Inferring Lower Runtime Bounds for Integer Programs

FLORIAN FROHN, Max Planck Institute for Informatics, Germany
MATTHIAS NAAF, RWTH Aachen University, Germany
MARC BROCKSCHMIDT, Microsoft Research, UK
JÜRGEN GIESL, LuFG Informatik 2, RWTH Aachen University, Germany

We present a technique to infer *lower* bounds on the worst-case runtime complexity of integer programs, where in contrast to earlier work, our approach is not restricted to tail-recursion. Our technique constructs symbolic representations of program executions using a framework for iterative, under-approximating program simplification. The core of this simplification is a method for (under-approximating) program acceleration based on recurrence solving and a variation of ranking functions. Afterwards, we deduce *asymptotic* lower bounds from the resulting simplified programs using a special-purpose calculus and an SMT encoding. We implemented our technique in our tool LoAT and show that it infers non-trivial lower bounds for a large class of examples.

CCS Concepts: • **Theory of computation → Complexity classes**; **Program analysis**; **Automated reasoning**; • **Software and its engineering → Software performance**.

Additional Key Words and Phrases: Integer Programs, Runtime Complexity, Lower Bounds, Automated Complexity Analysis

ACM Reference Format:

Florian Frohn, Matthias Naaf, Marc Brockschmidt, and Jürgen Giesl. 2020. Inferring Lower Runtime Bounds for Integer Programs. *ACM Trans. Program. Lang. Syst.* 1, 1 (July 2020), 51 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Recent advances in program analysis yield efficient methods to find *upper* bounds on the complexity of sequential integer programs. Here, one usually considers "worst-case complexity", i.e., for any variable valuation, one analyzes the length of the longest execution starting from that valuation. But in many cases, in addition to upper bounds, it is also important to find *lower* bounds for this notion of complexity. Together with an analysis for upper bounds, this can be used to infer *tight* complexity bounds. Lower bounds also have important applications in security analysis. If one can infer that there exists a family of inputs which lead to unacceptably large runtime

Authors' addresses: Florian Frohn, Max Planck Institute for Informatics, Saarland Informatics Campus, Campus E1 4, 66123 Saarbrücken, Germany, florian.frohn@mpi-inf.mpg.de; Matthias Naaf, RWTH Aachen University, Aachen, Germany, naaf@logic.rwth-aachen.de; Marc Brockschmidt, Microsoft Research, Cambridge, UK, mabrocks@microsoft.com; Jürgen Giesl, LuFG Informatik 2, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany, giesl@informatik.rwth-aachen.de.

0164-0925/2020/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of the program (i.e., a family for which there is a non-linear or probably even exponential lower bound on the runtime), then this family of inputs represents a possible denial-of-service attack. Thus, techniques for the computation of lower bounds on the worst-case complexity can be used to detect such attacks.[1]

While worst-case lower bounds are useful to *detect* attacks or performance bugs, *worst-case upper bounds* can *prove the absence* of such problems. *Best-case* lower bounds, which have also been investigated in the literature (see Section 7), are bounds on all program runs, whereas worst-case lower bounds hold for (usually infinite) families of (expensive) program runs. Thus, best-case lower bounds can, e.g., be used to decide whether a certain task is expensive enough to compensate the overhead of executing it remotely. So in general, the use cases of worst-case and best-case bounds are orthogonal.

We introduce the first technique to infer worst-case lower bounds for integer programs automatically. Besides *concrete* bounds, our technique can also deduce *asymptotic* bounds. In general, concrete lower bounds that hold for arbitrary variable valuations are difficult to express concisely. In contrast, asymptotic bounds are easily understood by humans and witness possible attacks in a convenient way.

We first introduce our program model in Section 2. Afterwards, Section 3 shows how to transform arbitrary tail-recursive integer programs into so-called *simplified programs* without loops. To this end, in Section 3.1 we introduce a variation of classical ranking functions which we call *metering functions*. These metering functions are used to under-estimate the number of iterations of simple loops, i.e., loops consisting of a single transition without nested loops or branching. Based on this concept, we present a framework to simplify programs iteratively in Sections 3.2 and 3.3. It transforms tail-recursive programs (with branching and sequences of possibly nested loops) into programs without loops. To this end, it *accelerates* and then eliminates simple loops by (under-)approximating their effect using a combination of metering functions and recurrence solving. In Section 4 we extend our technique to an automatic approach which also transforms non-tail-recursive integer programs into simplified programs.

Section 5 introduces techniques which allow us to infer *asymptotic* lower bounds from simplified programs. In Sections 5.1 and 5.2 we present a calculus to compute asymptotic bounds by repeatedly simplifying a *limit problem*, which is an abstraction of the path condition φ of a simplified program. This abstraction allows us to focus on φ's behavior in the limit, i.e., we search for an infinite family of inputs that satisfy φ. In addition, Section 5.3 shows how a limit problem can be encoded into a quantifier-free first-order formula with integer arithmetic. Then off-the-shelf SMT solvers can be used to find a model for this formula. This model in turn gives rise to the desired family of inputs that satisfy φ. Thus, in many cases we can benefit from the power of SMT solvers instead of applying the rules of our calculus from Section 5.2 heuristically. Note that the calculus from Section 5.2 can simplify limit problems such that our SMT encoding from Section 5.3 becomes applicable and on the other hand, our SMT encoding can be integrated into the calculus from Section 5.2, i.e., both techniques complement each other.

Finally, we evaluate our implementation in the tool LoAT in Section 6, discuss related work in Section 7, and conclude in Section 8.

A preliminary version of this paper was published in [29]. The current paper extends [29] significantly with the following novel contributions:

[1] In a joint project *CAGE* [23, 28] with Draper Inc. (https://www.draper.com) and the University of Innsbruck, we used our tool LoAT (that implements the techniques described in this paper) together with our tool KoAT [14] (that infers upper runtime bounds) to analyze the complexity of large Java programs in order to detect vulnerabilities.

(1) The new Theorem 3.4 integrates an optimization that we proposed in [29] into our notion of metering functions. The novel insight is that in this way we can infer more expressive "conditional" metering functions of the form $[\![\psi]\!] \cdot b$ where b is an ordinary arithmetic expression and $[\![\psi]\!]$ is the characteristic function of the arithmetic condition ψ (i.e., $[\![\psi]\!]$ yields 1 if ψ is satisfied and 0, otherwise). Such metering functions are also useful to treat terminating and non-terminating rules in a uniform way in our approach. To ease the use of such metering functions, in Theorem 3.8 we extend the technique for accelerating loops from [29, Theorem 7] accordingly. Thus, conditional metering functions are now seamlessly integrated into our framework, resulting in a more streamlined formalization and presentation than in [29].
(2) In Theorem 3.12, we present a technique to eliminate variables from the program. In order to apply it automatically, the new Lemma 3.15 clarifies how to check a crucial side condition which requires that certain arithmetic expressions evaluate to integers whenever one instantiates their variables with integers. A similar side condition is also needed for the automation of our calculus for limit problems in Section 5. Moreover, this check could also be used to ensure that the initial program is "well formed" before starting the analysis. In [29], the automation of this check has not been discussed.
(3) We lift our approach to non-tail-recursive programs in the new Section 4. In contrast, [29] was restricted to tail-recursive integer programs. While [29] used a graph-based program model, in the current paper we propose a different (rule-based) representation of integer programs. This representation allows an easy formulation of non-tail-recursion, by using rules whose right-hand sides are multisets of terms.
(4) The SMT encoding from Section 5.3 is completely new and improves the performance of our approach considerably. In particular, in our experiments of Section 6 it outperformed the calculus from Section 5.2 on examples with polynomial limit problems.
(5) We provide formal proofs for all lemmas and theorems, which were missing in [29].
(6) Throughout the paper, we added many more examples, discussions, and explanations.

2 PROGRAM MODEL

We consider sequential imperative integer programs, allowing non-linear arithmetic and non-determinism, whereas heap usage and concurrency are not supported. In [14] we used an equivalent program model, and showed how to deduce *upper* runtime bounds for integer programs.

Most existing abstractions that transform heap programs to integer programs are "over-approximations". However, in order to apply our approach to heap programs, we would need an under-approximating abstraction to ensure that the inference of worst-case *lower* bounds is sound. As in most related work, we treat numbers as mathematical integers $\mathbb{Z}$. However, one can use suitable transformations [24, 42] to handle machine integers correctly, e.g., by inserting explicit normalization steps at possible overflows.

In our program model, we use a rule-based representation of integer programs where *integer program rules* are of the form $f(\overline{x}) \xrightarrow{c} T \ [\varphi]$. The *left-hand side* $f(\overline{x})$ consists of a *function symbol* f and a vector of pairwise different variables $\overline{x}$. The sets of all function symbols and all variables are Σ and $\mathcal{V}$, respectively. While Σ is finite, we assume $\mathcal{V}$ to be countably infinite, as we rely on fresh variables to model non-determinism. The *arithmetic expression* c represents the *cost* of the rule, where arithmetic expressions are composed of variables from $\mathcal{V}$, numbers, and pre-defined operations like $+$, $-$, $*$, etc. Annotating rules with costs enables a modular analysis, as it allows us to summarize a sub-program $\mathcal{P}$ into a single rule whose cost is a lower bound on $\mathcal{P}$'s complexity. To ease readability, we sometimes omit the costs of rules. The *guard* φ is a *constraint* over $\mathcal{V}$, i.e.,

a finite conjunction[2] of inequations (built with $<$, $\leq$, $>$, or $\geq$) over arithmetic expressions, which we omit if it is empty (i.e., we write $f(\overline{x}) \xrightarrow{c} T$ instead of $f(\overline{x}) \xrightarrow{c} T$ [true]). So c is an expression like $x \cdot y + 2^y$ and φ is a formula like $x \cdot y \leq 2^y \wedge y > 0$, for example. The *right-hand side* T is a multiset of *terms* of the form $g(\overline{t})$ where $g \in \Sigma$ and $\overline{t}$ is a vector of arithmetic expressions. In the following, the notion of "term" always refers to terms of this specific form. The set of all terms is $\mathcal{T}$. We use $\mathcal{V}(\cdot)$ to denote all variables occurring in the argument expression (e.g., $\mathcal{V}(t)$ consists of all variables occurring in the term t).

Note that as in [14], we do not allow nested calls of function symbols from Σ in right-hand sides of rules. For that reason, our program model also does not support return values. Instead of a rule $f(\overline{x}) \xrightarrow{c} f(g(\overline{t}))$ $[\varphi]$ with $f, g \in \Sigma$, one has to represent the result of the inner call $g(\overline{t})$ by a fresh *temporary variable* tv. So one uses a rule $f(\overline{x}) \xrightarrow{c} \{g(\overline{t}), f(tv)\}$ $[\varphi \wedge \psi]$ instead, where ψ may restrict the possible values of tv by suitable inequations.

