holds iff $\vec{x} \in \vartheta(\mathbb{Z}^d)$ and ξ_{L_t} holds iff L_t does not terminate on $\vec{x}$. As shown in [17], non-termination of linear twn-loops with integer eigenvalues is NP-complete and it is semi-decidable for twn-loops with non-linear arithmetic.

[3] More precisely, $|\sigma|(r) \geq \inf\{n \in \mathbb{N} \mid \sigma(\eta_t^n(\neg\varphi_t))\}$ must hold for all $\sigma : \mathcal{V} \to \vartheta(\mathbb{Z}^d)$.

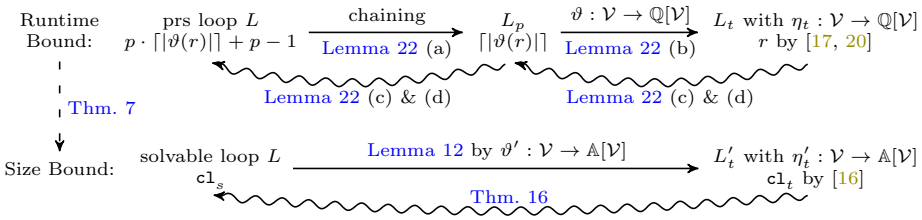

Fig. 1: Illustration of Runtime and Size Bound Computations

Since we can detect prs loops and their periods by Lemma 19, Lemma 22 allows us to compute runtime bounds for all terminating prs loops. This is illustrated in Fig. 1: For runtime bounds, L is transformed to L_p by chaining and L_p is transformed further to L_t by an automorphism ϑ. The runtime bound r for L_t can then be transformed into a runtime bound for L_p and further into a runtime bound for L. For size bounds, L is directly transformed to a twn-loop L'_t by an automorphism ϑ'. The closed form $\mathtt{cl}_t$ obtained for L'_t is transformed via the automorphism ϑ' into a closed form $\mathtt{cl}_s$ for L. Then the runtime bound for L is inserted into this closed form to yield a size bound for L. So in Fig. 1, standard arrows denote transformations of loops and wavy arrows denote transformations of runtime bounds or closed forms.

Theorem 23 (Completeness of Size and Runtime Bound Computation for Terminating PRS Loops). *For all terminating prs loops, polynomial runtime bounds and finite size bounds are computable. For terminating unit prs loops, all these size bounds are polynomial as well.*

Example 24. For the loop L from (1), we computed L_p for $p = 2$ in (2), see Ex. 21. As L_p is already a twn-loop, we can use the technique of [20] (implemented in our tool KoAT) to obtain the runtime bound x_3 for L_p. Lemma 22 yields the runtime bound $2 \cdot x_3 + 1$ for the original loop (1). Of course, here one could also use (incomplete) approaches based on linear ranking functions (also implemented in KoAT, see, e.g., [8, 19]) to directly infer the tighter runtime bound x_3 for the loop (1).

4 Size Bounds for Integer Programs

Up to now, we focused on *isolated* loops. In the following, we incorporate our complete approach from Sect. 2 and 3 into the setting of general *integer programs* where most questions regarding termination or complexity are undecidable. Formally, an integer program is a tuple $(\mathcal{V}, \mathcal{L}, \ell_0, \mathcal{T})$ with a finite set of variables $\mathcal{V}$, a finite set of locations $\mathcal{L}$, a fixed initial location $\ell_0 \in \mathcal{L}$, and a finite set of transitions $\mathcal{T}$. A *transition* is a 4-tuple $(\ell, \varphi, \eta, \ell')$ with a *start location* $\ell \in \mathcal{L}$, *target location* $\ell' \in \mathcal{L} \backslash \{\ell_0\}$, *guard* $\varphi \in \mathcal{F}(\mathcal{V})$, and *update* $\eta : \mathcal{V} \to \mathbb{Z}[\mathcal{V}]$. To simplify the presentation, we do not consider "temporary" variables (whose update is nondeterministic), but the approach can easily be extended accordingly. Transitions $(\ell_0, \_, \_, \_)$ are called *initial* and $\mathcal{T}_0$ denotes the set of all initial transitions.

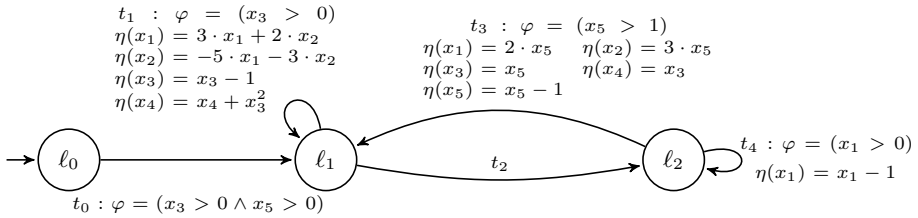

Fig. 2: An Integer Program with Non-Linear Size Bounds

Example 25. In the integer program of Fig. 2, we omitted identity updates $\eta(v) = v$ and guards where φ is `true`. Here, $\mathcal{V} = \{x_1, \ldots, x_5\}$ and $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$, where ℓ_0 is the initial location. Note that the loop in (1) *corresponds* to transition t_1.

