Proving Non-Termination and Lower Runtime Bounds with LoAT (System Description)*

Florian Frohn and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract. We present the *Loop Acceleration Tool* (LoAT), a powerful tool for proving non-termination and worst-case lower bounds for programs operating on integers. It is based on the novel calculus from [10,11] for *loop acceleration*, i.e., transforming loops into non-deterministic straight-line code, and for finding non-terminating configurations. To implement it efficiently, LoAT uses a new approach based on unsat cores. We evaluate LoAT's power and performance by extensive experiments.

1 Introduction

Efficiency is one of the most important properties of software. Consequently, *automated complexity analysis* is of high interest to the software verification community. Most research in this area has focused on deducing *upper* bounds on the worst-case complexity of programs. In contrast, the *Loop Acceleration Tool* LoAT aims to find performance bugs by deducing *lower* bounds on the worst-case complexity of programs operating on integers. Since non-termination implies the lower bound ∞, LoAT is also equipped with non-termination techniques.

LoAT is based on *loop acceleration* [4,5,9–11,15], which replaces loops by non-deterministic code: The resulting program chooses a value n, representing the number of loop iterations in the original program. To be sound, suitable constraints on n are synthesized to ensure that the original loop allows for at least n iterations. Moreover, the transformed program updates the program variables to the same values as n iterations of the original loop, but it does so in a single step. To achieve that, the loop body is transformed into a *closed form*, which is parameterized in n. In this way, LoAT is able to compute *symbolic under-approximations* of programs, i.e., every execution path in the resulting transformed program corresponds to a path in the original program, but not necessarily vice versa. In contrast to many other techniques for computing under-approximations, the symbolic approximations of LoAT cover *infinitely many runs* of *arbitrary length*.

Contributions: The main new feature of the novel version of LoAT presented in this paper is the *integration* of the *loop acceleration calculus* from [10, 11], which combines different loop acceleration techniques in a modular way, into LoAT's framework. This enables LoAT to use the loop acceleration calculus for the analysis of full integer programs, whereas the standalone implementation of

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

the calculus from [10,11] was only applicable to single loops without branching in the body. To control the application of the calculus, we use a new technique based on unsat cores (see Sect. 5). The new version of LoAT is evaluated in extensive experiments. See [14] for all missing proofs.

2 Preliminaries

Let $\mathcal{L} \supseteq \{main\}$ be a finite set of *locations*, where *main* is the *canonical start location* (i.e., the entry point of the program), and let $\vec{x} := [x_1, \ldots, x_d]$ be the vector of *program variables*. Furthermore, let $\mathcal{TV}$ be a countably infinite set of *temporary variables*, which are used to model non-determinism, and let $\sup \mathbb{Z} := \infty$. We call an arithmetic expression e an *integer expression* if it evaluates to an integer when all variables in e are instantiated by integers. LoAT analyzes tail-recursive programs operating on integers, represented as *integer transition systems* (ITSs), i.e., sets of *transitions* $f(\vec{x}) \xrightarrow{p} g(\vec{a})\,[\varphi]$ where $f, g \in \mathcal{L}$, the *update* $\vec{a}$ is a vector of d integer expressions over $\mathcal{TV} \cup \vec{x}$, the *cost* p is either an arithmetic expression over $\mathcal{TV} \cup \vec{x}$ or ∞, and the *guard* φ is a conjunction of inequations over integer expressions with variables from $\mathcal{TV} \cup \vec{x}$.[1] For example, consider the loop on the left and the corresponding transition t_{loop} on the right.

$$\textbf{while } x > 0 \textbf{ do } x \leftarrow x - 1 \qquad f(x) \xrightarrow{1} f(x-1)\,[x > 0] \quad (t_{loop})$$

Here, the cost 1 instructs LoAT to use the number of loop iterations as cost measure. LoAT allows for arbitrary *user defined* cost measures, since the user can choose any polynomials over the program variables as costs.

LoAT synthesizes transitions with cost ∞ to represent non-terminating runs, i.e., such transitions are not allowed in the input. A *configuration* is of the form $f(\vec{c})$ with $f \in \mathcal{L}$ and $\vec{c} \in \mathbb{Z}^d$. For any entity $s \notin \mathcal{L}$ and any arithmetic expressions $\vec{b} = [b_1, \ldots, b_d]$, let $s(\vec{b})$ denote the result of replacing each variable x_i in s by b_i, for all $1 \leq i \leq d$. Moreover, $\mathcal{V}ars(s)$ denotes the program variables and $\mathcal{TV}(s)$ denotes the temporary variables occurring in s. For an integer transition system $\mathcal{T}$, a configuration $f(\vec{c})$ *evaluates to* $g(\vec{c}')$ *with cost* $k \in \mathbb{Z} \cup \{\infty\}$, written $f(\vec{c}) \xrightarrow{k}_{\mathcal{T}} g(\vec{c}')$, if there exists a transition $f(\vec{x}) \xrightarrow{p} g(\vec{a})\,[\varphi] \in \mathcal{T}$ and an instantiation of its temporary variables with integers such that the following holds:

