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 ω. 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 ω 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 and
(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 $\boldsymbol{x}$ by bold letters and the i^{th} element of $\boldsymbol{x}$ by x_i. *Transitions* α have the form $\mathsf{f}(\boldsymbol{x}) \to \mathsf{g}(\boldsymbol{t})\ [\eta]$. The *left-hand side* $\mathrm{lhs}_\alpha = \mathsf{f}(\boldsymbol{x})$ consists of α's *source function symbol* $\mathrm{src}_\alpha = \mathsf{f} \in \Sigma$ and a vector of pairwise different variables $\boldsymbol{x} \subset \mathcal{V}$ ranging over $\mathbb{Z}$, where $\mathcal{V}$ is countably infinite. The set of function symbols Σ 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 "+", "·", etc. The *guard* $\mathrm{guard}_\alpha = \eta$ 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* $\mathrm{rhs}_\alpha = \mathsf{g}(\boldsymbol{t})$ consists of α's *destination* $\mathrm{dest}_\alpha = \mathsf{g} \in \Sigma$ and a vector $\boldsymbol{t} \subset \mathcal{A}$. The *substitution* $\mathrm{up}_\alpha = \{\boldsymbol{x} \mapsto \boldsymbol{t}\}$ is α's *update*.

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

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

funded by DFG grant 389792660 as part of TRR 248 and by DFG grant GI 274/6

[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 η.

Example 1 (Integer Program)**.** *Consider the function* `start`:

```
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:

$$\begin{array}{lrcll} \alpha_1: & \mathsf{start}(x,y) & \to & \mathsf{f}(x,y) & \\ \alpha_2: & \mathsf{f}(x,y) & \to & \mathsf{f}(x-y,y+1) & [x \geq 0] \\ \alpha_3: & \mathsf{f}(x,y) & \to & \mathsf{g}(x,y) & [x < 0] \\ \alpha_4: & \mathsf{g}(x,y) & \to & \mathsf{g}(x,y-x) & [y > 0] \end{array}$$

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* $\mathsf{f}(\boldsymbol{n})$ *where* $\boldsymbol{n} \subset \mathbb{Z}$ *is a* configuration*. An integer program* $\mathcal{T}$ *induces a relation* $\to_{\mathcal{T}}$ *on configurations: We have* $s \to_{\mathcal{T}} t$ *if there is an* $\alpha \in \mathcal{T}$ *and a model* σ *of* guard_α *such that* $\mathcal{V}(\alpha) \subseteq \mathrm{dom}(\sigma)$*,* $\sigma(\mathrm{lhs}_\alpha) = s$*, and* $\sigma(\mathrm{rhs}_\alpha) = t$*.*[2] *Then we say that* s evaluates *to* t*. As usual,* $\to^*_{\mathcal{T}}$ *is the transitive-reflexive closure of* $\to_{\mathcal{T}}$*.*

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

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

By definition, integer programs may contain transitions like $\mathsf{f}(x) \to \mathsf{f}(\frac{x}{2})$. While evaluations that would not yield integers get stuck (as, e.g., $\mathsf{f}(\frac{1}{2})$ 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* $\mathcal{T}$ *is* well formed *if for all transitions* $\alpha \in \mathcal{T}$ *and all models* σ *of* guard_α *with* $\mathcal{V}(\alpha) \subseteq \mathrm{dom}(\sigma)$*,* $\sigma(\mathrm{rhs}_\alpha)$ *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* $\mathcal{T}$ *where* proc *is defined:*

if $\mathsf{start}(\boldsymbol{n}) \to^*_{\mathrm{proc}(\mathcal{T})} \omega$ *or* $\mathsf{start}(\boldsymbol{n})$ *witnesses non-termination of* $\mathrm{proc}(\mathcal{T})$*,*
then $\mathsf{start}(\boldsymbol{n}) \to^*_{\mathcal{T}} \omega$ *or* $\mathsf{start}(\boldsymbol{n})$ *witnesses non-termination of* $\mathcal{T}$*.*

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

So we use the symbol ω to represent non-termination (and we omit its arguments for readability): If we can transform a program $\mathcal{T}$ into a simplified program $\mathcal{T}'$ via safe and sound processors and $\sigma \models \mathrm{guard}_\alpha$ for some $\alpha \in \mathcal{T}'$ with $\mathrm{rhs}_\alpha = \omega$, then $\sigma(\mathrm{lhs}_\alpha)$ witnesses non-termination of $\mathcal{T}$ 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, "$\forall \mathcal{V}(\mathcal{T}).\ \psi$" abbreviates "$\forall (\mathcal{V}(\psi) \cap \mathcal{V}(\mathcal{T})).\ \psi$", i.e., the quantifier binds all free variables of ψ that occur in $\mathcal{T}$.

Definition 5 (Invariants)**.** *Let* $\alpha \in \mathcal{T}$*. If*

$$\forall \mathcal{V}(\mathcal{T}).\ \mathrm{guard}_\alpha \wedge \varphi_{ci} \implies \mathrm{up}_\alpha(\varphi_{ci}) \qquad \text{(ci)}$$

is valid, then φ_{ci} *is a* conditional invariant *of* α*. If*

$$\forall \mathcal{V}(\mathcal{T}).\ \varphi_{si} \implies \mathrm{up}_\alpha(\varphi_{si}) \qquad \text{(si)}$$

is valid, then φ_{si} *is a* simple (conditional) invariant *of* α*. If* φ_{si} *is a simple invariant of* α *and*

$$\forall \mathcal{V}(\mathcal{T}).\ \varphi_{si} \wedge \mathrm{up}_\alpha(\varphi_{md}) \implies \varphi_{md} \qquad \text{(md)}$$

is valid, then φ_{md} *is* monotonically decreasing *for* φ_{si} *and* α*.*

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

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

Example 6 (Invariants)**.** *For* α_2 *from Ex. 1,* $y \geq 0$ *is a simple invariant and* $x \geq 0$ *is monotonically decreasing for* $y \geq 0$*, as*

$$\begin{array}{lll} \forall x,y.\ y \geq 0 & \implies & y+1 \geq 0 \quad \textit{and} \\ \forall x,y.\ y \geq 0 \wedge x-y \geq 0 & \implies & x \geq 0 \end{array}$$

are valid. Thus, $y \geq 0$ *is also a conditional invariant. Note*

[2] Throughout the paper, we use "=" for semantic (not syntactic) equality w.r.t. arithmetic, e.g., "$\mathsf{f}(1+2) = \mathsf{f}(3)$" holds.