Definition 26 (Correspondence between Loops and Transitions). *Let $t = (\ell, \varphi, \eta, \ell)$ be a transition with $\varphi \in \mathcal{F}(\mathcal{V}')$ for some variables $\mathcal{V}' \subseteq \mathcal{V}$ such that $\eta(x) = x$ for all $x \in \mathcal{V} \setminus \mathcal{V}'$ and $\eta(x) \in \mathbb{Z}[\mathcal{V}']$ for all $x \in \mathcal{V}'$. A loop (φ', η') with $\varphi' \in \mathcal{F}(\{x_1, \ldots, x_d\})$ and $\eta' : \{x_1, \ldots, x_d\} \to \mathbb{Z}[\{x_1, \ldots, x_d\}]$* corresponds *to the transition t via the variable renaming $\pi : \{x_1, \ldots, x_d\} \to \mathcal{V}'$ if φ is $\pi(\varphi')$ and for all $1 \leq i \leq d$ we have $\eta(\pi(x_i)) = \pi(\eta'(x_i))$.*

To define the semantics of integer programs, an evaluation step moves from one configuration $(\ell, \sigma) \in \mathcal{L} \times \Sigma$ to another configuration (ℓ', σ') via a transition $(\ell, \varphi, \eta, \ell')$ where $\sigma(\varphi)$ holds. Here, σ' is obtained by applying the update η on σ. From now on, we fix an integer program $\mathcal{P} = (\mathcal{V}, \mathcal{L}, \ell_0, \mathcal{T})$.

Definition 27 (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 $v \in \mathcal{V}$. 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 $k \geq 0$.*

Example 28. If we encode states as tuples $(\sigma(x_1), \ldots, \sigma(x_5)) \in \mathbb{Z}^5$, then $(-6, -8, 2, 1, 1) \to_{t_0} (-6, -8, 2, 1, 1) \to_{t_1}^2 (6, 8, 0, 6, 1) \to_{t_2} (6, 8, 0, 6, 1) \to_{t_4}^6 (0, 8, 0, 6, 1)$.

Now we define size bounds for variables v after evaluating a transition t: $\mathcal{SB}(t, v)$ is a *size bound* for v w.r.t. t if for any run starting in $\sigma_0 \in \Sigma$, $|\sigma_0|(\mathcal{SB}(t, v))$ is greater or equal to the largest absolute value of v after evaluating t.

Definition 29 (Size Bounds [8, 19]). *A function $\mathcal{SB} : (\mathcal{T} \times \mathcal{V}) \to \mathcal{B}$ is a* (global) size bound *for the program $\mathcal{P}$ if for all $(t, x) \in \mathcal{T} \times \mathcal{V}$ and all states $\sigma_0 \in \Sigma$ we have $|\sigma_0|(\mathcal{SB}(t, x)) \geq \sup\{|\sigma'(x)| \mid \exists\, \ell' \in \mathcal{L}.\ (\ell_0, \sigma_0)\ (\to^* \circ \to_t)\ (\ell', \sigma')\}$.*

Later in Lemma 35, we will compare the notion of size bounds for transitions in a program from Def. 29 to our earlier notion of size bounds for loops from Def. 6.

Example 30. As an example, we give size bounds for the transitions t_0 and t_3 in Fig. 2. Since t_0 does not change any variables, a size bound is $\mathcal{SB}(t_0, x_i) = x_i$ for

all $1 \leq i \leq 5$. Note that the value of x_5 is never increased and is bounded from below by 0 in any run through the program. Thus, $\mathcal{SB}(t_3, x_3) = x_5 = \mathcal{SB}(t_3, x_5)$. Similarly, we have $\mathcal{SB}(t_3, x_1) = 2 \cdot x_5$, $\mathcal{SB}(t_3, x_2) = 3 \cdot x_5$, and $\mathcal{SB}(t_3, x_4) = x_3$.

To infer size bounds for transitions as in Def. 29 automatically, we lift *local* size bounds (i.e., size bounds which only hold for a subprogram with transitions $\mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$) to global size bounds for the *complete* program. For the subprogram, one considers runs which start after evaluating an *entry transition* of $\mathcal{T}'$.

Definition 31 (Entry Transitions [8]). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$. The* entry transitions *of $\mathcal{T}'$ are $\mathcal{E}_{\mathcal{T}'} = \{t \mid t = (\_, \_, \_, \ell) \in \mathcal{T} \setminus \mathcal{T}' \text{ and there is a } (\ell, \_, \_, \_) \in \mathcal{T}'\}$.*

Example 32. For the program in Fig. 2, we have $\mathcal{E}_{\{t_1\}} = \{t_0, t_3\}$ and $\mathcal{E}_{\{t_4\}} = \{t_2\}$.

