Improving Automatic Complexity Analysis of Integer Programs*

Jürgen Giesl, Nils Lommen, Marcel Hark, and Fabian Meyer

LuFG Informatik 2, RWTH Aachen University, Aachen, Germany
{giesl,lommen,marcel.hark,fabian.meyer}@cs.rwth-aachen.de

Abstract. In [16], we developed an approach for automatic complexity analysis of integer programs, based on an alternating modular inference of upper runtime and size bounds for program parts. In this paper, we show how recent techniques to improve automated termination analysis of integer programs (like the generation of multiphase-linear ranking functions and control-flow refinement) can be integrated into our approach for the inference of runtime bounds. The power of the resulting approach is demonstrated by an extensive experimental evaluation with our new re-implementation of the corresponding tool KoAT.

1 Introduction

There are many techniques and tools for automated complexity analysis of programs, e.g., [2–6, 8–10, 16, 17, 22–24, 29, 32, 33, 40, 43, 46, 47]. Most of them infer variants of (mostly linear) polynomial ranking functions (see, e.g., [15, 44]) which are then combined to get a runtime bound for the overall program. However, approaches based on linear ranking functions are incomplete for termination and thus also for complexity analysis. For example, consider the loop from [12, 38] in Fig. 1, which terminates, but does not admit a linear ranking function. Its runtime is linear in the initial values of x and y, if they are positive initially. The reason is that if $y > 0$, then x grows first but it is decreased with the same "speed" once y has become negative.

```
while $x > 0$ do
    $x \leftarrow x + y$
    $y \leftarrow y - 1$
```

Fig. 1: Loop without Linear Ranking Function

Recently so-called multiphase-linear ranking functions have gained interest (see, e.g., [12, 13, 38, 50]). For loops as in Fig. 1, ranking functions of this form detect that the program has two phases: first y is decremented until it is negative. Afterwards, x is decremented until it is negative and the loop terminates. In [12], it is shown that the existence of a multiphase-linear ranking function for a loop implies linear runtime complexity. In the current paper, we embed multiphase-linear ranking functions into our modular approach for complexity analysis of integer programs from [16]. In contrast to [12], we infer multiphase-linear ranking functions for parts of the program and combine the so-obtained bounds to an overall runtime bound. In this way, we obtain a powerful technique which is able

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

```
while $x < 0$ do
    if $y < z$ then
        $y \leftarrow y - x$
    else
        $x \leftarrow x + 1$
```

Fig. 2: Original Loop

```
while $x < 0 \land y < z$ do
    $y \leftarrow y - x$
while $x < 0 \land y \geq z$ do
    $x \leftarrow x + 1$
```

Fig. 3: After Control-Flow Refinement

to infer finite runtime bounds for programs that contain loops such as Fig. 1.

Moreover, different forms of control-flow refinement were used to improve the automatic termination and complexity analysis of programs further, see, e.g., [20, 22]. The basic idea is to gain "more information" on the values of variables to sort out certain paths in the program. For example, the control-flow refinement technique from [20] detects that the programs in Fig. 2 and Fig. 3 are equivalent. Clearly, the program in Fig. 3 is easier to analyze as the two consecutive loops do not interfere with each other: x and z are constants in its first loop, while y and z are constants in its second loop. We show how to integrate the technique for control-flow refinement from [20] into our modular analysis in a non-trivial way. This increases the power of our approach further.

Structure: We first recapitulate our approach from [16] in Sect. 2. Afterwards, we adapt it to multiphase-linear ranking functions in Sect. 3. In Sect. 4, we discuss how to incorporate control-flow refinement from [20] into our analysis. We provide an extensive experimental evaluation of our corresponding new version of the tool KoAT [42] and compare it with existing tools in Sect. 5. Finally, we discuss related work and conclude (Sect. 6). All proofs can be found in App. A.

2 Preliminaries

In this section we recapitulate our approach for complexity analysis from [16]. We first introduce *constraints*, which are used in the guards of programs.

Definition 1 (Constraints). *Let $\mathcal{V}$ be a set of variables. The set of* constraints *$\mathcal{C}(\mathcal{V})$ over $\mathcal{V}$ is the smallest set containing $e_1 \leq e_2$ for all polynomials $e_1, e_2 \in \mathbb{Z}[\mathcal{V}]$ and $c_1 \land c_2$ for all $c_1, c_2 \in \mathcal{C}(\mathcal{V})$.*

In addition to "$\leq$", we also use relations like "$>$" and "$=$", which can be simulated by constraints (e.g., $e_1 > e_2$ is equivalent to $e_2 + 1 \leq e_1$ when regarding integers).

Now we define the notion of integer programs which we use in this paper. Instead of **while** loops as in Fig. 1 to 3, we use a formalism based on transitions (which of course also allows us to represent **while** programs easily).

Definition 2 (Integer Program). *An* integer program *$\mathcal{P}$ over a set of variables $\mathcal{V}$ is a tuple $(\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ of*

- *a finite set of* program variables *$\mathcal{PV} \subseteq \mathcal{V}$,*
- *a finite set of* locations *$\mathcal{L}$ with a distinguished* initial location *$\ell_0 \in \mathcal{L}$, and*
- *a finite set of* transitions *$\mathcal{T}$. A transition is a tuple $(\ell, \tau, \eta, \ell')$ consisting of*
  1. *the* start location *$\ell \in \mathcal{L}$ and the* target location *$\ell' \in \mathcal{L} \setminus \{\ell_0\}$,*
  2. *the* guard *$\tau \in \mathcal{C}(\mathcal{V})$ of t, and*

3. *the* update function $\eta\colon \mathcal{PV} \to \mathbb{Z}[\mathcal{V}]$ *of* t*, mapping every program variable to an update polynomial.*

We call $\mathcal{TV} = \mathcal{V} \setminus \mathcal{PV}$ *the set of* temporary variables*.*

Note that the initial location has *no* incoming transitions. The transitions $(\ell_0, \ldots)$ whose start location is ℓ_0 are called *initial* transitions.

Thus, integer programs contain two kinds of non-determinism. Non-deterministic branching is realized by multiple transitions with the same start location whose guards are non-exclusive. Non-deterministic sampling is modeled by temporary variables (which can be restricted in the guard of a transition). Temporary variables are not updated in the program. Intuitively, these variables are set by an adversary trying to "sabotage" the program in order to obtain long runtimes.

Example 3. Consider the integer program in Fig. 4 over the program variables $\mathcal{PV} = \{x, y, z\}$, the locations $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$, and the transitions $\mathcal{T} = \{t_0, t_1, t_2, t_3\}$. In Fig. 4, we omitted trivial guards, i.e., $\tau = \texttt{true}$, and trivial updates, i.e., updates of the form $\eta(v) = v$. This integer program corresponds to two nested loops: the inner loop is given by t_2, the outer loop by t_1 and t_3.

Transition t_0 just forwards the input values. If $z > 0$, t_1 sets x and y to $z - 1$. Then, t_2 decrements y by 1 and updates x to $x+y$ repeatedly as long as $x > 0$ (i.e., it corresponds to the loop in Fig. 1). Transition t_3 decrements z by 1 and leads back to the starting point of the outer loop.

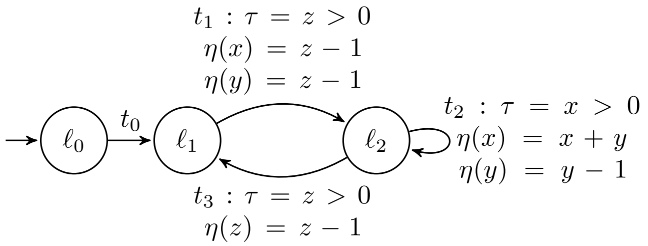


Fig. 4: Integer Program with Nested Loops

Note that t_2 and t_3 correspond to a non-deterministic branching as their guards are non-exclusive. If t_0 had the update $\eta(x) = u$ and the guard $u > 0$, then this would correspond to a non-deterministic sampling of a positive value.

From now on, we fix an integer program $\mathcal{P}$ over the variables $\mathcal{V}$. A mapping $\sigma : \mathcal{V} \to \mathbb{Z}$ is called a *state* and Σ denotes the set of all states. We also apply states to arithmetic expressions e and constraints c, where the number $\sigma(e)$ resp. the Boolean value $\sigma(c)$ results from e resp. c by replacing each variable v by $\sigma(v)$.

Definition 4 (Evaluation of Integer Programs). *A* configuration *is an element of* $\mathcal{L} \times \Sigma$*. For two configurations* (ℓ, σ) *and* (ℓ', σ')*, and a transition* $t = (\ell_t, \tau, \eta, \ell'_t) \in \mathcal{T}$*,* $(\ell, \sigma) \to_t (\ell', \sigma')$ *is an* evaluation *step by* t *if*

- $\ell = \ell_t$ *and* $\ell' = \ell'_t$*,*
- $\sigma(\tau) = \texttt{true}$*, and*
- *for every program variable* $v \in \mathcal{PV}$ *we have* $\sigma(\eta(v)) = \sigma'(v)$*.*

We denote the union of all relations $\to_t$ *for* $t \in \mathcal{T}$ *by* $\to_{\mathcal{T}}$*. Whenever it is clear from the context, we omit the transition* t *resp. the set* $\mathcal{T}$ *in the index. We also abbreviate* $(\ell_0, \sigma_0) \to_{t_1} (\ell_1, \sigma_1) \cdots \to_{t_k} (\ell_k, \sigma_k)$ *by* $(\ell_0, \sigma_0) \to^k (\ell_k, \sigma_k)$*.*

Example 5. For the integer program in Fig. 4, when denoting program states σ as tuples $(\sigma(x), \sigma(y), \sigma(z)) \in \mathbb{Z}^3$, we have $(\ell_0, (0,0,2)) \to_{t_0} (\ell_1, (0,0,2)) \to_{t_1} (\ell_2, (1,1,2)) \to_{t_2} (\ell_2, (2,0,2)) \to_{t_3} (\ell_1, (2,0,1))$.

For an integer program, the (worst-case) runtime complexity w.r.t. an initial state σ_0 is defined to be the length of the longest evaluation starting in σ_0.

Definition 6 (Runtime Complexity). *The (worst-case)* runtime complexity *of $\mathcal{P}$ is the function* $\mathrm{rc} : \Sigma \to \overline{\mathbb{N}}$ *with* $\overline{\mathbb{N}} = \mathbb{N} \cup \{\omega\}$ *and* $\mathrm{rc}(\sigma_0) = \sup\{k \in \mathbb{N} \mid \ell_k \in \mathcal{L}, \sigma_k \in \Sigma, (\ell_0, \sigma_0) \to^k (\ell_k, \sigma_k)\}$ *for all* $\sigma_0 \in \Sigma$.

As in [16], our approach combines bounds for program parts. We restrict ourselves to bounds that represent weakly monotonically increasing functions. Such bounds have the advantage that they can easily be "composed", i.e., if f and g are both weakly monotonically increasing upper bounds, then so is $f \circ g$.

Definition 7 (Bounds). *The set of* bounds $\mathcal{B}$ *is the smallest set with* $\overline{\mathbb{N}} \subseteq \mathcal{B}$, $\mathcal{PV} \subseteq \mathcal{B}$, $b_1 + b_2 \in \mathcal{B}$, $b_1 \cdot b_2 \in \mathcal{B}$, *and* $k^b \in \mathcal{B}$ *for all* $k \in \mathbb{N}$ *and* $b, b_1, b_2 \in \mathcal{B}$.

A bound which is only constructed from $\mathbb{N}$, $\mathcal{PV}$, $+$, *and* $\cdot$ *is called* polynomial. *A polynomial bound of degree at most* 1 *is called* linear.

For any $\sigma \in \Sigma$, $|\sigma|$ denotes the state with $|\sigma|(v) = |\sigma(v)|$ for all $v \in \mathcal{V}$. Clearly, a bound $b \in \mathcal{B}$ induces a weakly monotonic function on states by mapping any $\sigma \in \Sigma$ to $|\sigma|(b) \in \overline{\mathbb{N}}$. Then, $|\sigma| \le |\sigma'|$ implies $|\sigma|(b) \le |\sigma'|(b)$. As usual, we compare functions pointwise, i.e., $|\sigma| \le |\sigma'|$ means that $|\sigma|(v) \le |\sigma'|(v)$ for all $v \in \mathcal{V}$.

Example 8. For $\mathcal{PV} = \{x, y\}$, we have ω, x^2, $x+y$, $2^{x^2+y} \in \mathcal{B}$. Here, x^2 and $x+y$ are polynomial bounds and $x+y$ is linear. Consider the state σ with $\sigma(x) = 1$ and $\sigma(y) = -2$. Then, $|\sigma|(x+y) = |1| + |-2| = 3$.

To over-approximate the runtime complexity, we now introduce the concepts of runtime and size bounds. A runtime bound for a transition $t \in \mathcal{T}$ over-approximates the maximal number of occurrences of that transition in any evaluation starting with the initial state $\sigma_0 \in \Sigma$. Here, $\to^* \circ \to_t$ denotes the relation describing arbitrary many evaluation steps followed by a step with transition t.

Definition 9 (Runtime Bound). *The function* $\mathcal{RB} : \mathcal{T} \to \mathcal{B}$ *is a* runtime bound *if for all* $t \in \mathcal{T}$ *and all states* $\sigma_0 \in \Sigma$ *we have*

$$|\sigma_0|(\mathcal{RB}(t)) \ \ge \ \sup\{k \in \mathbb{N} \mid \ell \in \mathcal{L}, \sigma \in \Sigma, (\ell_0, \sigma_0)\, (\to^* \circ \to_t)^k\, (\ell, \sigma)\}.$$

Note that we require the runtime bound to only contain *program variables* since the values of temporary variables are "set by the adversary".

Example 10. For the program in Fig. 4, the technique from [16] obtains the following runtime bound. Trivially, $\mathcal{RB}(t_0) = 1$, as t_0 can only be applied once in any evaluation. Since the outer loop is only executed if $z > 0$ and every iteration of the outer loop decreases z by 1, we get $\mathcal{RB}(t_1) = \mathcal{RB}(t_3) = z$, i.e., these transitions can occur at most $|z_0|$ times, if z has the value $z_0 \in \mathbb{Z}$ initially. However, the implementation of [16] in the original version of the tool KoAT cannot infer a finite runtime bound for t_2 since this transition does not admit a linear ranking function, i.e., a linear function which decreases by at least one and is bounded from below for each iteration of the loop. Intuitively, the reason is that x is bounded, but it does not decrease in every iteration. In contrast, y decreases in every iteration, but it is not bounded. In Sect. 3, we will show how to improve our approach for complexity analysis such that it obtains a finite runtime bound for transitions like t_2 (see Ex. 21).

The following corollary shows that every runtime bound $\mathcal{RB}$ directly yields an upper bound for the program's runtime complexity: Instead of over-approximating the runtime complexity of the full program at once, one can compute runtime bounds for each transition separately and simply add these bounds.

Corollary 11 (Over-Approximating rc**).** *Let $\mathcal{RB}$ be a runtime bound. Then for all states $\sigma_0 \in \Sigma$ we have $|\sigma_0| \left(\sum_{t \in \mathcal{T}} \mathcal{RB}(t)\right) \ \geq \ \mathrm{rc}(\sigma_0)$.*

The framework in [16] performs a *modular* analysis of the program, i.e., parts of the program are analyzed as standalone programs and the results are then lifted to contribute to the overall analysis. For example, for the integer program in Fig. 4, the inner loop t_2 is analyzed separately in order to compute its runtime bound. But to lift a local runtime bound of t_2 to a runtime bound of t_2 in the full program, one has to take into account that the values of the variables when executing t_2 are not the input values of the program, but the values that the variables have after an execution of the previous transition t_1.

So to compute the runtime bound of a transition t', our approach considers all transitions t that can occur directly before t' in evaluations and it needs size bounds $\mathcal{SB}(t, v)$ to over-approximate the absolute values that the variables $v \in \mathcal{PV}$ may have *after* these "previous" transitions t. (This intuition will later be formalized in Thm. 20.) Here, we call $\mathcal{RV} = \mathcal{T} \times \mathcal{PV}$ the set of *result variables*.