$$\varphi(\vec{c}) \wedge \vec{c}\,' = \vec{a}(\vec{c}) \wedge k = p(\vec{c}).$$

As usual, we write $f(\vec{c}) \xrightarrow{k}{}^*_{\mathcal{T}} g(\vec{c}')$ if $f(\vec{c})$ evaluates to $g(\vec{c}')$ in arbitrarily many steps, and the sum of the costs of all steps is k. We omit the costs if they are irrelevant. The *derivation height* of $f(\vec{c})$ is

$$dh_{\mathcal{T}}(f(\vec{c})) := \sup\{k \mid \exists g(\vec{c}').\, f(\vec{c}) \xrightarrow{k}{}^*_{\mathcal{T}} g(\vec{c}')\}$$

and the *runtime complexity* of $\mathcal{T}$ is

$$rc_{\mathcal{T}}(n) := \sup\{dh_{\mathcal{T}}(main(c_1, \ldots, c_d)) \mid |c_1| + \ldots + |c_d| \leq n\}.$$

[1] LoAT can also analyze the complexity of certain non-tail-recursive programs, see [9]. For simplicity, we restrict ourselves to tail-recursive programs in the current paper.

$\mathcal{T}$ terminates if no configuration $\mathit{main}(\vec{c})$ admits an infinite $\to_{\mathcal{T}}$-sequence and $\mathcal{T}$ is *finitary* if no configuration $\mathit{main}(\vec{c})$ admits a $\to_{\mathcal{T}}$-sequence with cost ∞. Otherwise, $\vec{c}$ is a *witness of non-termination* or a *witness of infinitism*, respectively. Note that termination implies finitism for ITSs where no transition has cost ∞. However, our approach may transform non-terminating ITSs into terminating, infinitary ITSs, as it replaces non-terminating loops by transitions with cost ∞.

3 Overview of LoAT

The goal of LoAT is to compute a lower bound on $rc_{\mathcal{T}}$ or even prove non-termination of $\mathcal{T}$. To this end, it repeatedly applies program simplifications, so-called *processors*. When applying them with a suitable strategy (see [8, 9]), one eventually obtains *simplified transitions* of the form $\mathit{main}(\vec{x}) \xrightarrow{p} f(\vec{a})\,[\varphi]$ where $f \neq \mathit{main}$. As LoAT's processors are *sound for lower bounds* (i.e., if they transform $\mathcal{T}$ to $\mathcal{T}'$, then $dh_{\mathcal{T}} \geq dh_{\mathcal{T}'}$), such a simplified transition gives rise to the lower bound $I_\varphi \cdot p$ on $dh_{\mathcal{T}}(\mathit{main}(\vec{x}))$ (where I_φ denotes the indicator function of φ which is 1 for values where φ holds and 0 otherwise). This bound can be lifted to $rc_{\mathcal{T}}$ by solving a so-called *limit problem*, see [9].

LoAT's processors are also *sound for non-termination*, as they preserve finitism. So if $p = \infty$, then it suffices to prove satisfiability of φ to prove infinitism, which implies non-termination of the original ITS, where transitions with cost ∞ are forbidden (see Sect. 2). LoAT's most important processors are:

Loop Acceleration (Sect. 4) transforms a *simple loop*, i.e., a single transition $f(\vec{x}) \xrightarrow{p} f(\vec{a})\,[\varphi]$, into a non-deterministic transition that can simulate several loop iterations in one step. For example, loop acceleration transforms t_{loop} to

$$f(x) \xrightarrow{n} f(x-n)\,[x \geq n \wedge n > 0] \qquad (t_{loop^n})$$

where $n \in \mathcal{TV}$, i.e., the value of n can be chosen non-deterministically.