We say that a function symbol f has an *incoming* rule α if f occurs in α's right-hand side and f has an *outgoing* rule α if f occurs on (the root position of) α's left-hand side. Given a rule α, $\mathrm{root}(\alpha)$ denotes the root symbol of α's left-hand side $\mathrm{lhs}(\alpha)$. Furthermore, $\mathrm{cost}(\alpha)$, $\mathrm{rhs}(\alpha)$, and $\mathrm{guard}(\alpha)$ denote the cost, the right-hand side, and the guard of α. The number of elements of the multiset in α's right-hand side is called the *degree* of α. A rule is *tail-recursive* if its degree is at most 1.[3] For a rule α with degree 1, let $\mathrm{target}(\alpha)$ be the root symbol of the term in $\mathrm{rhs}(\alpha)$.

An *integer program* is a finite set of integer program rules. It is tail-recursive if all of its rules are tail-recursive.

Example 2.1 (Fibonacci). Consider the following imperative program, which computes the x-th Fibonacci number and returns 1 if x is negative.

```
int fib(int x) {
  if (x <= 1) return 1;
  else return fib(x - 1) + fib(x - 2);
}
```

A suitable abstraction of this program would yield the following integer program which represents its recursion pattern. For simplicity, we sometimes write t instead of $\{t\}$ for singleton (multi)sets of terms, i.e., in the first rule we write $\mathrm{f}_0(x) \xrightarrow{0} \mathrm{fib}(x)$ instead of $\mathrm{f}_0(x) \xrightarrow{0} \{\mathrm{fib}(x)\}$. Here, $\mathrm{f}_0 \in \Sigma$ is the *canonical start symbol*, i.e., the entry point of the program.

$$\mathrm{f}_0(x) \xrightarrow{0} \mathrm{fib}(x) \tag{1}$$

$$\mathrm{fib}(x) \xrightarrow{1} \{\mathrm{fib}(x-1), \mathrm{fib}(x-2)\} \quad [x > 1] \tag{2}$$

$$\mathrm{fib}(x) \xrightarrow{1} \emptyset \quad [x \leq 1] \tag{3}$$

As the right-hand side of Rule (2) consists of two terms, it is not tail-recursive. Note that the result of the fib program is not represented in the abstraction.

Using multisets as right-hand sides allows us to express that a function f calls several other functions $f_1, \ldots, f_n$. Note that a rule of the form $f(\ldots) \to \{f_1(\ldots), \ldots, f_n(\ldots)\}$ $[\varphi]$ is not equivalent to the n rules $f(\ldots) \to f_i(\ldots)$ $[\varphi]$ with $1 \leq i \leq n$: While the former rule expresses that f invokes *all* functions $f_1, \ldots, f_n$, the latter rules mean that f non-deterministically invokes *some*

[2]Note that negations can be expressed by negating inequations directly, and disjunctions in programs can be expressed using several rules. We write "$s = t$" as syntactic sugar for "$s \geq t \wedge s \leq t$".

[3]Note that rules with right-hand side $\emptyset$ can equivalently be transformed to rules with right-hand side "sink" where sink is a fresh function symbol of arity 0. Thus w.l.o.g., for tail-recursive programs we assume that all rules have degree 1.

function f_i. Thus, the recursive fib-rule (2) cannot be replaced by the rules

$$\mathsf{fib}(x) \xrightarrow{1} \mathsf{fib}(x-1)\ [x > 1] \quad \text{and} \tag{4}$$

$$\mathsf{fib}(x) \xrightarrow{1} \mathsf{fib}(x-2)\ [x > 1]\,. \tag{5}$$

These rules would mean that $\mathsf{fib}(x)$ *either* evaluates to $\mathsf{fib}(x-1)$ *or* to $\mathsf{fib}(x-2)$. In contrast, the recursive rule (2) expresses that $\mathsf{fib}(x)$ evaluates to *both* $\mathsf{fib}(x-1)$ and $\mathsf{fib}(x-2)$. Thus, the integer program $\{(1), (2), (3)\}$ has exponential complexity, but replacing its recursive rule (2) with (4) and (5) would result in a program with only linear complexity. So the recursion pattern of a non-tail-recursive program like fib cannot be modeled with rules whose right-hand sides are singleton sets.

Note that our notion of tail-recursion is a special case of the standard notion, where a procedure is considered to be tail-recursive if recursive calls are only performed as its last action. The reason is that in our program model the right-hand side of a rule is considered as a multiset and thus, there is no order imposed on the evaluation of its elements. Thus, the non-tail-recursive rule

$$\mathsf{f}(\ldots) \to \{\mathsf{f}(\ldots), \mathsf{g}(\ldots)\}\ \ [\varphi]$$

could correspond to either of the following procedures, where the first one is tail-recursive, but the second one is not.

```
f ( . . . )  {
   g ( . . . ) ;
   f ( . . . ) ;
}
```

```
f ( . . . )  {
   f ( . . . ) ;
   g ( . . . ) ;
}
```

On the other hand, programs which are tail-recursive w.r.t. our notion of tail-recursion are clearly also tail-recursive in the usual sense.

The guards of rules restrict the control flow of the program, i.e., a rule $f(\overline{x}) \xrightarrow{c} T\ [\varphi]$ can only be applied if the current valuation of the variables is a model of φ. As we are concerned with integer programs, all variables range over $\mathbb{Z}$, such that we only consider models of φ which map variables to integers.

Definition 2.2 (Substitutions). A substitution σ instantiates variables by arithmetic expressions. We sometimes denote substitutions σ by finite sets of key-value pairs $\{y_1/t_1, \ldots, y_k/t_k\}$ or $\{\overline{y}/\overline{t}\}$ for short, where $\overline{y}$ is the vector $(y_1, \ldots, y_k)$ and $\overline{t}$ is the vector $(t_1, \ldots, t_k)$. This substitution instantiates every variable $y_i \in \mathcal{V}$ by the arithmetic expression t_i. Furthermore, it maps each[4] $x \in \mathcal{V} \setminus \overline{y}$ to x, i.e., the *domain* of σ is $\mathrm{dom}(\sigma) = \{y_i \mid 1 \le i \le k, y_i \neq t_i\}$ and its *range* is $\{x\sigma \mid x \in \mathrm{dom}(\sigma)\}$. Substitutions are homomorphically extended to terms (i.e., $\sigma(t)$ instantiates all variables x in the term t by $\sigma(x)$) and we usually write $t\sigma$ instead of $\sigma(t)$. For two substitutions θ and σ, their *composition* is denoted by $\theta \circ \sigma$ where $t\,(\theta \circ \sigma) = t\theta\sigma$ (i.e., θ is applied first).

An *integer substitution* is a substitution that maps every variable $x \in \mathrm{dom}(\sigma)$ to an integer number. We write $\sigma \models \varphi$ for an integer substitution σ if $\mathcal{V}(\varphi) \subseteq \mathrm{dom}(\sigma)$ and σ is a model of φ.

We always assume that Σ contains the canonical start symbol f_0 and we are only interested in program runs that start with terms of the form $\mathsf{f}_0(\overline{n})$ where $\overline{n} \subset \mathbb{Z}$. Note that this is not a restriction, as we can simulate several start symbols $\mathsf{f}_1, \ldots, \mathsf{f}_k$ by adding corresponding rules from

[4]Slightly abusing notation, we sometimes use vectors as sets. So for a vector $\overline{y} = (y_1, \ldots, y_k)$, $\mathcal{V} \setminus \overline{y}$ denotes $\mathcal{V} \setminus \{y_1, \ldots, y_k\}$.

```
f0: y = 0;
f1: while (x > 0) {
      y = y + x;
      x = x - 1;
    }
    z = y;
f2: while (z > 0) {
      u = z - 1;
f3:   while (u > 0) {
        u = u - random(0, ω);
      }
      z = z - 1;
    }
```

(a) Example Integer Program

$$\alpha_0 : \mathrm{f}_0(x,y,z,u) \xrightarrow{1} \mathrm{f}_1(x,0,z,u)$$
$$\alpha_1 : \mathrm{f}_1(x,y,z,u) \xrightarrow{1} \mathrm{f}_1(x-1,y+x,z,u) \quad [x > 0]$$
$$\alpha_2 : \mathrm{f}_1(x,y,z,u) \xrightarrow{1} \mathrm{f}_2(x,y,y,u) \quad [x \leq 0]$$
$$\alpha_3 : \mathrm{f}_2(x,y,z,u) \xrightarrow{1} \mathrm{f}_3(x,y,z,z-1) \quad [z > 0]$$
$$\alpha_4 : \mathrm{f}_3(x,y,z,u) \xrightarrow{1} \mathrm{f}_3(x,y,z,u-tv) \quad [u > 0 \wedge tv > 0]$$
$$\alpha_5 : \mathrm{f}_3(x,y,z,u) \xrightarrow{1} \mathrm{f}_2(x,y,z-1,u) \quad [u \leq 0]$$

(b) Example Integer Program – Rule Representation

Fig. 1. Different Representations of Integer Programs

f_0 to $\mathrm{f}_1, \ldots, \mathrm{f}_k$. W.l.o.g., we assume that f_0 does not occur on right-hand sides of rules. Otherwise, one could rename f_0 to f'_0 and add a rule $\mathrm{f}_0(\overline{x}) \xrightarrow{0} \mathrm{f}'_0(\overline{x})$.

Figure 1b shows an example of a tail-recursive integer program, i.e., here every right-hand side just consists of a single term. Figure 1b corresponds to the pseudo-code in Figure 1a, where random(x, y) returns a random integer tv with $x < tv < y$ and ω is the smallest infinite ordinal, i.e., we have $-\omega < n < \omega$ for all numbers $n \in \mathbb{Z}$. The following definition clarifies how to evaluate integer programs.

Definition 2.3 (Integer Transition Relation). A *configuration* is a multiset of terms of the form $f(\overline{n})$ where $f \in \Sigma$ and $\overline{n} \subset \mathbb{Z}$. The set of all configurations is denoted by $\mathcal{C}$.

Let $\mathcal{P}$ be an integer program. For configurations $S, T \in \mathcal{C}$ and $k \in \mathbb{R}$, S *evaluates to* T *with cost* k ($S \xrightarrow{k}_{\mathcal{P}} T$) if there is an $s \in S$, a rule α of the form $f(\overline{x}) \xrightarrow{c} Q\ [\varphi] \in \mathcal{P}$, and an integer substitution σ with $\mathcal{V}(\alpha) \subseteq \mathrm{dom}(\sigma)$ such that $f(\overline{x})\sigma = s$,[5] $T = (S \setminus \{s\}) \cup Q\sigma$,[6] $\sigma \models \varphi$, and $c\sigma = k$.