Definition 12 (Size Bound). *The function $\mathcal{SB} : \mathcal{RV} \to \mathcal{B}$ is a* size bound *if for all $(t, v) \in \mathcal{RV}$ and all states $\sigma_0 \in \Sigma$ we have*

$$|\sigma_0| \left(\mathcal{SB}(t, v)\right) \ \geq \ \sup\{|\sigma(v)| \in \mathbb{N} \mid \ell \in \mathcal{L}, \sigma \in \Sigma, (\ell_0, \sigma_0) \left(\to^* \circ \to_t\right) (\ell, \sigma)\}.$$

Example 13. Consider again the program in Fig. 4. Here, $\mathcal{SB}(t_0, v) = v$ for $v \in \{x, y, z\}$, because t_0 does not change any variable. So if $(\ell_0, \sigma_0) \to_{t_0} (\ell_1, \sigma_1)$ then $|\sigma_0| \left(\mathcal{SB}(t_0, v)\right) = |\sigma_0| (v) = |\sigma_1| (v)$. Moreover, $\mathcal{SB}(t_1, z) = \mathcal{SB}(t_2, z) = \mathcal{SB}(t_3, z) = z$ as z is never increased in the program. For the computation of $\mathcal{SB}(t_1, x)$ and $\mathcal{SB}(t_1, y)$, the approach of [16] sums up the values of $\mathcal{SB}(t_0, z)$ and $\mathcal{SB}(t_3, z)$ (since t_0 and t_3 are the only transitions that can occur directly before t_1) and uses this as the "incoming size" of z. Hence, it obtains $\mathcal{SB}(t_1, x) = \mathcal{SB}(t_1, y) = z + z = 2 \cdot z$. The approach of [16] cannot compute finite size bounds for (t_2, x), (t_2, y), (t_3, x), and (t_3, y), since it needs a runtime bound for t_2 to over-approximate how often the "previous" transition t_2 may have been executed. In contrast, our results from Sect. 3 will enable the computation of finite size bounds for all result variables of this program, see Ex. 21.

So size bounds on previous transitions are needed to compute runtime bounds, and similarly, runtime bounds are needed to compute size bounds. The algorithm for the computation of size bounds in [16] is not needed to understand the techniques presented in the current paper and thus, we use it as a black box.

3 Runtime Bounds by Multiphase Ranking Functions

The approach for computing runtime bounds in [16] relies on polynomial ranking functions (see, e.g., [15, 44]). In this section, we extend this approach to so-called multiphase-linear ranking functions (MΦRFs) (see, e.g., [12, 13, 38, 50]). Our experiments in Sect. 5 demonstrate that this improves its power significantly.

In [12] it was already shown how to obtain a runtime bound from an MΦRF for a full integer program. We now adapt this result to our modular approach which allows for the computation of MΦRFs for parts of the program (Thm. 20).

3.1 Multiphase-Linear Ranking Functions

As mentioned, the idea of ranking functions is to construct a function which decreases by at least one in every evaluation step when a specific transition is applied. Moreover, the ranking function has to be non-negative before we apply a transition. Thus, if the function becomes negative, then the program terminates.

An MΦRF extends this idea and uses a ranking function f_i for every "phase" $1 \leq i \leq d$ of a program. When the phases 1 to $i-1$ are finished, the functions $f_1, \ldots, f_{i-1}$ remain negative and decreasing, but now the function f_i becomes decreasing as well. If all functions are negative, then the program terminates.

Def. 14 corresponds to so-called nested MΦRFs from [12, 38]. Here, the sum of f_{i-1} and f_i must be larger than the updated function f_i for all i. We set f_0 to 0. Then $f_0 + f_1 = f_1$ must be decreasing with each update. If f_1 becomes negative, then $f_1 + f_2 < f_2$ and thus, f_2 has to be decreasing with every update, and so on until f_d becomes decreasing. The program eventually terminates, since f_d must be non-negative whenever the program can be executed further. We restrict ourselves to such "nested" MΦRFs, as they are particularly easy to automate (i.e., one does not have to consider the mapping of evaluation steps to the different phases). As usual, we use an SMT solver to search for MΦRFs automatically.

In contrast to [12, 38], we define MΦRFs for sub-programs $\mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ which is crucial for our modular approach (see Thm. 20). Let $\mathbb{Z}[\mathcal{PV}]_{\text{lin}}$ denote the set of linear polynomials (i.e., of degree at most 1) over $\mathbb{Z}$ in the variables $\mathcal{PV}$.

Definition 14 (MΦRFs for Sub-Programs). *Let $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ and $d \geq 1$. A tuple $f = (f_1, \ldots, f_d)$ of functions $f_1, \ldots, f_d : \mathcal{L} \to \mathbb{Z}[\mathcal{PV}]_{\text{lin}}$ is an* MΦRF of depth *d for $\mathcal{T}'_>$ and $\mathcal{T}'$ if for all evaluation steps $(\ell, \sigma) \to_t (\ell', \sigma')$:*

(a) If $t \in \mathcal{T}'_>$, then we have $\sigma\,(f_{i-1}(\ell)) + \sigma\,(f_i(\ell)) \geq \sigma'\,(f_i(\ell')) + 1$ for all $1 \leq i \leq d$ and $\sigma\,(f_d(\ell)) \geq 0$.
(b) If $t \in \mathcal{T}' \setminus \mathcal{T}'_>$, then we have $\sigma\,(f_i(\ell)) \geq \sigma'\,(f_i(\ell'))$ for all $1 \leq i \leq d$.

Here, we set $f_0(\ell) = 0$ for all $\ell \in \mathcal{L}$. We say that $\mathcal{T}' \setminus \mathcal{T}'_>$ is the set of non-increasing *transitions and $\mathcal{T}'_>$ is the set of* decreasing *transitions of the MΦRF f.*

The definitions of MΦRFs and of linear ranking functions coincide in the special case of a single phase (i.e., if $d = 1$). Note that for $d > 1$, the requirement for decreasing transitions in (a) does not imply the requirement for non-increasing transitions in (b). The reason is that for decreasing transitions, f_i may increase in the beginning (if f_{i-1} is large enough), because eventually f_{i-1} will become negative. In contrast, for non-increasing transitions, (b) prohibits any increase of f_i, since the MΦRF does not represent any bound on the number of applications of these non-increasing transitions. Thus, we cannot replace (b) by $\sigma\left(f_{i-1}(\ell)\right) + \sigma\left(f_i(\ell)\right) \geq \sigma'\left(f_i(\ell')\right)$, because then such transitions might make f_i arbitrarily large if their repeated application does not change a positive f_{i-1}.

Example 15. Consider again the integer program in Fig. 4 and let $\mathcal{T}'_> = \{t_2\}$ and $\mathcal{T}' = \{t_2, t_3\}$. (See Alg. 1 for our heuristic to choose $\mathcal{T}'_>$ and $\mathcal{T}'$.) An execution of the loop $\mathcal{T}'_> = \{t_2\}$ has two phases: In the first phase, both x and y are positive. In every iteration, x increases until y is 0. The second phase starts when y is negative. This phase ends when x is negative, since then the guard $x > 0$ is not satisfied anymore. We now show that the tuple (f_1, f_2) is an MΦRF for $\mathcal{T}'_> = \{t_2\}$ and $\mathcal{T}' = \{t_2, t_3\}$ where $f_1(\ell_1) = f_1(\ell_2) = y + 1$ and $f_2(\ell_1) = f_2(\ell_2) = x$.

Since t_2 has the update function η with $\eta(x) = x + y$ and $\eta(y) = y - 1$, for any evaluation step $(\ell_2, \sigma) \to_{t_2} (\ell_2, \sigma')$, we have $\sigma'(x) = \sigma(\eta(x)) = \sigma(x) + \sigma(y)$ and $\sigma'(y) = \sigma(\eta(y)) = \sigma(y) - 1$. Hence, $\sigma\left(f_0(\ell_2)\right) + \sigma\left(f_1(\ell_2)\right) = 0 + \sigma\left(y+1\right) = \sigma\left(y\right) + 1 = \sigma'\left(y+1\right) + 1 = \sigma'\left(f_1(\ell_2)\right) + 1$ and $\sigma\left(f_1(\ell_2)\right) + \sigma\left(f_2(\ell_2)\right) = \sigma\left(y+1\right) + \sigma\left(x\right) = \sigma\left(x\right) + \sigma\left(y\right) + 1 = \sigma'\left(x\right) + 1 = \sigma'\left(f_2(\ell_2)\right) + 1$. Moreover, due to the guard $x > 0$, $\sigma\left(x > 0\right) = \texttt{true}$ implies $\sigma\left(f_2(\ell_2)\right) = \sigma\left(x\right) \geq 0$. Note that neither $y + 1$ (as y is not bounded) nor x (as x might increase) are ranking functions for t_2.

Similarly, since the update function η of t_3 does not modify x and y, for every evaluation step $(\ell_2, \sigma) \to_{t_3} (\ell_1, \sigma')$, we have $\sigma'(x) = \sigma(\eta(x)) = \sigma(x)$ and $\sigma'(y) = \sigma(\eta(y)) = \sigma(y)$. Hence, $\sigma\left(f_1(\ell_2)\right) = \sigma\left(y+1\right) = \sigma\left(y\right) + 1 = \sigma'\left(y+1\right) = \sigma'\left(f_1(\ell_1)\right)$ and $\sigma\left(f_2(\ell_2)\right) = \sigma\left(x\right) = \sigma'\left(x\right) = \sigma'\left(f_2(\ell_1)\right)$.

3.2 Computing Runtime Bounds

We now show how to compute runtime bounds using MΦRFs. As in [16], for a sub-program $\mathcal{T}'$, the *entry transitions of a location* ℓ are all transitions outside $\mathcal{T}'$ which reach ℓ. The *entry locations of* $\mathcal{T}'$ are all locations where an evaluation of the sub-program $\mathcal{T}'$ can begin. Finally, the *entry transitions of* $\mathcal{T}'$ are all entry transitions to entry locations of $\mathcal{T}'$.

Definition 16 (Entry Transitions and Entry Locations). *Let $\varnothing \neq \mathcal{T}' \subseteq \mathcal{T}$. We define the set of* entry transitions of $\ell \in \mathcal{L}$ *as $\mathcal{T}_\ell = \{t \mid t = (\ell', \tau, \eta, \ell) \wedge t \in \mathcal{T} \setminus \mathcal{T}'\}$. The set of* entry locations *is $\mathcal{E}_{\mathcal{T}'} = \{\ell_{in} \mid \mathcal{T}_{\ell_{in}} \neq \varnothing \wedge \exists \ell' : (\ell_{in}, \tau, \eta, \ell') \in \mathcal{T}'\}$. Finally, the* entry transitions of $\mathcal{T}'$ *are $\mathcal{ET}_{\mathcal{T}'} = \bigcup_{\ell \in \mathcal{E}_{\mathcal{T}'}} \mathcal{T}_\ell$.*

Example 17. Again, consider the integer program in Fig. 4 and $\mathcal{T}' = \{t_2, t_3\}$. Then we have $\mathcal{T}_{\ell_2} = \{t_1\}$, $\mathcal{E}_{\mathcal{T}'} = \{\ell_2\}$, and $\mathcal{ET}_{\mathcal{T}'} = \{t_1\}$.

In [12, Lemma 6], the authors considered programs consisting of a single looping transition and showed that an MΦRF for the loop yields a linear bound on the possible number of its executions. We now generalize their lemma to our modular setting where we regard sub-programs $\mathcal{T}'$ instead of the full program $\mathcal{T}$.[1] The sub-program $\mathcal{T}'$ may contain arbitrary many transitions and loops.

[1] So in the special case where $\mathcal{T}'_> = \mathcal{T}'$ and $\mathcal{T}'$ is a singleton, our Lemma 18 corresponds to [12, Lemma 6] for nested MΦRFs.

For a start configuration (ℓ, σ) where ℓ is an entry location of $\mathcal{T}'$ and an MΦRF $f = (f_1, \ldots, f_d)$ for $\mathcal{T}'_>$ and $\mathcal{T}'$, Lemma 18 gives a bound $\beta \in \mathbb{N}$ which ensures that whenever there is an evaluation of $\mathcal{T}'$ that begins with (ℓ, σ) and where transitions from $\mathcal{T}'_>$ are applied at least β times, then all ranking functions in f have become negative. As f is an MΦRF (and thus, in every application of a transition from $\mathcal{T}'_>$, some f_i must be decreasing and non-negative), this implies that in any evaluation of $\mathcal{T}'$ starting in (ℓ, σ), transitions from $\mathcal{T}'_>$ can be applied *at most* β times. Since the bound β depends *linearly* on the values $\sigma\left(f_1(\ell)\right), \ldots, \sigma\left(f_d(\ell)\right)$ of the ranking functions in the start configuration (ℓ, σ) and since all ranking functions f_i are linear as well, this means that we have inferred a linear bound on the number of applications of transitions from $\mathcal{T}'_>$. However, this is only a *local* bound w.r.t. the values of the variables at the start of the sub-program $\mathcal{T}'$. We lift these local bounds to global runtime bounds for the full program in Thm. 20. See App. A for the proofs of both Lemma 18 and Thm. 20.

Lemma 18 (Local Runtime Bound for Sub-Program). *Let $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$, $\ell \in \mathcal{E}_{\mathcal{T}'}$, $\sigma \in \Sigma$, and let $f = (f_1, \ldots, f_d)$ be an MΦRF for $\mathcal{T}'_>$ and $\mathcal{T}'$. For all $1 \leq i \leq d$, we define the constants $\gamma_i \in \mathbb{Q}$ and $\beta \in \mathbb{N}$ with $\gamma_i, \beta > 0$:*

- *$\gamma_1 = 1$ and $\gamma_i = 2 + \frac{\gamma_{i-1}}{i-1} + \frac{1}{(i-1)!}$ for $i > 1$*
- *$\beta = 1 + d! \cdot \gamma_d \ \cdot\ \max\{0, \sigma(f_1(\ell)), \ldots, \sigma(f_d(\ell))\}$*

Then for any evaluation $(\ell, \sigma) \left(\to^*_{\mathcal{T}' \setminus \mathcal{T}'_>} \circ \to_{\mathcal{T}'_>}\right)^n (\ell', \sigma')$ with $n \geq \beta$ and any $1 \leq i \leq d$, we have $\sigma'(f_i(\ell')) < 0$.

Note that the constants γ_i do not depend on the program or the MΦRF, and the factor $d! \cdot \gamma_d$ only depends on the depth d.

Example 19. Reconsider the MΦRF $= (f_1, f_2)$ that we found for $\mathcal{T}'_> = \{t_2\}$ and $\mathcal{T}' = \{t_2, t_3\}$ in Ex. 15. The constants of Lemma 18 are $\gamma_1 = 1$ and $\gamma_2 = 2 + \frac{1}{1} + \frac{1}{1} = 4$. Thus, when $\mathcal{T}'$ is interpreted as a standalone program, then transition t_2 can be executed at most $\beta = 1 + 2! \cdot \gamma_2 \cdot \max\{0, \sigma\left(f_1(\ell_2)\right), \sigma\left(f_2(\ell_2)\right)\} = 1 + 8 \cdot \max\left\{0, \sigma\left(y+1\right), \sigma\left(x\right)\right\}$ many times when starting in $\sigma \in \Sigma$.

Lemma 18 yields the runtime bound
$$1 + d! \cdot \gamma_d \ \cdot\ \max\{0, f_1(\ell), \ldots, f_d(\ell)\} \tag{1}$$
for the transitions $\mathcal{T}'_>$ in the standalone program consisting of the transitions $\mathcal{T}'$. However, (1) is not yet a bound from $\mathcal{B}$, because it contains "max" and because the polynomials $f_i(\ell)$ may have negative coefficients. To transform polynomials into (weakly monotonically increasing) bounds, we replace their coefficients by their absolute values (and denote this transformation by $\lceil \cdot \rceil$). So for example we have $\lceil -x + 2 \rceil = |-1| \cdot x + |2| = x + 2$. Moreover, to remove "max", we replace it by addition. In this way, we obtain the bound
$$\beta_\ell \ = \ 1 + d! \cdot \gamma_d \ \cdot\ (\lceil f_1(\ell) \rceil + \ldots + \lceil f_d(\ell) \rceil).$$

In an evaluation of the full program, we enter a sub-program $\mathcal{T}'$ by an entry transition $t \in \mathcal{T}_\ell$ to an entry location $\ell \in \mathcal{E}_{\mathcal{T}'}$. As explained in Sect. 2, to lift the local runtime bound β_ℓ for $\mathcal{T}'_>$ to a global bound, we have to instantiate the variables in β_ℓ by (over-approximations of) the values that the variables have when reaching the sub-program $\mathcal{T}'$, i.e., after the transition t. To this end, we use the size bound $\mathcal{SB}(t, v)$ which over-approximates the largest absolute value of v after the transition t. We also use the shorthand notation $\mathcal{SB}(t, \cdot) : \mathcal{PV} \to \mathcal{B}$, where $\mathcal{SB}(t, \cdot)(v)$ is defined to be $\mathcal{SB}(t, v)$ and for every arithmetic expression b, $\mathcal{SB}(t, \cdot)(b)$ results from b by replacing each variable v in b by $\mathcal{SB}(t, v)$. Hence, $\mathcal{SB}(t, \cdot)(\beta_\ell)$ is a (global) bound on the number of applications of transitions from $\mathcal{T}'_>$ if $\mathcal{T}'$ is entered once via the entry transition t. Here, weak monotonic increase of β_ℓ ensures that the over-approximation of the variables v in β_ℓ by $\mathcal{SB}(t, v)$ indeed leads to an over-approximation of $\mathcal{T}'_>$'s runtime.