Instantiation [9, Thm. 3.12] replaces temporary variables by integer expressions. For example, it could instantiate n with x in t_{loop^n}, resulting in

$$f(x) \xrightarrow{x} f(0)\,[x > 0]\,. \qquad (t_{loop^x})$$

Chaining [9, Thm. 3.18] combines two subsequent transitions into one transition. For example, chaining combines the transitions

$$\mathit{main}(x) \xrightarrow{1} f(x)$$
$$\text{and } t_{loop^x} \text{ to} \quad \mathit{main}(x) \xrightarrow{x+1} f(0)\,[x > 0]\,.$$

Nonterm (Sect. 6) searches for witnesses of non-termination, characterized by a formula ψ. So it turns, e.g.,

$$f(x_1, x_2) \xrightarrow{1} f(x_1 - x_2, x_2)\,[x_1 > 0] \qquad (t_{nonterm})$$
$$\text{into} \quad f(x_1, x_2) \xrightarrow{\infty} \mathit{sink}(x_1, x_2)\,[x_1 > 0 \wedge x_2 \leq 0]$$

(where $\mathit{sink} \in \mathcal{L}$ is fresh), as each $\vec{c} \in \mathbb{Z}^2$ with $c_1 > 0 \wedge c_2 \leq 0$ witnesses non-termination of $t_{nonterm}$, i.e., here ψ is $x_1 > 0 \wedge x_2 \leq 0$.

Intuitively, LoAT uses **Chaining** to transform non-simple loops into simple loops. **Instantiation** resolves non-determinism heuristically and thus reduces

the number of temporary variables, which is crucial for scalability. In addition to these processors, LoAT removes transitions after processing them, as explained in [9]. See [8,9] for heuristics and a suitable strategy to apply LoAT's processors.

4 Modular Loop Acceleration

For **Loop Acceleration**, LoAT uses *conditional acceleration techniques* [10]. Given two formulas ξ and $\check{\varphi}$, and a loop with update $\vec{a}$, a conditional acceleration technique yields a formula $\mathit{accel}(\xi, \check{\varphi}, \vec{a})$ which implies that ξ holds throughout n loop iterations (i.e., ξ is an *n-invariant*), provided that $\check{\varphi}$ is an n-invariant, too. In the following, let $\vec{a}^0(\vec{x}) := \vec{x}$ and $\vec{a}^{m+1}(\vec{x}) := \vec{a}(\vec{a}^m(\vec{x})) = \vec{a}[\vec{x}/\vec{a}^m(\vec{x})]$.

Definition 1 (Conditional Acceleration Technique). *A function accel is a* conditional acceleration technique *if the following implication holds for all formulas ξ and $\check{\varphi}$ with variables from $\mathcal{TV} \cup \vec{x}$, all updates $\vec{a}$, all $n > 0$, and all instantiations of the variables with integers:*

$$\big(\mathit{accel}(\xi, \check{\varphi}, \vec{a}) \land \forall i \in [0, n).\ \check{\varphi}(\vec{a}^i(\vec{x}))\big) \implies \forall i \in [0, n).\ \xi(\vec{a}^i(\vec{x})).$$

The prerequisite $\forall i \in [0, n).\ \check{\varphi}(\vec{a}^i(\vec{x}))$ is ensured by previous acceleration steps, i.e., $\check{\varphi}$ is initially $\top$ (*true*), and it is refined by conjoining a part ξ of the loop guard in each acceleration step. When formalizing acceleration techniques, we only specify the result of *accel* for certain arguments ξ, $\check{\varphi}$, and $\vec{a}$, and assume $\mathit{accel}(\xi, \check{\varphi}, \vec{a}) = \bot$ (*false*) otherwise.

Definition 2 (LoAT's Conditional Acceleration Techniques [10, 11]).

Increase	$\mathit{accel}_{\mathit{inc}}(\xi, \check{\varphi}, \vec{a}) := \xi$	*if* $\models \xi \land \check{\varphi} \implies \xi(\vec{a})$
Decrease	$\mathit{accel}_{\mathit{dec}}(\xi, \check{\varphi}, \vec{a}) := \xi(\vec{a}^{n-1}(\vec{x}))$	*if* $\models \xi(\vec{a}) \land \check{\varphi} \implies \xi$
Eventual Decrease	$\mathit{accel}_{\mathit{ev\text{-}dec}}(t > 0, \check{\varphi}, \vec{a}) := t > 0 \land t(\vec{a}^{n-1}(\vec{x})) > 0$	*if* $\models (t \geq t(\vec{a}) \land \check{\varphi}) \implies t(\vec{a}) \geq t(\vec{a}^2(\vec{x}))$
Eventual Increase	$\mathit{accel}_{\mathit{ev\text{-}inc}}(t > 0, \check{\varphi}, \vec{a}) := t > 0 \land t \leq t(\vec{a})$	*if* $\models (t \leq t(\vec{a}) \land \check{\varphi}) \implies t(\vec{a}) \leq t(\vec{a}^2(\vec{x}))$
Fixpoint	$\mathit{accel}_{\mathit{fp}}(t > 0, \check{\varphi}, \vec{a}) := t > 0 \land \bigwedge_{x \in \mathit{closure}_{\vec{a}}(t)} x = x(\vec{a})$	*where* $\mathit{closure}_{\vec{a}}(t) := \bigcup_{i \in \mathbb{N}} \mathcal{V}\mathit{ars}(t(\vec{a}^i(\vec{x})))$

The above five techniques are taken from [10,11], where only deterministic loops are considered (i.e., there are no temporary variables). Lifting them to non-deterministic loops in a way that allows for *exact* conditional acceleration techniques (which capture all possible program runs) is non-trivial and beyond the scope of this paper. Thus, we sacrifice exactness and treat temporary variables like additional constant program variables whose update is the identity, resulting in a sound under-approximation (that captures a subset of all possible runs).

