Proving Non-Termination via Loop Acceleration

Florian Frohn
Max Planck Institute for Informatics, Saarbrücken, Germany
florian.frohn@mpi-inf.mpg.de

Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Germany
giesl@informatik.rwth-aachen.de

Abstract—We present the first approach to prove non-termination of integer programs that is based on loop acceleration. If our technique cannot show non-termination of a loop, it tries to accelerate it instead in order to find paths to other non-terminating loops automatically. The prerequisites for our novel loop acceleration technique generalize a simple yet effective non-termination criterion. Thus, we can use the same program transformations to facilitate both non-termination proving and loop acceleration. In particular, we present a novel invariant inference technique that is tailored to our approach. An extensive evaluation of our fully automated tool LoAT shows that it is competitive with the state of the art.

I. INTRODUCTION

Proving non-termination of integer programs is an important research topic (e.g., [2, 7, 13, 14, 27, 33, 34, 35, 40, 41]). In another line of research, under-approximating loop acceleration is used to analyze safety [31] and runtime complexity [21]. Here, the idea is to replace a loop by code that mimics $k$ loop iterations, where $k$ is chosen non-deterministically.

Many non-termination techniques first search for a diverging configuration and then prove its reachability. For the latter, loop acceleration would be useful, as it allows reasoning about paths with loops without fixing the number of unrollings. Still, up to now acceleration has not been used for non-termination proving.

To fill this gap, we design a novel loop acceleration technique whose prerequisites generalize a well-known non-termination criterion. This correspondence is of great value: It allows us to develop an under-approximating program simplification framework that progresses incrementally towards the detection of non-terminating loops and the acceleration of other loops.

After introducing preliminaries in Sect. II, we present our approach in Sect. III and IV. It eliminates loops via acceleration and chaining, or by proving their non-termination and replacing them by a transition to a special symbol $\omega$. If a loop cannot be eliminated, then we strengthen its guard by synthesizing suitable invariants. Our approach also handles nested loops by eliminating inner loops before removing outer loops. Eventually, this leads to a loop-free program where a trace to $\omega$ yields a witness of non-termination. So our main contributions are:

(a) The applicability of existing under-approximating loop acceleration techniques is restricted: The technique from [31] is often inapplicable if the loop condition contains invariants and the technique from [21] requires metering functions which are often challenging to synthesize. Thus, in Sect. III we present a novel loop acceleration technique that generalizes [31] and does not require metering functions, and we integrate it into a program simplification framework inspired by [21].
(b) We combine our approach with a novel invariant inference technique in Sect. IV. So if the prerequisites of our non-termination criterion and our acceleration technique are violated, then we try to deduce invariants to make them applicable.

From a practical point of view, we contribute:

(c) an implementation in our open-source tool LoAT

(d) an extensive evaluation of our implementation, cf. Sect. V.

Finally, Sect. VI discusses related work and concludes. All proofs can be found in [22].

II. PRELIMINARIES

We denote vectors $x$ by bold letters and the $i^{th}$ element of $x$ by $x_i$. Transitions $\alpha$ have the form $f(x) \rightarrow g(t)$ [\eta]. The left-hand side $\text{lhs}_\alpha = f(x)$ consists of $\alpha$’s source function symbol $\text{src}_\alpha = f \in \Sigma$ and a vector of pairwise different variables $x \in \mathcal{V}$ ranging over $\mathbb{Z}$, where $\mathcal{V}$ is countably infinite. The set of function symbols $\Sigma$ is finite and we assume that all function symbols have the same arity (otherwise one can add unused arguments). We use $\mathcal{V}(\cdot)$ to denote all variables occurring in the argument. $\mathcal{A}$ denotes the set of all arithmetic expressions over $\mathcal{V}$, i.e., expressions built from variables, numbers, and arithmetic operations like $\cdot$, $\cdot\cdot$, etc. The guard $\text{guard}_\alpha = \eta$ is a constraint, i.e., a finite conjunction\(^1\) of inequations over $\mathcal{A}$, which we omit if it is empty. The right-hand side $\text{rhs}_\alpha = g(t)$ consists of $\alpha$’s destination $\text{dest}_\alpha = g \in \Sigma$ and a vector $t \in \mathcal{A}$. The substitution $u_{\alpha}(x) = \{x \mapsto t\}$ is $\alpha$’s update.

A substitution is a function $\sigma : \mathcal{V} \rightarrow \mathcal{A}$. The domain of $\sigma$ is $\text{dom}(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ and its range is defined as $\text{rng}(\sigma) = \{\sigma(x) \mid x \in \text{dom}(\sigma)\}$. We sometimes denote substitutions by sets of key-value pairs $\{y_1 \mapsto t_1, \ldots, y_k \mapsto t_k\}$ or just $\{y \mapsto t\}$. Then each $x \in \mathcal{V} \setminus y$ is mapped to itself. For every entity $e$, $\sigma(e)$ results from replacing all free variables in $e$ according to $\sigma$. If $\text{rng}(\sigma) \subset \mathbb{Z}$, then $\sigma$ is a valuation. A first-order formula $\varphi$ is valid if it is equivalent to true. Moreover, a valuation $\sigma$ is a model of $\varphi$ (or satisfies $\varphi$, denoted $\sigma \models \varphi$) if $\sigma$’s domain contains all free variables of $\varphi$ and $\sigma(\varphi)$ is valid.

An integer program $\mathcal{T}$ is a finite set of transitions. Their guards restrict the control flow, i.e., $f(x) \rightarrow g(t)$ [\eta] is only

\(^1\) Note that negations can be expressed by negating inequations directly, and disjunctions in programs can be expressed using several transitions.
applicable if the current valuation of the variables satisfies $\eta$.

**Example 1 (Integer Program).** Consider the function `start`:
```python
def start(x, y):
    while x >= 0: x = x - y; y = y + 1
    while y > 0: y = y - x
```

It corresponds to the following integer program:
\[
\begin{align*}
\alpha_1 : & \quad \text{start}(x, y) \rightarrow f(x, y) \\
\alpha_2 : & \quad f(x, y) \rightarrow f(x - y, y + 1) \quad [x \geq 0] \\
\alpha_3 : & \quad f(x, y) \rightarrow g(x, y) \quad [x < 0] \\
\alpha_4 : & \quad g(x, y) \rightarrow g(x, y - x) \quad [y > 0]
\end{align*}
\]
The function symbols $f$ and $g$ represent the first and the second loop, respectively. The program does not terminate if, e.g., $x$ and $y$ are initially $0$. After applying the first loop twice, $y$ is 2 and $x$ is $-1$, so that the second loop diverges.

**Definition 2 (Integer Transition Relation).** A term $f(n)$ where $n \in \mathbb{Z}$ is a configuration. An integer program $T$ induces a relation $\rightarrow_{T}$ on configurations: We have $s \rightarrow_{T} t$ if there is an $\alpha \in T$ and a model $\sigma$ of guard$_{\alpha}$ such that $\forall(\alpha) \subseteq \text{dom}(\sigma)$, $\sigma(\text{lhs}_{\alpha}) = s$, and $\sigma(\text{rhs}_{\alpha}) = t$.\(^2\) Then we say that $s$ evaluates to $t$. As usual, $\rightarrow_{T}^{*}$ is the transitive-reflexive closure of $\rightarrow_{T}$.

If there is an infinite $\rightarrow_{T}$-evaluation that starts with $\text{start}(n)$ where $\text{start} \in \Sigma$ is the canonical start symbol, then $T$ is non-terminating and $\text{start}(n)$ witnesses non-termination of $T$.