[3] One could also allow expressions like $\frac{1}{2} \cdot x^2 + \frac{1}{2} \cdot x$ in the initial program, as long as every arithmetic expression maps integers to integers.

that it is not a standard invariant as there are program runs where $y \geq 0$ is violated when α_2 is applied.

B. Loop Acceleration: The key idea of loop acceleration for a *simple loop*, i.e., a transition α with $\mathrm{src}_\alpha = \mathrm{dest}_\alpha$, is to generate a new transition $\overline{\alpha}$ that captures k iterations of α. Here, k is a fresh variable whose value can be chosen non-deterministically. 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 $\mathrm{up}_\alpha^k = \underbrace{\mathrm{up}_\alpha \circ \ldots \circ \mathrm{up}_\alpha}_{k \text{ times}}$. Then, as in [31], we exploit the following observation: If guard_α holds after $k-1$ loop iterations and $\mathrm{up}_\alpha(\mathrm{guard}_\alpha)$ implies guard_α (i.e., guard_α is monotonically decreasing), then guard_α also holds after $k-2, k-3, \ldots, 0$ iterations. Thus, adding $\mathrm{up}_\alpha^{k-1}(\mathrm{guard}_\alpha)$ to $\mathrm{guard}_{\overline{\alpha}}$ ensures that k only takes feasible values: If σ satisfies $\mathrm{up}_\alpha^{k-1}(\mathrm{guard}_\alpha)$, then α can be iterated at least $\sigma(k)$ times.

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

Theorem 7 (Accelerate). *Let $\mathcal{T}$ be well formed, let $\alpha \in \mathcal{T}$ be a simple loop with $\mathrm{lhs}_\alpha = \mathsf{f}(\boldsymbol{x})$, let $k \in \mathcal{V}$ be fresh, and let μ be a substitution such that $\mu(\boldsymbol{x}) = \mathrm{up}_\alpha^k(\boldsymbol{x})$ holds for all $k > 0$. Moreover, let $\mathrm{guard}_\alpha = \varphi_{ci} \wedge \varphi_{si} \wedge \varphi_{md}$ be monotonic. Finally, let $\mathrm{dec}_k = \{k \mapsto k-1\}$ and $\overline{\mathcal{T}} = \mathcal{T} \cup \{\overline{\alpha}\}$ where*

$$\overline{\alpha} = \boxed{\mathsf{f}(\boldsymbol{x}) \to \mathsf{f}(\mu(\boldsymbol{x}))\ \ [\varphi_{ci} \wedge \varphi_{si} \wedge \mathrm{dec}_k(\mu(\varphi_{md})) \wedge k > 0]}\,.$$

*Then the processor **Accelerate**: $\mathcal{T} \mapsto \overline{\mathcal{T}}$ is safe and sound.*

So to construct $\mathrm{rhs}_{\overline{\alpha}}$, we compute a closed form μ 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 $\boldsymbol{x}^{(k+1)} = \mathrm{up}_\alpha(\boldsymbol{x}^{(k)})$ with the initial condition $\boldsymbol{x}^{(1)} = \mathrm{up}_\alpha(\boldsymbol{x})$.

To see why **Accelerate** is sound, assume that $\mathrm{guard}_{\overline{\alpha}}$ holds. As[4] $\varphi_{si} \subseteq \mathrm{guard}_{\overline{\alpha}}$ and φ_{si} implies $\mathrm{up}_\alpha(\varphi_{si})$ by (si), we obtain

$$\mathrm{up}_\alpha^n(\varphi_{si}) \text{ for all } n \in \mathbb{N}. \tag{1}$$

Thus, as $\mathrm{guard}_{\overline{\alpha}}$ contains $\mathrm{dec}_k(\mu(\varphi_{md})) = \mathrm{up}_\alpha^{k-1}(\varphi_{md})$ and $\varphi_{si} \wedge \mathrm{up}_\alpha(\varphi_{md})$ implies φ_{md} by (md), we get

$$\mathrm{up}_\alpha^n(\varphi_{md}) \text{ for all } 0 \leq n < k. \tag{2}$$

So (1) and (2) imply $\varphi_{si} \wedge \varphi_{md}$. As $\varphi_{ci} \subseteq \mathrm{guard}_{\overline{\alpha}}$ and $\mathrm{guard}_\alpha = \varphi_{ci} \wedge \varphi_{si} \wedge \varphi_{md}$, this means that guard_α holds as well. As guard_α implies $\mathrm{up}_\alpha(\varphi_{ci})$ (since $\varphi_{ci} \subseteq \mathrm{guard}_\alpha$ and φ_{ci} is a conditional invariant), we obtain that $\mathrm{up}_\alpha(\varphi_{ci})$ holds. Together with (1) and (2) this means that $\mathrm{up}_\alpha(\mathrm{guard}_\alpha)$ holds (if $1 < k$). This in turn implies $\mathrm{up}_\alpha^2(\varphi_{ci})$, etc. Thus, we get

$$\mathrm{up}_\alpha^n(\varphi_{ci}) \text{ for all } 0 \leq n \leq k. \tag{3}$$

Due to (1) – (3), the constraint $\varphi_{ci} \wedge \varphi_{si} \wedge \mathrm{dec}_k(\mu(\varphi_{md}))$ ensures that $\mathrm{guard}_\alpha = \varphi_{ci} \wedge \varphi_{si} \wedge \varphi_{md}$ holds before the $1^{st}, \ldots, k^{th}$ iteration, as desired. Hence, every evaluation with $\overline{\alpha}$ can be replaced by k evaluation steps with α. Since $\mathrm{guard}_{\overline{\alpha}}$ enforces $k > 0$, every non-terminating run with $\overline{\mathcal{T}}$ can therefore be transformed into a non-terminating run of $\mathcal{T}$.