For any integer program rule α and any integer substitution σ, let $\sigma \models \alpha$ denote that $\mathcal{V}(\alpha) \subseteq \mathrm{dom}(\sigma)$ and $\sigma \models \mathrm{guard}(\alpha)$.

We write $S \xrightarrow{k}_{\alpha} T$ instead of $S \xrightarrow{k}_{\mathcal{P}} T$ if $\mathcal{P}$ is the singleton set containing α. Moreover,

$$S_0 \xrightarrow{k,m}_{\mathcal{P}} S_m \quad \text{denotes that} \quad S_0 \xrightarrow{k_1}_{\mathcal{P}} \ldots \xrightarrow{k_m}_{\mathcal{P}} S_m \quad \text{for} \quad k = \sum_{i=1}^{m} k_i.$$

If m is irrelevant, we write $S_0 \xrightarrow{k,*}_{\mathcal{P}} S_m$ if $m \geq 0$ and $S_0 \xrightarrow{k,+}_{\mathcal{P}} S_m$ if $m > 0$. Finally, we sometimes omit the costs (of both rules and evaluations) if they are not important.

We say that a program is *simplified* if $\mathrm{root}(\alpha) = \mathrm{f}_0$ for all rules α, i.e., if all left-hand sides of rules are constructed with the canonical start symbol f_0 which does not occur on right-hand sides. So any run of a simplified program that starts with a term of the form $\mathrm{f}_0(\overline{n})$ has at most length 1.

[5] Throughout this paper, "=" means equality modulo arithmetic and we assume that ground arithmetic expressions and comparisons are evaluated exhaustively, so we have, e.g., $f(2) = f(3-1)$ and $3 - 1 \in \mathbb{Z}$.

[6] As usual, $S \setminus \{s\}$ means that the number of occurrences of s in the multiset S (if any) is reduced by 1. We lift substitutions to (multi)sets of terms in the obvious way, i.e., $Q\sigma = \{q\sigma \mid q \in Q\}$.

Example 2.4 (Evaluation of Integer Programs). Using the rules from Figure 1b, we have, e.g.,

$$\mathsf{f}_0(3,2,1,0) \xrightarrow{1}_{\alpha_0} \mathsf{f}_1(3,0,1,0) \xrightarrow{1}_{\alpha_1} \mathsf{f}_1(2,3,1,0) \xrightarrow{1}_{\alpha_1} \mathsf{f}_1(1,5,1,0) \xrightarrow{1}_{\alpha_1} \dots$$

For the rules $\mathcal{P}_{\mathsf{fib}} = \{(1), (2), (3)\}$ from Example 2.1, we have

$$\mathsf{f}_0(3) \xrightarrow{0}_{\mathcal{P}_{\mathsf{fib}}} \mathsf{fib}(3) \xrightarrow{1}_{\mathcal{P}_{\mathsf{fib}}} \{\mathsf{fib}(2), \mathsf{fib}(1)\} \xrightarrow{1}_{\mathcal{P}_{\mathsf{fib}}} \{\mathsf{fib}(1), \mathsf{fib}(0), \mathsf{fib}(1)\} \xrightarrow{3}{}^{3}_{\mathcal{P}_{\mathsf{fib}}} \emptyset,$$

where $\{\mathsf{fib}(1), \mathsf{fib}(0), \mathsf{fib}(1)\} \xrightarrow{3}{}^{3}_{\mathcal{P}_{\mathsf{fib}}} \emptyset$ abbreviates the three steps

$$\{\mathsf{fib}(1), \mathsf{fib}(0), \mathsf{fib}(1)\} \xrightarrow{1}_{\mathcal{P}_{\mathsf{fib}}} \{\mathsf{fib}(0), \mathsf{fib}(1)\} \xrightarrow{1}_{\mathcal{P}_{\mathsf{fib}}} \{\mathsf{fib}(1)\} \xrightarrow{1}_{\mathcal{P}_{\mathsf{fib}}} \emptyset,$$

whose combined cost is $1 + 1 + 1 = 3$.

According to our definition, integer programs may also contain rules like $\mathsf{f}(x) \to \mathsf{f}(\frac{x}{2})$. While evaluations cannot yield non-integer values (e.g., we cannot evaluate $\mathsf{f}(1)$ to $\mathsf{f}(\frac{1}{2})$, as $\mathsf{f}(\frac{1}{2})$ is not a configuration), our technique assumes that arithmetic expressions on right-hand sides of rules always map integers to integers. Hence, throughout this paper we restrict ourselves to *well-formed* integer programs.

Definition 2.5 (Well-Formed Integer Program). An integer program rule α is *well formed* if $t_i\sigma \in \mathbb{Z}$ for each $f(t_1, \dots, t_k) \in \mathrm{rhs}(\alpha)$, for each $1 \le i \le k$, and for each integer substitution σ with $\sigma \models \alpha$. An integer program is well formed if each of its rules is well formed.

Note that for a well-formed rule α we do not require $\mathrm{cost}(\alpha)\sigma \in \mathbb{Z}$ for integer substitutions σ with $\sigma \models \alpha$.

To ensure that the analyzed program $\mathcal{P}_0$ is initially well formed, we just allow integer numbers, addition, subtraction, and multiplication in $\mathcal{P}_0$.[7] Our approach relies on several program transformations, i.e., the initial program $\mathcal{P}_0$ is transformed into other programs $\mathcal{P}_1, \mathcal{P}_2, \dots$ which may contain further operations like division and exponentiation. However, all our transformations preserve well-formedness.

To simplify the presentation, throughout this paper, we assume that all function symbols in Σ have the same arity. Otherwise, one can construct a variant of $\mathcal{P}$ where additional unused arguments are added to each function symbol whose arity is not maximal. Moreover, we assume that the left-hand sides of $\mathcal{P}$ only differ in their root symbols, i.e., the argument lists are equal (e.g., in Figure 1b, the variables on the left-hand sides are consistently named x, y, z, u). Otherwise, one can rename variables accordingly without affecting the relation $\to_{\mathcal{P}}$. The variables $\overline{x}$ on the left-hand sides are called *program variables* and for any rule α, all other variables $\mathcal{TV}(\alpha) = \mathcal{V}(\alpha) \setminus \overline{x}$ are called *temporary*. These temporary variables are used to model non-deterministic program data. So in Figure 1b, we have $\overline{x} = (x, y, z, u)$ and $tv \in \mathcal{TV}(\alpha_4)$.

In Figure 1, the loop at f_1 computes a value for y that is quadratic in the original value of x. Thus, the loop at f_2 is executed quadratically often where in each iteration, the inner loop at f_3 may also be repeated quadratically often. Thus, the program's (*worst-case*) runtime is a polynomial of degree 4 in x. In contrast, the *best-case* runtime of the program is only quadratic in the original value of x, because then the inner loop at f_3 would always set u to a non-positive value immediately. The goal of our paper is to infer lower bounds for worst-case runtimes automatically.

To formalize the (worst-case) runtime complexity of an integer program, we define the *derivation height* of a configuration S to be the cost of the most expensive evaluation starting with S. Here,

[7] One could also allow expressions with non-integer numbers like $\frac{1}{2}x^2 + \frac{1}{2}x$ in the initial program, as long as every arithmetic expression in the program evaluates to an integer when instantiating its variables by integers. We will present a criterion to detect such expressions in Lemma 3.15.

for any non-empty set $M \subseteq \mathbb{R} \cup \{\omega\}$, $\sup M$ is the least upper bound of M. In the following, let $\mathbb{R}_{\geq 0} = \{k \in \mathbb{R} \mid k \geq 0\}$.

Definition 2.6 (Derivation Height [44]). Let $\mathcal{P}$ be an integer program. Its *derivation height* function $\mathrm{dh}_\mathcal{P} : C \to \mathbb{R}_{\geq 0} \cup \{\omega\}$ is defined as $\mathrm{dh}_\mathcal{P}(S) = \sup\{k \in \mathbb{R} \mid S \xrightarrow{k}{}^*_\mathcal{P} T \text{ for some } T \in C\}$.

Clearly, we always have $\mathrm{dh}_\mathcal{P}(S) \geq 0$, since $\xrightarrow{k}{}^*_\mathcal{P}$ also permits evaluations with 0 steps. For the integer program $\mathcal{P}$ in Figure 1b, we obtain $\mathrm{dh}_\mathcal{P}(\mathrm{f}_0(0, y, z, u)) = 2$ for all $y, z, u \in \mathbb{Z}$, since then we can only apply the transitions α_0 and α_2 once. For all terms $\mathrm{f}_0(x, y, z, u)$ with $x > 1$, α_0 is executed once, then α_1 is executed x times. Afterwards, y has the value $\frac{(x+1)\cdot x}{2}$. Now α_2 is executed once and sets z to the value $\frac{(x+1)\cdot x}{2}$. The outer loop at f_2 is executed z times, where in each iteration, the inner loop at f_3 is executed $z-1$ time (in the worst case) and z is decreased by 1 in α_5. So overall, α_3 and α_5 are both executed z times and α_4 is executed $(z-1)+(z-2)+\ldots+1 = \frac{z\cdot(z-1)}{2}$ times. Hence, the worst-case runtime is $1+x+1+z+\frac{z\cdot(z-1)}{2}+z$, where $z = \frac{(x+1)\cdot x}{2}$, i.e., $\mathrm{dh}_\mathcal{P}(\mathrm{f}_0(x, y, z, u)) = \frac{1}{8}x^4+\frac{1}{4}x^3+\frac{7}{8}x^2+\frac{7}{4}x+2$. Our method will detect that the derivation height of $\mathrm{f}_0(x, y, z, u)$ is at least $\frac{1}{8}x^4 + \frac{1}{4}x^3 + \frac{7}{8}x^2 + \frac{7}{4}x$. From this concrete lower bound, our approach will infer that the asymptotic runtime complexity of the program is in $\Omega(n^4)$ where n is the size of the input, i.e., $n = |x| + |y| + |z| + |u|$. So the *size* of the input is measured by the sum of the absolute values of all program variables.

Note that the derivation height can be unbounded even if the integer program terminates.