W.l.o.g., start does not occur on right-hand sides. Otherwise, one can rename start to start$'$ and add a transition $\text{start}(x) \rightarrow \text{start'}(x)$. A program $T$ is simplified if src$_{\alpha}$ = start for all $\alpha \in T$. So any run of a simplified program has at most length one.

By definition, integer programs may contain transitions like $f(x) \rightarrow f(\frac{x}{2})$. While evaluations that would not yield integers get stuck (as, e.g., $f(\frac{1}{2})$ is not a configuration), our technique assumes that the arguments of functions are always integers. Hence, we restrict ourselves to well-formed integer programs.

**Definition 3 (Well-Formedness).** An integer program $T$ is well formed if for all transitions $\alpha \in T$ and all models $\sigma$ of guard$_{\alpha}$ with $\forall(\alpha) \subseteq \text{dom}(\sigma)$, $\sigma(\text{lhs}_{\alpha})$ is a configuration.

To ensure that the program is initially well-formed, we just allow integers, addition, subtraction, and multiplication in the original program.\(^3\) While our approach uses program transformations that may introduce further operations like division and exponentials, these transformations preserve well-formedness. We formalize our contributions in terms of processors.

**Definition 4 (Processor).** Let $\omega \in \Sigma$ be a dedicated fresh function symbol. A processor proc is a partial function which maps integer programs to integer programs. It is sound if the following holds for all $T$ where proc is defined:
\[
\begin{align*}
\text{if } & \quad \text{start}(n) \rightarrow_{\text{proc}(T)}^{*} \omega \quad \text{or} \\
\text{start}(n) & \quad \text{witnesses non-termination of proc(T)},
\end{align*}
\]
\(^2\)Throughout the paper, we use "=" for semantic (not syntactic) equality w.r.t. arithmetic, e.g., $-t(1 + 2) = f(3)$ holds.
\(^3\)One could also allow expressions like $\frac{1}{2} x^2 + \frac{1}{2} x$ in the initial program, as long as every arithmetic expression maps integers to integers.

then $\text{start}(n) \rightarrow_{\text{proc}(T)}^{*} \omega$ or $\text{start}(n)$ witnesses non-termination of proc(T).

If proc preserves well-formedness, then proc is called safe.

So we use the symbol $\omega$ to represent non-termination (and we omit its arguments for readability): If we can transform a program $T$ into a simplified program $T'$ via safe and sound processors and $\sigma \models \text{guard}_{\alpha}$ for some $\alpha \in T'$ with $\text{rhs}_{\alpha} = \omega$, then $\sigma(\text{lhs}_{\alpha})$ witnesses non-termination of $T$ due to Def. 4.

### III. Simplifying Integer Programs

We now present our Contribution (a) by defining suitable processors. In Sect. III-A, we introduce the notions of invariants which are the foundation of our loop acceleration technique, cf. Sect. III-B. The remaining processors of our approach are used to combine transitions (Sect. III-C) and to finally deduce non-termination (Sect. III-D).

**A. Invariants:** Our novel loop acceleration technique relies on the following notions of invariants. Here, "\(\forall \mathcal{V}(T)\). \psi" abbreviates "\(\forall (\mathcal{V}(\psi) \land \mathcal{V}(T))\). \psi\", i.e., the quantifier binds all free variables of $\psi$ that occur in $T$.

**Definition 5 (Invariants).** Let $\alpha \in T$. If
\[
\forall \mathcal{V}(T). \text{guard}_{\alpha} \land \varphi_{ci} \implies u\uparrow \alpha(\varphi_{ci}) \tag{ci}
\]
is valid, then $\varphi_{ci}$ is a conditional invariant of $\alpha$. If
\[
\forall \mathcal{V}(T). \varphi_{si} \implies u\uparrow \alpha(\varphi_{si}) \tag{si}
\]
is valid, then $\varphi_{si}$ is a simple (conditional) invariant of $\alpha$. If $\varphi_{si}$ is a simple invariant of $\alpha$ and
\[
\forall \mathcal{V}(T). \varphi_{si} \land u\uparrow \alpha(\varphi_{md}) \implies \varphi_{md} \tag{md}
\]
is valid, then $\varphi_{md}$ is monotonically decreasing for $\varphi_{si}$ and $\alpha$.

Recall that $\varphi$ is a (standard) invariant of a transition $\alpha$ if $\varphi$ holds whenever $\alpha$ is applied in a program run. If such a standard invariant $\varphi$ satisfies (ci), then $\varphi$ is usually called inductive. In contrast to inductive invariants, a conditional invariant $\varphi_{ci}$ does not have to hold when the control flow reaches $\alpha$, but if it does, then $\varphi_{ci}$ still holds after applying $\alpha$. Conditional invariants (resp. similar notions) are also used in e.g., [5, 9, 32, 33, 41]. Monotonic decreasingness is converse to invariance: $\varphi_{md}$ is preserved when the effect of $u\uparrow \alpha$ is undone.

We call constraints of the form $\varphi_{ci} \land \varphi_{si} \land \varphi_{md}$ monotonic if $\varphi_{ci}$ and $\varphi_{si}$ are conditional and simple invariants, and $\varphi_{md}$ is monotonically decreasing for $\varphi_{si}$. The reason is that the characteristic function $[\varphi]$ with $[\varphi] = 1 \iff \varphi$ and $[\varphi] = 0 \iff \neg \varphi$ of conditional invariants like $\varphi_{ci}$ and $\varphi_{si}$ is monotonically increasing w.r.t. $u\uparrow \alpha$ and (md) essentially requires that $[\varphi_{md}]$ is monotonically decreasing w.r.t. $u\uparrow \alpha$.

**Example 6 (Invariants).** For $\alpha$ from Ex. 1, $y \geq 0$ is a simple invariant and $x \geq 0$ is monotonically decreasing for $y \geq 0$, as
\[
\forall x, y. y \geq 0 \quad \implies \quad y + 1 \geq 0 \quad \text{and} \quad \forall x, y. y \geq 0 \land x - y \geq 0 \quad \implies \quad x \geq 0
\]
are valid. Thus, $y \geq 0$ is also a conditional invariant. Note
that it is not a standard invariant as there are program runs where $y \geq 0$ is violated when $\alpha_2$ is applied.

**B. Loop Acceleration:** The key idea of loop acceleration for a simple loop, i.e., a transition $\alpha$ with $\text{src}_\alpha = \text{dest}_\alpha$, is to generate a new transition $\pi$ that captures $k$ iterations of $\alpha$. Here, $k$ is a fresh variable whose value can be chosen nondeterministically. We first use recurrence solving to compute closed forms for the values of the program variables after a symbolic number of iterations, i.e., a closed form of $u_\alpha^{k} = u_\alpha \circ \ldots \circ u_\alpha$. Then, as in [31], we exploit the following observation: If $\text{guard}_\alpha$ holds after $k - 1$ loop iterations and $u_\alpha(\text{guard}_\alpha)$ implies $\text{guard}_\alpha$ (i.e., $\text{guard}_\alpha$ is monotonically decreasing), then $\text{guard}_\alpha$ also holds after $k - 2, k - 3, \ldots, 0$ iterations. Thus, adding $u_\alpha^{k-1}(\text{guard}_\alpha)$ to $\text{guard}_\alpha$ ensures that $k$ only takes feasible values: If $\sigma$ satisfies $u_\alpha^{k-1}(\text{guard}_\alpha)$, then $\alpha$ can be iterated at least $\sigma(k)$ times.

However, $u_\alpha(\text{guard}_\alpha) \implies \text{guard}_\alpha$ is rarely valid if $\text{guard}_\alpha$ contains invariants of $\alpha$. Thus, our novel loop acceleration technique only requires monotonicity of $\text{guard}_\alpha$ instead.