Example 8 (Ex. 1 continued)**.** *Consider the simple loop α_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 φ_{ci} : true, φ_{si} : $y \geq 0$, and φ_{md} : $x \geq 0$. Sect. IV will show how to find simple invariants like $y \geq 0$.*

To compute a substitution μ 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 α_2 to

$$\overline{\alpha}_2 : \quad \mathsf{f}(x,y) \to \mathsf{f}(\underbrace{x - y \cdot k - \tfrac{1}{2} \cdot k^2 + \tfrac{1}{2} \cdot k}_{\mu(x)}, \underbrace{y + k}_{\mu(y)})\ \ [\eta]$$

for $\eta : \underbrace{y \geq 0}_{\varphi_{si}} \wedge \mathrm{dec}_k(\mu(\varphi_{md})) \wedge k > 0$ where $\mathrm{dec}_k(\mu(\varphi_{md}))$ 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 $\mathsf{f}(\ldots) \to \mathsf{g}(\ldots)$ and $\mathsf{g}(\ldots) \to \mathsf{h}(\ldots)$ to a new transition $\mathsf{f}(\ldots) \to \mathsf{h}(\ldots)$ 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 $\mathrm{dest}_\alpha = \mathrm{src}_\beta$, the argument lists of lhs_α and lhs_β are equal, and $\mathcal{V}(\alpha) \cap \mathcal{V}(\beta) = \mathcal{V}(\mathrm{lhs}_\alpha)$.[5] Let $\mathcal{T}^\circ = \mathcal{T} \cup \{\alpha \circ \beta\}$ with*

$$\alpha \circ \beta = \boxed{\mathrm{lhs}_\alpha \to \mathrm{up}_\alpha(\mathrm{rhs}_\beta)\ \ [\mathrm{guard}_\alpha \wedge \mathrm{up}_\alpha(\mathrm{guard}_\beta)]}\,.$$

*Then the processor **Chain** : $\mathcal{T} \mapsto \mathcal{T}^\circ$ 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 α with itself in order to enable loop acceleration and to obtain better closed forms for up_α^k. For example, consider the following loop, where the sign of x alternates:

$$\alpha_{neg} : \quad \mathsf{f}(x,y) \to \mathsf{f}(-x, y-1)\ \ [y > x]$$

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 α_{neg} with itself, which results in

$$\alpha_{neg} \circ \alpha_{neg} : \quad \mathsf{f}(x,y) \to \mathsf{f}(x, y-2)\ \ [y > x \wedge y - 1 > -x]\,.$$

This transition can be accelerated to

$$\mathsf{f}(x,y) \to \mathsf{f}(x, y - 2 \cdot k)\ \ \left[\mathrm{dec}_k(\mu(\mathrm{guard}_{\alpha_{neg} \circ \alpha_{neg}})) \wedge k > 0\right]$$

[4] In the following, we identify conjunctions and sets of inequations.

[5] Otherwise, one can rename variables without affecting the relation $\to_{\mathcal{T}}$.

where $\mathrm{dec}_k(\mu(\mathrm{guard}_{\alpha_{neg}\circ\alpha_{neg}}))$ is

$$y - 2\cdot(k-1) > x \wedge y - 2\cdot(k-1) - 1 > -x,$$

i.e., the accelerated transition does not contain exponentials.

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

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

$$\alpha_{const}: \quad \mathsf{f}(x,y,z) \to \mathsf{f}(x-1,2,y) \;\; [x>0]$$

is $0^{k-1}\cdot y + (1-0^{k-1})\cdot 2$, which is again not polynomial. However, chaining α_{const} with itself yields

$$\alpha_{const}\circ\alpha_{const}: \quad \mathsf{f}(x,y,z) \to \mathsf{f}(x-2,2,2) \;\; [x>1]$$

(where we simplified the guard), which can be accelerated to

$$\mathsf{f}(x,y,z) \to \mathsf{f}(x-2\cdot k,2,2) \;\; [x-2\cdot(k-1)>1 \wedge k>0],$$

i.e., the accelerated transition again only contains polynomials.

Finally, chaining can also make acceleration applicable to loops that permute arguments:

$$\alpha_p: \quad \mathsf{f}(x,y) \to \mathsf{f}(y-1,x-1) \;\; [x>0]$$

While α_p violates the prerequisites of **Accelerate**,

$$\alpha_p\circ\alpha_p: \quad \mathsf{f}(x,y) \to \mathsf{f}(x-2,y-2) \;\; [x>0 \wedge y-1>0]$$

can be accelerated to:

$$\mathsf{f}(x,y) \to \mathsf{f}(x-2\cdot k, y-2\cdot k) \;\; \boxed{[\mathrm{dec}_k(\mu(\mathrm{guard}_{\alpha_p\circ\alpha_p})) \wedge k>0]}$$

So to handle simple loops α where some variables "stabilize" (i.e., $\mathrm{up}^n_\alpha(z) \in \mathbb{Z}$ for some $z \in \mathcal{V}$ and some $n>1$, as in α_{const}) or where arguments are permuted (as in α_p), we repeatedly chain α with itself as long as this reduces the size of

$$\{x \in \mathcal{V}(\mathrm{lhs}_\alpha) \mid \mathcal{V}(\mathrm{up}_\alpha(x)) \neq \varnothing \wedge x \notin \mathcal{V}(\mathrm{up}_\alpha(x))\}. \quad (4)$$

D. Proving Non-Termination: To detect non-terminating simple loops α, we check whether guard_α itself is a simple invariant (i.e., whether the valuations that satisfy guard_α correspond to a *recurrent set* of the relation $\to_{\{\alpha\}}$, cf. [27]).