Example 2.7 (Unbounded Costs without Non-Termination). Consider the integer program $\mathcal{P}$ with the rule $\mathrm{f}_0 \xrightarrow{tv} \mathrm{f}$ where tv is a temporary variable of the rule. By Definition 2.3, *every* integer substitution σ with $tv \in \mathrm{dom}(\sigma)$ can be used to evaluate f_0 to f. Thus, for any $n \in \mathbb{N}$ we have $\mathrm{f}_0 \xrightarrow{n} \mathrm{f}$ using an integer substitution σ_n with $tv\,\sigma_n = n$. Therefore we obtain $\mathrm{dh}_\mathcal{P}(\mathrm{f}_0) = \omega$ although the rule has no recursive call.

Similarly, for the program $\mathcal{P}'$ with the rules $\mathrm{f}_0(x) \xrightarrow{0} \mathrm{f}(tv)$ and $\mathrm{f}(x) \xrightarrow{1} \mathrm{f}(x-1)\,[x > 0]$, we also have $\mathrm{dh}_{\mathcal{P}'}(\mathrm{f}_0(1)) = \omega$, although every evaluation of the program is finite.

While $\mathrm{dh}_\mathcal{P}$ is defined on configurations, the complexity of a program is often defined as a function on $\mathbb{N}$, in particular when considering asymptotic complexity bounds. To bridge this gap, we use the common definition of complexity as a function of the size of the input. So the *runtime complexity* function $\mathrm{rc}_\mathcal{P}(n)$ maps a natural number n to the cost of the most expensive program run where the size of the input is bounded by n.

Definition 2.8 (Runtime Complexity). Let $\mathcal{P}$ be an integer program and let k be the arity of f_0. The *runtime complexity* function $\mathrm{rc}_\mathcal{P} : \mathbb{N} \to \mathbb{R}_{\geq 0} \cup \{\omega\}$ of $\mathcal{P}$ is defined as

$$\mathrm{rc}_\mathcal{P}(n) = \sup\{\mathrm{dh}_\mathcal{P}(\mathrm{f}_0(\overline{n})) \mid \overline{n} \in \mathbb{Z}^k, |\overline{n}| \leq n\},$$

where for the vector $\overline{n} = (n_1, \ldots, n_k)$, we have $|\overline{n}| = \sum_{i=1}^{k} |n_i|$.

For the program $\mathcal{P}$ from Figure 1b, recall that the derivation height $\mathrm{dh}_\mathcal{P}(\mathrm{f}_0(x, y, z, u)) = \frac{1}{8}x^4 + \frac{1}{4}x^3+\frac{7}{8}x^2+\frac{7}{4}x+2$ solely depends on the value of the first argument x of f_0. As n in Definition 2.8 is a bound on the sum of the absolute values of all arguments (i.e., $|x|+|y|+|z|+|u| \leq n$), setting $x = n$ and $y = z = u = 0$ maximizes $\mathrm{dh}_\mathcal{P}(\mathrm{f}_0(x, y, z, u))$. Hence, we have $\mathrm{rc}_\mathcal{P}(n) = \mathrm{dh}_\mathcal{P}(\mathrm{f}_0(n, 0, 0, 0)) = \frac{1}{8}n^4 + \frac{1}{4}n^3 + \frac{7}{8}n^2 + \frac{7}{4}n + 2$.

Obviously, $\mathrm{dh}_\mathcal{P}$ and $\mathrm{rc}_\mathcal{P}$ are not computable in general. Thus, our goal is to find a lower bound on the runtime complexity of a program $\mathcal{P}$ automatically which is as precise as possible (i.e., a lower bound which is, e.g., unbounded, exponential, or a polynomial of a degree as high as possible). So for the program in Figure 1b, we would like to derive $\mathrm{rc}_\mathcal{P}(n) \in \Omega(n^4)$, i.e., that the runtime

complexity is asymptotically bounded from below by n^4. As usual, $f(n) \in \Omega(g(n))$ means that there is an $m > 0$ and an $n_0 \in \mathbb{N}$ such that $f(n) \geq m \cdot g(n)$ holds for all $n \geq n_0$. In our example, we also have $\mathrm{rc}_{\mathcal{P}}(n) \in \mathcal{O}(n^4)$, i.e., n^4 is both an asymptotic *lower* and *upper* bound on the (worst-case) runtime complexity.

Note that according to Definition 2.8, $\mathrm{rc}_{\mathcal{P}}(n)$ takes all runs into account that start with $\mathrm{f}_0(\overline{n})$ where the size $|\overline{n}|$ is *n or smaller* (i.e., in the definition of $\mathrm{rc}_{\mathcal{P}}(n)$ we use "$|\overline{n}| \leq n$" instead of "$|\overline{n}| = n$"). This corresponds to the notion of "runtime complexity" used e.g., for complexity analysis of term rewriting [43] or for complexity analysis of integer transition systems in the *International Termination and Complexity Competition* [37]. To see the difference between $\mathrm{rc}_{\mathcal{P}}$ and the alternative definition

$$\mathrm{rc}'_{\mathcal{P}}(n) = \sup\{\mathrm{dh}_{\mathcal{P}}(\mathrm{f}_0(\overline{n})) \mid \overline{n} \in \mathbb{Z}^k, |\overline{n}| = n\},$$

consider a program $\mathcal{P}$ with the rule $\mathrm{f}_0(x) \xrightarrow{x} \mathrm{f}(x)\ [x \geq 0 \wedge x = 2 \cdot tv]$. For non-negative numbers n we have $\mathrm{rc}'_{\mathcal{P}}(n) = \mathrm{dh}_{\mathcal{P}}(\mathrm{f}_0(n)) = n$ and $\mathrm{rc}_{\mathcal{P}}(n) = n$ if n is even, but $\mathrm{rc}'_{\mathcal{P}}(n) = \mathrm{dh}_{\mathcal{P}}(\mathrm{f}_0(n)) = 0$ and $\mathrm{rc}_{\mathcal{P}}(n) = \mathrm{dh}_{\mathcal{P}}(\mathrm{f}_0(n-1)) = n-1$ if n is odd. As long as one is only interested in (asymptotic) upper bounds, the difference between $\mathrm{rc}_{\mathcal{P}}$ and $\mathrm{rc}'_{\mathcal{P}}$ is negligible, since we have both $\mathrm{rc}_{\mathcal{P}}(n) \in \mathcal{O}(n)$ and $\mathrm{rc}'_{\mathcal{P}}(n) \in \mathcal{O}(n)$. (More precisely, for any program $\mathcal{P}$ we have $\mathrm{rc}_{\mathcal{P}}(n) \in \mathcal{O}(g(n))$ iff $\mathrm{rc}'_{\mathcal{P}}(n) \in \mathcal{O}(g(n))$ if g is weakly monotonically increasing for large enough n.) But for (asymptotic) lower bounds, $\mathrm{rc}_{\mathcal{P}}$ and $\mathrm{rc}'_{\mathcal{P}}$ differ. For our example program we have $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n)$, but $\mathrm{rc}'_{\mathcal{P}}(n) \notin \Omega(n)$ (we only have $\mathrm{rc}'_{\mathcal{P}}(n) \in \Omega(1)$). Recall that two of our main motivations for the inference of worst-case lower bounds are

(A) to deduce *tight* bounds in combination with existing techniques for the inference of worst-case upper bounds and
(B) to find denial-of-service vulnerabilities (or, more generally, performance bugs),

see Section 1. The example above shows that in order to achieve (A), one should use our definition of $\mathrm{rc}_{\mathcal{P}}$ instead of $\mathrm{rc}'_{\mathcal{P}}$.[8] Regarding (B), note that $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(g(n))$ means that for large enough n one can always find inputs whose size is not greater than n, which lead to a runtime of at least $m \cdot g(n)$. Hence, our notion $\mathrm{rc}_{\mathcal{P}}(n)$ can indeed be used to find families of program inputs that lead to runtimes of at least length $m \cdot g(n)$, *for all large enough n*. So if g is unacceptably large (e.g., exponential or a high-degree polynomial), then such a family of program inputs witnesses a performance bug (which might, e.g., be exploited for denial-of-service attacks).

3 SIMPLIFYING TAIL-RECURSIVE INTEGER PROGRAMS

We now show how to transform any tail-recursive integer program $\mathcal{P}$ into a simplified program $\mathcal{P}'$ such that the runtime complexity of $\mathcal{P}'$ is smaller or equal to the runtime complexity of $\mathcal{P}$. Thus, any lower bound for $\mathrm{rc}_{\mathcal{P}'}$ is also a lower bound for $\mathrm{rc}_{\mathcal{P}}$. In Section 4 we will extend our transformation to non-tail-recursive integer programs, before inferring asymptotic lower bounds for the runtime complexity of simplified programs in Section 5.

[8] Nevertheless, almost all of our techniques would also work in order to infer a lower bound on $\mathrm{rc}'_{\mathcal{P}}$ instead of $\mathrm{rc}_{\mathcal{P}}$. The only problem is in Section 5 where we search for an infinite family of inputs that satisfy the guard of the program. Here, it is not required that this family can represent inputs of size n for *all* large enough n. So in our example, the technique of Section 5 would infer $\mathrm{rc}_{\mathcal{P}}(2 \cdot n) \in \Omega(n)$ (and indeed, we also have $\mathrm{rc}'_{\mathcal{P}}(2 \cdot n) \in \Omega(n)$), but the family of inputs "$2 \cdot n$" for all $n \geq 0$ does not represent *all* possible large enough numbers. For weakly monotonically increasing functions like $\mathrm{rc}_{\mathcal{P}}$, we present a technique in Section 5 (viz. Lemma 5.8) to transform a lower bound on $\mathrm{rc}_{\mathcal{P}}(2 \cdot n)$ into a lower bound on $\mathrm{rc}_{\mathcal{P}}(n)$, i.e., we show that $\mathrm{rc}_{\mathcal{P}}(2 \cdot n) \in \Omega(n)$ implies $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n)$. But the technique of Lemma 5.8 is not applicable to $\mathrm{rc}'_{\mathcal{P}}$, because $\mathrm{rc}'_{\mathcal{P}}$ is not weakly monotonically increasing.

We first show in Section 3.1 how to under-estimate the number of possible loop iterations for *simple loops* α of the form $f(\overline{x}) \to f(\overline{x})\mu\ [\varphi]$, where we define update$(\alpha) = \mu$ and require dom$(\mu) \subseteq \overline{x}$. So for instance, the rule

$$\alpha_1\colon\ \mathrm{f}_1(x, y, z, u) \ \to\ \mathrm{f}_1(x - 1, y + x, z, u) \quad [x > 0]$$

from Figure 1b is a simple loop where update(α_1) is the substitution $\mu = \{x/x - 1,\ y/y + x\}$. Based on the under-estimation of possible iterations, Section 3.2 presents our technique to accelerate simple loops. We introduce a technique to transform more complex loops into simple loops in Section 3.3.