**Theorem 7 (Accelerate).** Let $\mathcal{T}$ be well formed, let $\alpha \in \mathcal{T}$ be a simple loop with $\text{lhs}_\alpha = f(x)$, let $k \in \mathbb{V}$ be fresh, and let $\mu$ be a substitution such that $\mu(x) = u_\alpha^{k}(x)$ holds for all $k > 0$. Moreover, let $\text{guard}_\alpha = \varphi_{ci} \land \varphi_{si} \land \varphi_{md}$ be monotonic. Finally, let $\text{dec}_k = \{ k \mapsto k - 1 \}$ and $\mathcal{T}' = \mathcal{T} \cup \{ \pi \}$ where $\pi = [f(x) \rightarrow f(\mu(x)), \varphi_{ci} \land \varphi_{si} \land \text{dec}_k(\mu(\varphi_{md})) \land k > 0].$ Then the processor Accelerate: $\mathcal{T} \rightarrow \mathcal{T}'$ is safe and sound.

So to construct $\text{rhs}_{\pi}$, we compute a closed form $\mu$ that expresses $k$ iterations of the loop body as in [21, 31]. To do so, one can use state-of-the-art recurrence solvers like [1, 28, 42] to solve the system of recurrence relations $x^{(k+1)} = u_\alpha(x^{(k)})$ with the initial condition $x^{(1)} = u_\alpha(x)$.

To see why Accelerate is sound, assume that $\text{guard}_\alpha$ holds. As $\varphi_{si} \subseteq \text{guard}_\alpha$ and $\varphi_{si}$ implies $u_\alpha(\varphi_{si})$ by (si), we obtain $u_\alpha^n(\varphi_{si})$ for all $n \in \mathbb{N}.$ (1) Thus, as $\text{guard}_\pi$ contains $\text{dec}_k(\mu(\varphi_{md})) = u_\alpha^{k-1}(\varphi_{md})$ and $\varphi_{si} \land \text{dec}_k(\mu(\varphi_{md}))$ implies $\varphi_{md}$ by (md), we get $u_\alpha^n(\varphi_{md})$ for all $0 \leq n < k$. (2) So (1) and (2) imply $\varphi_{si} \land \varphi_{md}$. As $\varphi_{si} \subseteq \text{guard}_\alpha$ and $\text{guard}_\alpha = \varphi_{ci} \land \varphi_{si} \land \varphi_{md}$, this means that $\text{guard}_\alpha$ holds as well. As $\text{guard}_\alpha$ implies $u_\alpha(\varphi_{ci})$ (since $\varphi_{ci} \subseteq \text{guard}_\alpha$ and $\varphi_{ci}$ is a conditional invariant), we obtain that $u_\alpha(\varphi_{ci})$ holds. Together with (1) and (2) this means that $u_\alpha(\text{guard}_\alpha)$ holds (if $1 < k$). This in turn implies $u_\alpha^n(\varphi_{ci})$, etc. Thus, we get $u_\alpha^n(\varphi_{ci})$ for all $0 \leq n < k$. (3) Due to (1) – (3), the constraint $\varphi_{ci} \land \varphi_{si} \land \text{dec}_k(\mu(\varphi_{md}))$ ensures that $\text{guard}_\pi = \varphi_{ci} \land \varphi_{si} \land \varphi_{md}$ holds before the $1^{st}, \ldots, k^{th}$ iteration, as desired. Hence, every evaluation with $\pi$ can be replaced by $k$ evaluation steps with $\alpha$. Since $\text{guard}_{\pi}$ enforces $k > 0$, every non-terminating run with $\pi$ can therefore be transformed into a non-terminating run of $\mathcal{T}$.

**Example 8 (Ex. 1 continued).** Consider the simple loop $\alpha_2$ of Ex. 1. As $x \geq 0$ is not monotonic, Accelerate is not applicable. But if we strengthen the guard by adding the simple invariant $y \geq 0$, then $x \geq 0$ satisfies (md), cf. Ex. 6. Thus, we can apply Accelerate with $\varphi_{ci} : \text{true}$, $\varphi_{si} : y \geq 0$, and $\varphi_{md} : x \geq 0.$ Sect. IV will show how to find simple invariants like $y \geq 0$.

To compute a substitution $\mu$ that represents $k$ repeated updates, we solve the recurrence relations $y^{(k+1)} = y^{(k)} + 1$ and $x^{(k+1)} = x^{(k)} - y^{(k)}$ with the initial conditions $x^{(1)} = x - y$ and $y^{(1)} = y + 1$, resulting in the solutions $y^{(k)} = y + k$ and $x^{(k)} = x - y \cdot k - \frac{1}{2} \cdot k^2 + \frac{1}{2} \cdot k$, i.e., $\mu = \{ x \mapsto x - y \cdot k - \frac{1}{2} \cdot k^2 + \frac{1}{2} \cdot k, y \mapsto y + k \}$. Thus, we accelerate $\alpha_2$ to $\alpha_2': \begin{array}{rcl} f(x, y) & \rightarrow & f(x - y \cdot k - \frac{1}{2} \cdot k^2 + \frac{1}{2} \cdot k, y + k) \end{array}$

for $\eta : y \geq 0 \land \text{dec}_k(\mu(\varphi_{md})) \land k > 0$ where $\text{dec}_k(\mu(\varphi_{md}))$ is $x - y \cdot (k - 1) - \frac{1}{2} \cdot (k - 1)^2 + \frac{1}{2} \cdot (k - 1) \geq 0.$

**C. Chaining:** Accelerate only applies to simple loops. To transform loops with complex control flow into simple loops and to eventually obtain simplified programs, we use chaining, a standard technique to combine two transitions $f(\ldots) \rightarrow g(\ldots)$ and $g(\ldots) \rightarrow h(\ldots)$ to a new transition $f(\ldots) \rightarrow h(\ldots)$ that captures the effect of both transitions after each other.

**Theorem 9 (Chain).** Let $\mathcal{T}$ be well formed and let $\alpha, \beta \in \mathcal{T}$ where $\text{dest}_\alpha = \text{src}_\beta$, the argument lists of $\text{lhs}_\alpha$ and $\text{lhs}_\beta$ are equal, and $\mathcal{V}(\alpha) \cap \mathcal{V}(\beta) = \mathcal{V}(\text{lhs}_\alpha).$ Let $\mathcal{T}^\circ = \mathcal{T} \cup \{ \alpha \circ \beta \}$ with $\alpha \circ \beta = \text{lhs}_\alpha \rightarrow u_\alpha(\text{rhs}_\beta) \left[ \text{guard}_\alpha \land u_\alpha(\text{guard}_\beta) \right].$ Then the processor Chaining: $\mathcal{T} \rightarrow \mathcal{T}^\circ$ is safe and sound.