Theorem 10 (Nonterm). *Let $\mathcal{T}$ be well formed and let $\alpha \in \mathcal{T}$ be a simple loop such that guard_α is a simple invariant. Moreover, let $\mathcal{T}^\omega = \mathcal{T} \cup \{\alpha^\omega\}$ where*

$$\alpha^\omega = \boxed{\mathrm{lhs}_\alpha \to \omega \;\; [\mathrm{guard}_\alpha]}.$$

*Then the processor **Nonterm** : $\mathcal{T} \mapsto \mathcal{T}^\omega$ is safe and sound.*

Example 11 (Ex. 1 continued)**.** *Clearly, $y>0$ is not a simple invariant of α_4 from Ex. 1. But if we strengthen the guard by adding the simple invariant $x \le 0$, then **Nonterm** is applicable as $y>0 \wedge x \le 0$ implies $y - x > 0 \wedge x \le 0$. Thus, we obtain*

$$\alpha_4^\omega: \quad \mathsf{g}(x,y) \to \omega \;\; [y>0 \wedge x \le 0].$$

Again, we will see how to deduce suitable simple invariants like $x \le 0$ automatically in Sect. IV.

In some cases, chaining also helps to make **Nonterm** applicable. To see this, consider the simple loop

$$\alpha_{nt}: \quad \mathsf{f}(x,y) \to \mathsf{f}(0,y-x) \;\; [y>0]$$

where $y>0$ is no simple invariant. Chaining it with itself yields

$$\alpha_{nt}\circ\alpha_{nt}: \quad \mathsf{f}(x,y) \to \mathsf{f}(0,y-x) \;\; [y>0 \wedge y-x>0].$$

As $y>0 \wedge y-x>0 \implies y-x>0 \wedge y-x-0>0$ is valid, the prerequisites of **Nonterm** are satisfied and we obtain

$$\mathsf{f}(x,y) \to \omega \;\; [y>0 \wedge y-x>0].$$

So in general, we try to apply **Nonterm** not only to a simple loop α, but also to $\alpha\circ\alpha$. Apart from **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 $\mathrm{lhs}_\alpha = \mathsf{f}(\boldsymbol{x})$, and let $\mathrm{guard}_\alpha \wedge \boldsymbol{x} = \mathrm{up}_\alpha(\boldsymbol{x})$ be satisfiable. Let $\mathcal{T}^{\boldsymbol{fp}} = \mathcal{T} \cup \{\alpha^{\boldsymbol{fp}}\}$ where*

$$\alpha^{\boldsymbol{fp}} = \boxed{\mathrm{lhs}_\alpha \to \omega \;\; [\mathrm{guard}_\alpha \wedge \boldsymbol{x} = \mathrm{up}_\alpha(\boldsymbol{x})]}.$$

*Then the processor **Fixpoint** : $\mathcal{T} \mapsto \mathcal{T}^{\boldsymbol{fp}}$ is safe and sound.*

For example, $\{x \mapsto 0, y \mapsto 1\}$ is a fixpoint of α_{nt} which satisfies $y>0$ and $(x,y) = (0, y-x)$ (i.e., $x = 0 \wedge y = y-x$).

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 deleting 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 $\bot$

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

Algorithm 1: Proving Non-Termination

We present the algorithm $\mathbf{deduceInvariants}$ for Step 13 in

Sect. IV. It creates variants of α by extending guard_α with suitable constraints to make **Accelerate** or **Nonterm** applicable. Step 14 chains α with all preceding transitions that are no simple loops. Steps 17 and 18 eliminate a function symbol via chaining. Note that Alg. 1 could have non-terminating runs, as it may add new transitions in Step 13. However, this turned out to be unproblematic in our experiments, cf. Sect. V.

Example 13 (Ex. 1 finished)**.** *After accelerating α_2 in Step 12 (see Ex. 8), Alg. 1 computes*

$$\alpha_1 \circ \overline{\alpha}_2 : \quad \mathsf{start}(x,y) \to \mathsf{f}(x - y\cdot k - \tfrac{1}{2}\cdot k^2 + \tfrac{1}{2}\cdot k, y + k) \;\; [\eta]$$

in Step 14 where η is

$$y \geq 0 \wedge x - y\cdot(k-1) - \tfrac{1}{2}\cdot(k-1)^2 + \tfrac{1}{2}\cdot(k-1) \geq 0 \wedge k > 0.$$

*Next, it applies **Nonterm** to α_4 in Step 9 (see Ex. 11) and chains the resulting transition with α_3 in Step 14, which yields*

$$\alpha_3 \circ \alpha_4^\omega : \quad \mathsf{f}(x,y) \to \omega \;\; [x < 0 \wedge y > 0]\,.$$

Then, it chains $\alpha_1 \circ \overline{\alpha}_2$ and $\alpha_3 \circ \alpha_4^\omega$ in Step 17, resulting in

$$\alpha_1 \circ \overline{\alpha}_2 \circ \alpha_3 \circ \alpha_4^\omega : \quad \mathsf{start}(x,y) \to \omega \;\; [\psi]$$

where ψ is $\eta \wedge \underbrace{x - y\cdot k - \frac{1}{2}\cdot k^2 + \frac{1}{2}\cdot k < 0}_{\mathrm{up}_{\alpha_1\circ\overline{\alpha}_2}(x<0)} \wedge \underbrace{y + k > 0}_{\mathrm{up}_{\alpha_1\circ\overline{\alpha}_2}(y>0)}$.

To prove non-termination, we have to show satisfiability of ψ. As $\sigma \models \psi$ for $\sigma = \{x \mapsto 0, y \mapsto 0, k \mapsto 2\}$, the configuration $\sigma(\mathsf{start}(x,y)) = \mathsf{start}(0,0)$ 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 $\alpha_1 \circ \overline{\alpha}_2 \circ \alpha_3 \circ \alpha_4^\omega$, the instantiation $k \mapsto 2$ means that α_2 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 $\mathcal{T}$ be well formed, let $\alpha \in \mathcal{T}$, let φ be a constraint, and let $\mathcal{T}^\bullet = \mathcal{T} \cup \{\alpha^\bullet\}$ where*

$$\alpha^\bullet = \boxed{\mathrm{lhs}_\alpha \to \mathrm{rhs}_\alpha \;\; [\mathrm{guard}_\alpha \wedge \varphi]}\,.$$

*Then the processor **Strengthen** : $\mathcal{T} \mapsto \mathcal{T}^\bullet$ is safe and sound.*

The challenge is to find constraints φ that help to prove non-termination. We now explain how to automatically synthesize suitable simple invariants to strengthen a simple loop α, 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 α's guard is a simple invariant).