So essentially, **Increase** and **Decrease** handle inequations $t > 0$ in the loop guard where t increases or decreases (weakly) monotonically when applying the loop's update. The canonical examples where **Increase** or **Decrease** applies are

$$f(x, \ldots) \to f(x+1, \ldots)\,[x > 0 \land \ldots] \quad \text{or} \quad f(x, \ldots) \to f(x-1, \ldots)\,[x > 0 \land \ldots]\,,$$

respectively. **Eventual Decrease** applies if t never increases again once it starts to decrease. The canonical example is $f(x, y, \ldots) \to f(x+y, y-1, \ldots)\,[x > 0 \land \ldots]$.

Similarly, **Eventual Increase** applies if t never decreases again once it starts to increase. **Fixpoint** can be used for inequations $t > 0$ that do not behave (eventually) monotonically. It should only be used if $accel_{fp}(t > 0, \check{\varphi}, \vec{a})$ is satisfiable.

LoAT uses the *acceleration calculus* of [10]. It operates on *acceleration problems* $[\![\psi \mid \check{\varphi} \mid \widehat{\varphi}]\!]_{\vec{a}}$, where ψ (which is initially $\top$) is repeatedly refined. When it stops, ψ is used as the guard of the resulting accelerated transition. The formulas $\check{\varphi}$ and $\widehat{\varphi}$ are the parts of the loop guard that have already or have not yet been handled, respectively. So $\check{\varphi}$ is initially $\top$, and $\widehat{\varphi}$ and $\vec{a}$ are initialized with the guard φ and the update of the loop $f(\vec{x}) \xrightarrow{p} f(\vec{a})\,[\varphi]$ under consideration, i.e., the initial acceleration problem is $[\![\top \mid \top \mid \varphi]\!]_{\vec{a}}$. Once $\widehat{\varphi}$ is $\top$, the loop is accelerated to $f(\vec{x}) \xrightarrow{q} f(\vec{a}^n(\vec{x}))\,[\psi \wedge n > 0]$, where the cost q and a closed form for $\vec{a}^n(\vec{x})$ are computed by the recurrence solver PURRS [2].

Definition 3 (Acceleration Calculus for Conjunctive Loops). *The relation $\rightsquigarrow$ on acceleration problems is defined as*

$$\frac{accel(\xi, \check{\varphi}, \vec{a}) = \psi_2}{[\![\psi_1 \mid \check{\varphi} \mid \xi \wedge \widehat{\varphi}]\!]_{\vec{a}} \rightsquigarrow [\![\psi_1 \wedge \psi_2 \mid \check{\varphi} \wedge \xi \mid \widehat{\varphi}]\!]_{\vec{a}}} \qquad \begin{array}{l} \textit{accel is a conditional} \\ \textit{acceleration technique} \end{array}$$

So to accelerate a loop, one picks a not yet handled part ξ of the guard in each step. When accelerating $f(\vec{x}) \to f(\vec{a})\,[\xi]$ using a conditional acceleration technique *accel*, one may assume $\forall i \in [0, n).\ \check{\varphi}(\vec{a}^i(\vec{x}))$. The result of *accel* is conjoined to the result ψ_1 computed so far, and ξ is moved from the third to the second component of the problem, i.e., to the already handled part of the guard.

Example 4 (Acceleration Calculus). We show how to accelerate the loop

$$f(x, y) \xrightarrow{x} f(x - y, y)\,[x > 0 \wedge y \geq 0] \qquad \text{to}$$

$$f(x, y) \xrightarrow{(x + \frac{y}{2}) \cdot n - \frac{y}{2} \cdot n^2} f(x - n \cdot y, y)\,[y \geq 0 \wedge x - (n-1) \cdot y > 0 \wedge n > 0]\,.$$

The closed form $\vec{a}^n(x) = (x - n \cdot y, y)$ can be computed via recurrence solving. Similarly, the cost $(x + \frac{y}{2}) \cdot n - \frac{y}{2} \cdot n^2$ of n loop iterations is obtained by solving the following recurrence relation (where $c^{(n)}$ and $x^{(n)}$ denote the cost and the value of x after n applications of the transition, respectively).