Chaining is not only useful to transform complex into simple loops, but it can also be used to combine a simple loop $\alpha$ with itself in order to enable loop acceleration and to obtain better closed forms for $u_\alpha^n$. For example, consider the following loop, where the sign of $x$ alternates:

$\alpha_{neg} : \begin{array}{rcl} f(x, y) & \rightarrow & f(-x, y - 1) \end{array}$

$\{ y > x \}$

The closed form $(-1)^k \cdot x$ for the value of $x$ after $k$ iterations involves exponentials even though $x$ does not grow exponentially. This is disadvantageous, as our implementation relies on SMT solving, but SMT solvers have limited support for non-polynomial arithmetic. Moreover, Accelerate is not applicable, as $y > x$ is non-monotonic. However, this can be resolved by chaining $\alpha_{neg}$ with itself, which results in $\alpha_{neg} \circ \alpha_{neg} : \begin{array}{rcl} f(x, y) & \rightarrow & f(x, y - 2) \end{array}$

$\{ y > x \land y - 1 > x \}$

This transition can be accelerated to $f(x, y) \rightarrow f(x, y - 2 \cdot k)$

$\left[ \text{dec}_k(\mu(\text{guard}_{\alpha_{neg} \circ \alpha_{neg}})) \right]$
where \( \text{dec}_k(\mu(\text{guard}_{\alpha_{acg}})) \) is
\[
y - 2 \cdot (k - 1) > x \land y - 2 \cdot (k - 1) - 1 > -x,
\]
i.e., the accelerated transition does not contain exponentials.

So for simple loops \( \alpha \) that alternate the sign of a variable
(i.e., where \( \text{up}_\alpha(x) = c \cdot x + t \) for some \( x \in \mathcal{V}(\text{lhs}_\alpha) \), \( c < 0 \),
and \( t \in \mathcal{A} \) with \( x \notin \mathcal{V}(t) \)), we accelerate \( \alpha \circ \alpha \) instead of \( \alpha \).

Chaining can also help to obtain simpler closed forms for transitions
where variables are set to constants. For example, a closed form for the repeated update of the variable \( z \) in
\[
\alpha_{\text{const}}: \ f(x, y, z) \rightarrow f(x - 1, 2, y) \quad [x > 0]
\]
is \( 0^{k-1} \cdot y + (1 - 0^{k-1}) \cdot 2 \), which is again not polynomial.
However, chaining \( \alpha_{\text{const}} \) with itself yields
\[
\alpha_{\text{const}} \circ \alpha_{\text{const}} : \ f(x, y, z) \rightarrow f(x - 2, 2, 2) \quad [x > 1]
\]
(where we simplified the guard), which can be accelerated to
\[
f(x, y, z) \rightarrow f(x - 2 \cdot k, 2, 2) \quad [x - 2 \cdot (k - 1) > 1 \land k > 0]
\]
i.e., the accelerated transition again only contains polynomials.

Finally, chaining can also make acceleration applicable to
loops that permute arguments:
\[
\alpha_p : \ f(x, y) \rightarrow f(y, x - 1) \quad [x > 0]
\]
While \( \alpha_p \) violates the prerequisites of \( \text{Accelerate} \),
\( \alpha_p \circ \alpha_p : \ f(x, y) \rightarrow f(x - 2, y - 2) \quad [x > 0 \land y - 1 > 0] \)
can be accelerated to:
\[
f(x, y) \rightarrow f(x - 2 \cdot k, y - 2 \cdot k) \quad [\text{dec}_k(\mu(\text{guard}_{\alpha_p})) \land k > 0]
\]
So to handle simple loops \( \alpha \) where some variables “stabilize”
(i.e., \( \text{up}_\alpha(z) \in \mathbb{Z} \) for some \( z \in \mathcal{V} \))
or where arguments are permuted (as in \( \alpha_p \)), we repeatedly chain \( \alpha \) with itself as long as this reduces the size of
\[
\{ x \in \mathcal{V}(\text{lhs}_\alpha) \mid \mathcal{V}(\text{up}_\alpha(x)) \neq \emptyset \land x \notin \mathcal{V}(\text{up}_\alpha(x)) \}. \quad (4)
\]

### D. Proving Non-Termination:

To detect non-terminating simple loops \( \alpha \), we check whether \( \text{guard}_\alpha \) itself is a simple invariant
(i.e., whether the valuations that satisfy \( \text{guard}_\alpha \) correspond to a recurrent set of the relation \( \rightarrow(\alpha) \), cf. [27]).

**Theorem 10 (Nonterm).** Let \( \mathcal{T} \) be well formed and let \( \alpha \in \mathcal{T} \) be
a simple loop such that \( \text{guard}_\alpha \) is a simple invariant. Moreover, let \( \mathcal{T}^\omega = \mathcal{T} \cup \{ \alpha^\omega \} \) where
\[
\alpha^\omega = [\text{lhs}_\alpha \rightarrow \omega \mid \text{guard}_\alpha].
\]
Then the processor \( \text{Nonterm} : \mathcal{T} \rightarrow \mathcal{T}^\omega \) is safe and sound.

**Example 11 (Ex. 1 continued).** Clearly, \( y > 0 \) is not a simple
invariant of \( \alpha_4 \) from Ex. 1. But if we strengthen the guard
by adding the simple invariant \( x \leq 0 \), then \( \text{Nonterm} \) is applicable
as \( y > 0 \land x \leq 0 \) implies \( y - x > 0 \land x \leq 0 \). Thus, we obtain
\[
\alpha_4' : \ g(x, y) \rightarrow \omega \quad [y > 0 \land x \leq 0].
\]
Again, we will see how to deduce suitable simple invariants
like \( x \leq 0 \) automatically in Sect. IV.

In some cases, chaining also helps to make \( \text{Nonterm} \) applicable. To see this, consider the simple loop
\[
\alpha_{nt} : \ f(x, y) \rightarrow f(0, y - x) \quad [y > 0]
\]
where \( y > 0 \) is no simple invariant. Chaining it with itself yields
\[
\alpha_{nt} \circ \alpha_{nt} : \ f(x, y) \rightarrow f(0, y - x) \quad [y > 0 \land y - x > 0].
\]
As \( y > 0 \land y - x > 0 \implies y - x > 0 \land y - x > 0 \) is valid, the prerequisites of \( \text{Nonterm} \) are satisfied and we obtain
\[
f(x, y) \rightarrow \omega \quad [y > 0 \land y - x > 0].
\]
So in general, we try to apply \( \text{Nonterm} \) not only to a simple loop \( \alpha \), but also to \( \alpha \circ \alpha \). Apart from \( \text{Nonterm} \), we also use
SMT solving to check whether a loop has a fixpoint, which is
a standard technique to prove non-termination.

**Theorem 12 (Fixpoint).** Let \( \mathcal{T} \) be well formed, let \( \alpha \in \mathcal{T} \) be
a simple loop with \( \text{lhs}_\alpha = f(x) \), and let \( \text{guard}_\alpha \land x = \text{up}_\alpha(x) \)
be satisfiable. Let \( \mathcal{T}^{fp} = \mathcal{T} \cup \{ \alpha^{fp} \} \) where
\[
\alpha^{fp} = [\text{lhs}_\alpha \rightarrow \omega \mid \text{guard}_\alpha \land x = \text{up}_\alpha(x)].
\]
Then the processor \( \text{Fixpoint} : \mathcal{T} \rightarrow \mathcal{T}^{fp} \) is safe and sound.

For example, \( \{ x \rightarrow 0, y \rightarrow 1 \} \) is a fixpoint of \( \alpha_{nt} \) which
satisfies \( y > 0 \) and \( (x, y) = (0, y - x) \) (i.e., \( x = 0 \land y = y - x \)).

Alg. 1 shows a streamlined version of the strategy that we use
to apply the presented processors. It combines chaining, loop
acceleration, and our non-termination processors to transform
arbitrary programs into simplified programs. For nested loops,
the elimination starts with the inner loops. Note that

deducting transitions (Steps 2, 3, 5, and 18) is always sound in our setting.