However, for every entry transition t we also have to take into account how often the sub-program $\mathcal{T}'$ may be entered via t. We can over-approximate this value by $\mathcal{RB}(t)$. This leads to Thm. 20 which generalizes a result from [16] to MΦRFs. The analysis starts with a runtime bound $\mathcal{RB}$ and a size bound $\mathcal{SB}$ which map all transitions resp. result variables to ω, except for the transitions t which do not occur in cycles of $\mathcal{T}$, where $\mathcal{RB}(t) = 1$. Afterwards, $\mathcal{RB}$ and $\mathcal{SB}$ are refined repeatedly. Instead of using a single ranking function for the refinement of $\mathcal{RB}$ as in [16], Thm. 20 now allows us to replace $\mathcal{RB}$ by a refined bound $\mathcal{RB}'$ based on an MΦRF.

Theorem 20 (Refining Runtime Bounds Based on MΦRFs). *Let $\mathcal{RB}$ be a runtime bound, $\mathcal{SB}$ a size bound, and $\varnothing \neq \mathcal{T}'_> \subseteq \mathcal{T}' \subseteq \mathcal{T}$ such that $\mathcal{T}'$ does not contain any initial transitions. Let $f = (f_1, \ldots, f_d)$ be an MΦRF for $\mathcal{T}'_>$ and $\mathcal{T}'$. For any entry location $\ell \in \mathcal{E}_{\mathcal{T}'}$ we define $\beta_\ell = 1 + d! \cdot \gamma_d \cdot (\lceil f_1(\ell) \rceil + \ldots + \lceil f_d(\ell) \rceil)$, where γ_d is as in Lemma 18. Then $\mathcal{RB}'$ is also a runtime bound, where we define $\mathcal{RB}'$ by $\mathcal{RB}'(t) = \mathcal{RB}(t)$ for all $t \notin \mathcal{T}'_>$ and*

$$\mathcal{RB}'(t_>) = \sum\nolimits_{\ell \in \mathcal{E}_{\mathcal{T}'}} \sum\nolimits_{t \in \mathcal{T}_\ell} \mathcal{RB}(t) \cdot \mathcal{SB}(t, \cdot)(\beta_\ell) \quad \text{for all } t_> \in \mathcal{T}'_>.$$

Example 21. We use Thm. 20 to compute a runtime bound for t_2 in Fig. 4. In Ex. 17, we showed that $\mathcal{E}_{\{t_2, t_3\}} = \{\ell_2\}$ and $\mathcal{T}_{\ell_2} = \{t_1\}$. Thus, we obtain

$$\mathcal{RB}(t_2) = \mathcal{RB}(t_1) \cdot \mathcal{SB}(t_1, \cdot)(\beta_{\ell_2}).$$

Using our calculations from Ex. 19 we have $\beta_{\ell_2} = 1 + 2! \cdot \gamma_2 \cdot (\lceil f_1(\ell_2) \rceil + \lceil f_2(\ell_2) \rceil) = 1 + 8 \cdot (y + 1 + x) = 8 \cdot x + 8 \cdot y + 9$.

We use the runtime bound $\mathcal{RB}(t_1) = z$ and the size bounds $\mathcal{SB}(t_1, x) = \mathcal{SB}(t_1, y) = 2 \cdot z$ from Ex. 10 and 13 and get $\mathcal{RB}(t_2) = \mathcal{RB}(t_1) \cdot (8 \cdot \mathcal{SB}(t_1, x) + 8 \cdot \mathcal{SB}(t_1, y) + 9) = z \cdot (8 \cdot 2 \cdot z + 8 \cdot 2 \cdot z + 9) = 32 \cdot z^2 + 9 \cdot z$.

By Cor. 11 and Ex. 10, the runtime complexity of the program in Fig. 4 is at most $\sum_{j=0}^{3} \mathcal{RB}(t_j) = 1 + z + 32 \cdot z^2 + 9 \cdot z + z = 32 \cdot z^2 + 11 \cdot z + 1$, resp. $\operatorname{rc}(\sigma_0) \leq 32 \cdot |\sigma_0(z)|^2 + 11 \cdot |\sigma_0(z)| + 1$, i.e., the program's runtime complexity is at most quadratic in the initial absolute value of z. Thus, in contrast to [16], we can now infer a finite bound on the runtime complexity of this program.

3.3 Complete Algorithm

Based on Thm. 20, in Alg. 1 we present our complete algorithm which improves the approach for complexity analysis of integer programs from [16] by using MΦRFs to infer runtime bounds. As mentioned in Sect. 2, the computation of

```
    Input: An integer program $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$
1   Preprocess $\mathcal{P}$
2   Create an initial runtime bound $\mathcal{RB}$ and an initial size bound $\mathcal{SB}$ and set $d \leftarrow 1$
3   forall SCCs $\widetilde{\mathcal{T}}$ without initial transitions of $\mathcal{P}$ in topological order do
4       repeat
5           forall $t_> \in \widetilde{\mathcal{T}}$ with $\mathcal{RB}(t_>) = \omega$ do
6               repeat
7                   Search for an M$\Phi$RF with depth $d$ for a maximal subset $\mathcal{T}' \subseteq \widetilde{\mathcal{T}}$
                     that has a subset $\mathcal{T}'_> \subseteq \mathcal{T}'$ with $t_> \in \mathcal{T}'_>$
                     such that all transitions in $\mathcal{T}'_>$ are decreasing
                     and all transitions in $\mathcal{T}' \setminus \mathcal{T}'_>$ are non-increasing
8                   $d \leftarrow d + 1$
9               until M$\Phi$RF was found or $d > mdepth$
10              if M$\Phi$RF was found then
11                  Update $\mathcal{RB}(t)$ for all $t \in \mathcal{T}'_>$ using Thm. 20
12          Update all size bounds for transitions in $\widetilde{\mathcal{T}}$ and reset $d \leftarrow 1$
13      until No runtime or size bound improved
14      Update all size bounds for outgoing transitions of $\widetilde{\mathcal{T}}$.
    Output: Runtime Bound $\mathcal{RB}$ and Size Bound $\mathcal{SB}$
```

Algorithm 1: Inferring Global Runtime and Size Bounds

size bounds from [16] is used as a black box. We just take the alternating repeated improvement of runtime and size bounds into account. So in particular, size bounds are updated when runtime bounds have been updated (Lines 12 and 14).

First, we preprocess the program (Line 1) by eliminating unreachable locations and transitions with unsatisfiable guards, and infer program invariants using the Apron library [34]. In addition, we remove variables which clearly have no impact on the termination behavior of the program. Then, we set all runtime bounds for transitions outside of cycles to 1, and all other bounds to ω initially (Line 2).

For the computation of an MΦRF, the considered subset $\mathcal{T}'$ has to be chosen heuristically. We begin with regarding a strongly connected component[2] (SCC) $\widetilde{\mathcal{T}}$ of the program graph in Line 3. Then we try to generate an MΦRF, and choose $\mathcal{T}'$ to consist of a maximal subset of $\widetilde{\mathcal{T}}$ where all transitions are non-increasing or decreasing (and at least one of the unbounded transitions $t_>$ is decreasing). So for the program in Fig. 4, we would start with $\widetilde{\mathcal{T}} = \{t_1, t_2, t_3\}$, but when trying to generate an MΦRF for $\mathcal{T}'_> = \{t_2\}$, we can only make t_2 decreasing and t_3 non-increasing. For that reason, we then set $\mathcal{T}'$ to $\{t_2, t_3\}$.

We treat the SCCs in topological order such that improved bounds for previous transitions are already available when handling the next SCC. If an MΦRF was found, we update the runtime bound for all $t \in \mathcal{T}'_>$ using Thm. 20 (Line 11). If we do not find any MΦRF of the given depth that makes $t_>$ decreasing, we increase the depth and continue the search until we reach a fixed *mdepth*. We abort the computation of runtime bounds if no bound has been improved. Here,

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

we use a heuristic which compares polynomial bounds by their degrees.

Finally, let us elaborate on the choice of *mdepth*. For example, if *mdepth* is 1, then we just compute linear ranking functions. If *mdepth* is infinity, then we cannot guarantee that our algorithm always terminates. For certain classes of programs, it is possible to give a bound on *mdepth* such that if there is an MΦRF for the program, then there is also one of depth *mdepth* [11, 12, 50]. However, it is open whether such a bound is computable for general integer programs. As the amount of time the SMT solver needs to find an MΦRF increases with the depth, we decided to use 5 as a fixed maximal depth, which performed well in our examples. Still, we provide the option for the user to change this bound.

4 Improving Bounds by Control-Flow Refinement

Now we present another technique to improve the automated complexity analysis of integer programs, so-called *control-flow refinement*. The idea is to transform a program $\mathcal{P}$ into a new program $\mathcal{P}'$ which is "easier" to analyze. Of course, we ensure that the runtime complexity of $\mathcal{P}'$ is *at least* the runtime complexity of $\mathcal{P}$. Then it is sound to analyze upper runtime bounds for $\mathcal{P}'$ instead of $\mathcal{P}$.

Our approach is based on the *partial evaluation* technique of [20]. For termination analysis, [20] shows how to use partial evaluation of *constrained Horn clauses* locally on every SCC of the program graph. But for complexity analysis, [20] only discusses *global* partial evaluation as a preprocessing step for complexity analysis. In Sect. 4.1, we formalize the partial evaluation technique of [20] such that it operates directly on SCCs of integer programs and prove that it is sound for complexity analysis. Afterwards, we improve its locality further in Sect. 4.2 such that partial evaluation is only applied *on-demand* on those transitions of an integer program where our current runtime bounds are "not yet good enough". Our experimental evaluation in Sect. 5 shows that our local partial evaluation techniques of Sect. 4.1 and Sect. 4.2 lead to a significantly stronger tool than when performing partial evaluation only globally as a preprocessing step.

As indicated in Sect. 1, the loop in Fig. 2 can be transformed into two consecutive loops (see Fig. 3). The first loop in Fig. 3 covers the case $x < 0 \land y < z$ and the second one covers the case $x < 0 \land y \geq z$. These cases correspond to the conjunction of the loop guard with the conditions of the two branches of the **if**-instruction. Here, partial evaluation detects that these cases occur *after* each other, i.e., if $y \geq z$, then the case $y < z$ does not occur again afterwards. In Ex. 22, we illustrate how our algorithm for partial evaluation performs this transformation. In the refined program of Fig. 3, it is easy to see that the runtime complexity is at most linear. Thus, the original loop has at most linear runtime complexity as well. Note that our tool KoAT can also infer a linear bound for both programs corresponding to Fig. 2 and 3 *without* control-flow refinement. In fact, for these examples it suffices to use just linear ranking functions, i.e., Thm. 20 with MΦRFs of depth $d = 1$. Still, we illustrate partial evaluation using this small example to ease readability. We will discuss the relationship between MΦRFs and partial evaluation at the end of Sect. 4.2, where we also show examples to demonstrate that these techniques do not subsume each other (see Ex. 26 and 27).

```
    Input: A program $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ and a non-trivial SCC $\mathcal{T}_{SCC} \subseteq \mathcal{T}$
 1  $\mathcal{L}_1 \leftarrow \{\langle \ell', \mathtt{true} \rangle \mid \ell' \in \mathcal{E}_{\mathcal{T}_{SCC}}\}$
 2  $\mathcal{T}_{res} \leftarrow \{(\ell, \tau, \eta, \langle \ell', \mathtt{true} \rangle) \mid \ell' \in \mathcal{E}_{\mathcal{T}_{SCC}} \wedge (\ell, \tau, \eta, \ell') \in \mathcal{T} \setminus \mathcal{T}_{SCC}\}$
 3  $\mathcal{L}_0 \leftarrow \varnothing$, $\mathcal{L}_{done} \leftarrow \varnothing$
 4  repeat
 5      $\mathcal{L}_0 \leftarrow \mathcal{L}_1$, $\mathcal{L}_1 \leftarrow \varnothing$
 6      forall $\langle \ell, \varphi \rangle \in \mathcal{L}_0$ do
 7          forall $(\ell, \tau, \eta, \ell') \in \mathcal{T}_{SCC}$ do
 8              Compute $\varphi_{new}$ from $\varphi$, $\tau$, and $\eta$ such that $\models (\varphi \wedge \tau) \rightarrow \eta(\varphi_{new})$
 9              if $\langle \ell', \alpha_{\ell'}(\varphi_{new}) \rangle \notin \mathcal{L}_{done}$ then
10                  $\mathcal{L}_1 \leftarrow \mathcal{L}_1 \cup \{\langle \ell', \alpha_{\ell'}(\varphi_{new}) \rangle\}$
11              $\mathcal{T}_{res} \leftarrow \mathcal{T}_{res} \cup \{(\langle \ell, \varphi \rangle, \varphi \wedge \tau, \eta, \langle \ell', \alpha_{\ell'}(\varphi_{new}) \rangle)\}$
12          forall $(\ell, \tau, \eta, \ell') \in \mathcal{T} \setminus \mathcal{T}_{SCC}$ do
13              $\mathcal{T}_{res} \leftarrow \mathcal{T}_{res} \cup \{(\langle \ell, \varphi \rangle, \varphi \wedge \tau, \eta, \ell')\}$
14          $\mathcal{L}_{done} \leftarrow \mathcal{L}_{done} \cup \{\langle \ell, \varphi \rangle\}$
15  until $\mathcal{L}_1 = \varnothing$
    Output: $\mathcal{P}' = (\mathcal{PV}, (\mathcal{L} \setminus \{\ell \mid \ell$ occurs as source or target in $\mathcal{T}_{SCC}\}) \cup \mathcal{L}_{done}, \ell_0,$
            $(\mathcal{T} \setminus \{(\ell, \tau, \eta, \ell') \mid \ell$ or $\ell'$ occurs as source or target in $\mathcal{T}_{SCC}\}) \cup \mathcal{T}_{res})$
```

Algorithm 2: Partial Evaluation for an SCC

4.1 SCC-Based Partial Evaluation

We now formalize the partial evaluation of [20] as an SCC-based refinement technique for integer programs in Alg. 2 and show its correctness for complexity analysis in Thm. 24. The intuitive idea of Alg. 2 is to refine a non-trivial[3] SCC $\mathcal{T}_{SCC}$ of an integer program into multiple SCCs by considering "abstract" evaluations which do not operate on concrete states but on sets of states. These sets of states are characterized by constraints, i.e., a constraint φ stands for all states σ with $\sigma(\varphi) = \mathtt{true}$. To this end, we label every location ℓ in the SCC by a constraint $\varphi \in \mathcal{C}(\mathcal{PV})$ which describes (a superset of) those states σ which can occur in this location. So all reachable configurations with the location ℓ have the form (ℓ, σ) such that $\sigma(\varphi) = \mathtt{true}$. We begin with labeling the entry locations of $\mathcal{T}_{SCC}$ by the constraint $\mathtt{true}$. The constraints for the other locations in the SCC are obtained by considering how the updates of the transitions affect the constraints of their source locations and their guards. The pairs of locations and constraints then become the new locations in the refined program.

Since locations can be reached by different paths, the same location may get different constraints, i.e., partial evaluation can transform a former location ℓ into several new locations $\langle \ell, \varphi_1 \rangle, \ldots, \langle \ell, \varphi_n \rangle$. So the constraints are not necessarily invariants that hold for *all* evaluations that reach a location ℓ but instead of "widening" (or "generalizing") constraints when a location can be reached by different states, we perform a case analysis and split up a location ℓ according to the different sets of states that may reach ℓ.

After labeling every entry location ℓ of $\mathcal{T}_{SCC}$ by the constraint $\mathtt{true}$ in Line 1 of Alg. 2, we modify the entry transitions to ℓ such that they now reach the new

[3] As usual, an SCC is *non-trivial* if it contains at least one transition.

location $\langle \ell, \texttt{true} \rangle$ instead (Line 2). The sets $\mathcal{L}_0$ and $\mathcal{L}_1$ (the new locations whose outgoing transitions need to be processed) and $\mathcal{L}_{done}$ (the new locations whose outgoing transitions were already processed) are used for bookkeeping. We then apply partial evaluation in Lines 4 to 15 until there are no new locations with transitions to be processed anymore (see Line 15).

In each iteration of the outer loop in Line 4, the transitions of the current new locations in $\mathcal{L}_1$ are processed. To this end, $\mathcal{L}_0$ is set to $\mathcal{L}_1$ and $\mathcal{L}_1$ is set to $\varnothing$ in Line 5. During the handling of the locations in $\mathcal{L}_0$, we might create new locations and these will be stored in $\mathcal{L}_1$ again.

We handle all locations $\langle \ell, \varphi \rangle$ in $\mathcal{L}_0$ (Line 6) by using all outgoing transitions $(\ell, \tau, \eta, \ell')$. We first consider those transitions which are part of the considered SCC (Line 7), whereas the transitions which leave the SCC are handled in Line 12.

The actual partial evaluation step is in Line 8. Given a new location $\langle \ell, \varphi \rangle$ and a transition $t = (\ell, \tau, \eta, \ell')$, we compute a constraint φ_{new} which over-approximates the set of states that can result from those states that satisfy the constraint φ and the guard τ of the transition when applying the update η. More precisely, φ_{new} has to satisfy $\models (\varphi \land \tau) \to \eta(\varphi_{new})$, i.e., $(\varphi \land \tau) \to \eta(\varphi_{new})$ is a tautology. For example, if $\varphi = (x = 0)$, $\tau = \texttt{true}$, and $\eta(x) = x - 1$, we derive $\varphi_{new} = (x = -1)$. However, if we now created the new location $\langle \ell', \varphi_{new} \rangle$, this might lead to non-termination of our algorithm. The reason is that if ℓ' is within a loop, then whenever one reaches ℓ' again, one might obtain a new constraint. In this way, one would create infinitely many new locations $\langle \ell', \varphi_1 \rangle, \langle \ell', \varphi_2 \rangle, \ldots$. For instance, if in our example the transition with the update $\eta(x) = x - 1$ is a self-loop, then we would derive further new locations with the constraints $x = -2$, $x = -3$, etc.