3.1 Under-Estimating the Number of Iterations

For a simple loop α of the form $f(\overline{x}) \to f(\overline{x})\mu\ [\varphi]$, our goal is to infer an arithmetic expression b such that for all integer substitutions σ with $\sigma \models \alpha$, the rule α can be executed at least $b\sigma$ times, i.e., there is an integer substitution σ' with $f(\overline{x})\sigma \to_\alpha^{\lceil b\sigma \rceil} f(\overline{x})\sigma'$. Here, as usual, $\lceil x \rceil$ is the smallest integer n with $n \geq x$.

To find such estimates, we use an adaptation of ranking functions [6, 9, 13, 56] which we call *metering functions*. In the following, we say that a quantifier-free formula φ is *valid* if we have $\sigma \models \varphi$ for every integer substitution σ with $\mathcal{V}(\varphi) \subseteq$ dom(σ).

Definition 3.1 (Ranking Function). An arithmetic expression b is a *ranking function* for a simple loop α with update$(\alpha) = \mu$ and $\mathcal{TV}(\alpha) = \emptyset$ if the following conditions are valid:

$$\text{guard}(\alpha) \implies b > 0 \tag{6}$$

$$\text{guard}(\alpha) \implies b\mu \leq b - 1 \tag{7}$$

So for example, x is a ranking function for the rule α_1 in Figure 1b, since both $x > 0 \implies x > 0$ and $x > 0 \implies x - 1 \leq x - 1$ are clearly valid. If b is a ranking function for a rule α, then for any integer substitution σ with $\mathcal{V}(\alpha) \subseteq$ dom(σ), $b\sigma$ *over-estimates* the number of possible iterations of the loop α: (7) ensures that $b\sigma$ decreases at least by 1 in each loop iteration (i.e., $b\mu\sigma \leq b\sigma - 1$ holds whenever guard$(\alpha)\sigma$ is true), and (6) requires that $b\sigma$ is positive whenever the loop can be executed.

Note that Definition 3.1 would be incorrect for the case $\mathcal{TV}(\alpha) \neq \emptyset$. For example, consider the rule $\alpha:\ \mathrm{f}(x) \to \mathrm{f}(x + 1)\ [x < tv]$. If we omitted the requirement $\mathcal{TV}(\alpha) = \emptyset$, then $tv - x$ would be a ranking function for α since guard(α) implies both $tv - x > 0$ and $tv - (x + 1) \leq tv - x - 1$. However, there are non-terminating evaluations like $\mathrm{f}(0) \to_\alpha \mathrm{f}(1) \to_\alpha \mathrm{f}(2) \to_\alpha \ldots$, since tv can be instantiated differently in each evaluation step. Thus, $tv - x$ is not a correct over-estimation for the number of loop iterations.

To cover the case $\mathcal{TV}(\alpha) \neq \emptyset$, Definition 3.1 would need to reflect that the values of temporary variables may change non-deterministically in every iteration. We chose the simple definition above as it nicely exposes the analogy to our following novel concept of *metering functions*. In contrast to ranking functions, metering functions are *under-estimates* for the maximal number of iterations of a simple loop.

Definition 3.2 (Metering Function). We call an arithmetic expression b a *metering function* for a simple loop α with update$(\alpha) = \mu$ if the following conditions are valid:

$$\neg\text{guard}(\alpha) \implies b \leq 0 \tag{8}$$

$$\text{guard}(\alpha) \implies b\mu \geq b - 1 \tag{9}$$

Here, (9) ensures that $b\sigma$ decreases at most by 1 in each loop iteration, and (8) requires that $b\sigma$ is non-positive if the loop cannot be executed. Thus, the loop can be executed *at least* $b\sigma$ times (i.e., $b\sigma$ is an under-estimate).

In contrast to our definition of ranking functions, Definition 3.2 also covers the case $\mathcal{TV}(\alpha) \neq \emptyset$. As we will show in Theorem 3.3, the reason is that a metering function b for a simple loop α is a witness that α can be applied at least b times for *fixed* values of α's temporary variables. In particular, metering functions can also contain temporary variables to express that the number of loop iterations is unbounded, see Example 3.6.

As an example, for the loop α_1 in Figure 1b, x is also a metering function. Condition (8) requires the validity of $\neg(x > 0) \implies x \leq 0$ and (9) requires $x > 0 \implies x - 1 \geq x - 1$. While x is a metering *and* a ranking function, $\frac{1}{2}x$ is a metering, but not a ranking function for α_1. Similarly, x^2 is a ranking, but not a metering function for α_1. Theorem 3.3 states that if b is a metering function for a simple loop α, then α can be executed at least $\lceil b\sigma \rceil$ times when starting the evaluation with $\mathrm{lhs}(\alpha)\sigma$. Thus, if every rule has the constant cost 1, then $\mathrm{dh}_{\{\alpha\}}(\mathrm{lhs}(\alpha)\sigma) \geq b\sigma$ holds for all integer substitutions σ with $\mathcal{V}(\alpha) \cup \mathcal{V}(b) \subseteq \mathrm{dom}(\sigma)$. Recall that we have $\mathrm{rhs}(\alpha) = \mathrm{lhs}(\alpha)\mu$ for $\mu = \mathrm{update}(\alpha)$, since α is a simple loop. Hence, the evaluation has the form

$$\mathrm{lhs}(\alpha)\,\sigma \;\to_\alpha\; \mathrm{rhs}(\alpha)\,\sigma \;=\; \mathrm{lhs}(\alpha)\,\mu\,\sigma \;\to_\alpha\; \mathrm{rhs}(\alpha)\,\mu\,\sigma \;=\; \mathrm{lhs}(\alpha)\,\mu^2\,\sigma \;\to_\alpha\; \ldots$$

Here, for any $k \in \mathbb{N}$, μ^k stands for k applications of μ. So for example, μ^3 stands for $\mu \circ \mu \circ \mu$ and μ^0 is the identity substitution.

Theorem 3.3 (Metering Functions Under-Estimate Simple Loops). *Let b be a metering function for a well-formed simple loop α with $\mu = \mathrm{update}(\alpha)$. Then for all integer substitutions σ with $\mathcal{V}(\alpha) \subseteq \mathrm{dom}(\sigma)$ and $\sigma \models b \geq 0$, there is the following evaluation of length $\lceil b\sigma \rceil$:*

$$\mathrm{lhs}(\alpha)\,\sigma \to_\alpha \mathrm{lhs}(\alpha)\,\mu\,\sigma \to_\alpha \mathrm{lhs}(\alpha)\,\mu^2\,\sigma \to_\alpha \ldots \to_\alpha \mathrm{lhs}(\alpha)\,\mu^{\lceil b\sigma \rceil}\,\sigma$$

where $\mu^k \circ \sigma \models \mathrm{guard}(\alpha)$ for all $0 \leq k < b\sigma$.

Proof. For any integer substitution σ with $\mathcal{V}(\alpha) \subseteq \mathrm{dom}(\sigma)$, let $m_\sigma \in \mathbb{N} \cup \{\omega\}$ be the length of the longest evaluation of the form $\mathrm{lhs}(\alpha)\,\sigma \to_\alpha^{m_\sigma} \mathrm{lhs}(\alpha)\,\mu^{m_\sigma}\,\sigma$ where $\mu^k \circ \sigma \models \mathrm{guard}(\alpha)$ for all $0 \leq k < m_\sigma$. So the loop α can be executed m_σ times when starting with $\mathrm{lhs}(\alpha)\,\sigma$. We prove that $m_\sigma \geq b\sigma$.

If $m_\sigma = \omega$, then the claim is trivial. For $m_\sigma \neq \omega$, we use induction on m_σ. In the base case $m_\sigma = 0$, we have $\sigma \not\models \mathrm{guard}(\alpha)$. Thus, (8) implies $b\sigma \leq 0 = m_\sigma$.

For the induction step $m_\sigma \geq 1$, we must have $\sigma \models \mathrm{guard}(\alpha)$ which implies:

$$b\mu\sigma \;\geq\; b\sigma - 1 \qquad \text{by (9)} \tag{10}$$

$$\mathrm{lhs}(\alpha)\,\sigma \;\to_\alpha\; \mathrm{lhs}(\alpha)\,\mu\,\sigma \tag{11}$$

Due to (11), the longest evaluation $\mathrm{lhs}(\alpha)\,\sigma \to_\alpha^{m_\sigma} \mathrm{lhs}(\alpha)\,\mu^{m_\sigma}\,\sigma$ has the form

$$\mathrm{lhs}(\alpha)\,\sigma \to_\alpha \mathrm{lhs}(\alpha)\,\mu\,\sigma \to_\alpha^{m_{\mu\circ\sigma}} \mathrm{lhs}(\alpha)\,\mu^{m_\sigma}\,\sigma,$$

i.e., $m_\sigma = m_{\mu\circ\sigma} + 1$. Since $\mathcal{V}(\alpha) \subseteq \mathrm{dom}(\mu\circ\sigma)$ and $\mu\circ\sigma$ is an integer substitution (as α is well formed), the induction hypothesis implies $m_{\mu\circ\sigma} \geq b\mu\sigma$. Hence, we have $m_\sigma = m_{\mu\circ\sigma} + 1 \geq b\mu\sigma + 1 \geq b\sigma$ by (10). □

Note that if one regards a single simple loop $f(\overline{x}) \to f(\overline{x})\mu\ [\varphi]$ without any other f-rules that may lead to non-determinism, then the only remaining possible non-determinism is due to the temporary variables. So then the number of iterations of the loop in the worst and in the

best case only depends on the instantiation of the temporary variables. Since the requirements (8) and (9) for the metering function must hold for all instantiations of the variables (i.e., also for all instantiations of the temporary variables), then a metering function is also a lower bound on the number of iterations of the loop in the best case. To exploit that we only need lower bounds on the worst-case runtime of the loop, in Section 3.2 we will present a technique which can instantiate temporary variables by suitable values which (hopefully) lead to long runtimes.

Our implementation builds upon a well-known transformation based on Farkas' Lemma [13, 56] to find *linear* metering functions. The basic idea is to search for coefficients of a linear template polynomial b such that (8) and (9) hold for all possible instantiations of the variables $\mathcal{V}(\alpha)$. In addition to (8) and (9), we also require (6) to avoid trivial solutions like $b = 0$. Here, the coefficients of b are existentially quantified, while the variables from $\mathcal{V}(\alpha)$ are universally quantified. As in [13, 56], eliminating the universal quantifiers using Farkas' Lemma allows us to use standard SMT solvers to search for b's coefficients.[9]

If guard(α) contains constraints that are irrelevant for α's termination (provided that guard(α) is satisfiable), then one can improve our approach. More precisely, if guard(α) $= \varphi \wedge \psi$, then ψ is irrelevant for termination of the loop α if it always holds after executing the loop (given that it holds before the loop), i.e., if guard(α) implies $\psi\mu$. In this case, one can infer "conditional" metering functions of the form $[\![\psi]\!] \cdot b$. Here, $[\![\psi]\!]$ is the *characteristic function* of ψ, i.e., for any integer substitution σ with $\mathcal{V}(\psi) \subseteq \mathrm{dom}(\sigma)$ we have $[\![\psi]\!]\,\sigma = 1$ if $\sigma \models \psi$ and $[\![\psi]\!]\,\sigma = 0$ otherwise. So for example, $[\![y + z = 1]\!]\,\sigma = 1$ holds for an integer substitution σ iff $y\sigma + z\sigma = 1$.