**Input:** A program \( \mathcal{T} \)

**Output:** A witness for non-termination of \( \mathcal{T} \) or \( \perp \)

1 while \( \mathcal{T} \) is not simplified :
   2 \( \mathcal{T} \leftarrow \{ \alpha \in \mathcal{T} \mid \text{src}_\alpha \text{ is reachable from start} \} \)
   3 \( \mathcal{T} \leftarrow \{ \alpha \in \mathcal{T} \mid \text{guard}_\alpha \text{ is satisfiable} \} \)
   4 while \( \exists \alpha \in \mathcal{T} \). \alpha \text{ is a simple loop} :
      5 \( \mathcal{T} \leftarrow \mathcal{T} \setminus \{ \alpha \} \)
      6 \( \alpha \leftarrow \alpha \circ \alpha \) if \( \alpha \) alternates the sign of a variable
      7 \( \alpha_{orig} \leftarrow \alpha \)
      8 do \( \alpha \leftarrow \alpha \circ \alpha_{orig} \) while it reduces the size of (4)
      9 if \( \text{Nonterm} \) applies to \( \alpha \) : \( \alpha \rightarrow \alpha^\omega \)
     10 elif \( \text{Nonterm} \) applies to \( \alpha \circ \alpha \) : \( \alpha \leftarrow (\alpha \circ \alpha)^\omega \)
     11 elif \( \text{Fixpoint} \) applies to \( \alpha \) : \( \alpha \rightarrow \alpha^{fp} \)
     12 elif \( \text{Accelerate} \) applies to \( \alpha \) : \( \alpha \rightarrow \alpha^{orig} \)
     13 else : \( \mathcal{T} \leftarrow \mathcal{T} \cup \text{deduceInvariants}(\alpha) \)
     14 \( S \leftarrow S \cup \{ \beta \mid \alpha \rightarrow \beta \in \mathcal{T}, \text{src}_\beta \neq \text{dest}_\beta = \text{src}_\alpha \} \)
     15 \( \mathcal{T} \leftarrow \mathcal{T} \cup S \)
     16 if \( \exists \alpha, \beta \in \mathcal{T} \). \text{dest}_\alpha = \text{src}_\beta = f : \)
     17 \( \mathcal{T} \leftarrow \mathcal{T} \cup \{ \alpha \circ \beta \mid \alpha, \beta \in \mathcal{T}, \text{dest}_\alpha = \text{src}_\beta = f \}
     18 \( \{ \alpha \in \mathcal{T} \mid f \notin \{ \text{src}_\alpha, \text{dest}_\alpha \} \} \)
     19 if \( \exists \alpha \in \mathcal{T} \). \text{rhs}_\alpha = \omega \land \sigma \mid \text{guard}_\alpha : \text{return} \text{\scriptsize \{lhs}_\alpha \}
     20 else : \text{return} \perp

**Algorithm 1:** Proving Non-Termination

We present the algorithm deduceInvariants for Step 13 in
As then it chains

Then the processor Strengthen α in Step 14 where

Nonterm y in Step 14 where

Next, it applies Nonterm to α in Step 9 (see Ex. 11) and chains the resulting transition with α in Step 14, which yields

α ∪ α:

\[ \text{start}(x, y) \rightarrow f(x - y \cdot k - \frac{1}{2} \cdot k^2 + \frac{1}{2} \cdot k, y + k) \] [η]

in Step 14 where η is

\[ y \geq 0 \land x - y \cdot (k - 1) - \frac{1}{2} \cdot (k - 1)^2 + \frac{1}{2} \cdot (k - 1) \geq 0 \land k > 0. \]

Then, it chains α ∪ α in Step 17, resulting in

α ∪ α ∪ α:

\[ \text{start}(x, y) \rightarrow \omega [\psi] \]

where ψ is \( \eta \land x - y \cdot k - \frac{1}{2} \cdot k^2 + \frac{1}{2} \cdot k \land y + k > 0. \)

To prove non-termination, we have to show satisfiability of ψ.

As \( \sigma \models \psi \) for \( \sigma = \{ x \rightarrow 0, y \rightarrow 0, k \rightarrow 2 \} \), the configuration \( \sigma(\text{start}(x, y)) = \text{start}(0, 0) \) witnesses non-termination of Ex. 1.

So loop acceleration introduces a new variable \( k \) for the number of loop unrollings. Later, \( k \) is instantiated when searching for models of the guards of the simplified transitions which result from repeated acceleration and chaining. In Ex. 13, when inferring a model for the guard of α ∪ α ∪ α, the instantiation \( k \rightarrow 2 \) means that α ∪ α is applied twice in the corresponding non-terminating run of the original program.

IV. Deducing Simple Invariants

In Sect. III, we have seen that we sometimes need to deduce suitable simple invariants to apply our novel loop acceleration technique or to prove non-termination. Soundness of adding constraints to transitions is ensured by the following processor.

Theorem 14 (Strengthen [21]). Let \( T \) be well formed, let \( \alpha \in T \), let \( \varphi \) be a constraint, and let \( T^* = T \cup \{ \alpha \} \) where

\[ \alpha^* = \text{lhs}_\alpha \rightarrow \text{rhs}_\alpha [\text{guard}_\alpha \land \varphi] \]

Then the processor Strengthen \( T \rightarrow T^* \) is safe and sound.

The challenge is to find constraints \( \varphi \) that help to prove non-termination. We now explain how to automatically synthesize suitable simple invariants to strengthen a simple loop \( \alpha \), cf. Contribution (b) from Sect. I. Our approach iteratively generates simple invariants such that larger and larger parts of guard_α become monotonic. To this end, it constructs arithmetic formulas and uses constraint solvers to instantiate their free variables (or parameters) such that they become valid. This results in simple invariants that are suitable for strengthening. Eventually, our technique either fails to synthesize further invariants or the whole guard becomes monotonic, so that we can apply Accelerate or even Nonterm (if \( \alpha \)'s guard is a simple invariant).

To synthesize simple invariants, we first compute a maximal subset \( \varphi_1 \) of guard_α such that \( \varphi_1 \) is a conditional invariant. However, to apply Accelerate, not all constraints of guard_α need to be conditional invariants, as long as the remaining constraints are monotonically decreasing. Hence, we next compute a maximal subset \( \varphi_2 \) of \( \varphi_1 \) such that \( \varphi_2 \) is a simple invariant. Then we can determine a maximal subset \( \varphi_m \) of guard_α \( \varphi_1 \) which is monotonically decreasing for \( \varphi_2 \).

A. Generating New Invariants: Let the set of parameters \( P \subset V \) be countable and disjoint from the program variables \( \mathcal{V}(T) \). Moreover, let \( \varphi_2 = \text{guard}_\alpha \setminus \{ \varphi_1 \cup \varphi_m \} \), i.e., \( \varphi_2 \) causes non-monotonicity of guard_α. For each inequation \( \rho \in \varphi_2 \), we construct a linear template \( \tau_\rho \) over the relevant variables \( \mathcal{V}_\rho \) of \( \rho \), i.e., \( \mathcal{V}_\rho \) is the smallest set such that \( \mathcal{V}(\rho) \subsetneq \mathcal{V}_\rho \) \( \mathcal{V}(\rho') \cap \mathcal{V}_\rho \neq \emptyset \) implies \( \mathcal{V}(\rho') \subsetneq \mathcal{V}_\rho \) for each \( \rho' \in \text{guard}_\alpha \), and \( \rho \in \mathcal{V}_\rho \) implies \( \mathcal{V}(\uparrow \alpha(x)) \subset \mathcal{V}_\rho \). So \( \tau_\rho \) has the form \( \sum_{x \in \mathcal{V}_\rho} c_x \cdot x \geq c \) where \( \{ c_x \mid x \in \mathcal{V}_\rho \} \subsetneq P \).