$$c^{(n)} = c^{(n-1)} + x^{(n-1)} = c^{(n-1)} + x - (n-1) \cdot y \qquad \text{and} \qquad c^{(1)} = x.$$

The guard is computed as follows:

$$\begin{aligned} [\![\top \mid \top \mid x > 0 \wedge y \geq 0]\!]_{\vec{a}} &\rightsquigarrow [\![y \geq 0 \mid y \geq 0 \mid x > 0]\!]_{\vec{a}} \\ &\rightsquigarrow [\![y \geq 0 \wedge x - (n-1) \cdot y > 0 \mid y \geq 0 \wedge x > 0 \mid \top]\!]_{\vec{a}}\,. \end{aligned}$$

In the 1^{st} step, we have $\xi = (y \geq 0)$ and $accel_{inc}(y \geq 0, \top, \vec{a}) = (y \geq 0)$. In the 2^{nd} step, we have $\xi = (x > 0)$ and $accel_{dec}(x > 0, y \geq 0, \vec{a}) = (x - (n-1) \cdot y > 0)$. So the inequation $x - (n-1) \cdot y > 0$ ensures n-invariance of $x > 0$.

5 Efficient Loop Acceleration using Unsat Cores

Each attempt to apply a conditional acceleration technique other than **Fixpoint** requires proving an implication, which is implemented via SMT solving by proving

unsatisfiability of its negation. For **Fixpoint**, satisfiability of $accel_{fp}(t > 0, \check{\varphi}, \vec{a})$ is checked via SMT. So even though LoAT restricts ξ to atoms, up to $\Theta(m^2)$ attempts to apply a conditional acceleration technique are required to accelerate a loop whose guard contains m inequations using a naive strategy ($5 \cdot m$ attempts for the 1^{st} $\rightsquigarrow$-step, $5 \cdot (m-1)$ attempts for the 2^{nd} step, ...).

To improve efficiency, LoAT uses a novel encoding that requires just $5 \cdot m$ attempts. For any $\alpha \in AT_{imp} = \{inc, dec, ev\text{-}dec, ev\text{-}inc\}$, let $encode_\alpha(\xi, \check{\varphi}, \vec{a})$ be the implication that has to be valid in order to apply $accel_\alpha$, whose premise is of the form $\ldots \wedge \check{\varphi}$. Instead of repeatedly refining $\check{\varphi}$, LoAT tries to prove validity[2] of $encode_{\alpha,\xi} := encode_\alpha(\xi, \varphi \setminus \{\xi\}, \vec{a})$ for each $\alpha \in AT_{imp}$ and each $\xi \in \varphi$, where φ is the (conjunctive) guard of the transition that should be accelerated. Again, proving validity of an implication is equivalent to proving unsatisfiability of its negation. So if validity of $encode_{\alpha,\xi}$ can be shown, then SMT solvers can also provide an *unsat core* for $\neg encode_{\alpha,\xi}$.

Definition 5 (Unsat Core). *Given a conjunction ψ, we call each unsatisfiable subset of ψ an* unsat core *of ψ.*

Thm. 6 shows that when handling an inequation ξ, one only has to require n-invariance for the elements of $\varphi \setminus \{\xi\}$ that occur in an unsat core of $\neg encode_{\alpha,\xi}$. Thus, an unsat core of $\neg encode_{\alpha,\xi}$ can be used to determine which prerequisites $\check{\varphi}$ are needed for the inequation ξ. This information can then be used to find a suitable order for handling the inequations of the guard. Thus, in this way one only has to check (un)satisfiability of the $4 \cdot m$ formulas $\neg encode_{\alpha,\xi}$. If no such order is found, then LoAT cannot accelerate the loop under consideration.

Theorem 6 (Unsat Core Induces $\rightsquigarrow$-Step). *Let $deps_{\alpha,\xi}$ be the intersection of $\varphi \setminus \{\xi\}$ and an unsat core of $\neg encode_{\alpha,\xi}$. If $\check{\varphi}$ implies $deps_{\alpha,\xi}$, then $accel_\alpha(\xi, \check{\varphi}, \vec{a}) = accel_\alpha(\xi, \varphi \setminus \{\xi\}, \vec{a})$.*

Example 7 (Controlling Acceleration Steps via Unsat Cores). Reconsider Ex. 4. Here, LoAT would try to prove, among others, the following implications:

$$encode_{dec,x>0} \quad = \quad (x - y > 0 \wedge y > 0) \implies x > 0 \tag{1}$$

$$encode_{inc,y>0} \quad = \quad (y > 0 \wedge x > 0) \implies y > 0 \tag{2}$$

To do so, it would try to prove unsatisfiability of $\neg encode_{\alpha,\xi}$ via SMT. For (1), we get $\neg encode_{dec,x>0} = (x - y > 0 \wedge y > 0 \wedge x \leq 0)$, whose only unsat core is $\neg encode_{dec,x>0}$, and its intersection with $\varphi \setminus \{x > 0\} = \{y > 0\}$ is $\{y > 0\}$.