To synthesize simple invariants, we first compute a maximal subset φ_i of guard_α such that φ_i 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 φ_{si} of φ_i such that φ_{si} is a simple invariant. Then we can determine a maximal subset φ_{md} of $\mathrm{guard}_\alpha \setminus \varphi_i$ which is monotonically decreasing for φ_{si}.

A. Generating New Invariants: Let the set of *parameters* $\mathcal{P} \subset \mathcal{V}$ be countably infinite and disjoint from the program variables $\mathcal{V}(\mathcal{T})$. Moreover, let $\varphi_{nm} = \mathrm{guard}_\alpha \setminus (\varphi_i \cup \varphi_{md})$, i.e., φ_{nm} causes non-monotonicity of guard_α. For each inequation $\rho \in \varphi_{nm}$, we construct a linear template τ_ρ over the *relevant* variables $\mathcal{V}_\rho$ of ρ, i.e., $\mathcal{V}_\rho$ is the smallest set such that $\mathcal{V}(\rho) \subseteq \mathcal{V}_\rho$, $\mathcal{V}(\rho') \cap \mathcal{V}_\rho \neq \varnothing$ implies $\mathcal{V}(\rho') \subseteq \mathcal{V}_\rho$ for each $\rho' \in \mathrm{guard}_\alpha$, and $x \in \mathcal{V}_\rho$ implies $\mathcal{V}(\mathrm{up}_\alpha(x)) \subseteq \mathcal{V}_\rho$. So τ_ρ has the form $\sum_{x \in \mathcal{V}_\rho} c_x \cdot x \geq c$ where $\{c_x \mid x \in \mathcal{V}_\rho\} \cup \{c\} \subset \mathcal{P}$.

For α_4 from Ex. 1, we obtain $\varphi_i = \varnothing$, $\varphi_{md} = \varnothing$, and $\varphi_{nm} = \{y > 0\}$. As $x \in \mathrm{up}_{\alpha_4}(y)$, we have $\mathcal{V}_{y>0} = \{x, y\}$. Hence, $\tau_{y>0}$ is $c_x \cdot x + c_y \cdot y \geq c$ where $c_x, c_y, c \in \mathcal{P}$.

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

$$\forall \mathcal{V}(\mathcal{T}).\; \varphi_{si} \wedge \bigwedge_{\rho \in \varphi_{nm}} \tau_\rho \implies \bigwedge_{\rho \in \varphi_{nm}} \mathrm{up}_\alpha(\tau_\rho). \qquad (\tau\text{-si})$$

So for α_4, we search for a valuation of c_x, c_y, and c that satisfies

$$\forall x, y.\; c_x \cdot x + c_y \cdot y \geq c \implies c_x \cdot x + c_y \cdot (y - x) \geq c. \quad (\tau\text{-si-}\alpha_4)$$

B. Improving Towards Monotonicity: By construction, the constraint $\varphi_i \wedge \varphi_{md}$ is monotonic. Furthermore, (τ-si) ensures that $\varphi_{si} \wedge \bigwedge_{\rho \in \varphi_{nm}} \tau_\rho$ is a simple invariant, i.e., we know that

$$\varphi_i \wedge \bigwedge_{\rho \in \varphi_{nm}} \tau_\rho \wedge \varphi_{md} \qquad (5)$$

is monotonic. Eventually, our goal is to turn guard_α 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 $\rho \in \varphi_{nm}$ to (5) without violating monotonicity. To this end, we enforce that (ci) or (md) holds for some $\rho \in \varphi_{nm}$ by requiring:

$$\bigvee_{\rho \in \varphi_{nm}} \forall \mathcal{V}_\mathcal{T}.\; \mathrm{guard}_\alpha \wedge \bigwedge_{\xi \in \varphi_{nm}} \tau_\xi \implies \mathrm{up}_\alpha(\rho) \quad \text{or} \qquad \text{(some-ci)}$$

$$\bigvee_{\rho \in \varphi_{nm}} \forall \mathcal{V}_\mathcal{T}.\; \varphi_{si} \wedge \bigwedge_{\xi \in \varphi_{nm}} \tau_\xi \wedge \mathrm{up}_\alpha(\varphi_{md} \wedge \rho) \implies \rho \quad \text{(some-md)}$$

Note that (some-ci) can also help to apply **Nonterm** as guard_α 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 φ_{nm} as possible can be added to (5) without violating monotonicity. Hence, we require

$$\forall \mathcal{V}(\mathcal{T}).\ \mathrm{guard}_\alpha \wedge \bigwedge_{\xi \in \varphi_{nm}} \tau_\xi \implies \mathrm{up}_\alpha(\rho) \quad \text{or} \qquad (\rho\text{-ci})$$

$$\forall \mathcal{V}(\mathcal{T}).\ \varphi_{si} \wedge \bigwedge_{\xi \in \varphi_{nm}} \tau_\xi \wedge \mathrm{up}_\alpha(\varphi_{md} \wedge \rho) \implies \rho \qquad (\rho\text{-md})$$

for as many $\rho \in \varphi_{nm}$ as possible, i.e., each $\rho \in \varphi_{nm}$ gives rise to a *soft requirement* (ρ-ci) $\vee$ (ρ-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 (τ-si) and (some-ci) $\vee$ (some-md) must hold.

For α_4, φ_{nm} is a singleton set and hence (some-ci) and (ρ-ci) resp. (some-md) and (ρ-md) coincide for $\rho : y > 0$.