Theorem 3.4 (Inferring Conditional Metering Functions). *Let α be a simple loop such that* update(α) $= \mu$ *and* guard(α) $= \varphi \wedge \psi$ *where* guard(α) $\implies \psi\mu$ *is valid. If the conditions*

$$\neg\varphi \wedge \psi \implies b \leq 0 \tag{12}$$

$$\varphi \wedge \psi \implies b\mu \geq b - 1 \tag{13}$$

are valid, then $[\![\psi]\!] \cdot b$ is a metering function for α.

Proof. If $\sigma \models$ guard(α), i.e., $\sigma \models \varphi \wedge \psi$, then we extend σ arbitrarily to the variables in $\psi\mu$ that do not occur in φ or ψ. Then we have $\sigma \models \psi\mu$ as guard(α) implies $\psi\mu$. Thus, $[\![\psi]\!]\,\mu\sigma = 1$ and $[\![\psi]\!]\,\sigma = 1$. So by (13) we obtain $([\![\psi]\!] \cdot b)\,\mu\sigma = b\mu\sigma \geq b\sigma - 1 = ([\![\psi]\!] \cdot b)\,\sigma - 1$, i.e., then $[\![\psi]\!] \cdot b$ satisfies (9).

Now we regard the case where $\sigma \models \neg$guard(α). If $\sigma \models \neg\psi$, then $([\![\psi]\!] \cdot b)\,\sigma = 0$, i.e., then $[\![\psi]\!] \cdot b$ satisfies (8). Otherwise, we have $\sigma \models \neg\varphi$ and $\sigma \models \psi$. Then (12) implies $([\![\psi]\!] \cdot b)\,\sigma = b\sigma \leq 0$, i.e., then $[\![\psi]\!] \cdot b$ also satisfies (8). □

While our implementation of Definition 3.2 was restricted to the search for linear metering functions b, with Theorem 3.4 our implementation can now also be used to obtain conditional metering functions of the form $[\![\psi]\!] \cdot b$ for linear arithmetic expressions b.

Compared to Definition 3.2, Theorem 3.4 weakens the conditions for metering functions: If b is a metering function according to (8) and (9), then we can also prove that $[\![\psi]\!] \cdot b$ is a metering

[9] Since Farkas' Lemma is only applicable for linear constraints, for loops with non-linear arithmetic, our implementation uses simplifications in order to linearize the constraints that have to be satisfied for metering functions: We may substitute a non-linear term by a fresh variable, provided that the variables of the non-linear term do not appear elsewhere (the reverse substitution is applied to the metering function afterwards), and we may omit irrelevant non-linear constraints or updates. For example, if the update of a variable x which does not occur in the guard is non-linear, then we use a linear template polynomial b without the variable x for the metering function. But if such simplifications are not possible, then we fail when trying to infer metering functions for loops with non-linear arithmetic.

function via Theorem 3.4 (but the converse does not hold). The reason is that in (8) we require $b \leq 0$ whenever $\neg(\varphi \wedge \psi)$ holds, whereas in (12) we only require $b \leq 0$ whenever $\neg\varphi \wedge \psi$ holds.

Example 3.5 (Conditional Metering Function). To illustrate the use of conditional metering functions, consider the following rule α:

$$\mathrm{f}(x, y, z) \to \mathrm{f}(x - y - z, y, z)\ [x > 0 \wedge y + z = 1]\,.$$

Here, we can choose φ to be $x > 0$ and ψ to be $y + z = 1$, since $x > 0 \wedge y + z = 1$ implies $(y + z = 1)\,\mu$ for α's update $\mu = \{x/x - y - z\}$, i.e., it implies $y + z = 1$. Hence, to infer the metering function $[\![y + z = 1]\!] \cdot x$, according to Theorem 3.4 it suffices to show that $\neg(x > 0) \wedge y + z = 1 \implies x \leq 0$ and $x > 0 \wedge y + z = 1 \implies x - y - z \geq x - 1$ are valid. Using this metering function, our approach can show that the rule can be applied at least linearly often. In contrast, without Theorem 3.4 our implementation would not be able to generate a useful metering function for this example, since it would only search for linear arithmetic expressions b that satisfy (8) and (9). However, x is not a metering function, since Condition (8) would not be satisfied (i.e., $\neg(x > 0 \wedge y + z = 1) \implies x \leq 0$ is not valid).

In [29], we already sketched a related optimization, which is however weaker than Theorem 3.4. There the idea was to omit ψ completely when searching for metering functions. So with this optimization, one would check the implications $\neg(x > 0) \implies x \leq 0$ and $x > 0 \implies x - y - z \geq x - 1$ to prove that $[\![y + z = 1]\!] \cdot x$ is a metering function for the loop of Example 3.5. As the second implication is not valid, this approach is not sufficient to handle Example 3.5. In contrast, Theorem 3.4 adds ψ to the premise in (12) and (13), i.e., the approach of Theorem 3.4 for inferring conditional metering functions is strictly more powerful than the optimization from [29].

Conditional metering functions are also particularly useful to integrate the handling of non-terminating rules in our approach.

Example 3.6 (Unbounded Loops). Let α be a simple loop whose update is μ. If $\mathrm{guard}(\alpha) \implies \mathrm{guard}(\alpha)\mu$ is valid and hence the *whole* guard is irrelevant for α's termination, then α does not terminate (provided that $\mathrm{guard}(\alpha)$ is satisfiable). In such cases, we can choose $\psi = \varphi$ in Theorem 3.4 and thus, (12) and (13) from Theorem 3.4 become

$$\mathrm{false} \implies b \leq 0 \quad \text{and} \quad \mathrm{guard}(\alpha) \implies b\mu \geq b - 1.$$

This is valid for a fresh temporary variable $b = tv$. Thus, for

$$\mathcal{P} = \{\mathrm{f}_0(x, y) \xrightarrow{0} \mathrm{f}(x, y),\ \alpha\} \quad \text{where } \alpha \text{ is the rule} \quad \mathrm{f}(x, y) \xrightarrow{y} \mathrm{f}(x + 1, y)\ [0 < x]\,,$$

we obtain the metering function $[\![0 < x]\!] \cdot tv$. As $\mathrm{guard}(\alpha) = 0 < x$ is satisfiable, this indicates that the runtime of the program is unbounded, i.e., $\mathrm{dh}_{\mathcal{P}}(\mathrm{f}(x, y)\sigma) \geq tv\,\sigma$ and thus $\mathrm{dh}_{\mathcal{P}}(\mathrm{f}(x, y)\sigma) = \omega$ for all integer substitutions σ with $\{x, y, tv\} \subseteq \mathrm{dom}(\sigma)$ and $0 < x\sigma$.

Note that in this example, Theorem 3.4 succeeds when choosing b to be tv (i.e., $[\![0 < x]\!] \cdot tv$ is a metering function). In contrast, tv is not a metering function, since (8) does not hold (i.e., $\neg(0 < x)$ does not imply $tv \leq 0$). Thus, conditional metering functions allow us to handle terminating and non-terminating rules in a uniform way.

3.2 Accelerating Simple Loops

We now define *sound processors* that simplify integer programs. A sound processor is essentially a program transformation which preserves lower runtime bounds.

Definition 3.7 (Processor). A *processor* proc is a partial function which maps integer programs to integer programs. It is *sound* if $\mathrm{rc}_{\mathcal{P}}(n) \geq \mathrm{rc}_{\mathrm{proc}(\mathcal{P})}(n)$ holds for all $n \in \mathbb{N}$ and all $\mathcal{P}$ where proc is defined.

In our framework, processors are applied repeatedly until the extraction of a concrete lower bound is straightforward. We first show how to *accelerate* a simple loop α to a rule which is equivalent to applying α multiple times (according to a metering function for α). In Section 3.3 we will show that the resulting integer program can be simplified by *chaining* subsequent rules which may result in new simple loops. Moreover, we describe a simplification strategy which alternates these steps repeatedly. In this way, we eventually obtain a *simplified* program without loops which directly gives rise to a concrete lower bound. Section 3.2 only deals with simple loops and in Section 3.3, we consider arbitrary tail-recursive rules α of the form $f(\overline{x}) \to g(\overline{t})\ [\varphi]$ with $\mathrm{update}(\alpha) = \{\overline{x}/\overline{t}\}$ and $\mathrm{target}(\alpha) = g$. We extend our approach to arbitrary rules in Section 4.

First, consider a simple loop α with $\mathrm{update}(\alpha) = \mu$ and $\mathrm{cost}(\alpha) = c$. To accelerate α, we compute its *iterated* update and cost, i.e., a substitution μ_{it} that is a closed form of μ^{tv} and an arithmetic expression c_{it} that is an under-approximation of $\sum_{i=0}^{tv-1} c\mu^i$ for a fresh temporary variable tv, where μ^i again denotes the i-fold composition $\mu \circ \ldots \circ \mu$ of the substitution μ. So $\mu_{\mathrm{it}} = \mu^{tv}$ and $c_{\mathrm{it}} \leq \sum_{i=0}^{tv-1} c\mu^i$ must hold for all $tv > 0$. If $[\![\psi]\!] \cdot b$ is a metering function for α, then we add the *accelerated* rule

$$\mathrm{lhs}(\alpha) \xrightarrow{c_{\mathrm{it}}} \mathrm{lhs}(\alpha)\,\mu_{\mathrm{it}}\ [\mathrm{guard}(\alpha) \wedge \psi \wedge 0 < tv < b + 1]$$

to the program. It summarizes tv iterations of α, where tv is positive[10] and bounded by $\lceil b \rceil$. Note that μ_{it} and c_{it} may also contain operations which are not allowed in the input program like division and exponentiation (i.e., in this way we can also infer non-polynomial bounds).