Definition 33 (Local Size Bounds). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$ and $t' \in \mathcal{T}'$. $\mathcal{SB}_{t'} : \mathcal{V} \to \mathcal{B}$ is a* local size bound *for t' w.r.t. $\mathcal{T}'$ if for all $x \in \mathcal{V}$ and all $\sigma \in \Sigma$:*[4] *$|\sigma|(\mathcal{SB}_{t'}(x)) \geq \sup\{|\sigma'(x)| \mid \exists \ell' \in \mathcal{L}, (\_, \_, \_, \ell) \in \mathcal{E}_{\mathcal{T}'}.\ (\ell, \sigma)\ (\to^*_{\mathcal{T}'} \circ \to_{t'})\ (\ell', \sigma')\}$.*

Thm. 34 below yields a novel *modular* procedure to infer (global) size bounds from previously computed local size bounds. A local size bound for a transition t' w.r.t. a subprogram $\mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$ is lifted by inserting size bounds for all entry transitions. Again, this is possible because we only use weakly monotonically increasing functions as bounds. Here, "$b\,[v/p_v \mid v \in \mathcal{V}]$" denotes the bound which results from replacing every variable v by p_v in the bound b.

Theorem 34 (Lifting Local Size Bounds). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T} \setminus \mathcal{T}_0$, let $\mathcal{SB}_{t'}$ be a local size bound for a transition t' w.r.t. $\mathcal{T}'$ and let $\mathcal{SB} : (\mathcal{T} \times \mathcal{V}) \to \mathcal{B}$ be a size bound for $\mathcal{P}$. Let $\mathcal{SB}'(t', x) = \sum_{r \in \mathcal{E}_{\mathcal{T}'}} \mathcal{SB}_{t'}(x)\,[v/\mathcal{SB}(r, v) \mid v \in \mathcal{V}]$ and $\mathcal{SB}'(t, x) = \mathcal{SB}(t, x)$ for all $t' \neq t$. Then $\mathcal{SB}'$ is also a size bound for $\mathcal{P}$.*

To obtain local size bounds which can then be lifted via Thm. 34, we look for transitions t_L that correspond to a loop L and then we compute a size bound for L as in Sect. 2 and 3. The following lemma shows that size bounds for loops as in Def. 6 indeed yield local size bounds for the corresponding transitions.[5]

Lemma 35 (Local Size Bounds via Loops). *Let $\mathcal{SB}_L$ be a size bound for a loop L (as in Def. 6) which corresponds to a transition t_L via a variable renaming π. Then $\pi \circ \mathcal{SB}_L \circ \pi^{-1}$ is a local size bound for t_L w.r.t. $\{t_L\}$ (as in Def. 33).*

Example 36. $\mathcal{SB}_L(x_4) = x_4 + 3 \cdot x_3^3 + 2 \cdot x_3^2 + x_3$ is a size bound for x_4 in the loop (1), see Ex. 8. This loop corresponds to transition t_1 in the program of Fig. 2. Since $\mathcal{E}_{\{t_1\}} = \{t_0, t_3\}$ by Ex. 32, Thm. 34 yields the following (non-linear) size bound for x_4 in the full program of Fig. 2 (see Ex. 30 for $\mathcal{SB}(t_0, v)$ and $\mathcal{SB}(t_3, v)$):

$$\mathcal{SB}(t_1, x_4) = \mathcal{SB}_L(x_4)\,[v/\mathcal{SB}(t_0, v) \mid v \in \mathcal{V}] + \mathcal{SB}_L(x_4)\,[v/\mathcal{SB}(t_3, v) \mid v \in \mathcal{V}]$$

[4] To simplify the formalism, in this definition, we consider every possible configuration (ℓ, σ) and not only configurations which are reachable from the initial location ℓ_0.

[5] Local or global size bounds for transitions only have to hold if the transition is indeed taken. In contrast, size bounds for loops also have to hold if there is no loop iteration. This will be needed in Thm. 38 to compute local size bounds for simple cycles.

$$= (x_4 + 3 \cdot x_3^3 + 2 \cdot x_3^2 + x_3) + (x_3 + 3 \cdot x_5^3 + 2 \cdot x_5^2 + x_5)$$
$$= 2 \cdot x_3 + 2 \cdot x_3^2 + 3 \cdot x_3^3 + x_4 + x_5 + 2 \cdot x_5^2 + 3 \cdot x_5^3$$

Analogously, we infer the remaining size bounds $\mathcal{SB}(t_1, x_i)$, e.g., $\mathcal{SB}(t_1, x_1) = (4{\cdot}x_1 + 2{\cdot}x_2)\,[v/\mathcal{SB}(t_0, v) \mid v \in \mathcal{V}] + (4{\cdot}x_1 + 2{\cdot}x_2)\,[v/\mathcal{SB}(t_3, v) \mid v \in \mathcal{V}] = 4{\cdot}x_1 + 2{\cdot}x_2 + 14{\cdot}x_5$.

Our approach alternates between improving size and runtime bounds for individual transitions. We start with $\mathcal{SB}(t_0, x) = |\eta(x)|$ for initial transitions $t_0 \in \mathcal{T}_0$ where η is t_0's update, and $\mathcal{SB}(t, \_) = \omega$ for $t \in \mathcal{T} \setminus \mathcal{T}_0$. Here, similar to the notion $\lceil |p| \rceil$ in Sect. 2, for every polynomial $p = \sum_j c_j \cdot \beta_j$ with normalized monomials β_j, $|p|$ is the polynomial $\sum_j |c_j| \cdot \beta_j$. To improve the size bounds of transitions that correspond to (possibly non-linear) solvable loops, we can use closed forms (Thm. 7) and the lifting via Thm. 34. Otherwise, we use an existing incomplete technique [8] to improve size bounds (where [8] essentially only succeeds for updates without non-linear arithmetic). In this way, we can automatically compute polynomial size bounds for all remaining transitions and variables in the program of Fig. 2 (e.g., we obtain $\mathcal{SB}(t_2, x_1) = \mathcal{SB}(t_1, x_1) = 4{\cdot}x_1 + 2{\cdot}x_2 + 14{\cdot}x_5$).