For \( \alpha_4 \) from Ex. 1, we obtain \( \varphi_1 = \emptyset \), \( \varphi_m = \emptyset \), and \( \varphi_2 = \{ y > 0 \} \). As \( x \in \mathcal{V}_\rho(y) \), we have \( \mathcal{V}_\rho(y) = \{ x, y \} \). Hence, \( \tau_{y>0} = c_x \cdot x + c_y \cdot y \geq c \) where \( c_x, c_y, c \in P \).

To find a valuation of the parameters such that all templates can be added to \( \varphi_2 \) without violating the definition of simple invariants, we enforce (si) for \( \varphi_2 \sqcup \bigwedge_{\rho \in \varphi_2} \tau_\rho \) by requiring

\[ \forall \mathcal{V}(T). \varphi_2 \land \bigwedge_{\rho \in \varphi_2} \tau_\rho \implies \bigwedge_{\rho \in \varphi_2} \uparrow \alpha(\tau_\rho). \] (τ-si)

So for \( \alpha_4 \), we search for a valuation of \( c_x, c_y, c \) such that

\[ \forall x, y, c_x \cdot x + c_y \cdot y \geq c \implies c_x \cdot x + c_y \cdot (y-x) \geq c. \] (τ-si-α4)

B. Improving Towards Monotonicity: By construction, the constraint \( \varphi_2 \land \varphi_m \) is monotonic. Furthermore, (τ-si) ensures that \( \varphi_2 \land \bigwedge_{\rho \in \varphi_2} \tau_\rho \) is a simple invariant, i.e., we know that

\[ \varphi_2 \land \bigwedge_{\rho \in \varphi_2} \tau_\rho \] (5)

is monotonic. Eventually, our goal is to turn \( \text{guard}_\alpha \) into a simple invariant and apply Nonterm or to make it monotonic and apply Accelerate. To progress towards this goal incrementally, we ensure that we can add at least one \( \rho \in \varphi_2 \) to (5) without violating monotonicity. To this end, we enforce that (ei) or (md) holds for some \( \rho \in \varphi_2 \) by requiring:

\[ \bigwedge_{\rho \in \varphi_2} \text{guard}_\alpha \land \bigwedge_{\xi \in \varphi_2} \tau_\xi \implies \uparrow \alpha(\tau_\rho) \] or (some-ci)

\[ \bigwedge_{\rho \in \varphi_2} \text{guard}_\alpha \land \bigwedge_{\xi \in \varphi_2} \tau_\xi \land \uparrow \alpha(\varphi_m \land \rho) \implies \rho \] (some-md)

Note that (some-ci) can also help to apply Nonterm as \( \text{guard}_\alpha \) is a conditional invariant iff it is a simple invariant.

C. Maximizing the Improvement: It is clearly advantageous to instantiate the parameters in such a way that as many inequations from \( \varphi_2 \) as possible can be added to (5) without violating monotonicity. Hence, we require
for as many \( \rho \in \varphi_{nm} \) as possible, i.e., each \( \rho \in \varphi_{nm} \) gives rise to a soft requirement \((\rho\text{-ci})\) or \((\rho\text{-md})\). Later, soft requirements will be associated with weights. We then try to maximize the weight of all valid soft requirements, but some of them may be violated. However, all hard requirements like \((\tau\text{-si})\) and \((\text{some-ci})\) or \((\text{some-md})\) must hold.

For \( \alpha_4 \), \( \varphi_{nm} \) is a singleton set and hence \((\text{some-ci})\) or \((\rho\text{-ci})\) resp. \((\text{some-md})\) and \((\rho\text{-md})\) coincide for \( \rho : y > 0 \).

\( \text{(some-ci)}/(\rho\text{-ci}) : \)
\[
\forall x, y. \ y > 0 \land c_x \cdot x + c_y \cdot y \geq c \implies y - x > 0 \quad (\rho\text{-ci}\alpha_4)
\]
\( \text{(some-md)}/(\rho\text{-md}) : \)
\[
\forall x, y. \ c_x \cdot x + c_y \cdot y \geq c \land y - x > 0 \implies y > 0 \quad (\rho\text{-md}\alpha_4)
\]

**D. Preferring Local Invariants:** If we strengthen a transition \( \alpha \) with an inequation \( \xi \), then the case \( \neg \xi \) is not covered by the resulting transition. So we split \( \alpha \) relative to \( \xi \), i.e., we also strengthen \( \alpha \) with \( \neg \xi \). However, this increases the size of the program. Thus, we try to deduce standard invariants whenever possible, i.e., we try to deduce constraints \( \xi \) that are valid whenever \( \alpha \) is applied in a program run so that the case \( \neg \xi \) is irrelevant. To detect such invariants in a modular way, we only consider local invariants, i.e., constraints whose invariance can be proven by reasoning about \( \alpha \) and all transitions \( \beta \) with \( \text{dest}_\beta = \text{src}_\alpha \), whereas all other transitions are ignored. A similar idea is also used in [33] to synthesize invariants.

**Definition 15** (Local Invariants). Let \( \alpha \in T \). If \( \varphi_{li} \) is a conditional invariant of \( \alpha \) and for all \( \beta \in T \setminus \{ \alpha \} \) with \( \text{dest}_\beta = \text{src}_\alpha \),
\[
\forall \forall \forall (T). \ \text{guard}_\beta \land \text{up}_\beta(\text{guard}_\alpha) \implies \text{up}_\beta(\varphi_{li}) \quad (\text{li})
\]
is valid, then \( \varphi_{li} \) is a local invariant of \( \alpha \).

Def. 15 requires that whenever \( \beta \) can be applied (\( \text{guard}_\beta \) in the premise of (li)) and \( \alpha \) can be applied afterwards (\( \text{dest}_\beta = \text{src}_\alpha \) and \( \text{up}_\beta(\text{guard}_\alpha) \) in the premise of (li)), then \( \varphi_{li} \) must hold after applying \( \beta \) (which is the conclusion of (li)).

So for \( \alpha_4 \), \( x \leq 0 \) is clearly a simple invariant, as \( \alpha_4 \) does not update \( x \). Moreover, the guard \( x < 0 \) of \( \alpha_3 \) (which is the only other transition whose destination is \( \text{src}_{\alpha_4} \)) implies \( x \leq 0 \). Thus, \( x \leq 0 \) is a local invariant of \( \alpha_4 \).

To guide the search towards local invariants, we add a soft requirement corresponding to (li) for each \( \rho \in \varphi_{nm} \):
\[
\bigwedge_{\beta \in T \setminus \{ \alpha \}} \forall \forall \forall (T). \ \text{guard}_\beta \land \text{up}_\beta(\text{guard}_\alpha) \implies \text{up}_\beta(\tau_\rho) \quad (\tau\text{-li})
\]
So for \( \rho : y > 0 \) in our example, due to transition \( \alpha_3 \) we get:
\[
\forall x, y. \ x < 0 \land y > 0 \implies c_x \cdot x + c_y \cdot y \geq c \quad (\tau\text{-li}\alpha_4)
\]

**E. Excluding Inapplicable Transitions:** So far we do not exclude solutions that result in inapplicable transitions. To solve this problem, we add the hard requirement
\[
\bigvee_{\forall \forall \forall (T).} \text{guard}_\beta \land \text{up}_\beta(\text{guard}_\alpha) \land \bigwedge_{\rho \in \varphi_{nm}} \text{up}_\beta(\tau_\rho). \quad \text{(sat)}
\]
So we require that there is a transition \( \beta \) with \( \text{dest}_\beta = \text{src}_\alpha \) (due to the leading \( \bigvee \ldots \) ) and a valuation (due to the existential quantifier) such that \( \beta \) is applicable (due to \( \text{guard}_\beta \)) and \( \alpha \) is applicable afterwards (due to \( \text{up}_\beta(\text{guard}_\alpha) \land \bigwedge_{\rho \in \varphi_{nm}} \text{up}_\beta(\tau_\rho) \)), as we will strengthen \( \alpha \)’s guard with \( \bigwedge_{\rho \in \varphi_{nm}} \tau_\rho \) after instantiating the parameters in the templates). Thus, for \( \alpha_4 \) we require
\[
\exists x, y. \ x < 0 \land y > 0 \land c_x \cdot x + c_y \cdot y \geq c \quad \text{(sat}\alpha_4)
\]
due to the transition \( \alpha_3 \).