(some-ci)/(ρ-ci) :

$$\forall x, y.\ y > 0 \wedge c_x \cdot x + c_y \cdot y \geq c \implies y - x > 0 \quad (\rho\text{-ci-}\alpha_4)$$

(some-md)/(ρ-md) :

$$\forall x, y.\ c_x \cdot x + c_y \cdot y \geq c \wedge y - x > 0 \implies y > 0 \ (\rho\text{-md-}\alpha_4)$$

D. Preferring Local Invariants: If we strengthen a transition α with an inequation ξ, then the case $\neg\xi$ is not covered by the resulting transition. So we split α relative to ξ, i.e., we also strengthen α 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 ξ that are valid whenever α 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 α and all transitions β with $\mathrm{dest}_\beta = \mathrm{src}_\alpha$, whereas all other transitions are ignored. A similar idea is also used in [33] to synthesize invariants.

Definition 15 (Local Invariants)**.** *Let $\alpha \in \mathcal{T}$. If φ_{li} is a conditional invariant of α and for all $\beta \in \mathcal{T} \setminus \{\alpha\}$ with $\mathrm{dest}_\beta = \mathrm{src}_\alpha$,*

$$\forall \mathcal{V}(\mathcal{T}).\ \mathrm{guard}_\beta \wedge \mathrm{up}_\beta(\mathrm{guard}_\alpha) \implies \mathrm{up}_\beta(\varphi_{li}) \qquad (\mathrm{li})$$

is valid, then φ_{li} is a local invariant *of α.*

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

So for α_4, $x \leq 0$ is clearly a simple invariant, as α_4 does not update x. Moreover, the guard $x < 0$ of α_3 (which is the only other transition whose destination is src_{α_4}) implies $x \leq 0$. Thus, $x \leq 0$ is a local invariant of α_4.

To guide the search towards local invariants, we add a soft requirement corresponding to (li) for each $\rho \in \varphi_{nm}$:

$$\bigwedge_{\substack{\beta \in \mathcal{T} \setminus \{\alpha\} \\ \mathrm{dest}_\beta = \mathrm{src}_\alpha}} \forall \mathcal{V}(\mathcal{T}).\ \mathrm{guard}_\beta \wedge \mathrm{up}_\beta(\mathrm{guard}_\alpha) \implies \mathrm{up}_\beta(\tau_\rho) \quad (\tau_\rho\text{-li})$$

So for $\rho : y > 0$ in our example, due to transition α_3 we get:

$$\forall x, y.\ x < 0 \wedge y > 0 \implies c_x \cdot x + c_y \cdot y \geq c \quad (\tau_\rho\text{-li-}\alpha_4)$$

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

$$\bigvee_{\substack{\beta \in \mathcal{T} \setminus \{\alpha\} \\ \mathrm{dest}_\beta = \mathrm{src}_\alpha}} \exists \mathcal{V}(\mathcal{T}).\ \mathrm{guard}_\beta \wedge \mathrm{up}_\beta(\mathrm{guard}_\alpha) \wedge \bigwedge_{\rho \in \varphi_{nm}} \mathrm{up}_\beta(\tau_\rho). \qquad (\mathrm{sat})$$

So we require that there is a transition β with $\mathrm{dest}_\beta = \mathrm{src}_\alpha$ (due to the leading $\bigvee \ldots$) and a valuation (due to the existential quantifier) such that β is applicable (due to guard_β) and α is applicable afterwards (due to $\mathrm{up}_\beta(\mathrm{guard}_\alpha) \wedge \bigwedge_{\rho \in \varphi_{nm}} \mathrm{up}_\beta(\tau_\rho)$, as we will strengthen α's guard with $\bigwedge_{\rho \in \varphi_{nm}} \tau_\rho$ after instantiating the parameters in the templates). Thus, for α_4 we require

$$\exists x, y.\ x < 0 \wedge y > 0 \wedge c_x \cdot x + c_y \cdot y \geq c \qquad (\mathrm{sat}\text{-}\alpha_4)$$

due to the transition α_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, (τ_ρ-li) and (sat) can result in large formulas, which leads to performance issues. Hence, our implementation only considers transitions β with $\mathrm{src}_\beta \neq \mathrm{dest}_\beta$ when constructing (τ_ρ-li) and (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 φ_{md} is empty:

$$\forall \mathcal{V}(\mathcal{T}).\ \mathrm{guard}_\alpha \wedge \bigwedge_{\rho \in \varphi_{nm}} \tau_\rho \implies \bigwedge_{\rho \in \varphi_{nm}} \mathrm{up}_\alpha(\rho) \qquad (\mathrm{nt})$$

In our example, (nt) equals (ρ-ci-α_4) as φ_{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 $\boldsymbol{w}$ corresponds to the weight of the i^{th} soft requirement χ_i and $\mathbf{solve}(\zeta, \boldsymbol{\chi}, \boldsymbol{w})$ searches an instantiation σ of the parameters such that $\sigma \models \zeta$ and $\sum_{\substack{1 \leq i \leq |\boldsymbol{\chi}| \\ \sigma \models \chi_i}} w_i$ is maximized. We explain how to implement **solve** in Sect. IV-H. The weights are chosen in such a way that a solution σ is preferred over σ' if σ turns more templates τ_ρ into local invariants than σ': The weight $m+2 = |\varphi_{nm}|+2$ of the formulas resulting from (τ_ρ-li) (where $|\varphi_{nm}|$ is the number of inequations in φ_{nm}) ensures that each formula from (τ_ρ-li) has a higher weight than the sum of all other soft requirements (ρ-ci) $\vee$ (ρ-md) and (nt).

Note that Step 15 updates guard_α in each iteration and φ_{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 (some-ci) $\vee$ (some-md), which ensures that some $\rho \in \varphi_{nm}$ becomes part of φ_i or φ_{md}. Moreover, the hard requirement (τ-si) ensures that each $\sigma(\tau_\rho)$ becomes part of φ_{si}, so Alg. 2 never adds elements to φ_{nm}.