For the program variables $\overline{x} = (x_1, \ldots, x_k)$, the iterated update μ_{it} is computed by solving the recurrence equations $x^{(1)} = x\mu$ and $x^{(tv+1)} = x\mu\,\{x_1/x_1^{(tv)}, \ldots, x_k/x_k^{(tv)}\}$ for all $x \in \overline{x}$. So for the rule α_1 from Figure 1b we get the recurrence equations $x^{(1)} = x - 1$, $x^{(tv_1+1)} = x^{(tv_1)} - 1$, $y^{(1)} = y + x$, and $y^{(tv_1+1)} = y^{(tv_1)} + x^{(tv_1)}$. Usually, the resulting equations can easily be solved using state-of-the-art recurrence solvers, e.g., [8, 41, 61]. In our example, we obtain the closed forms

$$x\mu_{\mathrm{it}} = x^{(tv_1)} = x - tv_1 \quad \text{and} \quad y\mu_{\mathrm{it}} = y^{(tv_1)} = y + tv_1 \cdot x - \tfrac{1}{2}tv_1^2 + \tfrac{1}{2}tv_1.$$

While $y\mu_{\mathrm{it}}$ contains rational coefficients, our approach ensures that μ_{it} always maps integers to integers. Thus, our technique to accelerate loops preserves well-formedness. We proceed similarly for the iterated cost of a rule, where we may under-approximate the solution of the recurrence equations $c^{(1)} = c$ and $c^{(tv+1)} = c^{(tv)} + c\,\{x_1/x_1^{(tv)}, \ldots, x_k/x_k^{(tv)}\}$. For the rule α_1 in Figure 1, we get $c^{(1)} = 1$ and $c^{(tv_1+1)} = c^{(tv_1)} + 1$ which leads to the closed form $c_{\mathrm{it}} = c^{(tv_1)} = tv_1$. Hence, when using α_1's metering function x, it is accelerated to the following rule:

$$\mathrm{f}_1(x, y, z, u) \xrightarrow{tv_1} \mathrm{f}_1(x - tv_1, y + tv_1 \cdot x - \tfrac{1}{2}tv_1^2 + \tfrac{1}{2}tv_1, z, u) \quad [x > 0 \wedge 0 < tv_1 < x + 1] \tag{14}$$

Here, the guard can be simplified to $0 < tv_1 < x + 1$. (We will perform such simplifications in all examples to ease readability.)

Theorem 3.8 (Loop Acceleration). *Let $\mathcal{P}$ be a well-formed integer program with the program variables $\overline{x}$, let $\alpha \in \mathcal{P}$ be a simple loop with* $\mathrm{update}(\alpha) = \mu$ *and* $\mathrm{cost}(\alpha) = c$, *let tv be a fresh temporary*

[10] The accelerated rule does not cover the case that α is not applied at all, i.e., it does not cover the case where $tv = 0$. We excluded this case in order to ease the inference of the closed forms μ_{it} and c_{it}. To see this, consider a loop $\mathrm{f}(x) \to \mathrm{f}(0)$ with $x\mu = 0$. Here, we would get $x\mu_{\mathrm{it}} = 0$ if $tv > 0$ and $x\mu_{\mathrm{it}} = x$ for $tv = 0$. Hence, even in such simple examples it would be difficult to express the iterated update in closed form when considering the case $tv = 0$ as well.

variable, and let $[\![\psi]\!] \cdot b$ be a metering function for α, where b is an arithmetic expression. Moreover, for all $tv > 0$, let $x\mu_{\text{it}} = x\mu^{tv}$ be valid for all $x \in \overline{x}$, let $c_{\text{it}} \leq \sum_{i=0}^{tv-1} c\mu^i$ be valid, and let

$$\mathcal{P}' = \mathcal{P} \cup \{\alpha_{\text{it}}\} \quad \text{where } \alpha_{\text{it}} \text{ is the rule } \quad \text{lhs}(\alpha) \xrightarrow{c_{\text{it}}} \text{lhs}(\alpha)\,\mu_{\text{it}}\ [\text{guard}(\alpha) \wedge \psi \wedge 0 < tv < b + 1]\,.$$

Then $\mathcal{P}'$ is well formed and the processor that maps $\mathcal{P}$ to $\mathcal{P}'$ is sound.

Proof. Let σ be an integer substitution such that $\sigma \models \alpha_{\text{it}}$, i.e., we have

$$\text{lhs}(\alpha)\,\sigma \xrightarrow{c_{\text{it}}\sigma}_{\alpha_{\text{it}}} \text{lhs}(\alpha)\,\mu_{\text{it}}\,\sigma. \tag{15}$$

Note that $\sigma \models 0 < tv < b + 1$ implies $\sigma \models 0 < tv \leq \lceil b \rceil$ and we have $b\sigma = ([\![\psi]\!] \cdot b)\,\sigma$, because $\sigma \models \text{guard}(\alpha_{\text{it}})$ implies $\sigma \models \psi$ and thus, $[\![\psi]\!]\,\sigma = 1$. Since $\sigma \models \text{guard}(\alpha_{\text{it}})$ also implies $\sigma \models [\![\psi]\!] \cdot b \geq 0$ and $[\![\psi]\!] \cdot b$ is a metering function for α, by Theorem 3.3 there is the following evaluation of length $tv\,\sigma$:

$$\text{lhs}(\alpha)\,\sigma \to_\alpha \text{lhs}(\alpha)\,\mu\,\sigma \to_\alpha \ldots \to_\alpha \text{lhs}(\alpha)\,\mu^{tv\,\sigma}\,\sigma$$

where $\mu^k \circ \sigma \models \text{guard}(\alpha)$ for all $0 \leq k < tv\,\sigma$. For that reason, the costs of the rule applications are $c\sigma, c\mu\sigma, \ldots, c\mu^{tv\,\sigma-1}\sigma$, i.e.,

$$\text{lhs}(\alpha)\,\sigma \xrightarrow{c\sigma}_\alpha \text{lhs}(\alpha)\,\mu\,\sigma \xrightarrow{c\mu\sigma}_\alpha \ldots \xrightarrow{c\mu^{tv\,\sigma-1}\sigma}_\alpha \text{lhs}(\alpha)\,\mu^{tv\,\sigma}\,\sigma. \tag{16}$$

For soundness, it suffices to show that every evaluation step (15) with α_{it} can be simulated using a sequence of evaluation steps with α where the costs are at least the same. As α is well formed, this also implies well-formedness of α_{it}. By definition of μ_{it} and c_{it}, $x\mu_{\text{it}} = x\mu^{tv}$ for all $x \in \overline{x}$ and $c_{\text{it}} \leq \sum_{i=0}^{tv-1} c\mu^i$ are valid. Thus, the evaluation (16) indeed simulates the evaluation step (15) with α_{it}. □

Note that Theorem 3.8 shows that when using conditional metering functions of the form $[\![\psi]\!] \cdot b$ (which we infer via Theorem 3.4) to accelerate loops, the characteristic function $[\![\psi]\!]$ is not needed in the accelerated rule. Instead, the condition ψ is simply added to its guard. Thus, whenever the accelerated rule is applicable, then $[\![\psi]\!] \cdot b$ is equal to b and hence, $\lceil b \rceil$ under-estimates the number of consecutive iterations of the original loop. Clearly, Theorem 3.8 is also applicable if the metering function is not conditional, i.e., if it is an ordinary arithmetic expression (by choosing $\psi = \text{true}$).

The following example illustrates that the iterated update and cost may also contain non-polynomial arithmetic, which may lead to exponential bounds.

Example 3.9 (Non-Polynomial Arithmetic due to Loop Acceleration). Consider the program with the rule $\text{f}_0(x, y) \xrightarrow{0} \text{f}(x, y)$ and the simple loop $\text{f}(x, y) \xrightarrow{y} \text{f}(x - 1, 2y)\ [x > 0]$. Here, the update is $\mu = \{x/x - 1,\ y/2y\}$ and hence, the resulting iterated update is $\mu_{\text{it}} = \{x/x - tv,\ y/2^{tv} \cdot y\}$. Moreover, the cost is $c = y$ and the iterated cost is $c_{\text{it}} = \sum_{i=0}^{tv-1} 2^i y = (2^{tv} - 1) \cdot y$. Thus, both the iterated update and the iterated cost are exponential. Accelerating the simple loop via Theorem 3.8 with the metering function x yields $\text{f}(x, y) \xrightarrow{(2^{tv}-1)\cdot y} \text{f}(x - tv, 2^{tv} \cdot y)\ [0 < tv < x + 1]$, where we again simplified the guard $x > 0 \wedge 0 < tv < x + 1$ to $0 < tv < x + 1$. Using this accelerated rule, our approach can infer an exponential lower bound for the program's runtime complexity.

Recall that the fresh variable tv represents the number of loop iterations which are summarized by an accelerated rule. While tv ranges over the integers, its upper bound $b + 1$ can be rational, as the following example shows.

Example 3.10 (Non-Integer Metering Functions). Theorem 3.8 also allows bounds that do not map to the integers. Consider the program

$$\mathcal{P} = \{\text{f}_0(x) \xrightarrow{0} \text{f}(x),\ \alpha\} \quad \text{where } \alpha \text{ is the rule } \quad \text{f}(x) \xrightarrow{1} \text{f}(x - 2)\ [0 < x]\,.$$

Clearly, $\frac{1}{2}x$ is a metering function for α, as $\neg(0 < x) \implies \frac{1}{2}x \leq 0$ and $0 < x \implies \frac{1}{2}(x-2) \leq \frac{1}{2}x-1$ are valid. For $\mu = \{x/x-2\}$ we have $\mu_{\text{it}} = \mu^{tv} = \{x/x-2\,tv\}$ and we choose $c_{\text{it}} = \sum_{i=0}^{tv-1} 1 = tv$. Hence, accelerating α with the metering function $\frac{1}{2}x$ yields

$$\mathrm{f}(x) \quad \xrightarrow{tv} \quad \mathrm{f}(x-2\,tv) \quad \left[0 < tv < \tfrac{1}{2}x+1\right]. \tag{17}$$

Note that $0 < tv < \frac{1}{2}x + 1$ implies $0 < x$ as tv ranges over $\mathbb{Z}$. Hence, $0 < x$ can be omitted in the resulting guard.

If a (non-terminating) simple loop has the metering function $[\![\varphi]\!] \cdot tv$ where tv is a fresh temporary variable, then the upper bound $b + 1 = tv + 1$ on the number of summarized loop iterations can take arbitrary values.