To ensure that every former location ℓ' only gives rise to finitely many new locations $\langle \ell', \varphi \rangle$, we perform *property-based abstraction* as in [20, 28]: For every location ℓ' we use a finite so-called abstraction layer $\alpha_{\ell'} \subseteq \{e_1 \leq e_2 \mid e_1, e_2 \in \mathbb{Z}[\mathcal{PV}]\}$. So $\alpha_{\ell'}$ is a finite set of atomic constraints (i.e., of polynomial inequations). Then $\alpha_{\ell'}$ is extended to a function on constraints such that $\alpha_{\ell'}(\varphi_{new}) = \varphi'_{new}$ where φ'_{new} is a conjunction of inequations from $\alpha_{\ell'}$ and $\models \varphi_{new} \to \varphi'_{new}$. This guarantees that partial evaluation terminates, but it can lead to an exponential blow-up, since for every location ℓ' there can now be $2^{|\alpha_{\ell'}|}$ many possible constraints. In our example, instead of the infinitely many inequations $x = 0, x = -1, x = -2, \ldots$ the abstraction layer might just contain the inequation $x \leq 0$. Then we would only obtain the new location with the constraint $x \leq 0$.

Afterwards, in Lines 9 and 10 we add the new location $\langle \ell', \alpha_{\ell'}(\varphi_{new}) \rangle$ to $\mathcal{L}_1$ if it was not processed before. Moreover, the transition $(\ell, \tau, \eta, \ell')$ which we used for the refinement must now get the new location as its target (Line 11) and $\langle \ell, \varphi \rangle$ as its source. In addition, we extend the transition's guard τ by φ.

Finally, we also have to process the transitions $(\ell, \tau, \eta, \ell')$ which leave the SCC. Thus, we replace the source transition ℓ by $\langle \ell, \varphi \rangle$ and again extend the guard τ of the transition by the constraint φ in Lines 12 and 13. Since we have now processed all outgoing transitions of $\langle \ell, \varphi \rangle$ we can add it to $\mathcal{L}_{done}$ in Line 14.

In the end, we output the program where the considered SCC and all transitions in or out of this SCC were refined (and thus, have to be removed from the original program). We now illustrate Alg. 2 using the program from Fig. 2.

Example 22. Fig. 5 represents the program from Fig. 2 in our formalism for integer programs. Here, we used an explicit location ℓ_3 for the end of the program to illustrate how Alg. 2 handles transitions which leave the SCC.

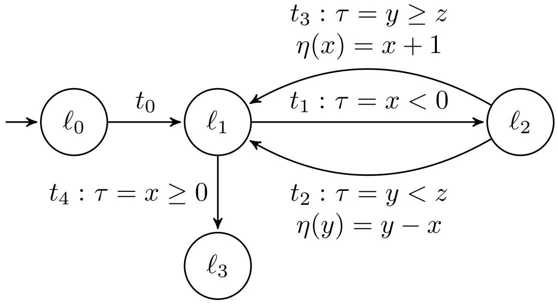


Fig. 5: Integer Program Corresponding to Fig. 2

We apply Alg. 2 to the program in Fig. 5 and refine the SCC $\mathcal{T}_{SCC} = \{t_1, t_2, t_3\}$. The entry location is $\mathcal{E}_{\mathcal{T}_{SCC}} = \{\ell_1\}$. To increase readability, let τ_i be the guard and η_i be the update of transition t_i for all $0 \leq i \leq 3$.

For the abstraction layers, we choose[4] $\alpha_{\ell_1} = \alpha_{\ell_2} = \{x < 0, y \geq z\}$. It is not necessary to define abstraction layers for ℓ_0 and ℓ_3, as they are not part of the SCC. So for any constraint φ_{new} and $i \in \{1,2\}$, $\alpha_{\ell_i}(\varphi_{new})$ can only be a conjunction of the inequations in α_{ℓ_i} (i.e., $\alpha_{\ell_i}(\varphi_{new})$ is $\texttt{true}$, $x < 0$, $y \geq z$, or $x < 0 \land y \geq z$, where $\texttt{true}$ corresponds to the empty conjunction).

Since t_0 is the only entry transition to the entry location ℓ_1, we initialize $\mathcal{T}_{res}$ with $\{(\ell_0, \tau_0, \eta_0, \langle \ell_1, \texttt{true} \rangle)\}$ and $\mathcal{L}_1$ with $\{\langle \ell_1, \texttt{true} \rangle\}$.

In the first iteration $\mathcal{L}_0$ only consists of $\langle \ell_1, \texttt{true} \rangle$. We have two possible transitions which we can apply in ℓ_1: $t_1 = (\ell_1, \tau_1, \eta_1, \ell_2) \in \mathcal{T}_{SCC}$ or $t_4 = (\ell_1, \tau_4, \eta_4, \ell_3) \in \mathcal{T} \setminus \mathcal{T}_{SCC}$. We start with transition t_1. Since the update η_1 is the identity, from the guard $\tau_1 = (x < 0)$ we obtain the resulting constraint $\varphi_{new} = (x < 0)$. We apply the abstraction layer and get $\alpha_{\ell_2}(x < 0) = (x < 0)$ because $\models (x < 0) \to (x < 0)$. Now we add the new transition

$$(\langle \ell_1, \texttt{true} \rangle, \texttt{true} \land \tau_1, \eta_1, \langle \ell_2, x < 0 \rangle)$$

to $\mathcal{T}_{res}$ and $\langle \ell_2, x < 0 \rangle$ to $\mathcal{L}_1$. For transition $t_4 = (\ell_1, \tau_4, \eta_4, \ell_3)$, we update its source location and get the resulting transition

$$(\langle \ell_1, \texttt{true} \rangle, \texttt{true} \land \tau_4, \eta_4, \ell_3)$$

in $\mathcal{T}_{res}$. We add $\langle \ell_1, \texttt{true} \rangle$ to $\mathcal{L}_{done}$. Now $\mathcal{L}_1$ consists of $\langle \ell_2, x < 0 \rangle$. There are two transitions t_2 and t_3 which can be applied in ℓ_2. For t_2, from the previous constraint $x < 0$ and the guard $\tau_2 = (y < z)$ we can infer that after the update $\eta_2(y) = y - x$ we have $x < 0 \land y < z - x$. As the abstraction layer α_{ℓ_1} consists of $x < 0$ and $y \geq z$, we have $\alpha_{\ell_1}(x < 0 \land y < z - x) = x < 0$, since $\not\models (x < 0 \land y < z - x) \to (y \geq z)$. Thus, we add the new transition

$$(\langle \ell_2, x < 0 \rangle, x < 0 \land \tau_2, \eta_2, \langle \ell_1, x < 0 \rangle)$$

to $\mathcal{T}_{res}$ and $\langle \ell_1, x < 0 \rangle$ to the set $\mathcal{L}_1$. Similarly, for t_3, from $x < 0$ and the guard $\tau_3 = (y \geq z)$ we infer that after $\eta_3(x) = x + 1$ we have $x < 1 \land y \geq z$. Here, $\alpha_{\ell_1}(x < 1 \land y \geq z) = y \geq z$, since $\not\models (x < 1 \land y \geq z) \to (x < 0)$. Hence, we add

$$(\langle \ell_2, x < 0 \rangle, x < 0 \land \tau_3, \eta_3, \langle \ell_1, y \geq z \rangle)$$

[4] In [20], different heuristics are presented to choose such abstraction layers. In our implementation, we use these heuristics as a black box.

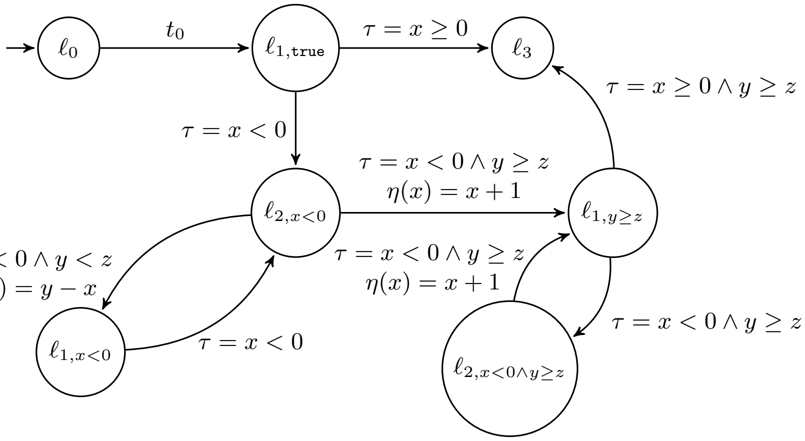


Fig. 6: Applying Partial Evaluation to Fig. 5

to $\mathcal{T}_{res}$ and $\langle \ell_1, y \geq z\rangle$ to $\mathcal{L}_1$. So $\mathcal{L}_1$ now consists of $\langle \ell_1, x < 0\rangle$ and $\langle \ell_1, y \geq z\rangle$. For $\langle \ell_1, x < 0\rangle$, in the same way as before we obtain the following new transitions:

$$(\langle \ell_1, x < 0\rangle, x < 0 \land \tau_1, \eta_1, \langle \ell_2, x < 0\rangle)$$
$$(\langle \ell_1, x < 0\rangle, x < 0 \land \tau_4, \eta_4, \ell_3)$$

Note that the guard $x < 0 \land \tau_4$ of the last transition is unsatisfiable. For that reason, we always remove transitions with unsatisfiable guard after partial evaluation was applied. For $\langle \ell_1, y \geq z\rangle$, we obtain the following new transitions:

$$(\langle \ell_1, y \geq z\rangle, y \geq z \land \tau_1, \eta_1, \langle \ell_2, x < 0 \land y \geq z\rangle)$$
$$(\langle \ell_1, y \geq z\rangle, y \geq z \land \tau_4, \eta_4, \ell_3)$$

Thus, $\mathcal{L}_1$ now consists of the new location $\langle \ell_2, x < 0 \land y \geq z\rangle$. For this location, we finally get the following new transitions:

$$(\langle \ell_2, x < 0 \land y \geq z\rangle, x < 0 \land y \geq z \land \tau_2, \eta_2, \langle \ell_1, x < 0 \land y \geq z\rangle)$$
$$(\langle \ell_2, x < 0 \land y \geq z\rangle, x < 0 \land y \geq z \land \tau_3, \eta_3, \langle \ell_1, y \geq z\rangle)$$

Since the guard $x < 0 \land y \geq z \land \tau_2$ of the penultimate transition is again unsatisfiable, it will be removed. For that reason, then the location $\langle \ell_1, x < 0 \land y \geq z\rangle$ will be unreachable and will also be removed.

Fig. 6 shows the refined integer program where we wrote $\ell_{i,\varphi}$ instead of $\langle \ell_i, \varphi\rangle$ for readability. Moreover, transitions with unsatisfiable guard or unreachable locations were removed. The first SCC with the locations $\langle \ell_2, x < 0\rangle$ and $\langle \ell_1, x < 0\rangle$ is applied *before* the second SCC with the locations $\langle \ell_1, y \geq z\rangle$ and $\langle \ell_2, x < 0 \land y \geq z\rangle$. So we have detected that these two SCCs occur *after* each other. Indeed, the integer program in Fig. 6 corresponds to the one in Fig. 3.

Alg. 2 is sound because partial evaluation transforms a program $\mathcal{P}$ into an *equivalent* program $\mathcal{P}'$. Therefore, it does not change the runtime.

Definition 23 (Equivalence of Programs). *Let $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ and $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}', \ell_0, \mathcal{T}')$ be integer programs over $\mathcal{V}$. $\mathcal{P}$ and $\mathcal{P}'$ are* equivalent *iff the*

```
Input: A program $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ and a non-empty subset $\mathcal{T}_{cfr}$ of a
       non-trivial SCC from $\mathcal{T}$.
$\mathcal{S} \leftarrow \varnothing$
forall $t = (\ell, \tau, \eta, \ell') \in \mathcal{T}_{cfr}$ do
    $\mathcal{T}_t \leftarrow$ a shortest path from $\ell'$ to $\ell$
    $\mathcal{T}_t \leftarrow \mathcal{T}_t \cup \{t\}$
    $\mathcal{T}_t \leftarrow \mathcal{T}_t \cup \{(\hat{\ell}, \_, \_, \hat{\ell}') \in \mathcal{T} \mid (\hat{\ell}, \_, \_, \hat{\ell}') \in \mathcal{T}_t\}$
    forall entry transitions $(\overline{\ell}, \overline{\tau}, \overline{\eta}, \overline{\ell}') \in \mathcal{ET}_{\mathcal{T}_t}$ do
        Add transition $(\ell_{new}, \overline{\tau}, \overline{\eta}, \overline{\ell}')$ to $\mathcal{T}_t$.
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{(\mathcal{PV}, \mathcal{L}, \ell_{new}, \mathcal{T}_t)\}$
repeat
    if there exist $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}, \ell_{new}, \mathcal{T}')$ and $\mathcal{P}'' = (\mathcal{PV}, \mathcal{L}, \ell_{new}, \mathcal{T}'')$ with
      $\mathcal{P}', \mathcal{P}'' \in \mathcal{S}$, $\mathcal{P}' \neq \mathcal{P}''$, and a location $\ell \neq \ell_{new}$ occurs in both $\mathcal{T}'$ and $\mathcal{T}''$
      then
        $\mathcal{S} \leftarrow (\mathcal{S} \setminus \{\mathcal{P}', \mathcal{P}''\}) \cup \{(\mathcal{PV}, \mathcal{L}, \ell_{new}, \mathcal{T}' \cup \mathcal{T}'')\}$
until $\mathcal{S}$ does not change anymore
forall $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}, \ell_{new}, \mathcal{T}') \in \mathcal{S}$ do
    $\mathcal{P}'' = (\mathcal{PV}, \mathcal{L}'', \ell_{new}, \mathcal{T}'') \leftarrow$ apply Alg. 2 to $\mathcal{P}'$ and the single non-trivial
      SCC $\mathcal{T}_{SCC}$ in $\mathcal{T}'$
    Extend the transitions $\mathcal{T}$ of $\mathcal{P}$ by the transitions $\mathcal{T}''$.
    forall entry transitions $t = (\ell, \tau, \eta, \ell') \in \mathcal{ET}_{\mathcal{T}'}$ do
        Replace $t$ by $(\ell, \tau, \eta, \langle \ell', \mathtt{true} \rangle)$ in $\mathcal{P}$.
    forall outgoing transitions $t = (\ell, \tau, \eta, \ell') \in \mathcal{ET}_{\mathcal{T} \setminus \mathcal{T}'}$ do
        Replace $t$ by $(\langle \ell, \varphi \rangle, \tau, \eta, \ell')$ in $\mathcal{P}$ for all $\langle \ell, \varphi \rangle \in \mathcal{L}''$.
Remove unreachable locations and transitions, and transitions with
  unsatisfiable guard.
Output: Refined program $\mathcal{P}$
```

Algorithm 3: Partial Evaluation for a Subset of an SCC

following holds for all states $\sigma_0 \in \Sigma$: There is an evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}}^{k} (\ell, \sigma)$ for some $\sigma \in \Sigma$, some $k \in \mathbb{N}$, and some $\ell \in \mathcal{L}$ iff there is an evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}'}^{k} (\ell', \sigma)$ for the same $\sigma \in \Sigma$ and $k \in \mathbb{N}$, and some location ℓ'.

Theorem 24 (Soundness of Partial Evaluation in Alg. 2). *Let $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ be an integer program and let $\mathcal{T}_{SCC} \subseteq \mathcal{T}$ be a non-trivial SCC of the program graph. Let $\mathcal{P}'$ be the integer program resulting from applying Alg. 2 to $\mathcal{P}$ and $\mathcal{T}_{SCC}$. Then $\mathcal{P}$ and $\mathcal{P}'$ are equivalent.*

4.2 Sub-SCC-Based Partial Evaluation

As control-flow refinement may lead to an exponential blow-up of the program, we now present an algorithm where we heuristically minimize the strongly connected part of the program on which we apply partial evaluation (Alg. 3) and we discuss how to integrate it into our approach for complexity analysis. Our experiments in Sect. 5 show that such a sub-SCC-based partial evaluation leads to significantly shorter runtimes than the SCC-based partial evaluation of Alg. 2.

The idea of Alg. 3 is to find a minimal cycle of the program graph containing the transitions $\mathcal{T}_{cfr}$ whose runtime bound we aim to improve by partial evaluation.

On the one hand, in this way we minimize the input set $\mathcal{T}_{SCC}$ for the partial evaluation algorithm. On the other hand, we keep enough of the original program's control flow such that partial evaluation can produce useful results.