Input: A simple loop α
Output: A set of strengthened variants of α

if $\varphi_{nm} = \varnothing$ **: return** $\varnothing$
else : $res \leftarrow \varnothing$
while $\varphi_{nm} \neq \varnothing$ **:**
  $i \leftarrow 0;\ m \leftarrow |\varphi_{nm}|$
  for $\rho \in \varphi_{nm}$ **:**
    $i \leftarrow i + 1$
    $\chi_i \leftarrow (\tau_\rho\text{-li});\quad w_i \leftarrow m + 2$
    $\chi_{i+m} \leftarrow (\rho\text{-ci}) \vee (\rho\text{-md});\quad w_{i+m} \leftarrow 1$
  if $\varphi_{md} = \varnothing$ **:** $\chi_{i+m+1} \leftarrow (\text{nt});\quad w_{i+m+1} \leftarrow 1$
  $\zeta \leftarrow (\tau\text{-si}) \wedge ((\text{some-ci}) \vee (\text{some-md})) \wedge (\text{sat})$
  $\sigma \leftarrow \mathbf{solve}(\zeta, \boldsymbol{\chi}, \boldsymbol{w})$
  return res **if solve** failed
  for $\rho \in \varphi_{nm}$ *where* $\sigma(\tau_\rho)$ *is not a local invariant* **:**
    $res \leftarrow res \cup \{\mathrm{lhs}_\alpha \to \mathrm{rhs}_\alpha\ [\mathrm{guard}_\alpha \wedge \neg\sigma(\tau_\rho)]\}$
  $\mathrm{guard}_\alpha \leftarrow \mathrm{guard}_\alpha \wedge \bigwedge_{\rho \in \varphi_{nm}} \sigma(\tau_\rho)$
return $\{\alpha\} \cup res$

Algorithm 2: deduceInvariants

In our example, (τ-si-α_4), (ρ-ci-α_4), (τ_ρ-li-α_4), and (sat-α_4) are valid if $c_x = -1$ and $c_y = c = 0$. Hence, Alg. 2 successfully generates the local invariant $-x \geq 0$, i.e., $x \leq 0$. Afterwards, we can apply **Nonterm** to the strengthened loop as in Ex. 11.

Example 16 (Deducing Simple Invariants for α_2)**.** *Reconsider the simple loop α_2 from Ex. 1, where $\varphi_i = \varphi_{md} = \varnothing$ and $\varphi_{nm} = \{x \geq 0\}$ as α_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 $y \in \mathcal{V}(\mathrm{up}_\alpha(x))$. So ($\tau$-si) becomes*

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

Again, (some-ci) $\vee$ (some-md) coincides with (ρ-ci) $\vee$ (ρ-md) for $\rho: x \geq 0$.

$$\forall x, y.\ x \geq 0 \wedge c_x \cdot x + c_y \cdot y \geq c \implies x - y \geq 0 \ \vee \quad (\rho\text{-ci-}\alpha_2)$$

$$\forall x, y.\ c_x \cdot x + c_y \cdot y \geq c \wedge x - y \geq 0 \implies x \geq 0 \quad (\rho\text{-md-}\alpha_2)$$

Next, (τ_ρ-li) gives rise to the requirement

$$\forall x, y.\ x \geq 0 \implies c_x \cdot x + c_y \cdot y \geq c. \quad (\tau_\rho\text{-li-}\alpha_2)$$

Moreover, (sat) becomes

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

Finally, (nt) equals (ρ-ci-α_2). Thus, the hard requirement ζ is

$$(\tau\text{-si-}\alpha_2) \wedge ((\rho\text{-ci-}\alpha_2) \vee (\rho\text{-md-}\alpha_2)) \wedge (\text{sat-}\alpha_2).$$

The soft requirements are (τ_ρ-li-α_2), (ρ-ci-α_2)$\vee$(ρ-md-α_2), and (ρ-ci-α_2) with weights 3, 1, and 1, respectively. The valuation $\sigma = \{c_x \mapsto 0, c_y \mapsto 1, c \mapsto 0\}$ satisfies ζ and (ρ-ci-α_2) $\vee$ (ρ-md-α_2), but not the other soft constraints. As $\zeta \wedge$ (τ_ρ-li-α_2) and $\zeta \wedge$ (ρ-ci-α_2) are unsatisfiable, σ 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 $\mathcal{V}(\mathcal{T})$. 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[6] 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.

	LoAT	T2	VeryMax
NO	462	420	392
YES	0	607	623
MAYBE	760	195	207
Unique NO	22	9	23
Avg. time	8.3s	8.8s	13.8s

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

[6]Note that rational constants can be eliminated by multiplying with the least common multiple.

[7]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.

	LoAT	Anant
NO	24	25
MAYBE	5	4
Unique NO	4	5
Avg. time	0.5s	32.5s

Finally, we compared LoAT with the tools from the category *Termination of C Integer Programs* at TermComp '19[8] (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.

	LoAT	AProVE	Ultimate	VeryMax
NO	96	99	88	102
YES	0	214	206	212
MAYBE	239	22	41	21
Unique NO	2	0	0	2
Avg. time	3.1s	6.3s	8.7s	5.2s

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 $\varphi_{ci} = \varphi_{si} = \mathtt{true}$, 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 α and a *stem*, i.e., a path from the program's entry point to α. 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 α. However, if it cannot find a recurrent set it may still successfully enforce monotonicity of guard_α and hence allow us to accelerate α.

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.

[8] Ultimate and AProVE were also the two most powerful tools in the "termination" category for C programs at SV-COMP '19 [3].

References

[1] R. Bagnara, A. Pescetti, A. Zaccagnini, and E. Zaffanella. "PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis". In: *CoRR* abs/cs/0512056 (2005).

[2] A. M. Ben-Amram, J. J. Doménech, and S. Genaim. "Multiphase-Linear Ranking Functions and their Relation to Recurrent Sets". In: *CoRR* abs/1811.07340 (2018).

[3] D. Beyer. "Automatic Verification of C and Java Programs: SV-COMP 2019". In: *TACAS '19*. LNCS 11429. 2019, pp. 133–155.

[4] M. Bofill, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. "The Barcelogic SMT Solver". In: *CAV '08*. LNCS 5123. 2008, pp. 294–298.

[5] C. Borralleras, M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. "Proving Termination Through Conditional Termination". In: *TACAS '17*. LNCS 10205. 2017, pp. 99–117.