Both the technique from [8] and our approach from Thm. 7 rely on runtime bounds to compute size bounds. On the other hand, as shown in [8, 19, 27], size bounds for "previous" transitions are needed to infer (global) runtime bounds for transitions in a program. For that reason, the alternated computation resp. improvement of global size and runtime bounds for the transitions is repeated until all bounds are finite. We will illustrate this in more detail in Sect. 5.

In Def. 26 and Lemma 35 we considered transitions with the same start and target location that directly correspond to loops. To increase the applicability of our approach, as in [27] now we consider so-called *simple cycles*, where iterations through the cycle can only be done in a unique way. So the cycle must not have subcycles and there must not be any indeterminisms concerning the next transition to be taken. Formally, $\mathcal{C} = \{t_1, \ldots, t_n\} \subseteq \mathcal{T}$ is a simple cycle if 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 $t_n = (\ell_n, \_, \_, \ell_1)$. To handle simple cycles, we *chain* transitions.[6]

Definition 37 (Chaining (see, e.g., [27])). *Let $t_1, \ldots, t_n \in \mathcal{T}$ where $t_i = (\ell_i, \varphi_i, \eta_i, \ell_{i+1})$ for all $1 \leq i \leq n-1$. 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*

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

Now we want to compute a *local* size bound for the transition t_n w.r.t. a simple cycle $\mathcal{C} = \{t_1, \ldots, t_n\}$ where a loop L corresponds to $t_1 \star \ldots \star t_n$ via π. Then a size bound $\mathcal{SB}_L$ for the loop L yields the size bound $\pi \circ \mathcal{SB}_L \circ \pi^{-1}$ for t_n regarding runs through $\mathcal{C}$ starting in t_1. However, to obtain a local size bound $\mathcal{SB}_{t_n}$ w.r.t. $\mathcal{C}$, we have to consider runs starting after any entry transition $(\_, \_, \_, \ell_i) \in \mathcal{E}_\mathcal{C}$. Hence,

[6] The chaining of a loop L in Def. 20 corresponds to $p-1$ chaining steps of a transition t_L via Def. 37, i.e., to $t_L \star \ldots \star t_L$.

we use $|\,\eta_n(\dots\eta_i(\pi(\mathcal{SB}_L(\pi^{-1}(x))))\dots)\,|$ for any $(\_,\_,\_,\ell_i) \in \mathcal{E}_\mathcal{C}$. In this way, we also capture evaluations starting in ℓ_i, i.e., without evaluating the complete cycle.

Theorem 38 (Local Size Bounds for Simple Cycles). *Let $\mathcal{C} = \{t_1, \dots, t_n\} \subseteq \mathcal{T}$ be a simple cycle and let $\mathcal{SB}_L$ be a size bound for a loop L which corresponds to $t_1 \star \dots \star t_n$ via a variable renaming π. Then a* local size bound $\mathcal{SB}_{t_n}$ *for t_n w.r.t. $\mathcal{C}$ is $\mathcal{SB}_{t_n}(x) = \sum_{1 \leq i \leq n, (\_,\_,\_,\ell_i) \in \mathcal{E}_\mathcal{C}} |\,\eta_n(\dots\eta_i(\pi(\mathcal{SB}_L(\pi^{-1}(x))))\dots)\,|$.*

Example 39. As an example, in the program of Fig. 2 we replace $t_1 = (\ell_1, x_3 > 0, \eta_1, \ell_1)$ by $t_{1a} = (\ell_1, \texttt{true}, \eta_{1a}, \ell_1')$ and $t_{1b} = (\ell_1', x_3 > 0, \eta_{1b}, \ell_1)$ with a new location ℓ_1', where $\eta_{1a}(v) = \eta_1(v)$ for $v \in \{x_1, x_2\}$, $\eta_{1b}(v) = \eta_1(v)$ for $v \in \{x_3, x_4\}$, and η_{1a} resp. η_{1b} are the identity on the remaining variables. Then $\{t_{1a}, t_{1b}\}$ forms a simple cycle and Thm. 38 allows us to compute local size bounds $\mathcal{SB}_{t_{1b}}$ and $\mathcal{SB}_{t_{1a}}$ w.r.t. $\{t_{1a}, t_{1b}\}$, because the chained transitions $t_{1a} \star t_{1b} = t_1$ and $t_{1b} \star t_{1a}$ both correspond to the loop (1). They can then be lifted to global size bounds as in Ex. 36 using size bounds for the entry transitions $\mathcal{E}_{\{t_{1a}, t_{1b}\}} = \{t_0, t_3\}$.