For (2), we get $\neg encode_{inc,y>0} = (y > 0 \wedge x > 0 \wedge y \leq 0)$, whose minimal unsat core is $y > 0 \wedge y \leq 0$, and its intersection with $\varphi \setminus \{y > 0\} = \{x > 0\}$ is empty. So by Thm. 6, we have $accel_{inc}(y > 0, \top, \vec{a}) = accel_{inc}(y > 0, x > 0, \vec{a})$.

In this way, validity of $encode_{\alpha_1,x>0}$ and $encode_{\alpha_2,y>0}$ is proven for all $\alpha_1 \in AT_{imp} \setminus \{inc\}$ and all $\alpha_2 \in AT_{imp}$. However, the premise $x \leq x - y \wedge y > 0$ of $encode_{ev\text{-}inc,x>0}$ is unsatisfiable and thus a corresponding acceleration step would yield a transition with unsatisfiable guard. To prevent that, LoAT only uses a technique $\alpha \in AT_{imp}$ for ξ if the premise of $encode_{\alpha,\xi}$ is satisfiable.

[2] Here and in the following, we unify conjunctions of atoms with sets of atoms.

So for each inequation ξ from φ, LoAT synthesizes up to 4 potential $\rightsquigarrow$-steps corresponding to $accel_\alpha(\xi, deps_{\alpha,\xi}, \vec{a})$, where $\alpha \in AT_{imp}$. If validity of $encode_{\alpha,\xi}$ cannot be shown for any $\alpha \in AT_{imp}$, then LoAT tries to prove satisfiability of $accel_{fp}(\xi, \top, \vec{a})$ to see if **Fixpoint** should be applied. Note that the 2^{nd} argument of $accel_{fp}$ is irrelevant, i.e., **Fixpoint** does not benefit from previous acceleration steps and thus $\rightsquigarrow$-steps that use it do not have any dependencies.

It remains to find a suitably ordered subset S of m $\rightsquigarrow$-steps that constitutes a successful $\rightsquigarrow$-sequence. In the following, we define $AT = AT_{imp} \cup \{fp\}$ and we extend the definition of $deps_{\alpha,\xi}$ to the case $\alpha = fp$ by defining $deps_{fp,\xi} := \varnothing$.

Lemma 8. *Let $C \subseteq AT \times \varphi$ be the smallest set such that $(\alpha, \xi) \in C$ implies*
(a) if $\alpha \in AT_{imp}$, then $encode_{\alpha,\xi}$ is valid and its premise is satisfiable,
(b) if $\alpha = fp$, then $accel_{fp}(\xi, \top, \vec{a})$ is satisfiable, and
(c) $deps_{\alpha,\xi} \subseteq \{\xi' \mid (\alpha', \xi') \in C \text{ for some } \alpha' \in AT\}$.
Let $S := \{(\alpha, \xi) \in C \mid \alpha \geq_{AT} \alpha' \text{ for all } (\alpha', \xi) \in C\}$ where $>_{AT}$ is the total order inc $>_{AT}$ dec $>_{AT}$ ev-dec $>_{AT}$ ev-inc $>_{AT}$ fp. We define $(\alpha', \xi') \prec (\alpha, \xi)$ if $\xi' \in deps_{\alpha,\xi}$. Then $\prec$ is a strict (and hence, well-founded) order on S.

The order $>_{AT}$ in Lemma 8 corresponds to the order proposed in [10]. Note that the set C can be computed without further (potentially expensive) SMT queries by a straightforward fixpoint iteration and well-foundedness of $\prec$ follows from minimality of C. For Ex. 7, we get

$$C = \{(dec, x > 0), (ev\text{-}dec, x > 0)\} \cup \{(\alpha, y > 0) \mid \alpha \in AT\} \qquad \text{and}$$
$$S = \{(dec, x > 0), (inc, y > 0)\} \text{ with } (inc, y > 0) \prec (dec, x > 0).$$

Finally, we can construct a valid $\rightsquigarrow$-sequence via the following theorem.