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. "Linear Ranking with Reachability". In: *CAV '05*. LNCS 3576. 2005, pp. 491–504.

[7] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl. "Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode". In: *FoVeOOS '11*. LNCS 7421. 2011, pp. 123–141.

[8] M. Brockschmidt and A. Rybalchenko. *TermComp Proposal: Pushdown Systems as a Model for Programs with Procedures*. 2014. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/SMTPushdownPrograms.pdf.

[9] M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. "Compositional Safety Verification with Max-SMT". In: *FMCAD '15*. 2015, pp. 33–40.

[10] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. "Analyzing Runtime and Size Complexity of Integer Programs". In: *ACM TOPLAS* 38.4 (2016), 13:1–13:50.

[11] M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. "T2: Temporal Property Verification". In: *TACAS '16*. LNCS 9636. 2016, pp. 387–393.

[12] Y. Chen, M. Heizmann, O. Lengál, Y. Li, M. Tsai, A. Turrini, and L. Zhang. "Advanced Automata-Based Algorithms for Program Termination Checking". In: *PLDI '18*. 2018, pp. 135–150.

[13] H. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. W. O'Hearn. "Proving Nontermination via Safety". In: *TACAS '14*. LNCS 8413. 2014, pp. 156–171.

[14] B. Cook, C. Fuhs, K. Nimkar, and P. W. O'Hearn. "Disproving Termination with Overapproximation". In: *FMCAD '14*. 2014, pp. 67–74.

[15] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS '08*. LNCS 4963. 2008, pp. 337–340.

[16] J. J. Doménech and S. Genaim. "iRankFinder". In: *WST '18*. 2018, p. 83.

[17] J. J. Doménech, S. Genaim, and P. Gallagher. "Control-Flow Refinement via Partial Evaluation". In: *WST '18*. 2018, pp. 55–59.

[18] B. Dutertre. "Yices 2.2". In: *CAV '14*. LNCS 8559. 2014, pp. 737–744.

[19] A. Farzan and Z. Kincaid. "Compositional Recurrence Analysis". In: *FMCAD '15*. 2015, pp. 57–64.

[20] A. Flores-Montoya. "Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations". In: *FM '16*. LNCS 9995. 2016, pp. 254–273.

[21] F. Frohn, M. Naaf, J. Hensel, M. Brockschmidt, and J. Giesl. "Lower Runtime Bounds for Integer Programs". In: *IJCAR '16*. LNCS 9706. 2016, pp. 550–567.

[22] F. Frohn and J. Giesl. "Proving Non-Termination via Loop Acceleration". In: *CoRR* abs/1905.111187 (2019).

[23] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. "Analyzing Program Termination and Complexity Automatically with AProVE". In: *J. Autom. Reasoning* 58.1 (2017), pp. 3–31.

[24] J. Giesl, A. Rubio, C. Sternagel, J. Waldmann, and A. Yamada. "The Termination and Complexity Competition". In: *TACAS '19*. LNCS 11429. 2019, pp. 156–166.

[25] L. Gonnord and N. Halbwachs. "Combining Widening and Acceleration in Linear Relation Analysis". In: *SAS '06*. LNCS 4134. 2006, pp. 144–160.

[26] S. Gulwani, S. Jain, and E. Koskinen. "Control-Flow Refinement and Progress Invariants for Bound Analysis". In: *PLDI '09*. 2009, pp. 375–385.

[27] A. Gupta, T. A. Henzinger, R. Majumdar, A. Rybalchenko, and R. Xu. "Proving Non-Termination". In: *POPL '08*. 2008, pp. 147–158.

[28] A. Heck. *Introduction to Maple (2. ed.)* Springer, 1996.

[29] B. Jeannet, P. Schrammel, and S. Sankaranarayanan. "Abstract Acceleration of General Linear Loops". In: *POPL '14*. 2014, pp. 529–540.

[30] C. Kop and N. Nishida. "Constrained Term Rewriting tooL". In: *LPAR '15*. LNCS 9450. 2015, pp. 549–557.

[31] D. Kroening, M. Lewis, and G. Weissenbacher. "Under-Approximating Loops in C Programs for Fast Counterexample Detection". In: *FMSD* 47.1 (2015), pp. 75–92.

[32] D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. "Proving Termination of Imperative Programs Using Max-SMT". In: *FMCAD '13*. 2013, pp. 218–225.

[33] D. Larraz, K. Nimkar, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. "Proving Non-Termination Using Max-SMT". In: *CAV '14*. LNCS 8559. 2014, pp. 779–796.

[34] T. C. Le, S. Qin, and W. Chin. "Termination and Non-Termination Specification Inference". In: *PLDI '15*. 2015, pp. 489–498.

[35] J. Leike and M. Heizmann. "Geometric Nontermination Arguments". In: *TACAS '18*. LNCS 10806. 2018, pp. 266–283.

[36] K. Madhukar, B. Wachter, D. Kroening, M. Lewis, and M. K. Srivas. “Accelerating Invariant Generation”. In: *FMCAD ’15*. 2015, pp. 105–111.

[37] A. Podelski and A. Rybalchenko. “A Complete Method for the Synthesis of Linear Ranking Functions”. In: *VMCAI ’04*. LNCS 2937. 2004, pp. 239–251.

[38] A. Stump, G. Sutcliffe, and C. Tinelli. “StarExec: A Cross-Community Infrastructure for Logic Solving”. In: *IJCAR ’14*. LNCS 8562. 2014, pp. 367–373.

[39] *TPDB*. URL: http://termination-portal.org/wiki/TPDB.

[40] C. Urban, A. Gurfinkel, and T. Kahsai. “Synthesizing Ranking Functions from Bits and Pieces”. In: *TACAS ’16*. LNCS 9636. 2016, pp. 54–70.

[41] H. Velroyen and P. Rümmer. “Non-Termination Checking for Imperative Programs”. In: *TAP ’08*. 2008, pp. 154–170.

[42] S. Wolfram. “Mathematica: A System for Doing Mathematics by Computer”. In: *SIAM Review* 34.3 (1992), pp. 519–522.