This shows how we choose t' and $\mathcal{T}'$ when lifting local size bounds to global ones with Thm. 34: For a transition t' we search for a simple cycle $\mathcal{T}'$ such that chaining the cycle results in a twn- or suitable solvable loop and the size bounds of $\mathcal{E}_{\mathcal{T}'}$ are finite. For all other transitions, we compute size bounds as in [8].

5 Completeness of Size and Runtime Analysis for Programs

For individual loops, we showed in Thm. 23 that polynomial runtime bounds and finite size bounds are computable for all terminating prs loops. In this section, we discuss completeness of the size bound technique from the previous section and of termination and runtime complexity analysis for general integer programs. We show that for a large class of programs consisting of consecutive prs loops, in case of termination we can always infer finite runtime and size bounds.

To this end, we briefly recapitulate how size bounds are used to compute runtime bounds for general integer programs, and show that our new technique to infer size bounds also results in better runtime bounds. We call $\mathcal{RB} : \mathcal{T} \to \mathcal{B}$ a *(global) runtime bound* if for every transition $t \in \mathcal{T}$ and state $\sigma_0 \in \Sigma$, $|\sigma_0|(\mathcal{RB}(t))$ over-approximates the number of evaluations of t in any run starting in (ℓ_0, σ_0).

Definition 40 (Runtime Bound [8, 19]). *A function $\mathcal{RB} : \mathcal{T} \to \mathcal{B}$ is a* (global) runtime bound *if for all $t \in \mathcal{T}$ and all states $\sigma_0 \in \Sigma$, we have $|\sigma_0|(\mathcal{RB}(t)) \geq \sup\{n \in \mathbb{N} \mid \exists\, (\ell', \sigma').\ (\ell_0, \sigma_0)\ (\to_\mathcal{T}^* \circ \to_t)^n\ (\ell', \sigma')\}$.*

For our example in Fig. 2, a global runtime bound for t_0, t_2, and t_3 is $\mathcal{RB}(t_0) = 1$ and $\mathcal{RB}(t_2) = \mathcal{RB}(t_3) = x_5$, as x_5 is bounded from below by t_3's guard $x_5 > 1$ and the value of x_5 decreases by 1 in t_3, and no transition increases x_5.

To infer global runtime bounds automatically, similar as for size bounds, we first consider a smaller subprogram $\mathcal{T}' \subseteq \mathcal{T}$ and compute *local runtime bounds* for non-empty subsets $\mathcal{T}_>' \subseteq \mathcal{T}'$. A local runtime bound measures how often a transition $t \in \mathcal{T}_>'$ can occur in a run through $\mathcal{T}'$ that starts after an entry

transition $r \in \mathcal{E}_{\mathcal{T}'}$. Thus, local runtime bounds do not consider how many $\mathcal{T}'$-runs take place in a global run and they do not consider the sizes of the variables before starting a $\mathcal{T}'$-run. We lift these local bounds to global runtime bounds for the complete program afterwards.

Definition 41 (Local Runtime Bound [27]). *Let $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$. $\mathcal{RB}_{\mathcal{T}'_>} \in \mathcal{B}$ 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 $|\sigma|(\mathcal{RB}_{\mathcal{T}'_>}) \geq \sup\{n \in \mathbb{N} \mid \exists\, \sigma_0, (\ell', \sigma').\ (\ell_0, \sigma_0) \to^*_{\mathcal{T}} \circ \to_r\ (\ell, \sigma)\ (\to^*_{\mathcal{T}'} \circ \to_t)^n\ (\ell', \sigma')\}$.*

Example 42. In Fig. 2, local runtime bounds for $\mathcal{T}'_> = \mathcal{T}' = \{t_1\}$ and for $\mathcal{T}'_> = \mathcal{T}' = \{t_4\}$ are $\mathcal{RB}_{\{t_1\}} = x_3$ and $\mathcal{RB}_{\{t_4\}} = x_1$. Local runtime bounds can often be inferred automatically by approaches based on ranking functions (see, e.g., [8]) or by the complete technique for terminating prs loops (see Thm. 23).

If we have a local runtime bound $\mathcal{RB}_{\mathcal{T}'_>}$ w.r.t. $\mathcal{T}'$, then setting $\mathcal{RB}(t)$ to $\sum_{r \in \mathcal{E}_{\mathcal{T}'}} \mathcal{RB}(r) \cdot (\mathcal{RB}_{\mathcal{T}'_>}[v/\mathcal{SB}(r, v) \mid v \in \mathcal{V}])$ for all $t \in \mathcal{T}'_>$ yields a global runtime bound [27]. Here, we over-approximate the number of local $\mathcal{T}'$-runs which are started by an entry transition $r \in \mathcal{E}_{\mathcal{T}'}$ by an already computed global runtime bound $\mathcal{RB}(r)$. Moreover, we instantiate each $v \in \mathcal{V}$ by a size bound $\mathcal{SB}(r, v)$ to consider the size of v before a local $\mathcal{T}'$-run is started. So as mentioned in Sect. 4, we need runtime bounds to infer size bounds (see Thm. 7 and the inference of global size bounds in [8]), and on the other hand we need size bounds to compute runtime bounds. Thus, our implementation alternates between size bound and runtime bound computations (see [8, 27] for a more detailed description of this alternation).