Our local control-flow refinement technique in Alg. 3 consists of three parts. In the first loop in Lines 2 to 8, we find a minimal cycle $\mathcal{T}_t$ for each transition t from $\mathcal{T}_{cfr}$. Afterwards, $\mathcal{T}_t$ is extended by all transitions which are parallel to some transition in $\mathcal{T}_t$ in Line 5. Otherwise, we would not be able to correctly insert the refined program afterwards. We add a fresh initial location ℓ_{new}, take all entry transitions to the previously computed cycle and extend $\mathcal{T}_t$ by new corresponding entry transitions which start in ℓ_{new} instead (Lines 6 and 7). We collect all these programs in a set $\mathcal{S}$, where the programs have ℓ_{new} as their initial location.

So for our example from Fig. 5 and $\mathcal{T}_{cfr} = \{t_3\}$, $\mathcal{S}$ only contains one program with locations $\ell_1, \ell_2, \ell_{new}$, transitions t_1, t_2, t_3, and a transition from ℓ_{new} to ℓ_1.

As the next step, in the second loop in Lines 9 to 12, we merge those programs which share a location other than ℓ_{new}. Again, this allows us to correctly insert the refined program afterwards (see the proof of Thm. 25).

The last loop in Lines 13 to 19 performs partial evaluation on each strongly connected part of the programs in $\mathcal{S}$, and inserts the refined programs into the original one by redirecting the entry and the outgoing transitions. Here, an outgoing transition is simply an entry transition of the complement.

At the end of Alg. 3, one should remove unreachable locations and transitions, as well as transitions with unsatisfiable guard. This is needed, because the refined transitions $\mathcal{T}'$ are simply added to the old transitions $\mathcal{T}$, and entry and outgoing transitions are redirected. So the previous transitions might become unreachable.

Instead of implementing Alg. 2 ourselves, our complexity analyzer KoAT calls the implementation of [20] in the tool iRankFinder [19] as a backend for partial evaluation.[5] So in particular, we rely on iRankFinder's heuristics to compute the abstraction layers $\alpha_{\ell'}$ and the new constraints φ_{new} resp. $\alpha_{\ell'}(\varphi_{new})$ in Alg. 2.

So in our example, partial evaluation on the program in $\mathcal{S}$ would result in a program like the one in Fig. 6, but instead of the transition from ℓ_0 to $\ell_{1,\texttt{true}}$ there would be a transition from ℓ_{new} to $\ell_{1,\texttt{true}}$. Moreover, the location ℓ_3 and the transitions to ℓ_3 would be missing. The redirection of the entry and the outgoing transitions would finally create the program from Fig. 6.

The advantage of our technique in contrast to the naïve approach (i.e., applying partial evaluation on the full program as a preprocessing step) and also to the SCC-based approach in Alg. 2, is that Alg. 3 allows us to apply partial evaluation "on-demand" just on those transitions where our bounds are still "improvable". Thus, to integrate partial evaluation into our overall approach, Alg. 1 is modified such that after the treatment of an SCC $\widetilde{\mathcal{T}}$ in Lines 5 to 12, we let $\mathcal{T}_{cfr}$ consists of all transitions $t \in \widetilde{\mathcal{T}}$ where $\mathcal{RB}(t)$ is not linear (and not constant). So this is our heuristic to detect transitions with "improvable" bounds. If $\mathcal{T}_{cfr} \neq \varnothing$, then we call Alg. 3 to perform partial evaluation and afterwards we execute Lines 5

[5] To ensure the *equivalence* of the transformed program according to Def. 23, we call iRankFinder with a flag to prevent the "chaining" of transitions. This ensures that partial evaluation does not change the lengths of evaluations.

to 12 of Alg. 1 once more for the SCC that results from refining $\widetilde{\mathcal{T}}$.

Theorem 25 (Soundness of Partial Evaluation in Alg. 3). *Let $\mathcal{P} = (\mathcal{PV}, \mathcal{L}, \ell_0, \mathcal{T})$ be a program and $\mathcal{T}_{cfr} \subseteq \mathcal{T}$ a non-empty set of transitions from some non-trivial SCC. Then $\mathcal{P}$ and the program computed by Alg. 3 are equivalent.*

Both MΦRFs and control-flow refinement detect "phases" of the program. An MΦRF represents these phases via different ranking functions, whereas control-flow refinement makes these phases explicit by modifying the program, e.g., by splitting an SCC into several new ones as in Ex. 22. Ex. 26 and 27 show that there are programs where one of the techniques allows us to infer a finite bound on the runtime complexity while the other one does not. This is also demonstrated by our experiments with different configurations of KoAT in Sect. 5.

Example 26. For the program corresponding to the loop in Fig. 1 we can only infer a finite runtime bound if we search for MΦRFs of *at least* depth 2. In contrast, control-flow refinement via partial evaluation does not help here, because it does not change the loop. The used MΦRF (f_1, f_2) with $f_1(\ell_1) = f_1(\ell_2) = y + 1$ and $f_2(\ell_1) = f_2(\ell_2) = x$ (see Ex. 15) corresponds *implicitly* to the case analysis $y \geq 0$ resp. $y < 0$. However, this case analysis is not detected by Alg. 2, because $y < 0$ only holds after $|y_0| + 1$ executions of this loop if we have $y = y_0$ initially. Thus, this cannot be inferred when evaluating the loop partially for a finite number of times (as this number depends on the initial values of the variables). As Fig. 1 does not admit a linear ranking function, this means that we fail to infer a finite runtime bound if we only use linear ranking functions and control-flow refinement. The same argument explains why we cannot infer a finite runtime bound for our running example in Fig. 4 (which contains the loop in Fig. 1) with only linear ranking functions and control-flow refinement. For this example, we again need MΦRFs of at least depth 2 (see Ex. 21). So control-flow refinement via partial evaluation does not subsume MΦRFs.

Example 27. Now we show an example where MΦRFs are not strong enough to infer a finite runtime bound, whereas this is possible using just linear ranking functions (i.e., MΦRFs of depth 1) if we apply partial evaluation before. Moreover, it illustrates Alg. 3 which only performs partial evaluation on a subset of an SCC.

Consider the program in Fig. 7 where $\mathcal{PV} = \{x, y\}$ are the program variables and $\mathcal{TV} = \{u, w\}$ are the temporary variables. It has two independent components (the self-loop t_1 at location ℓ_1 and the cycle of t_2 and t_3 between ℓ_1 and ℓ_2) which do not influence each other, since t_1 operates only on the variable x and the cycle of t_2 and t_3 depends only on y. The choice which component is evaluated is non-deterministic since it depends on the value of the temporary variable w. Since the value of x is between 1 and 3 in the self-loop, t_1 is only evaluated at most 3 times. Similarly, t_2 and t_3 are each executed at most y times. Hence, the runtime complexity of the program is at most $1 + 3 + 2 \cdot y = 4 + 2 \cdot y$.

However, our approach does not find a finite runtime bound when using only MΦRFs *without* control-flow refinement. To make the transition t_1 in the self-loop decreasing, we need an MΦRF f where the variable x occurs in at least one function f_i of the MΦRF. So $f_i(\ell_1)$ contains x and thus, β_{ℓ_1} (as defined in

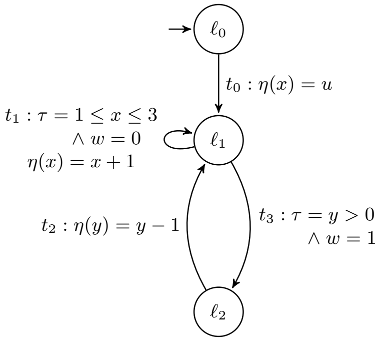


Fig. 7: Original Program

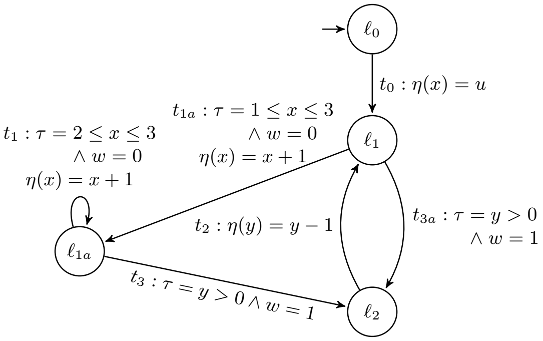


Fig. 8: Result of Alg. 3 with $\mathcal{T}_{cfr} = \{t_1\}$

Thm. 20) contains x as well. When constructing the global bound $\mathcal{RB}(t_1)$ by Thm. 20, we have to instantiate x in β_{ℓ_1} by $\mathcal{SB}(t_0, x)$, i.e., by the size-bound for x of the entry transition t_0. Since x is set to an arbitrary integer value u non-deterministically, its size is unbounded, i.e., $\mathcal{SB}(t_0, x) = \omega$. Thus, Thm. 20 yields $\mathcal{RB}(t_1) = \omega$. The alternative solution of turning t_0 into a non-initial transition and adding it to the subset $\mathcal{T}'$ in Thm. 20 does not work either. Since the value of x after t_0 is an arbitrary integer, t_0 violates the requirement of being non-increasing for every MΦRF where $f_i(\ell_1)$ contains x.

In this example, only the self-loop t_1 is problematic for the computation of runtime bounds. We can directly infer a linear runtime bound for all other transitions, using just linear ranking functions. Thus, when applying control-flow refinement via partial evaluation, according to our heuristic we call Alg. 3 on just $\mathcal{T}_{cfr} = \{t_1\}$. The result of Alg. 3 is presented in Fig. 8. Since partial evaluation is restricted to the problematic transition t_1, the other transitions t_2 and t_3 in the SCC remain unaffected, which avoids a too large increase of the program.

As before, in the program of Fig. 8 we infer linear runtime bounds for t_0, t_2, t_3, and t_{3a} using linear ranking functions. To obtain linear bounds for t_{1a} and t_1, we can now use the following MΦRF f of depth 1 for the subset $\mathcal{T}' = \{t_1, t_{1a}, t_3, t_{3a}\}$ and the decreasing transition $\mathcal{T}'_> = \{t_{1a}\}$ resp. $\mathcal{T}'_> = \{t_1\}$:

$$f(\ell_1) = 3 \qquad f(\ell_{1a}) = 3 - x \qquad f(\ell_2) = 0$$

Thus, while this example cannot be solved by MΦRFs, we can indeed infer linear runtime bounds when using control-flow refinement and just linear ranking functions. Hence, MΦRFs do not subsume control-flow refinement.

5 Evaluation

As mentioned, we implemented both Alg. 3 and the refined version of Alg. 1 which calls Alg. 3 in a new re-implementation of our tool KoAT which is written in OCaml. To find MΦRFs, it uses the SMT Solver Z3 [41] and it uses the tool iRankFinder [19] for the implementation of Alg. 2 to perform partial evaluation.

To distinguish our re-implementation of KoAT from the original version of the tool from [16], let KoAT1 refer to the tool from [16] and let KoAT2 refer to our new re-implementation. We now evaluate KoAT2 in comparison to the main other state-of-the-art tools for complexity analysis of integer programs: CoFloCo [22, 23], KoAT1 [16], Loopus [46], and MaxCore [6]. Moreover, we also evaluate the performance of KoAT1 and KoAT2 when control-flow refinement using iRankFinder [19] is performed on the complete program as a preprocessing step. We do not compare with RaML [33], as it does not support programs whose complexity depends on (possibly negative) integers (see [45]). We also do not compare with PUBS [2], because as stated in [20] by one of the authors of PUBS, CoFloCo is stronger than PUBS. Note that MaxCore is a tool chain which preprocesses the input program and then passes it to either CoFloCo or PUBS for the computation of the bound. As the authors' evaluation in [6] shows that MaxCore with CoFloCo as a backend is substantially stronger than with PUBS as a backend, we only consider the former configuration and refer to it as "MaxCore".

For our evaluation, we use the two sets for complexity analysis of integer programs from the *Termination Problems Data Base (TPDB)* [48] that are used in the annual *Termination and Complexity Competition (TermComp)* [31]: Complexity_ITS (CITS), consisting of integer transition systems, and Complexity_C_Integer (CINT), consisting of C programs with only integer variables. The integers in both CITS and CINT are interpreted as mathematical integers (i.e., without overflows).

Both Loopus and MaxCore only accept C programs as in CINT as input. While it is easily possible to transform the input format of CINT to the input format of CITS automatically, the other direction is not so straightforward. Hence, we compare with Loopus and MaxCore only on the benchmarks from the CINT collection. Our tool KoAT2 is evaluated in 7 different configurations to make the effects of both control-flow refinement and MΦRFs explicit:

1. KoAT2 denotes the configuration which uses Alg. 1 with maximal depth *mdepth* set to 1, i.e., we only compute linear ranking functions.
2. CFR + KoAT2 first preprocesses the complete program by performing control-flow refinement using iRankFinder. Afterwards, the refined program is analyzed with KoAT2 where $mdepth = 1$.
3. KoAT2 + CFRSCC is the configuration where control-flow refinement is applied to SCCs according to Alg. 2 and $mdepth = 1$.
4. KoAT2 + CFR uses Alg. 3 instead to apply control-flow refinement on sub-SCCs and has $mdepth = 1$.
5. KoAT2 + MΦRF5 applies Alg. 1 with maximal depth $mdepth = 5$, i.e., we use MΦRFs with up to 5 components, but no control-flow refinement.
6. KoAT2 + MΦRF5 + CFRSCC applies control-flow refinement to SCCs (Alg. 2) and uses $mdepth = 5$.
7. KoAT2 + MΦRF5 + CFR uses sub-SCC control-flow refinement (Alg. 3) and MΦRFs with maximal depth $mdepth = 5$.

Furthermore, we evaluate the tool KoAT1 in two configurations: KoAT1 corresponds to the standalone version, whereas for CFR + KoAT1, the complete program is first preprocessed using control-flow refinement via the tool iRankFinder before analyzing the resulting program with KoAT1. The second configuration was also used in the evaluation of iRankFinder in [20].

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(EXP)$	$< \infty$	AVG$^+$(s)	AVG(s)
KoAT2 + MΦRF5 + CFR	131	255	101	13	6	506	4.31	26.50
KoAT2+MΦRF5+CFRSCC	131	255	102	12	6	506	6.00	27.47
KoAT2 + CFR	131	245	101	10	6	493	5.00	21.81
KoAT2 + CFRSCC	131	245	101	9	6	492	6.37	23.45
KoAT2 + MΦRF5	126	235	100	13	6	480	2.19	13.32
KoAT1	132	214	104	14	5	469	0.65	9.38
CFR + KoAT1	128	231	93	10	5	467	5.54	40.81
CFR + KoAT2	130	231	93	6	6	466	10.44	43.05
CoFloCo	126	231	95	9	0	461	3.44	18.40
KoAT2	126	218	97	10	6	457	2.29	9.45

Fig. 9: Evaluation on Complexity_ITS

We compare the runtime bounds computed by the tools asymptotically as functions which depend on the largest initial absolute value n of all program variables. All tools were run inside an Ubuntu Docker container on a machine with an AMD Ryzen 7 3700X octa-core CPU and 32 GB of RAM. The benchmarks were evaluated in parallel such that at most 8 containers were running at once, each limited to 1.9 CPU cores. In particular, the runtimes of the tools include the times to start and remove the container. As in *TermComp*, we applied a timeout of 5 minutes for every program. See [42] for a binary and the source code of our tool KoAT2, a Docker image, web interfaces to test our implementation, and full details on all our experiments in the evaluation.

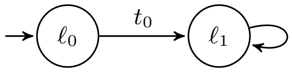
5.1 Evaluation on Complexity_ITS

The set CITS consists of 781 integer programs, where at most 564 of them *might* have finite runtime (since the tool LoAT [26, 27] proves unbounded runtime complexity for 217 examples). The results of our experiments on this set can be found in Fig. 9. So for example, there are $131 + 255 = 386$ programs where KoAT2 + MΦRF5 + CFR can show that $\mathrm{rc}(\sigma_0) \in \mathcal{O}(n)$ holds for all initial states σ_0 where $|\sigma_0(v)| \leq n$ for all $v \in \mathcal{PV}$. For 131 of these programs, KoAT2 + MΦRF5 + CFR can even show that $\mathrm{rc}(\sigma_0) \in \mathcal{O}(1)$, i.e., their runtime complexity is constant. In Fig. 9, "$< \infty$" is the number of examples where a finite bound on the runtime complexity could be computed by the respective tool within the time limit. "AVG$^+$(s)" is the average runtime of the tool on successful runs in seconds, i.e., where the tool proved a finite time bound before reaching the timeout, whereas "AVG(s)" is the average runtime of the tool on all runs including timeouts.