Theorem 9 (Finding $\rightsquigarrow$-Sequences). *Let S be defined as in Lemma 8 and assume that for each $\xi \in \varphi$, there is an $\alpha \in AT$ such that $(\alpha, \xi) \in S$. W.l.o.g., let $\varphi = \bigwedge_{i=1}^{m} \xi_i$ where $(\alpha_1, \xi_1) \prec' \ldots \prec' (\alpha_m, \xi_m)$ for some strict total order $\prec'$ containing $\prec$, and let $\breve{\varphi}_j := \bigwedge_{i=1}^{j} \xi_i$. Then for all $j \in [0, m)$, we have:*

$$\left[\!\!\left[\bigwedge_{i=1}^{j} accel_{\alpha_i}(\xi_i, \breve{\varphi}_{i-1}, \vec{a}) \;\middle|\; \breve{\varphi}_j \;\middle|\; \bigwedge_{i=j+1}^{m} \xi_i \right]\!\!\right]_{\vec{a}} \rightsquigarrow \left[\!\!\left[\bigwedge_{i=1}^{j+1} accel_{\alpha_i}(\xi_i, \breve{\varphi}_{i-1}, \vec{a}) \;\middle|\; \breve{\varphi}_{j+1} \;\middle|\; \bigwedge_{i=j+2}^{m} \xi_i \right]\!\!\right]_{\vec{a}}$$

In our example, we have $\prec' = \prec$ as $\prec$ is total. Thus, we obtain a $\rightsquigarrow$-sequence by first processing $y > 0$ with **Increase** and then processing $x > 0$ with **Decrease**.

6 Proving Non-Termination of Simple Loops

To prove non-termination, LoAT uses a variation of the calculus from Sect. 4, see [11]. To adapt it for proving non-termination, further restrictions have to be imposed on the conditional acceleration techniques, resulting in the notion of *conditional non-termination techniques*, see [11, Def. 10]. We denote a $\rightsquigarrow$-step that uses a conditional non-termination technique with $\rightsquigarrow_{nt}$.

Theorem 10 (Proving Non-Termination via $\rightsquigarrow_{nt}$). *Let $f(\vec{x}) \to f(\vec{a})\,[\varphi] \in \mathcal{T}$. If $[\![\top \mid \top \mid \varphi]\!]_{\vec{a}} \rightsquigarrow_{nt}^{*} [\![\psi \mid \varphi \mid \top]\!]_{\vec{a}}$, then for every $\vec{c} \in \mathbb{Z}^d$ where $\psi(\vec{c})$ is satisfiable, the configuration $f(\vec{c})$ admits an infinite $\to_{\mathcal{T}}$-sequence.*

The conditional non-termination techniques used by LoAT are **Increase**, **Eventual Increase**, and **Fixpoint**. So non-termination proofs can be synthesized while trying to accelerate a loop with very little overhead. After successfully accelerating a loop as explained in Sect. 5, LoAT tries to find a second suitably ordered $\rightsquigarrow$-sequence, where it only considers the conditional non-termination techniques mentioned above. If LoAT succeeds, then it has found a $\rightsquigarrow_{nt}$-sequence which gives rise to a proof of non-termination via Thm. 10.

7 Implementation, Experiments and Conclusion

Our implementation in LoAT can parse three widely used formats for ITSs (see [13]), and it is configurable via a minimalistic set of command-line options:
`--timeout` to set a timeout in seconds
`--proof-level` to set the verbosity of the proof output
`--plain` to switch from colored to monochrome proof-output
`--limit-strategy` to choose a strategy for solving limit problems, see [9]
`--mode` to choose an analysis mode for LoAT (`complexity` or `non_termination`)
We evaluate three versions of LoAT: LoAT '19 uses templates to find invariants that facilitate loop acceleration for proving non-termination [8]; LoAT '20 deduces worst-case lower bounds based on loop acceleration via *metering functions* [9]; and LoAT '22 applies the calculus from [10,11] as described in Sect. 5 and 6. We also include three other state-of-the-art termination tools in our evaluation: T2 [6], VeryMax [16], and iRankFinder [3,7]. Regarding complexity, the only other tool for worst-case lower bounds of ITSs is LOBER [1]. However, we do not compare with LOBER, as it only analyses (multi-path) loops instead of full ITSs.

We use the examples from the categories *Termination* (1222 examples) and *Complexity of ITSs* (781 examples), respectively, of the *Termination Problems Data Base* [19]. All benchmarks have been performed on *StarExec* [18] (Intel Xeon E5-2609, 2.40GHz, 264GB RAM [17]) with a timeout of 300 s.