Example 43. Based on the local runtime bounds in Ex. 42, we can compute the remaining global runtime bounds for our example. We obtain $\mathcal{RB}(t_1) = \mathcal{RB}(t_0) \cdot (x_3[v/\mathcal{SB}(t_0, v) \mid v \in \mathcal{V}]) + \mathcal{RB}(t_3) \cdot (x_3[v/\mathcal{SB}(t_3, v) \mid v \in \mathcal{V}]) = x_3 + x_5^2$ and $\mathcal{RB}(t_4) = \mathcal{RB}(t_2) \cdot (x_1[v/\mathcal{SB}(t_2, v) \mid v \in \mathcal{V}]) = x_5 \cdot (4 \cdot x_1 + 2 \cdot x_2 + 14 \cdot x_5)$. Thus, overall we have a quadratic runtime bound $\sum_{1 \leq i \leq 5} \mathcal{RB}(t_i)$. Note that it is due to our new size bound technique from Sect. 2–4 that we obtain polynomial runtime bounds in this example. In contrast, to the best of our knowledge, all other state-of-the-art tools fail to infer polynomial size or runtime bounds for this example. Similarly, if one modifies t_4 such that instead of x_1, x_4 is decreased as long as $x_4 > 0$ holds, then our approach again yields a polynomial runtime bound, whereas none of the other tools can infer finite runtime bounds.

Finally, we state our completeness results for integer programs. For a set $\mathcal{C} \subseteq \mathcal{T}$ and $\ell, \ell' \in \mathcal{L}$, let $\ell \rightsquigarrow_{\mathcal{C}} \ell'$ hold iff there is a transition $(\ell, \_, \_, \ell') \in \mathcal{C}$. We say that $\mathcal{C}$ is a *component* if we have $\ell \rightsquigarrow^+_{\mathcal{C}} \ell'$ for all locations ℓ, ℓ' occurring in $\mathcal{C}$, where $\rightsquigarrow^+_{\mathcal{C}}$ is the transitive closure of $\rightsquigarrow_{\mathcal{C}}$. So in particular, we must also have $\ell \rightsquigarrow^+_{\mathcal{C}} \ell$ for all locations ℓ in the transitions of $\mathcal{C}$. We call an integer program *simple* if every component is a simple cycle that is "reachable" from any initial state.

Definition 44 (Simple Integer Program). *An integer program $(\mathcal{V}, \mathcal{L}, \ell_0, \mathcal{T})$ is* simple *if every component $\mathcal{C} \subseteq \mathcal{T}$ is a simple cycle, and for every entry transition $(\_, \_, \_, \ell) \in \mathcal{E}_{\mathcal{C}}$ and every $\sigma_0 \in \Sigma$, there is an evaluation $(\ell_0, \sigma_0) \to^*_{\mathcal{T}} (\ell, \sigma_0)$.*

In Fig. 2, $\mathcal{T} \setminus \{t_0\}$ is a component that is no simple cycle. However, if we remove t_3 and replace t_0's guard by `true`, then the resulting program $\mathcal{P}'$ is simple (but not linear). A simple program terminates iff each of its isolated simple cycles terminates. Thus, if we can prove termination for every simple cycle, then the overall program terminates. Hence, if after chaining, every simple cycle corresponds to a linear, unit prs loop, then we can decide termination and infer polynomial runtime and size bounds for the overall integer program. For terminating, non-unit prs loops, runtime bounds are still polynomial but size bounds can be exponential. Hence, then the global runtime bounds can be exponential as well. Note that in the example program $\mathcal{P}'$ above, the eigenvalues of the update matrices of t_1 and t_4 have absolute value 1, i.e., t_1 and t_4 correspond to unit prs loops. Hence, by Thm. 45 we obtain polynomial runtime and size bounds for $\mathcal{P}'$.

Theorem 45 (Completeness Results for Integer Programs).

(a) Termination is decidable for all simple linear *integer programs where after chaining, all simple cycles correspond to prs loops.*
(b) Finite runtime and size bounds are computable for all simple integer programs where after chaining, all simple cycles correspond to terminating *prs loops.*
(c) If in addition to (b), all simple cycles correspond to unit *prs loops, then the runtime and size bounds are* polynomial*.*

In the definition of simple integer programs (Def. 44), we required that for every component $\mathcal{C}$ and every entry transition $(\_, \_, \_, \ell) \in \mathcal{E}_{\mathcal{C}}$, there is an evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}}^* (\ell, \sigma_0)$ for every $\sigma_0 \in \Sigma$. If one strengthens this by requiring that one can reach ℓ from ℓ_0 using only transitions whose guard is `true` and whose update is the identity, then the class of programs in Thm. 45 (a) is decidable (there are only n ways to chain a simple cycle with n transitions and checking whether a loop is a prs loop is decidable by Lemma 19).

6 Conclusion and Evaluation

Conclusion In this paper, we developed techniques to infer size bounds automatically and to use them in order to obtain bounds on the runtime complexity of programs. This yields a complete procedure to prove termination and to infer runtime and size bounds for a large class of integer programs. Moreover, we showed how to integrate the complete technique into an (incomplete) modular technique for general integer programs. To sum up, we presented the following new contributions in this paper:

(a) We showed how to use closed forms in order to infer size bounds for loops with possibly non-linear arithmetic in Thm. 7.
(b) We proved completeness of our novel approach for terminating prs loops (see Thm. 23) in Sect. 3.
(c) We embedded our approach for loops into the setting of general integer programs in Sect. 4 and showed completeness of our approach for simple integer programs with only prs loops in Sect. 5.