KoAT2 without MΦRFs and control-flow refinement infers a finite bound for 457 of the 781 examples, while CoFloCo solves 461 and KoAT1 solves 469 examples. In contrast to KoAT2, both CoFloCo and KoAT1 always apply some form of control-flow refinement. However, KoAT1's control-flow refinement is weaker than the one in Sect. 4, since it only performs loop unrolling via "chaining" to combine subsequent transitions. Indeed, when adding control-flow refinement as a preprocessing technique (in CFR + KoAT1 and CFR + KoAT2), the tools are almost equally powerful.

```
while $x > 0$ do
    $x \leftarrow x + y$
    $y \leftarrow y + z$
    $z \leftarrow z - 1$
```

Fig. 10: Loop With Three Phases



$$\begin{aligned} t_1 : \tau &= x > 0 \\ \eta(x) &= x + y \\ \eta(y) &= y + z \\ \eta(z) &= z - 1 \end{aligned}$$

Fig. 11: Integer Program

However, for efficiency it is much better to integrate control-flow refinement into KoAT2 as in Alg. 2 or Alg. 3 (KoAT2 + CFRSCC resp. KoAT2 + CFR) than to use it as a preprocessing step (CFR + KoAT2). This integration reduces the number of timeouts and therefore increases power. The corresponding configurations already make KoAT2 stronger than all previous tools on this benchmark. Nevertheless, while control-flow improves power substantially, it increases the resulting runtimes. The reason is that partial evaluation can lead to an exponential blow-up of the program. Moreover, we have to analyze parts of the program twice: we first analyze parts where we do not find a linear or a constant bound. Then, we apply control-flow refinement and afterwards, we analyze them again.

If instead of using control-flow refinement, the maximum depth of MΦRFs is increased from 1 to 5, KoAT2 can compute a finite runtime bound for 480 examples. As explained in Sect. 3, MΦRFs are a proper extension of classical linear ranking functions as used in KoAT1, for example. Thus, CoFloCo, KoAT1, and KoAT2 + CFR fail to compute a finite bound on the runtime complexity of our running example in Fig. 4, while KoAT2 + MΦRF5 succeeds on this example. In particular, this shows that KoAT2 + CFR does *not* subsume KoAT2 + MΦRF5 but the two techniques presented in Sect. 3 and 4 can have orthogonal effects and combining them leads to an even more powerful tool. Indeed, KoAT2 + MΦRF5 + CFR proves a finite bound for more examples than KoAT2 + MΦRF5 and KoAT2 + CFR, in total 506. The configuration KoAT2 + MΦRF5 + CFRSCC has approximately the same power, but a slightly higher runtime.

5.2 Evaluation on Complexity_C_Integer

The benchmark suite CINT consists of 484 C programs, where 366 of them *might* have finite runtime (since iRankFinder can show non-termination of 118 examples). To apply KoAT1 and KoAT2 on these benchmarks, one has to translate the C programs into integer programs as in Def. 2. To this end, we use the tool llvm2kittel [21] which performs this translation by using an intermediate representation of C programs as LLVM bytecode [37], obtained from the Clang compiler frontend [18]. The output of this transformation is then analyzed by KoAT1 and KoAT2.

The results of our evaluation on CINT can be found in Fig. 12. Here, Loopus solved 239 benchmarks, KoAT2 solved 281, KoAT1 solved 285, and CoFloCo solved 288 out of the 484 examples. Additionally, both MaxCore and KoAT2 + MΦRF5 solve 310 examples and KoAT2 + CFRSCC solves 320 examples. This makes KoAT2 the strongest tool on both benchmark sets. Applying partial evaluation on sub-SCCs instead of SCCs improves the average runtime of successful runs, without reducing the number of solved examples. When enabling both control-flow

	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{>2})$	$\mathcal{O}(EXP)$	$< \infty$	AVG$^+$(s)	AVG(s)
KoAT2 + MΦRF5 + CFR	24	228	65	11	0	328	4.77	16.40
KoAT2+MΦRF5+CFRSCC	24	228	65	11	0	328	5.72	16.53
KoAT2 + CFR	25	216	68	11	0	320	5.14	11.67
KoAT2 + CFRSCC	28	216	66	10	0	320	6.00	11.93
MaxCore	23	214	66	7	0	310	1.94	5.24
KoAT2 + MΦRF5	23	204	71	12	0	310	2.11	5.16
CFR + KoAT2	27	200	70	2	1	300	11.26	19.92
CFR + KoAT1	29	187	74	7	0	297	5.34	12.64
CoFloCo	22	195	66	5	0	288	0.81	2.95
KoAT1	25	168	74	12	6	285	2.36	2.97
KoAT2	23	176	70	12	0	281	2.05	2.76
Loopus	17	169	49	4	0	239	0.84	0.72

Fig. 12: Evaluation on Complexity_C_Integer

refinement and multiphase-linear ranking functions then KoAT2 is even stronger, as KoAT2 + MΦRF5 + CFR solves 328 examples. Moreover, it is faster than the equally powerful configuration KoAT2 + MΦRF5 + CFRSCC.

In contrast to KoAT1 and CoFloCo, MaxCore also proves a linear runtime bound for our example in Fig. 1, as it detects that y is eventually negative. However, when generalizing Fig. 1 to three phases as in [12] (see Fig. 10 and 11), KoAT2 with MΦRFs can infer the finite bound $27 \cdot x + 27 \cdot y + 27 \cdot z + 56$ on the runtime by using the MΦRF $(z + 1, y + 1, x)$, whereas the other tools fail. Moreover, KoAT2 with MΦRFs is the only tool that proves a finite time bound for the program in Fig. 4. To evaluate Loopus and MaxCore on this example, we translated it into C. While these tools failed, KoAT2 also succeeded on the integer program that was obtained by applying llvm2kittel to the translated program. This shows the robustness of MΦRFs for programs consisting of several phases.

For the example in Fig. 7 which demonstrates that MΦRFs do not subsume control-flow refinement (Ex. 27), KoAT2 with its control-flow refinement technique of Sect. 4 infers a linear runtime bound whereas KoAT1 fails, since its loop unrolling technique is a substantially weaker form of control-flow refinement. Besides our tool, only MaxCore was able to infer a finite runtime bound for the C version of this program, where however this bound was quadratic instead of linear.

To sum up, both multiphase-linear ranking functions and control-flow refinement lead to significant improvements. Combining the two techniques, our tool KoAT2 outperforms *all* existing state-of-the-art tools on both benchmark sets.

6 Related Work and Conclusion

Related Work. As mentioned in Sect. 1, many other techniques for automated complexity analysis of integer programs have been developed. The approach in [8] uses lexicographic combinations of linear ranking functions and Ehrhart polynomials to over-approximate the runtime complexity of integer programs. In [46], difference logic is used to analyze C programs. The works in [2–4, 22, 23] over-approximate so-called cost relations which are closely related to recurrence relations. In [6], a tool chain is presented which uses conditional termination proofs as in [14] to guide the inference of complexity bounds via cost relations by a complexity analyzer in the backend. Based on tools for complexity analysis of integer programs, there also exist approaches to analyze complexity for full programming languages like Java [24, 40]. In this way, they complement successful tools for functional verification of Java programs like [1]. Other approaches use the potential method from amortized analysis or type systems to analyze the complexity of C (see, e.g., [17]) or ML programs [32, 33]. An approach to verify whether a given resource bound for a program is valid is presented in [47]. While all of these works focus on over-approximating the worst-case runtime complexity of programs, there is also work on the inference of lower bounds on the worst-case runtime complexity, see, e.g., [7, 27, 49]. Moreover, our tool KoAT also offers the possibility to analyze the expected runtime complexity of probabilistic integer programs, because we also transferred the approach from [16] to probabilistic integer programs [39] and we also integrated decision procedures for the termination and complexity of restricted classes of probabilistic programs in KoAT [30]. See [35] for an overview on runtime analysis for probabilistic programs.

A fundamentally different concept to integer programs are so-called term rewrite systems. These systems model recursion and algebraic data structures, but they do not have any built-in data type for integers. There is also a wealth of techniques and tools to analyze the runtime complexity of term rewrite systems automatically (see [9, 10, 25, 29, 43], for example).

Multiphase-linear ranking functions are studied in [12, 13, 38, 50], but these works mainly focus on termination instead of complexity analysis. Moreover, [12] shows how to obtain a linear bound on the runtime complexity of a program with a *single* MΦRF, while we developed a technique to combine MΦRFs on program parts to obtain bounds on the runtime complexity of the full program.

Using control-flow refinement for inferring runtime bounds is studied in [20, 22]. Here, [22] focuses on cost relations, while we embed the approach of [20] into our analysis of integer programs, where we do not apply this method globally but only locally on parts where we do not yet have a linear runtime bound.

Conclusion. In this paper, we showed how to adapt the approach for the computation of runtime and size bounds for integer programs from [16] to multiphase-linear ranking functions and to the use of control-flow refinement. As shown by our experimental evaluation, due to these new improvements, the resulting implementation in our new version of the tool KoAT outperforms the other existing tools for complexity analysis of integer programs.

KoAT's source code, a binary, and a Docker image are available at https://aprove-developers.github.io/ComplexityMprfCfr/. This web site also provides details on our experiments and *web interfaces* to run KoAT directly online.

Acknowledgments. This paper is dedicated to Reiner Hähnle whose groundbreaking results on functional verification and symbolic execution of Java programs with the KeY tool [1], on automatic resource analysis [22], and on its combination with deductive verification (e.g., [5]) were a major inspiration for us. Reiner's work motivated us to develop and improve KoAT such that it can be used as a backend for complexity analysis of languages like Java [24].

We are indebted to Samir Genaim and Jesús J. Doménech for their help and advice with integrating multiphase-linear ranking functions and partial evaluation into our approach, and for providing us with a suitable version of iRankFinder which we could use in KoAT's backend. Moreover, we are grateful to Albert Rubio and Enrique Martín-Martín for providing us with a static binary of MaxCore, to Antonio Flores-Montoya and Florian Zuleger for their help in running CoFloCo and Loopus for our experiments, and to Florian Frohn for help and advice.

References

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich. *Deductive Software Verification - The* KeY *Book - From Theory to Practice*. LNCS 10001. 2016. DOI: 10.1007/978-3-319-49812-6.

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

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. "Cost Analysis of Object-Oriented Bytecode Programs". In: *Theor. Comput. Sci.* 413.1 (2012), pp. 142–159. DOI: 10.1016/j.tcs.2011.07.009.

[4] E. Albert, S. Genaim, and A. N. Masud. "On the Inference of Resource Usage Upper and Lower Bounds". In: *ACM Trans. Comput. Log.* 14.3 (2013), 22:1–22:35. DOI: 10.1145/2499937.2499943.

[5] E. Albert, R. Bubel, S. Genaim, R. Hähnle, G. Puebla, and G. Román-Díez. "A Formal Verification Framework for Static Analysis - As well as its Instantiation to the Resource Analyzer COSTA and Formal Verification Tool KeY". In: *Softw. Syst. Model.* 15.4 (2016), pp. 987–1012. DOI: 10.1007/s10270-015-0476-y.

[6] E. Albert, M. Bofill, C. Borralleras, E. Martín-Martín, and A. Rubio. "Resource Analysis driven by (Conditional) Termination Proofs". In: *Theory Pract. Log. Program.* 19.5-6 (2019), pp. 722–739. DOI: 10.1017/S1471068419000152.

[7] E. Albert, S. Genaim, E. Martín-Martín, A. Merayo, and A. Rubio. "Lower-Bound Synthesis Using Loop Specialization and Max-SMT". In: *Proc. CAV*. LNCS 12760. 2021, pp. 863–886. DOI: 10.1007/978-3-030-81688-9_40.

[8] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. "Multi-Dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs". In: *Proc. SAS*. LNCS 6337. 2010, pp. 117–133. DOI: 10.1007/978-3-642-15769-1_8.

[9] M. Avanzini and G. Moser. "A Combination Framework for Complexity". In: *Proc. RTA*. LIPIcs 21. 2013, pp. 55–70. DOI: 10.4230/LIPIcs.RTA.2013.55.

[10] M. Avanzini, G. Moser, and M. Schaper. “TcT: Tyrolean Complexity Tool”. In: *Proc. TACAS*. LNCS 9636. 2016, pp. 407–423. DOI: 10.1007/978-3-662-49674-9_24.

[11] A. M. Ben-Amram and S. Genaim. “Ranking Functions for Linear-Constraint Loops”. In: *J. ACM* 61.4 (2014), 26:1–26:55. DOI: 10.1145/2629488.

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

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

[14] C. Borralleras, M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. “Proving Termination Through Conditional Termination”. In: *Proc. TACAS*. LNCS 10205. 2017, pp. 99–117. DOI: 10.1007/978-3-662-54577-5_6.

[15] A. R. Bradley, Z. Manna, and H. B. Sipma. “The Polyranking Principle”. In: *Proc. ICALP*. LNCS 3580. 2005, pp. 1349–1361. DOI: 10.1007/11523468_109.

[16] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. “Analyzing Runtime and Size Complexity of Integer Programs”. In: *ACM Trans. Program. Lang. Syst.* 38.4 (2016), 13:1–13:50. DOI: 10.1145/2866575.

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

[18] Clang compiler. URL: https://clang.llvm.org/.

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

[20] J. J. Doménech, J. P. Gallagher, and S. Genaim. “Control-Flow Refinement by Partial Evaluation, and its Application to Termination and Cost Analysis”. In: *Theory Pract. Log. Program.* 19.5-6 (2019), pp. 990–1005. DOI: 10.1017/S1471068419000310.

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

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

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

[24] F. Frohn and J. Giesl. “Complexity Analysis for Java with AProVE”. In: *Proc. iFM*. LNCS 10510. 2017, pp. 85–101. DOI: 10.1007/978-3-319-66845-1_6.

[25] F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder. “Lower Bounds for Runtime Complexity of Term Rewriting”. In: *J. Autom. Reason.* 59.1 (2017), pp. 121–163. DOI: 10.1007/s10817-016-9397-x.

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

[27] F. Frohn, M. Naaf, M. Brockschmidt, and J. Giesl. “Inferring Lower Runtime Bounds for Integer Programs”. In: *ACM Trans. Program. Lang. Syst.* 42.3 (2020), 13:1–13:50. DOI: 10.1145/3410331.

[28] J. P. Gallagher. “Polyvariant Program Specialisation with Property-Based Abstraction”. In: *VPT@Programming.* EPTCS 299. 2019, pp. 34–48. DOI: 10.4204/EPTCS.299.6.

[29] 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. Reason.* 58.1 (2017), pp. 3–31. DOI: 10.1007/s10817-016-9388-y.

[30] J. Giesl, P. Giesl, and M. Hark. “Computing Expected Runtimes for Constant Probability Programs”. In: *Proc. CADE.* LNCS 11716. 2019, pp. 269–286. DOI: 10.1007/978-3-030-29436-6_16.

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

[32] J. Hoffmann, K. Aehlig, and M. Hofmann. “Multivariate Amortized Resource Analysis”. In: *ACM Trans. Program. Lang. Syst.* 34.3 (2012), 14:1–14:62. DOI: 10.1145/2362389.2362393.

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

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

[35] B. L. Kaminski, J.-P. Katoen, and C. Matheja. “Expected Runtime Analysis by Program Verification”. In: *Foundations of Probabilistic Programming.* Cambridge University Press, G. Barthe, J.-P. Katoen, and A. Silva (eds.) 2020, pp. 185–220. DOI: 10.1017/9781108770750.007.

[36] K. Königsberger. *Analysis 1.* 6. Aufl. 2004. Springer. DOI: 10.1007/978-3-642-18490-1.

[37] C. Lattner and V. S. Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proc. CGO.* 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665.

[38] J. Leike and M. Heizmann. “Ranking Templates for Linear Loops”. In: *Log. Methods Comput. Sci.* 11.1 (2015). DOI: 10.2168/LMCS-11(1:16)2015.

[39] F. Meyer, M. Hark, and J. Giesl. “Inferring Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes”. In: *Proc. TACAS.* LNCS 12651. 2021, pp. 250–269. DOI: 10.1007/978-3-030-72016-2_14.

[40] G. Moser and M. Schaper. "From Jinja Bytecode to Term Rewriting: A Complexity Reflecting Transformation". In: *Inf. Comput.* 261 (2018), pp. 116–143. DOI: 10.1016/j.ic.2018.05.007.

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

[42] KoAT: Web Interface, Experiments, Source Code, Binary, and Docker Image. URL: https://aprove-developers.github.io/ComplexityMprfCfr/.

[43] L. Noschinski, F. Emmes, and J. Giesl. "Analyzing Innermost Runtime Complexity of Term Rewriting by Dependency Pairs". In: *J. Autom. Reason.* 51.1 (2013), pp. 27–56. DOI: 10.1007/s10817-013-9277-6.

[44] A. Podelski and A. Rybalchenko. "A Complete Method for the Synthesis of Linear Ranking Functions". In: *Proc. VMCAI*. LNCS 2937. 2004, pp. 239–251. DOI: 10.1007/978-3-540-24622-0_20.

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

[46] M. Sinn, F. Zuleger, and H. Veith. "Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints". In: *J. Autom. Reason.* 59.1 (2017), pp. 3–45. DOI: 10.1007/s10817-016-9402-4.

[47] A. Srikanth, B. Sahin, and W. R. Harris. "Complexity Verification Using Guided Theorem Enumeration". In: *Proc. POPL*. 2017, pp. 639–652. DOI: 10.1145/3009837.3009864.