Alg. 1 essentially compresses each path through a multi-path loop (e.g., a loop whose body contains case analyses) into a simple loop via chaining in order to apply Nonterm, Fixpoint, or Accelerate afterwards. So our technique tends to generate many simple loops for function symbols that correspond to entry points of multi-path loops. Therefore, \((\tau_\rho\text{-li})\) and \((\text{sat})\) can result in large formulas, which leads to performance issues. Hence, our implementation only considers transitions \( \beta \) with \( \text{src}_\beta \neq \text{dest}_\beta \) when constructing \((\tau_\rho\text{-li})\) and \((\text{sat})\). Note that this is uncritical for correctness, as the technique presented in the current section is only a heuristic to generate constraints to be added via Strengthen (which is always sound).

**F. Preferring Nonterm:** Finally, we prefer simple invariants that allow us to apply Nonterm, our main technique to prove non-termination. To this end, we add a soft requirement to prefer solutions where the guard of the resulting strengthened transition is a conditional invariant whenever \( \varphi_{md} \) is empty:
\[
\forall \forall \forall (T). \ \text{guard}_\alpha \land \bigwedge_{\rho \in \varphi_{nm}} \tau_\rho \implies \bigwedge_{\rho \in \varphi_{nm}} \text{up}_\rho(\rho) \quad \text{(nt)}
\]
In our example, \((\text{nt})\) equals \((\rho\text{-ci}\alpha_4)\) as \( \varphi_{nm} \) is a singleton set.

**G. Algorithm for Inferring Simple Invariants:** Alg. 2 summarizes our approach to deduce simple invariants. Here, the \( i \)th entry of the weight vector \( \omega \) corresponds to the weight of the \( i \)th soft requirement \( \chi_i \) and \( \text{solve}(\zeta, X, \omega) \) searches an instantiation \( \sigma \) of the parameters such that \( \sigma \models \zeta \) and \( \sum_{1 \leq i \leq |X|} w_i \) is maximized. We explain how to implement \text{solve} in Sect. IV-H. The weights are chosen in such a way that a solution \( \sigma \) is preferred over \( \sigma' \) if \( \sigma \) turns more templates \( \tau_\rho \) into local invariants than \( \sigma' \). The weight \( m + 2 = |\varphi_{nm}| + 2 \) of the formulas resulting from \((\tau_\rho\text{-li})\) (where \( |\varphi_{nm}| \) is the number of inequations in \( \varphi_{nm} \)) ensures that each formula from \((\tau_\rho\text{-li})\) has a higher weight than the sum of all other soft requirements \((\rho\text{-ci}) \lor (\rho\text{-md})\) and \((\text{nt})\).

Note that Step 15 updates \( \text{guard}_\alpha \) in each iteration and \( \varphi_{nm} \) is recomputed before checking the condition of the while-loop in Step 3. Alg. 2 terminates: \(|\varphi_{nm}| \) decreases in every iteration due to the hard requirement \((\text{some-ci}) \lor (\text{some-md})\), which ensures that some \( \rho \in \varphi_{nm} \) becomes part of \( \varphi_{si} \) or \( \varphi_{md} \). Moreover, the hard requirement \((\tau\text{-si})\) ensures that each \( \sigma(\tau_\rho) \) becomes part of \( \varphi_{si} \), so Alg. 2 never adds elements to \( \varphi_{nm} \).
The soft requirements are \( \varphi_{nm} = \emptyset \) : return \( \emptyset \).

Example 16 (Deducing Simple Invariants for \( \alpha_2 \)). Reconsider the simple loop \( \alpha_2 \) from Ex. 1, where \( \varphi_i = \varphi_{md} = \emptyset \) and \( \varphi_{nm} = \{ x \geq 0 \} \) as \( \alpha_2 \)'s guard \( x \geq 0 \) is not monotonic. Here, \( \tau_{x \geq 0} \) is \( c_x \cdot x + c_y \cdot y \geq c \) as \( y \in V(\text{up}(x)) \). So \( \tau_{\text{si}} \) becomes

\[
\forall x, y, c_x \cdot x + c_y \cdot y \geq c \quad \implies c_x \cdot (x - y) + c_y \cdot (y + 1) \geq c. \quad (\tau_{\text{si}}-\alpha_2)
\]

Again, \( (\text{some-ci}) \lor (\text{some-md}) \) coincides with \( (\rho_{\text{ci}}) \lor (\rho_{\text{md}}) \) for \( \rho : x \geq 0 \).

\[
\forall x, y, c_x \cdot x + c_y \cdot y \geq c \quad \implies x - y \geq 0 \lor (\rho_{\text{ci}}-\alpha_2)
\]

and

\[
\forall x, y, c_x \cdot x + c_y \cdot y \geq c \land x - y \geq 0 \quad \implies x \geq 0 \quad (\rho_{\text{md}}-\alpha_2)
\]

Next, \( (\rho_{\text{md}}) \) gives rise to the requirement

\[
\forall x, y, x \geq 0 \implies c_x \cdot x + c_y \cdot y \geq c. \quad (\rho_{\text{li}}-\alpha_2)
\]

Moreover, \( (\text{sat}) \) becomes

\[
\exists x, y, x \geq 0 \land c_x \cdot x + c_y \cdot y \geq c. \quad (\text{sat-}\alpha_2)
\]

Finally, \( (\text{nt}) \) equals \( (\rho_{\text{ci}}-\alpha_2) \). Thus, the hard requirement \( \zeta \) is

\[
(\tau_{\text{si}}-\alpha_2) \land ((\rho_{\text{ci}}-\alpha_2) \lor (\rho_{\text{md}}-\alpha_2)) \land (\text{sat-}\alpha_2).
\]

The soft requirements are \( (\tau_{\text{li}}-\alpha_2) \lor (\rho_{\text{md}}-\alpha_2) \), and \( (\rho_{\text{ci}}-\alpha_2) \) with weights 3, 1, and 1, respectively. The valuation \( \sigma = \{ c_x \mapsto 0, c_y \mapsto 1, c \mapsto 0 \} \) satisfies \( \zeta \) and \( (\rho_{\text{ci}}-\alpha_2) \lor (\rho_{\text{md}}-\alpha_2) \), but not the other soft constraints. As \( \zeta \land (\tau_{\text{li}}-\alpha_2) \) and \( \zeta \land (\rho_{\text{ci}}-\alpha_2) \) are unsatisfiable, \( \sigma \) is an optimal solution. It corresponds to the simple invariant \( y \geq 0 \). After deducing it, the strengthened transition can be accelerated as in Ex. 8.