(d) Finally, we implemented a prototype of our procedure in our re-implementation of the tool KoAT, written in OCaml. It integrates the computation of size bounds via closed forms for twn-loops and homogeneous (and thus linear) solvable loops into the complexity analysis for general integer programs.[7]

To infer local runtime bounds as in Def. 41, KoAT first applies multiphase-linear ranking functions (see [5, 19]), which can be done very efficiently. For twn-loops where no finite bound was found, it then uses the computability of runtime bounds for terminating twn-loops (see [17, 20, 27]). When computing size bounds, KoAT first applies the technique of [8] for reasons of efficiency and in case of exponential or infinite size bounds, it tries to compute size bounds via closed forms as in the current paper. Here, SymPy [30] is used to compute Jordan normal forms for the transformation to twn-loops. Moreover, KoAT applies a local control-flow refinement technique [19] (using the tool iRankFinder [13]) and preprocesses the program in the beginning, e.g., by extending the guards of transitions with invariants inferred by Apron [24]. For all SMT problems, KoAT uses Z3 [31]. In the future, we plan to extend the runtime bound inference of KoAT to prs loops and to extend our size bound computations also to suitable non-linear non-twn-loops.

Evaluation To evaluate our new technique, we tested KoAT on the 504 benchmarks for *Complexity of* C *Integer Programs* (CINT) from the *Termination Problems Data Base* [35] which is used in the annual *Termination and Complexity Competition (TermComp)* [18]. Here, all variables are interpreted as integers over $\mathbb{Z}$ (i.e., without overflows). To distinguish the original version of KoAT [8] from our re-implementation, we refer to them as KoAT1 resp. KoAT2. We used the following configurations of KoAT2, which apply different techniques to infer size bounds.

- KoAT2orig uses the original technique from [8] to infer size bounds.
- KoAT2+SIZE additionally uses our novel approach with Thm. 7, 34, and 38.

The CINT collection contains almost only examples with linear arithmetic and the existing tools can already solve most of its benchmarks which are not known to be non-terminating.[8] While most complexity analyzers are essentially restricted to programs with linear arithmetic, our new approach also succeeds on programs with *non-linear* arithmetic. Some programs with non-linear arithmetic could already be handled by KoAT due to our integration of the complete technique for the inference of local runtime bounds in [27]. But the approach from the current paper increases KoAT's power substantially for programs (possibly with non-linear arithmetic) where the values of variables computed in "earlier" loops influence the runtime of "later" loops (e.g., the modification of our example from Fig. 2 where t_4 decreases x_4 instead of x_1, see the end of Ex. 43).

Therefore, we extended CINT by 15 new typical benchmarks including the programs in (1), Fig. 2, and the modification of Fig. 2 discussed above, as well

[7] For a homogeneous solvable loop, the closed form of the twn-loop over $\mathbb{A}$ that results from its transformation is particularly easy to compute.

[8] iRankFinder [13] proves non-termination for 119 programs in CINT. KoAT2orig already infers finite runtimes for 343 of the remaining $504 - 119 = 386$ examples in CINT.

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(EXP)$	$< \omega$	AVG$^+$(s)	AVG(s)
KoAT2+SIZE	26	233 (2)	71 (1)	25 (9)	3 (2)	358 (14)	9.97	22.88
KoAT2orig	26	232 (1)	70	15	5 (4)	348 (5)	8.29	21.52
MaxCore	23	220 (4)	67 (1)	7	0	317 (5)	1.96	5.25
CoFloCo	22	197 (1)	66	5	0	290 (1)	0.59	2.68
KoAT1	25	170 (1)	74	12	8 (3)	289 (4)	0.96	3.49
Loopus	17	171 (1)	50 (1)	6 (1)	0	244 (3)	0.40	0.40

Table 1: Evaluation on the Collection CINT$^+$

as several benchmarks from the literature (e.g., [3, 6]), resulting in the collection CINT$^+$. For KoAT2 and KoAT1, we used Clang [11] and llvm2kittel [14] to transform C into integer programs as in Sect. 4. We compare KoAT2 with KoAT1 [8] and the tools CoFloCo [15], MaxCore [2] with CoFloCo in the backend, and Loopus [33]. These tools also rely on variants of size bounds: CoFloCo uses a set of constraints to measure the size of variables w.r.t. their initial and final values, MaxCore's size bound computations build upon [12], and Loopus considers suitable bounding invariants to infer size bounds.

Table 1 gives the results of our evaluation, where as in *TermComp*, we used a timeout of 5 minutes per example. The first entry in every cell denotes the number of benchmarks from CINT$^+$ for which the tool inferred the respective bound. The number in brackets only considers the 15 new examples. 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 example, KoAT2+SIZE proved a linear runtime bound for $231 + 2 = 233$ benchmarks, i.e., $\mathrm{rc}(\sigma) \in \mathcal{O}(n)$ holds for all initial states where $|\sigma(v)| \leq n$ for all $v \in \mathcal{V}$. Overall, this configuration succeeds on 358 examples, i.e., "$< \omega$" is the number of examples where a finite bound on the runtime complexity could be computed by the tool within the time limit. "AVG$^+$(s)" denotes the average runtime of successful runs in seconds, whereas "AVG(s)" is the average runtime of all runs.