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

[49] D. Wang and J. Hoffmann. "Type-Guided Worst-Case Input Generation". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 13:1–13:30. DOI: 10.1145/3290326.

[50] Y. Yuan, Y. Li, and W. Shi. "Detecting Multiphase Linear Ranking Functions for Single-Path Linear-Constraint Loops". In: *Int. J. Softw. Tools Tech. Transf.* 23.1 (2021), pp. 55–67. DOI: 10.1007/s10009-019-00527-1.

A Proofs

A.1 Proof of Lemma 18

We first present lemmas which give an upper and a lower bound for sums of powers. These lemmas will be needed in the proof of Lemma 18.

Lemma 28 (Upper Bound for Sums of Powers). *For any $i \geq 2$ and $k \geq 1$ we have $\sum_{j=1}^{k-1} j^{i-2} \leq \frac{k^{i-1}}{i-1}$.*

Proof. We have $\sum_{j=1}^{k-1} j^{i-2} \leq \sum_{j=1}^{k-1} \int_j^{j+1} x^{i-2}\, dx \leq \int_0^k x^{i-2}\, dx = \frac{k^{i-1}}{i-1}$. □

For the lower bound, we use the summation formula of Euler (see, e.g., [36]).

Lemma 29 (Summation Formula of Euler). *We define the periodic function $H : \mathbb{R} \to \mathbb{R}$ as $H(x) = x - \lfloor x \rfloor - \frac{1}{2}$ if $x \in \mathbb{R} \setminus \mathbb{Z}$ and as $H(x) = 0$ if $x \in \mathbb{Z}$. Note that $H(x)$ is bounded by $-\frac{1}{2}$ and $\frac{1}{2}$. Then for any continuously differentiable* *function* $f : [1,n] \to \mathbb{C}$ *with* $n \in \mathbb{N}$*, we have* $\sum_{j=1}^{k} f(j) \;=\; \int_1^k f(x)\,dx + \frac{1}{2}\cdot(f(1) + f(k)) + \int_1^k H(x)\cdot f'(x)\,dx$*.*

This then leads to the following result.

Lemma 30 (Lower Bound for Sums of Powers). *For any* $i \geq 2$ *and* $k \geq 1$ *we have* $\sum_{j=1}^{k-1} j^{i-1} \geq \frac{k^i}{i} - k^{i-1}$*.*

Proof. Consider $f(x) = x^i$ with the derivative $f'(x) = i \cdot x^{i-1}$. We get

$$\begin{aligned}
&\sum\nolimits_{j=1}^{k} j^i \\
=\;& \int_1^k x^i\,dx + \tfrac{1}{2}\cdot(1+k^i) + \int_1^k H(x)\cdot i\cdot x^{i-1}\,dx && \text{(by Lemma 29)} \\
=\;& \tfrac{k^{i+1}}{i+1} - \tfrac{1}{i+1} + \tfrac{1}{2}\cdot(1+k^i) + \int_1^k H(x)\cdot i\cdot x^{i-1}\,dx \\
=\;& \tfrac{k^{i+1}}{i+1} + R && \text{(for } R = -\tfrac{1}{i+1} + \tfrac{1}{2}\cdot(1+k^i) + \textstyle\int_1^k H(x)\cdot i\cdot x^{i-1}\,dx\text{)}
\end{aligned}$$

Since $|H(x)| \leq \frac{1}{2}$, we have $\left|\int_1^k H(x)\cdot i\cdot x^{i-1}\,dx\right| \;\leq\; \frac{1}{2}\cdot\left|\int_1^k i\cdot x^{i-1}\,dx\right| \;=\; \frac{1}{2}\cdot i\cdot\left|\frac{k^i}{i} - \frac{1}{i}\right| \;=\; \frac{k^i-1}{2}$. Thus, we obtain

$$-\tfrac{1}{i+1} + \tfrac{1}{2}\cdot(1+k^i) + \tfrac{k^i-1}{2} \geq R \geq -\tfrac{1}{i+1} + \tfrac{1}{2}\cdot(1+k^i) - \tfrac{k^i-1}{2}$$

or, equivalently $-\frac{1}{i+1} + k^i \geq R \geq -\frac{1}{i+1} + 1$. This implies $k^i > R > 0$. Hence, we get $\sum_{j=1}^{k} j^i \;=\; \frac{k^{i+1}}{i+1} + R \;\geq\; \frac{k^{i+1}}{i+1}$ and thus, $\sum_{j=1}^{k-1} j^i \;=\; \sum_{j=1}^{k} j^i - k^i \;\geq\; \frac{k^{i+1}}{i+1} - k^i$. With the index shift $i \to i-1$ we finally obtain the lower bound $\sum_{j=1}^{k-1} j^{i-1} \geq \frac{k^i}{i} - k^{i-1}$. □

Proof of Lemma 18. To ease notation, in this proof ℓ_0 does not denote the initial location of the program $\mathcal{T}$, but an arbitrary location from $\mathcal{L}$. Then we can write (ℓ_0, σ_0) instead of (ℓ, σ), (ℓ_n, σ_n) instead of (ℓ', σ'), and consider an evaluation

$$(\ell_0, \sigma_0)\,(\to^*_{\mathcal{T}'\setminus\mathcal{T}'_>} \circ \to_{\mathcal{T}'_>})\,(\ell_1, \sigma_1)\,(\to^*_{\mathcal{T}'\setminus\mathcal{T}'_>} \circ \to_{\mathcal{T}'_>})\,\ldots\,(\to^*_{\mathcal{T}'\setminus\mathcal{T}'_>} \circ \to_{\mathcal{T}'_>})\,(\ell_n, \sigma_n).$$

Let $M = \max\{0, \sigma_0\,(f_1(\ell_0)), \ldots, \sigma_0\,(f_d(\ell_0))\}$. We first prove that for all $1 \leq i \leq d$ and all $0 \leq k \leq n$, we have

$$\sigma_k(f_i(\ell_k)) \;\leq\; -k \text{ if M} = 0 \quad\text{and}\quad \sigma_k(f_i(\ell_k)) \;\leq\; \gamma_i{\cdot}M{\cdot}k^{i-1} - \tfrac{k^i}{i!} \;\text{ if } M > 0. \quad (2)$$

The proof is done by induction on i. So in the base case, we have $i = 1$. Since $\gamma_1 = 1$, we have to show that $\sigma_k\,(f_1(\ell_k)) \leq M \cdot k^0 - \frac{k^1}{1!} = M - k$.

For all $0 \leq j \leq k-1$, the step from (ℓ_j, σ_j) to $(\ell_{j+1}, \sigma_{j+1})$ corresponds to the evaluation of transitions from $\mathcal{T}' \setminus \mathcal{T}'_>$ followed by a transition from $\mathcal{T}'_>$, i.e., we have $(\ell_j, \sigma_j) \to^*_{\mathcal{T}'\setminus\mathcal{T}'_>} (\ell'_j, \sigma'_j) \to_{\mathcal{T}'_>} (\ell_{j+1}, \sigma_{j+1})$ for some configuration (ℓ'_j, σ'_j). Since f is an MΦRF and all transitions in $\mathcal{T}' \setminus \mathcal{T}'_>$ are non-increasing, we obtain $\sigma_j(f_1(\ell_j)) \geq \sigma'_j(f_1(\ell'_j))$. Moreover, since the transitions in $\mathcal{T}'_>$ are decreasing, we have $\sigma'_j(f_0(\ell'_j)) + \sigma'_j(f_1(\ell'_j)) = \sigma'_j(f_1(\ell'_j)) \geq \sigma_{j+1}(f_1(\ell_{j+1})) + 1$. So together, this implies $\sigma_j(f_1(\ell_j)) \geq \sigma_{j+1}(f_1(\ell_{j+1})) + 1$ and thus, $\sigma_0\,(f_1(\ell_0)) \geq \sigma_1\,(f_1(\ell_1)) + 1 \geq$

$\ldots \geq \sigma_k\left(f_1(\ell_k)\right)+k$ or equivalently, $\sigma_0\left(f_1(\ell_0)\right)-k \geq \sigma_k\left(f_1(\ell_k)\right)$. Furthermore, we have $\sigma_0\left(f_1(\ell_0)\right) \leq \max\{0, \sigma_0\left(f_1(\ell_0)\right), \ldots, \sigma_0\left(f_d(\ell_0)\right)\} = M$. Hence, we obtain $\sigma_k\left(f_1(\ell_k)\right) \leq \sigma_0\left(f_1(\ell_0)\right) - k \leq M-k$. So in particular, if $M=0$, then we have $\sigma_k(f_1(\ell_k)) \leq -k$.

In the induction step, we assume that for all $0 \leq k \leq n$, we have $\sigma_k(f_{i-1}(\ell_k)) \leq -k$ if $M=0$ and $\sigma_k(f_{i-1}(\ell_k)) \leq \gamma_{i-1} \cdot M \cdot k^{i-2} - \frac{k^{i-1}}{(i-1)!}$ if $M>0$. To show that the inequations also hold for i, we first transform $\sigma_k(f_i(\ell_k))$ into a telescoping sum.

$$\sigma_k\left(f_i(\ell_k)\right) = \sigma_0\left(f_i(\ell_0)\right) + \sum_{j=0}^{k-1}(\sigma_{j+1}\left(f_i(\ell_{j+1})\right) - \sigma_j\left(f_i(\ell_j)\right))$$

For all $0 \leq j \leq k-1$, the step from (ℓ_j, σ_j) to $(\ell_{j+1}, \sigma_{j+1})$ again has the form $(\ell_j, \sigma_j) \to^*_{\mathcal{T}' \setminus \mathcal{T}'_>} (\ell'_j, \sigma'_j) \to_{\mathcal{T}'_>} (\ell_{j+1}, \sigma_{j+1})$ for some configuration (ℓ'_j, σ'_j). Since f is an MΦRF and all transitions in $\mathcal{T}' \setminus \mathcal{T}'_>$ are non-increasing, we obtain $\sigma_j(f_{i-1}(\ell_j)) \geq \sigma'_j(f_{i-1}(\ell'_j))$ and $\sigma_j(f_i(\ell_j)) \geq \sigma'_j(f_i(\ell'_j))$. Moreover, since the transitions in $\mathcal{T}'_>$ are decreasing, we have $\sigma'_j(f_{i-1}(\ell'_j)) + \sigma'_j(f_i(\ell'_j)) \geq \sigma_{j+1}(f_i(\ell_{j+1})) + 1$. So together, this implies $\sigma_j(f_{i-1}(\ell_j)) + \sigma_j(f_i(\ell_j)) \geq \sigma_{j+1}(f_i(\ell_{j+1})) + 1$ or equivalently, $\sigma_{j+1}\left(f_i(\ell_{j+1})\right) - \sigma_j\left(f_i(\ell_j)\right) < \sigma_j\left(f_{i-1}(\ell_j)\right)$. Hence, we obtain

$$\begin{aligned}\sigma_k\left(f_i(\ell_k)\right) &= \sigma_0\left(f_i(\ell_0)\right) + \sum_{j=0}^{k-1}(\sigma_{j+1}\left(f_i(\ell_{j+1})\right) - \sigma_j\left(f_i(\ell_j)\right)) \\ &< \sigma_0\left(f_i(\ell_0)\right) + \sum_{j=0}^{k-1}\sigma_j\left(f_{i-1}(\ell_j)\right).\end{aligned}$$

If $M=0$, then we obviously have $\sigma_0(f_i(\ell_0)) \leq 0$ for all $1 \leq i \leq d$. For $k \geq 1$, we obtain

$$\begin{aligned}&\sigma_0\left(f_i(\ell_0)\right) + \sum_{j=0}^{k-1}\sigma_j\left(f_{i-1}(\ell_j)\right) \\ \leq\; & 0 + \sum_{j=0}^{k-1} -j && \text{(by the induction hypothesis)} \\ \leq\; & -k+1.\end{aligned}$$

Hence, we have $\sigma_k\left(f_i(\ell_k)\right) < -k+1$ and thus, $\sigma_k\left(f_i(\ell_k)\right) \leq -k$.

If $M>0$, then we obtain

$$\begin{aligned}&\sigma_0\left(f_i(\ell_0)\right) + \sum_{j=0}^{k-1}\sigma_j\left(f_{i-1}(\ell_j)\right) \\ \leq\; & 2 \cdot M + \sum_{j=1}^{k-1}\sigma_j\left(f_{i-1}(\ell_j)\right) && \text{(as } \sigma_0\left(f_i(\ell_0)\right) \leq M \text{ and } \sigma_0\left(f_{i-1}(\ell_0)\right) \leq M\text{)} \\ \leq\; & 2 \cdot M + \sum_{j=1}^{k-1}(\gamma_{i-1} \cdot M \cdot j^{i-2} - \tfrac{j^{i-1}}{(i-1)!}) && \text{(by the induction hypothesis)}\end{aligned}$$

$$
\begin{aligned}
&= 2 \cdot M + \gamma_{i-1} \cdot M \cdot \left(\sum_{j=1}^{k-1} j^{i-2} \right) - \tfrac{1}{(i-1)!} \cdot \left(\sum_{j=1}^{k-1} j^{i-1} \right) \\
&\leq 2 \cdot M + \gamma_{i-1} \cdot M \cdot \tfrac{k^{i-1}}{i-1} - \tfrac{1}{(i-1)!} \cdot \left(\tfrac{k^i}{i} - k^{i-1} \right) && \text{(by Lemma 28 and 30)} \\
&= 2 \cdot M + \gamma_{i-1} \cdot M \cdot \tfrac{k^{i-1}}{i-1} + \tfrac{k^{i-1}}{(i-1)!} - \tfrac{k^i}{i!} \\
&\leq 2 \cdot M \cdot k^{i-1} + \gamma_{i-1} \cdot M \cdot \tfrac{k^{i-1}}{i-1} + \tfrac{k^{i-1}}{(i-1)!} - \tfrac{k^i}{i!} \\
&\leq M \cdot k^{i-1} \cdot \left(\underbrace{2 + \tfrac{\gamma_{i-1}}{i-1} + \tfrac{1}{(i-1)!}}_{\gamma_i} \right) - \tfrac{k^i}{i!} && \text{(as } M \geq 1\text{)} \\
&= M \cdot k^{i-1} \cdot \gamma_i - \tfrac{k^i}{i!}.
\end{aligned}
$$

Hence, (2) is proved.

In the case $M = 0$, (2) implies $\sigma_n(f_i(\ell_n)) \leq -n \leq -\beta = -1 < 0$ for all $1 \leq i \leq d$ which proves the lemma.

Hence, it remains to regard the case $M > 0$. Now (2) implies

$$\sigma_n(f_i(\ell_n)) \;\leq\; \gamma_i \cdot M \cdot n^{i-1} - \tfrac{n^i}{i!}. \tag{3}$$

We now prove that for $i > 1$ we always have $i! \cdot \gamma_i \geq (i-1)! \cdot \gamma_{i-1}$.

$$
\begin{aligned}
& i! \cdot \gamma_i \\
&= i! \cdot \left(2 + \tfrac{\gamma_{i-1}}{i-1} + \tfrac{1}{(i-1)!} \right) \\
&= i! \cdot 2 + i \cdot (i-2)! \cdot \gamma_{i-1} + i \\
&\geq (i-1) \cdot (i-2)! \cdot \gamma_{i-1} \\
&= (i-1)! \cdot \gamma_{i-1}.
\end{aligned}
$$

Thus,

$$d! \cdot \gamma_d \geq i! \cdot \gamma_i \quad \text{for all } 1 \leq i \leq d. \tag{4}$$

Hence, for $n \geq \beta = 1 + d! \cdot \gamma_d \cdot M$ we obtain:

$$
\begin{aligned}
& \sigma_n(f_i(\ell_n)) \\
&\leq \gamma_i \cdot M \cdot n^{i-1} - \tfrac{n^i}{i!} && \text{(by (3))} \\
&= \tfrac{n^{i-1}}{i!} \cdot (i! \cdot \gamma_i \cdot M - n) \\
&\leq \tfrac{n^{i-1}}{i!} \cdot (\beta - 1 - n) && \text{(by (4))} \\
&< 0 && \text{(since } n \geq \beta\text{)}
\end{aligned}
$$

Finally, to show that $\beta \in \mathbb{N}$, note that by induction on i, one can easily prove that $(i-1)! \cdot \gamma_i \in \mathbb{N}$ holds for all $i \geq 1$. Hence, in contrast to γ_i, the number $i! \cdot \gamma_i$ is a natural number for all $i \in \mathbb{N}$. This implies $\beta \in \mathbb{N}$. □

A.2 Proof of Thm. 20

Proof. We prove Thm. 20 by showing that for all $t \in \mathcal{T}$ and all $\sigma_0 \in \Sigma$ we have

$$|\sigma_0| \left(\mathcal{RB}'(t) \right) \geq \sup\{k \in \mathbb{N} \mid \ell \in \mathcal{L}, \sigma \in \Sigma, (\ell_0, \sigma_0) \left(\to^* \circ \to_t \right)^k (\ell, \sigma)\}. \tag{5}$$

The case $t \notin \mathcal{T}'_>$ is trivial, since $\mathcal{RB}'(t) = \mathcal{RB}(t)$ and $\mathcal{RB}$ is a runtime bound.