H. Greedy Algorithm for Max-SMT Solving: We now explain how to implement the function solve that is called in Alg. 2 to instantiate the parameters in the formulas. Our implementation is restricted to the case that these formulas are linear w.r.t. the program variables \( V(T) \). Then the universally quantified variables can be eliminated by applying Farkas’ Lemma \([6, 37]\). In this way, we obtain a Max-SMT obligation over the theory of non-linear integer arithmetic. While there exist powerful Max-SMT solvers \([4, 15, 18]\), we use a straightforward greedy algorithm based on incremental SMT solving. This approach turned out to be be more efficient than sophisticated Max-SMT techniques in our setting, presumably as it does not aim to find provably optimal solutions.

V. Experiments

We implemented our approach in our tool LoAT \([21]\) which uses the recurrence solver PURRS \([1]\) and the SMT solver Z3 \([15]\). It supports the SMT-LIB input format \([8]\) and the native formats of the tools KoAT \([10]\) and T2 \([11]\). We evaluated it on the benchmark suite from the Termination and Complexity Competition (TermComp \([24]\)) consisting of 1222 programs (TPDB \([39]\), category Termination of Integer Transition Systems). All experiments were executed on StarExec \([38]\) with a timeout of 60 seconds per example.

We first compared our new implementation with our technique to prove lower complexity bounds of integer programs from \([21]\) (LoAT LB), which can also deduce non-termination as a byproduct. LoAT LB proves non-termination in 390 cases, whereas the new version of LoAT succeeds for 462 examples.

Then, we compared LoAT with two state-of-the-art termination analyzers for integer programs: VeryMax \([5, 33]\) (resp. its predecessor CppInv) won the category Termination of Integer Transition Systems at TermComp in 2014 and 2016 – 2019. T2 was the winner in 2015. We also tested with our tool AProVe \([23]\), but excluded it as it uses a similar approach like T2, but finds fewer non-termination proofs. The remaining participants of the respective category of TermComp, Ctrl \([30]\) and iRankFinder \([2, 16]\), cannot prove non-termination.7 We used the TermComp ‘19 version of VeryMax and the TermComp ‘17 version of T2 (as T2 has not been developed further since 2017). Our experiments did not reveal any conflicts, i.e., there is no example where one tool proved termination and another proved non-termination. As the table on the right shows, LoAT proves non-termination more often than any other tool. According to the second last row, it solves 22 examples where all other tools fail. Together, T2 and all TermComp participants succeed on 1130 examples. So LoAT solves 23.9% of the 92 remaining potentially non-terminating examples.

The TPDB examples mostly use linear arithmetic and T2 and VeryMax are restricted to such programs \([11, 33]\). To

\footnote{Note that rational constants can be eliminated by multiplying with the least common multiple.}

\footnote{iRankFinder can prove non-termination of simple loops \([2]\), but according to its authors it cannot yet check reachability of diverging configurations.}
evaluate LoAT on examples with non-linear arithmetic, we also compared with the tool Anant [14], which has been specifically designed to handle non-linearity. Here, we used the 29 non-terminating programs with non-linear arithmetic from the evaluation of [14]. As we were not able to run Anant, even though the authors kindly provided the source code and old binaries, we compared with the results presented in [14].

Together, Anant and LoAT prove non-termination of all examples. LoAT solves one example less than Anant, but it is significantly faster: It always terminates within less than three seconds whereas Anant takes up to 4 minutes in some cases. However, both tools were run on different machines.

Finally, we compared LoAT with the tools from the category Termination of C Integer Programs at TermComp ’198 (AProve [23], Ultimate [12], and VeryMax [5, 33]) on the 355 examples from that category of the TPDB. As LoAT cannot parse C, we coupled it with a version of AProVE that converts C programs into equivalent integer programs. The results of LoAT are competitive, but it succeeds on less examples than AProVE and VeryMax. VeryMax and LoAT are the only tools that find unique non-termination proofs. Finally, LoAT is the fastest tool, although its runtime includes AProVE’s conversion from C. However, all tools but LoAT also spend time on attempting to prove termination, which may explain their longer runtime.

To explain the discrepancy between the results for integer programs and for C programs, note that the integer programs from the TPDB often contain several loops. Here, our loop acceleration technique is particularly successful, because the challenge is not only to prove non-termination of one of the loops, but also to prove its reachability. In contrast, many C programs from the TPDB consist of a single multi-path loop. So to prove non-termination, one has to find a suitable pattern to execute the paths through the loop’s body. To improve the handling of such examples, we will extend our approach by control flow refinement techniques [17, 20, 26, 31] in future work.

See https://ffrohn.github.io/acceleration for a pre-compiled binary (Linux, 64 bit) of LoAT, tables with detailed results for all benchmarks, and the full output of the tools for all examples (the detailed results of Anant can be found in [14]). The source code of the implementation in our tool LoAT is available at https://github.com/aprove-developers/LoAT/tree/nonterm.

VI. CONCLUSION AND RELATED WORK

A. Conclusion: We presented the first non-termination technique based on loop acceleration. It accelerates terminating loops in order to prove reachability of non-terminating configurations, even if this requires reasoning about program parts that contain loops themselves. As we use a non-termination criterion which is a special case of the prerequisites of our novel loop acceleration technique (see Sect. III), we can use the same new invariant inference technique (Sect. IV) to facilitate both loop acceleration and non-termination proving. The experimental evaluation of our approach shows that it is competitive with state-of-the-art tools, cf. Sect. V.

B. Related Work: Loop Acceleration is mostly used in over-approximating settings (e.g., [19, 25, 29, 36]), whereas our setting is under-approximating. We only know of two other under-approximating loop acceleration techniques [21, 31]: One requires metering functions [21], an adaption of ranking functions, that can be challenging to synthesize. The other [31] is a special case of Thm. 7 where \( \varphi_{cs} = \varphi_{si} = true \), which restricts its applicability in comparison to our approach. To facilitate acceleration, [31] splits disjunctive guards, which is orthogonal to our splitting of guards by adding conjuncts (cf. Sect. IV-D).

Most techniques to prove non-termination first generate lassos consisting of a simple loop \( \alpha \) and a stem, i.e., a path from the program’s entry point to \( \alpha \). Then they try to prove non-termination of these lassos. However, a program with consecutive or nested loops usually has infinitely many possible lassos. In contrast, our program simplification framework yields a loop-free simplified program with finitely many transitions.

In [27], recurrent sets were proposed to prove non-termination. A set of configurations is recurrent if each element has a successor in the set. Hence, a non-empty recurrent set that contains an initial configuration witnesses non-termination. There are many techniques to find recurrent sets for simple loops [2], lassos [7, 13, 27, 41], or more complex sub-programs [33]. Essentially, our invariant inference technique of Sect. IV also searches for a recurrent set for a simple loop \( \alpha \). However, if it cannot find a recurrent set it may still successfully enforce monotonicity of \( \text{guard}_{\alpha} \) and hence allow us to accelerate \( \alpha \).

An alternative to recurrent sets is presented in [35]. It represents infinite runs as sums of geometric series. In general, we could use any technique to prove non-termination of simple loops or lassos as an alternative to our non-termination criteria.

Further approaches to prove non-termination are, e.g., based on Hoare-style reasoning [34] or safety proving [40].

While most related techniques to prove non-termination focus on linear arithmetic, [14] has been specifically designed to handle non-linear arithmetic via live abstractions and a variation of recurrent sets. As shown in Sect. V, our approach is also competitive on programs with non-linear arithmetic.

C. Future Work: We will integrate control flow refinement techniques and more powerful non-termination criteria (e.g., to find disjunctive recurrent sets, which we cannot handle yet). We will also consider techniques to infer non-linear invariants, as our current invariant inference is restricted to linear arithmetic.

D. Acknowledgments: We thank Marc Brockschmidt and Matthias Naaf for important initial discussions.