Already on the original benchmarks CINT, integrating our novel technique for the inference of size bounds leads to the most powerful approach for runtime complexity analysis. The effect of the new size bound technique becomes even clearer when also considering our new examples which contain non-linear arithmetic and loops whose runtime depends on the results of earlier loops in the program. Thus, the new contributions of the paper are crucial in order to extend automated complexity analysis to larger programs with non-linear arithmetic.

KoAT's source code, a binary, and a Docker image are available at https://koat.verify.rwth-aachen.de/size. This website also has details on our experiments, a list and description of the new examples, and *web interfaces* to run KoAT's configurations directly online.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. "Cost Analysis of Object-Oriented Bytecode Programs". In: *Theoretical Computer Science* 413 (2012), pp. 142–159. DOI: 10.1016/j.tcs.2011.07.009.

[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, N. D. Jones, and L. Kristiansen. "Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time". In: *Proc. CiE '08*. LNCS 5028. 2008, pp. 67–76. DOI: 10.1007/978-3-540-69407-6_7.

[4] A. M. Ben-Amram and A. Pineles. "Flowchart Programs, Regular Expressions, and Decidability of Polynomial Growth-Rate". In: *Proc. VPT '16*. EPTCS 216. 2016, pp. 24–49. DOI: 10.4204/EPTCS.216.2.

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

[6] A. M. Ben-Amram and G. W. Hamilton. "Tight Worst-Case Bounds for Polynomial Loop Programs". In: *Proc. FoSSaCS '19*. LNCS 11425. 2019, pp. 80–97. DOI: 10.1007/978-3-030-17127-8_5.

[7] M. Braverman. "Termination of Integer Linear Programs". In: *Proc. CAV '06*. LNCS 4144. 2006, pp. 372–385. DOI: 10.1007/11817963_34.

[8] 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.

[9] J.-Y. Cai. "Computing Jordan Normal Forms Exactly for Commuting Matrices in Polynomial Time". In: *International Journal of Foundations of Computer Science* 5.3/4 (1994), pp. 293–302. DOI: 10.1142/S0129054194000165.

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

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

[12] P. Cousot and N. Halbwachs. "Automatic Discovery of Linear Restraints Among Variables of a Program". In: *Proc. POPL '78*. 1978, pp. 84–96. DOI: 10.1145/512760.512770.

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

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

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

[16] F. Frohn and J. Giesl. "Termination of Triangular Integer Loops Is Decidable". In: *Proc. CAV '19*. LNCS 11562. 2019, pp. 426–444. DOI: 10.1007/978-3-030-25543-5_24.

[17] F. Frohn, M. Hark, and J. Giesl. "Termination of Polynomial Loops". In: *Proc. SAS '20*. 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.

[18] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. "The Termination and Complexity Competition". In: *Proc. TACAS '19*. LNCS 11429. 2019, pp. 156–166. DOI: 10.1007/978-3-030-17502-3_10.

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

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

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

[22] M. Hosseini, J. Ouaknine, and J. Worrell. "Termination of Linear Loops over the Integers". In: *Proc. ICALP '19*. LIPIcs 132. 2019. DOI: 10.4230/LIPIcs.ICALP.2019.118.

[23] A. Humenberger, M. Jaroschek, and L. Kovács. "Invariant Generation for Multi-Path Loops with Polynomial Assignments". In: *Proc. VMCAI '18*. LNCS 10747. 2018, pp. 226–246. DOI: 10.1007/978-3-319-73721-8_11.

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

[25] Z. Kincaid, J. Breck, J. Cyphert, and T. Reps. "Closed Forms for Numerical Loops". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019). DOI: 10.1145/3290368.

[26] L. Kovács. "Reasoning Algebraically About P-Solvable Loops". In: *Proc. TACAS '08*. LNCS 4963. 2008, pp. 249–264. DOI: 10.1007/978-3-540-78800-3_18.

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

[28] N. Lommen and J. Giesl. "Targeting Completeness: Using Closed Forms for Size Bounds of Integer Programs". In: *CoRR* abs/2307.06921 (2023). DOI: 10.48550/arXiv.2307.06921.

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

[30] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. "SymPy: Symbolic Computing in Python". In: *PeerJ Computer Science* 3 (2017). DOI: 10.7717/peerj-cs.103.

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

[32] E. Rodríguez-Carbonell and D. Kapur. "Automatic Generation of Polynomial Loop Invariants: Algebraic Foundation". In: *Proc. ISSAC '04*. 2004, pp. 266–273. DOI: 10.1145/1005285.1005324.

[33] 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.

[34] A. Tiwari. "Termination of Linear Programs". In: *Proc. CAV '04*. LNCS 3114. 2004, pp. 70–82. DOI: 10.1007/978-3-540-27813-9_6.

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

[36] 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.