Now we prove (5) for a transition $t_> \in \mathcal{T}'_>$, i.e., we show that for all $\sigma_0 \in \Sigma$ we have

$$\begin{aligned} |\sigma_0|\left(\mathcal{RB}'(t_>)\right) \; &= \; \textstyle\sum_{\ell \in \mathcal{E}_{\mathcal{T}'}} \sum_{t \in \mathcal{T}_\ell} |\sigma_0|\left(\mathcal{RB}(t)\right) \cdot |\sigma_0|\left(\mathcal{SB}(t,\cdot)(\beta_\ell)\right) \\ &\geq \; \sup\{k \in \mathbb{N} \mid \ell \in \mathcal{L}, \sigma \in \Sigma, (\ell_0, \sigma_0)\,(\to^* \circ \to_{t_>})^k\,(\ell, \sigma)\}. \end{aligned}$$

So let $(\ell_0, \sigma_0)\,(\to^* \circ \to_{t_>})^k\,(\ell, \sigma)$ and we have to show $|\sigma_0|\left(\mathcal{RB}'(t_>)\right) \geq k$. If $k = 0$, then we clearly have $|\sigma_0|\left(\mathcal{RB}'\,(t_>)\right) \geq 0 = k$. Hence, we consider $k > 0$. We represent the evaluation as follows:

$$\begin{array}{llll} (\ell_0, \sigma_0) & \to^{\tilde{k}_0}_{\mathcal{T} \setminus \mathcal{T}'} & (\tilde{\ell}_1, \tilde{\sigma}_1) & \to^{k'_1}_{\mathcal{T}'} \\ (\ell_1, \sigma_1) & \to^{\tilde{k}_1}_{\mathcal{T} \setminus \mathcal{T}'} & (\tilde{\ell}_2, \tilde{\sigma}_2) & \to^{k'_2}_{\mathcal{T}'} \\ & & \ldots & \\ (\ell_{m-1}, \sigma_{m-1}) & \to^{\tilde{k}_{m-1}}_{\mathcal{T} \setminus \mathcal{T}'} & (\tilde{\ell}_m, \tilde{\sigma}_m) & \to^{k'_m}_{\mathcal{T}'} \\ (\ell_m, \sigma_m) & & & \end{array}$$

So for the evaluation from (ℓ_i, σ_i) to $(\tilde{\ell}_{i+1}, \tilde{\sigma}_{i+1})$ we only use transitions from $\mathcal{T} \setminus \mathcal{T}'$, and for the evaluation from $(\tilde{\ell}_i, \tilde{\sigma}_i)$ to (ℓ_i, σ_i) we only use transitions from $\mathcal{T}'$. Thus, $t_>$ can only occur in the following finite sequences of evaluation steps:

$$(\tilde{\ell}_i, \tilde{\sigma}_i) \to_{\mathcal{T}'} (\tilde{\ell}_{i,1}, \tilde{\sigma}_{i,1}) \to_{\mathcal{T}'} \cdots \to_{\mathcal{T}'} (\tilde{\ell}_{i,k'_i-1}, \tilde{\sigma}_{i,k'_i-1}) \to_{\mathcal{T}'} (\ell_i, \sigma_i). \tag{6}$$

For every $1 \leq i \leq m$, let $k_i \leq k'_i$ be the number of times that $t_>$ is used in the evaluation (6). Clearly, we have

$$\sum_{i=1}^{m} k_i = k. \tag{7}$$

By Lemma 18, all functions $f_1, \ldots, f_d$ are negative after executing $t_>$ at least $1 + d! \cdot \gamma_d \cdot \max\{0, \tilde{\sigma}_i(f_1(\tilde{\ell}_i)), \ldots, \tilde{\sigma}_i(f_d(\tilde{\ell}_i))\}$ times in an evaluation with $\mathcal{T}'$. If all the f_i are negative, then $t_>$ cannot be executed anymore as f is an MΦRF for $\mathcal{T}'_>$ with $t_> \in \mathcal{T}'_>$ and $\mathcal{T}'$. Thus, for all $1 \leq i \leq m$ we have

$$1 + d! \cdot \gamma_d \cdot \max\left\{0, \tilde{\sigma}_i\left(f_1(\tilde{\ell}_i)\right), \ldots, \tilde{\sigma}_i\left(f_d(\tilde{\ell}_i)\right)\right\} \geq k_i. \tag{8}$$

Let t_i be the entry transition reaching $(\tilde{\ell}_i, \tilde{\sigma}_i)$, i.e., $\tilde{\ell}_i \in \mathcal{E}_{\mathcal{T}'}$ and $t_i \in \mathcal{T}_{\tilde{\ell}_i}$. As $(\ell_0, \sigma_0) \to^*_{\mathcal{T}} \circ \to_{t_i} (\tilde{\ell}_i, \tilde{\sigma}_i)$, by Def. 12 we have $|\sigma_0|\left(\mathcal{SB}(t_i, v)\right) \geq |\tilde{\sigma}_i(v)|$ for all $v \in \mathcal{PV}$ and thus,

$$\begin{aligned} & |\sigma_0|\left(\mathcal{SB}(t_i, \cdot)(\beta_{\tilde{\ell}_i})\right) \\ \geq \; & |\tilde{\sigma}_i|\left(\beta_{\tilde{\ell}_i}\right) && \text{(since } \beta_{\tilde{\ell}_i} \in \mathcal{B}\text{)} \\ \geq \; & \tilde{\sigma}_i(\beta_{\tilde{\ell}_i}) \\ \geq \; & 1 + d! \cdot \gamma_d \cdot \max\left\{0, \tilde{\sigma}_i\left(f_1(\tilde{\ell}_i)\right), \ldots, \tilde{\sigma}_i\left(f_d(\tilde{\ell}_i)\right)\right\} \\ & && \text{(by definition of } \lceil\cdot\rceil \text{ and } \beta_{\tilde{\ell}_i}\text{)} \\ \geq \; & k_i && \text{(by (8))} \end{aligned}$$

In the last part of this proof we need to analyze how often such evaluations $(\tilde{\ell}_i, \tilde{\sigma}_i) \to^*_{\mathcal{T}'} (\ell_i, \sigma_i)$ can occur. Again, let t_i be the entry transition reaching $(\tilde{\ell}_i, \tilde{\sigma}_i)$. Every entry transition t_i can occur at most $|\sigma_0|\left(\mathcal{RB}(t_i)\right)$ times in the complete evaluation, as $\mathcal{RB}$ is a runtime bound. Thus, we have

$$\begin{aligned}
|\sigma_0|\left(\mathcal{RB}'\left(t_>\right)\right) &= \sum_{\ell \in \mathcal{E}_{\mathcal{T}'}} \sum_{t \in \mathcal{T}_\ell} |\sigma_0|\left(\mathcal{RB}(t)\right) \cdot |\sigma_0|\left(\mathcal{SB}(t,\cdot)(\beta_\ell)\right) \\
&\geq \sum_{i=1}^{m} |\sigma_0|\left(\mathcal{SB}(t_i,\cdot)(\beta_{\tilde{\ell}_i})\right) \\
&\geq \sum_{i=1}^{m} k_i && \text{(as shown above)} \\
&= k && \text{(by (7))}
\end{aligned}$$

□

A.3 Proof of Thm. 24

Let $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}', \ell_0, \mathcal{T}')$. First note that for every evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}'}^{k} (\ell', \sigma)$ there is obviously also a corresponding evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}}^{k} (\ell, \sigma)$. To obtain the evaluation with $\mathcal{T}$ one simply has to remove the labels from the locations. Then the claim follows because the guards of the transitions in $\mathcal{T}'$ always imply the guards of the respective original transitions in $\mathcal{T}$ and the updates of the transitions have not been modified in the transformation from $\mathcal{T}$ to $\mathcal{T}'$.

For the other direction, we show by induction on $k \in \mathbb{N}$ that for every evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}}^{k} (\ell, \sigma)$ there is a corresponding evaluation $(\ell_0, \sigma_0) \to_{\mathcal{T}'}^{k} (\ell', \sigma)$ where either $\ell' = \ell$ or $\ell' = \langle \ell, \varphi \rangle$ for some constraint φ with $\sigma(\varphi) = \texttt{true}$.

In the induction base, we have $k = 0$ and the claim is trivial. In the induction step $k > 0$ the evaluation has the form

$$(\ell_0, \sigma_0) \to_{t_1} (\ell_1, \sigma_1) \to_{t_2} \cdots \to_{t_{k-1}} (\ell_{k-1}, \sigma_{k-1}) \to_{t_k} (\ell_k, \sigma_k)$$

with $t_1, \dots, t_k \in \mathcal{T}$. By the induction hypothesis, there is a corresponding evaluation

$$(\ell_0, \sigma_0) \to_{t_1'} (\ell_1', \sigma_1) \to_{t_2'} \cdots \to_{t_{k-1}'} (\ell_{k-1}', \sigma_{k-1})$$

with $t_1', \dots, t_k' \in \mathcal{T}'$ where $\ell_{k-1}' = \ell_{k-1}$ or $\ell_{k-1}' = \langle \ell_{k-1}, \varphi \rangle$ for some constraint φ with $\sigma_{k-1}(\varphi) = \texttt{true}$. We distinguish two cases:

Case 1: $t_k \notin \mathcal{T}_{SCC}$. If $\ell_{k-1}' = \ell_{k-1}$ and $\ell_k \notin \mathcal{E}_{\mathcal{T}_{SCC}}$, then t_k has not been modified in the transformation from $\mathcal{P}$ to $\mathcal{P}'$. Thus, we have the evaluation $(\ell_0, \sigma_0) \to_{t_1'} (\ell_1', \sigma_1) \to_{t_2'} \cdots \to_{t_{k-1}'} (\ell_{k-1}', \sigma_{k-1}) = (\ell_{k-1}, \sigma_{k-1}) \to_{t_k} (\ell_k, \sigma_k)$ with $t_k \in \mathcal{T}'$. If $\ell_{k-1}' = \ell_{k-1}$ and $\ell_k \in \mathcal{E}_{\mathcal{T}_{SCC}}$, then for $t_k = (\ell_{k-1}, \tau, \eta, \ell_k)$, we set $\ell_k' = \langle \ell_k, \texttt{true} \rangle$ and obtain that $t_k' = (\ell_{k-1}, \tau, \eta, \ell_k') \in \mathcal{T}'$. So we get the evaluation $(\ell_0, \sigma_0) \to_{t_1'} (\ell_1', \sigma_1) \to_{t_2'} \cdots \to_{t_{k-1}'} (\ell_{k-1}', \sigma_{k-1}) = (\ell_{k-1}, \sigma_{k-1}) \to_{t_k'} (\ell_k', \sigma_k)$. Finally, we regard the case $\ell_{k-1}' = \langle \ell_{k-1}, \varphi \rangle$ where $\sigma_{k-1}(\varphi) = \texttt{true}$. As $t_k = (\ell_{k-1}, \tau, \eta, \ell_k) \in \mathcal{T} \setminus \mathcal{T}_{SCC}$, and $\mathcal{T}_{SCC}$ is an SCC, there is a $t_k' = (\langle \ell_{k-1}, \varphi \rangle, \varphi \wedge \tau, \eta, \ell_k) \in \mathcal{T}'$. Then $(\ell_0, \sigma_0) \to_{t_1'} (\ell_1', \sigma_1) \to_{t_2'} \cdots \to_{t_{k-1}'} (\ell_{k-1}', \sigma_{k-1}) = (\langle \ell_{k-1}, \varphi \rangle, \sigma_{k-1}) \to_{t_k'} (\ell_k, \sigma_k)$ is an evaluation with $\mathcal{T}'$. The evaluation step with t_k' is possible, since $\sigma_{k-1}(\varphi) = \texttt{true}$ and $\sigma_{k-1}(\tau) = \texttt{true}$ (due to the evaluation step $(\ell_{k-1}, \sigma_{k-1}) \to_{t_k} (\ell_k, \sigma_k)$). Note that the step with t_k' also results in the state σ_k, because both t_k and t_k' have the same update η.

Case 2: $t_k \in \mathcal{T}_{SCC}$**.** Here, ℓ'_{k-1} has the form $\langle \ell_{k-1}, \varphi \rangle$ where $\sigma_{k-1}(\varphi) = \texttt{true}$. As ℓ_k is part of the SCC and hence has an incoming transition from $\mathcal{T}_{SCC}$, at some point it is refined by Alg. 2. Thus, for $t_k = (\ell_{k-1}, \tau, \eta, \ell_k)$, there is some $t'_k = (\langle \ell_{k-1}, \varphi \rangle, \varphi \wedge \tau, \eta, \langle \ell_k, \alpha_{\ell_k}(\varphi_{new}) \rangle) \in \mathcal{T}'$ where $\alpha_{\ell_k}(\varphi_{new})$ is constructed as in Line 8. This leads to the corresponding evaluation $(\ell_0, \sigma_0) \to_{t'_1} (\ell'_1, \sigma_1) \to_{t'_2} \cdots \to_{t'_{k-1}} (\langle \ell_{k-1}, \varphi \rangle, \sigma_{k-1}) \to_{t'_k} (\langle \ell_k, \alpha_{\ell_k}(\varphi_{new}) \rangle, \sigma_k)$. Again, the evaluation step with t'_k is possible, since $\sigma_{k-1}(\varphi) = \texttt{true}$ and $\sigma_{k-1}(\tau) = \texttt{true}$ (due to the evaluation step $(\ell_{k-1}, \sigma_{k-1}) \to_{t_k} (\ell_k, \sigma_k)$). And again, the step with t'_k also results in the state σ_k, because both t_k and t'_k have the same update η. Finally, note that we have $\sigma_k(\alpha_{\ell_k}(\varphi_{new})) = \texttt{true}$. The reason is that $\models (\varphi \wedge \tau) \to \eta(\varphi_{new})$ and $\sigma_{k-1}(\varphi \wedge \tau) = \texttt{true}$ implies $\sigma_{k-1}(\eta(\varphi_{new})) = \texttt{true}$. Hence, we also have $\sigma_k(\varphi_{new}) = \sigma_{k-1}(\eta(\varphi_{new})) = \texttt{true}$. Therefore, $\models \varphi_{new} \to \alpha_{\ell_k}(\varphi_{new})$ implies $\sigma_k(\alpha_{\ell_k}(\varphi_{new})) = \texttt{true}$. □

A.4 Proof of Thm. 25

Let $\mathcal{P}' = (\mathcal{PV}, \mathcal{L}', \ell_0, \mathcal{T}')$ result from $\mathcal{P}$ by Alg. 3. As in the proof of Thm. 24, for every evaluation $(\ell_0, \sigma_0) \to^k_{\mathcal{T}'} (\ell', \sigma)$ there is also a corresponding evaluation $(\ell_0, \sigma_0) \to^k_{\mathcal{T}} (\ell, \sigma)$, which is obtained by removing the labels from the locations.

For the other direction, we show that for each evaluation $(\ell_0, \sigma_0) \to_{t_1} (\ell_1, \sigma_1) \to_{t_2} \cdots \to_{t_k} (\ell_k, \sigma_k)$ with $t_1, \ldots, t_k \in \mathcal{T}$ there is a corresponding evaluation $(\ell_0, \sigma_0) \to^k_{\mathcal{T}'} (\ell'_k, \sigma_k)$ in $\mathcal{P}'$. To obtain this evaluation, we handle all evaluation fragments separately which use programs $\mathcal{Q}$ from $\mathcal{S}$. This is possible, since different programs in $\mathcal{S}$ do *not* share locations, i.e., entry and outgoing transitions of $\mathcal{Q}$ cannot be part of another $\mathcal{Q}'$ from $\mathcal{S}$. Such an evaluation fragment has the form

$$(\ell_i, \sigma_i) \to_{t_{i+1}} (\ell_{i+1}, \sigma_{i+1}) \to_{t_{i+2}} \cdots \to_{t_{n-1}} (\ell_{n-1}, \sigma_{n-1}) \to_{t_n} (\ell_n, \sigma_n) \qquad (9)$$

where t_{i+1} is an entry transition to $\mathcal{Q}$, t_n is an outgoing transition from $\mathcal{Q}$, and the transitions $t_{i+2}, \ldots, t_{n-1}$ belong to $\mathcal{Q}$. By Thm. 24 it follows that there is a corresponding evaluation using the transitions $t'_{i+2}, \ldots, t'_{n-1}$ from the refined version of $\mathcal{Q}$, such that with the new redirected entry transition t'_{i+1} and the new redirected outgoing transition t'_n we have

$$(\ell_i, \sigma_i) \to_{t'_{i+1}} (\ell'_{i+1}, \sigma_{i+1}) \to_{t'_{i+2}} \cdots \to_{t'_{n-1}} (\ell'_{n-1}, \sigma_{n-1}) \to_{t'_n} (\ell_n, \sigma_n) \qquad (10)$$

Thus, by substituting each evaluation fragment (9) in an evaluation of $\mathcal{P}$ by its refinement (10), we get a corresponding evaluation in $\mathcal{P}'$. □