	No	Yes	Avg. Rt	Median Rt	Std. Dev. Rt
LoAT '22	493	0	9.4	0.2	41.5
LoAT '19	459	0	22.6	1.5	67.5
T2	438	610	22.6	1.2	66.7
VeryMax	419	628	29.9	1.0	66.7
iRankFinder	399	634	44.1	4.9	89.1

		LoAT '22						
	$rc_{\mathcal{T}}(n)$	$\Omega(1)$	$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^{>2})$	EXP	$\Omega(\omega)$
LoAT '20	$\Omega(1)$	180	63	1	–	–	12
	$\Omega(n)$	6	218	3	–	–	–
	$\Omega(n^2)$	–	1	69	–	–	–
	$\Omega(n^{>2})$	–	–	–	7	–	–
	EXP	1	–	–	–	4	–
	$\Omega(\omega)$	–	–	–	–	–	216

By the table on the left, LoAT '22 is the most powerful tool for non-termination. The improvement over LoAT '19 demonstrates that the calculus from [10,11] is more powerful and efficient than the approach from [8]. The last three columns show the average, the median, and the standard deviation of the runtime, including examples where the timeout was reached.

The table on the right shows the results for complexity. The diagonal corresponds to examples where LoAT '20 and LoAT '22 yield the same result. The entries above or below the diagonal correspond to examples where LoAT '22 or LoAT '20 is better, respectively. There are 8 regressions and 79 improvements, so the calculus from [10,11] used by LoAT '22 is also beneficial for lower bounds.

LoAT is open source and its source code is available on GitHub [12]. See [13,14] for details on our evaluation, related work, all proofs, and a pre-compiled binary.

References

1. Albert, E., Genaim, S., Martin-Martin, E., Merayo, A., Rubio, A.: Lower-bound synthesis using loop specialization and Max-SMT. In: CAV 21. pp. 863–886. LNCS 12760 (2021). https://doi.org/10.1007/978-3-030-81688-9_40
2. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. CoRR **abs/cs/0512056** (2005), https://arxiv.org/abs/cs/0512056
3. Ben-Amram, A.M., Doménech, J.J., Genaim, S.: Multiphase-linear ranking functions and their relation to recurrent sets. In: SAS '19. pp. 459–480. LNCS 11822 (2019). https://doi.org/10.1007/978-3-030-32304-2_22
4. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: TACAS '09. pp. 337–351. LNCS 5505 (2009). https://doi.org/10.1007/978-3-642-00768-2_29
5. Bozga, M., Iosif, R., Konecný, F.: Fast acceleration of ultimately periodic relations. In: CAV '10. pp. 227–242. LNCS 6174 (2010). https://doi.org/10.1007/978-3-642-14295-6_23
6. Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H., Piterman, N.: T2: temporal property verification. In: TACAS '16. pp. 387–393. LNCS 9636 (2016). https://doi.org/10.1007/978-3-662-49674-9_22
7. Doménech, J.J., Genaim, S.: iRankFinder. In: WST '18. p. 83 (2018), http://wst2018.webs.upv.es/wst2018proceedings.pdf
8. Frohn, F., Giesl, J.: Proving non-termination via loop acceleration. In: FMCAD '19. pp. 221–230 (2019). https://doi.org/10.23919/FMCAD.2019.8894271
9. Frohn, F., Naaf, M., Brockschmidt, M., Giesl, J.: Inferring lower runtime bounds for integer programs. ACM TOPLAS **42**(3), 13:1–13:50 (2020). https://doi.org/10.1145/3410331, revised and extended version of a paper which appeared in IJCAR '16, pp. 550–567, LNCS 9706 (2016).
10. Frohn, F.: A calculus for modular loop acceleration. In: TACAS '20. pp. 58–76. LNCS 12078 (2020). https://doi.org/10.1007/978-3-030-45190-5_4
11. Frohn, F., Fuhs, C.: A calculus for modular loop acceleration and non-termination proofs. CoRR **abs/2111.13952** (2021), https://arxiv.org/abs/2111.13952, to appear in STTT.
12. Frohn, F.: LoAT on GitHub, https://github.com/aprove-developers/LoAT
13. Frohn, F., Giesl, J.: Empirical evaluation of: Proving non-termination and lower runtime bounds with LoAT, https://ffrohn.github.io/loat-tool-paper-evaluation
14. Frohn, F., Giesl, J.: Proving non-termination and lower runtime bounds with LoAT (System Description). CoRR **abs/2202.04546** (2022), https://arxiv.org/abs/2202.04546
15. Kroening, D., Lewis, M., Weissenbacher, G.: Under-approximating loops in C programs for fast counterexample detection. FMSD **47**(1), 75–92 (2015). https://doi.org/10.1007/s10703-015-0228-1
16. Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Proving non-termination using Max-SMT. In: CAV '14. pp. 779–796. LNCS 8559 (2014). https://doi.org/10.1007/978-3-319-08867-9_52
17. StarExec hardware specifications, https://www.starexec.org/starexec/public/machine-specs.txt
18. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: IJCAR '14. pp. 367–373. LNCS 8562 (2014). https://doi.org/10.1007/978-3-319-08587-6_28
19. Termination Problems Data Base (TPDB, Git SHA 755775), https://github.com/TermCOMP/TPDB