Example 3.11 (Unbounded Loops Continued). In Example 3.6, $[\![0 < x]\!] \cdot tv$ is a metering function for $\alpha : \mathrm{f}(x, y) \xrightarrow{y} \mathrm{f}(x+1, y)\ [0 < x]$. The resulting accelerated rule α_{it} is

$$\mathrm{f}(x, y) \xrightarrow{tv_1 \cdot y} \mathrm{f}(x + tv_1, y)\ [0 < x \wedge 0 < tv_1 < tv + 1].$$

Since tv does not have any upper bound, the value of tv_1 is not bounded by the values of the program variables x and y. Thus, the condition of the rule could be replaced by $0 < x \wedge 0 < tv_1$, i.e., we obtain

$$\mathrm{f}(x, y) \xrightarrow{tv_1 \cdot y} \mathrm{f}(x + tv_1, y)\ [0 < x \wedge 0 < tv_1]. \tag{18}$$

After accelerating a simple loop α according to Theorem 3.8 using the metering function $[\![\psi]\!] \cdot b$, we eliminate the fresh variable tv by instantiating it with b, provided that b maps to $\mathbb{Z}$ (i.e., for every integer substitution σ with $\sigma \models \alpha\{tv/b\}$ we have $b\sigma \in \mathbb{Z}$). The reason is that we want to keep the number of variables small for the sake of efficiency. However, this is just a heuristic which can also lead to worse results (e.g., if there is a non-terminating run where the original non-accelerated loop must not be applied more than $b - 1$ times after each other). If we cannot verify that $b\sigma \in \mathbb{Z}$ holds for every integer substitution σ with $\sigma \models \alpha\{tv/b\}$, then we do not eliminate tv, but keep the inequation $0 < tv < b + 1$ in the accelerated rule.

We apply the following processor for the instantiation of temporary variables.

Theorem 3.12 (Instantiation). *Let $\mathcal{P}$ be a well-formed integer program, let $\alpha \in \mathcal{P}$, let $tv \in \mathcal{TV}(\alpha)$, let b be an arithmetic expression such that for every integer substitution σ with $\sigma \models \alpha\{tv/b\}$ we have $b\sigma \in \mathbb{Z}$, and let $\mathcal{P}' = \mathcal{P} \cup \{\alpha\{tv/b\}\}$. Then $\mathcal{P}'$ is well formed and the processor mapping $\mathcal{P}$ to $\mathcal{P}'$ is sound.*

Proof. To show the soundness of the processor, let σ be an integer substitution with $\sigma \models \alpha\{tv/b\}$ and $b\sigma = m \in \mathbb{Z}$. Then let σ' be the integer substitution with $\mathrm{dom}(\sigma') = \mathrm{dom}(\sigma) \cup \{tv\}$, $\sigma'(tv) = m$, and $\sigma'(x) = \sigma(x)$ for all $x \in \mathrm{dom}(\sigma) \setminus \{tv\}$. Clearly, $\sigma \models \mathrm{guard}(\alpha\{tv/b\})$ iff $\sigma' \models \mathrm{guard}(\alpha)$ and moreover, $\mathrm{cost}(\alpha\{tv/b\})\sigma = \mathrm{cost}(\alpha)\sigma'$. Thus,

$$\mathrm{lhs}(\alpha\{tv/b\})\sigma \xrightarrow{k}_{\alpha\{tv/b\}} \mathrm{rhs}(\alpha\{tv/b\})\sigma \quad \text{implies} \quad \mathrm{lhs}(\alpha)\sigma' \xrightarrow{k}_{\alpha} \mathrm{rhs}(\alpha)\sigma'.$$

This shows that every step with $\alpha\{tv/b\}$ can also be done with α, because we have $\mathrm{lhs}(\alpha\{tv/b\})\sigma = \mathrm{lhs}(\alpha)\{tv/b\}\sigma = \mathrm{lhs}(\alpha)\{tv/m\}\sigma = \mathrm{lhs}(\alpha)\sigma'$ (and $\mathrm{rhs}(\alpha\{tv/b\})\sigma = \mathrm{rhs}(\alpha)\sigma'$ can be derived analogously).

To show that $\mathcal{P}'$ is well formed, recall that $\sigma \models \mathrm{guard}(\alpha\{tv/b\})$ iff $\sigma' \models \mathrm{guard}(\alpha)$. If $\mathrm{rhs}(\alpha)$ contains $f(t_1, \ldots, t_k)$, then $t_i\sigma' \in \mathbb{Z}$ holds for all $1 \leq i \leq k$ by well-formedness of $\mathcal{P}$. As $t_i\sigma' = t_i\{tv/b\}\sigma$, this implies well-formedness of $\mathcal{P}'$. □

Example 3.13 (Instantiation of Fresh Temporary Variables). For our example from Figure 1b, accelerating α_1 results in the rule (14). By instantiating its temporary variable tv_1 with the metering function x, the above processor yields

$$\alpha_{\overline{1}}\colon\ \mathrm{f}_1(x,y,z,u) \xrightarrow{x} \mathrm{f}_1(0, y + \tfrac{1}{2}x^2 + \tfrac{1}{2}x, z, u) \quad [x > 0]\,.$$

In [29, Theorem 10], we presented a processor which can extend the guard of a rule by arbitrary conjuncts. This processor could be used as an alternative to Theorem 3.12, because instead of instantiating tv, one could add the constraint "$tv = b$" to the guard of α. In practice, however, it is preferable to instantiate tv in order to keep the number of variables as small as possible.

If we cannot apply Theorem 3.8 for a simple loop α, because our implementation fails to solve the recurrence equations needed to compute the closed forms μ_{it} or c_{it}, or because it cannot find a useful metering function, then we can simplify α by eliminating temporary variables. To do so, we fix their values via Theorem 3.12. As we are interested in witnesses for maximal computations, we use a heuristic that sets tv to a for temporary variables tv where the arithmetic expression a is a minimal upper or a maximal lower bound on tv's values, i.e., guard(α) implies $tv \leq a$ but not $tv \leq a - 1$, or guard(α) implies $tv \geq a$ but not $tv \geq a + 1$. This elimination of temporary variables is repeated until we find constraints which allow us to apply loop acceleration.

Example 3.14 (Instantiation of Other Temporary Variables). For the rule α_4 from Figure 1b, guard(α_4) contains the constraint $tv > 0$. So guard(α_4) implies the bound $tv \geq 1$ since tv must be instantiated by an integer. Hence, we instantiate the rule α_4 by replacing tv with 1. Thus, the update $\{u/u - tv\}$ of the instantiated rule α_4' becomes $\{u/u - 1\}$. Hence, now u is a metering function for α_4' (whereas it was not a metering function for α_4, as u's value could decrease by more than 1 in each application of α_4). Thus, α_4' can be accelerated similarly to α_1, resulting in the rule

$$\mathrm{f}_3(x,y,z,u) \xrightarrow{tv_4} \mathrm{f}_3(x,y,z,u - tv_4)\ [0 < tv_4 < u + 1]\,.$$

Now the temporary variable tv_4 that results from loop acceleration can be eliminated by instantiating it with the metering function u. In this way, we obtain

$$\alpha_{\overline{4}}\colon\ \mathrm{f}_3(x,y,z,u) \xrightarrow{u} \mathrm{f}_3(x,y,z,0) \quad [u > 0]\,.$$

If b is a polynomial, then we can use the following generalization of an observation from [18] to check the side condition of Theorem 3.12 that b needs to map to $\mathbb{Z}$. Note that this check does not take the guard of the rule into account. So for the rule

$$\mathrm{f}(x,y) \to \mathrm{f}(x-2, y-1)\ [x > 0 \wedge x = 2 \cdot y]$$

with the metering function $\frac{x}{2}$ it would fail to recognize that $\sigma\left(\frac{x}{2}\right)$ is an integer for every model σ of $x > 0 \wedge x = 2 \cdot y$.

Lemma 3.15 (Polynomials Mapping to $\mathbb{Z}$). *Let $f : \mathbb{Z}^k \to \mathbb{R}$, where $f(x_1, \ldots, x_k)$ is a polynomial over the variables $x_1, \ldots, x_k$ with degrees $d_1, \ldots, d_k$ w.r.t. $x_1, \ldots, x_k$, respectively (i.e., for each $1 \leq i \leq k$, $f(x_1, \ldots, x_k)$ can be rearranged to the form $\sum_{j=0}^{d_i} p_j \cdot x_i^j$ where each p_j is a polynomial over the variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k$). If there are numbers $n_1, \ldots, n_k \in \mathbb{Z}$ such that $f(m_1, \ldots, m_k) \in \mathbb{Z}$ for all $m_1, \ldots, m_k \in \mathbb{Z}$ with $n_i \leq m_i \leq n_i + d_i + 1$, then $\mathrm{img}(f) \subseteq \mathbb{Z}$, i.e., then we have $f(m_1, \ldots, m_k) \in \mathbb{Z}$ for all $m_1, \ldots, m_k \in \mathbb{Z}$.*

Proof. We use induction on $d = \sum_{i=1}^{k} d_i$. If $d = 0$, then f is a constant and thus the claim is trivial. If $d > 0$, then there exists a $1 \leq j \leq k$ with $d_j > 0$. Then $g(x_1, \ldots, x_k) = f(x_1, \ldots, x_{j-1}, x_j + 1, x_{j+1}, \ldots, x_k) - f(x_1, \ldots, x_k)$ is a polynomial whose degree w.r.t. x_j is $d_j - 1$ and whose degree w.r.t. all x_i with $i \neq j$ is at most d_i. To see this, note that all monomials that do not contain x_j vanish

in $g(x_1, \ldots, x_k)$. Thus, $g(x_1, \ldots, x_k)$ is a finite sum of expressions of the form $m \cdot (x_j+1)^e \cdot p - m \cdot x_j^e \cdot p$ where $m \in \mathbb{R}$, $e \leq d_j$, and p is a product of $x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_k$ where each x_i, $i \neq j$, occurs at most d_i times. We get:

$$\begin{array}{lll} & m \cdot (x_j + 1)^e \cdot p - m \cdot x_j^e \cdot p & \\ = & m \cdot \left(\sum_{i=0}^{e} \binom{e}{i} \cdot x_j^i\right) \cdot p - m \cdot x_j^e \cdot p & \text{by the Binomial theorem} \\ = & m \cdot \left(\binom{e}{e} \cdot x_j^e + q\right) \cdot p - m \cdot x_j^e \cdot p & \text{where } q \text{ is a univariate polynomial over } x_j \\ & & \text{whose degree is smaller than } e \\ = & m \cdot (x_j^e + q) \cdot p - m \cdot x_j^e \cdot p & \text{as } \binom{e}{e} = 1 \\ = & m \cdot q \cdot p & \end{array}$$

Moreover, since $f(m_1, \ldots, m_j + 1, \ldots, m_k) \in \mathbb{Z}$ and $f(m_1, \ldots, m_j, \ldots, m_k) \in \mathbb{Z}$ for all $m_1, \ldots, m_k \in \mathbb{Z}$ with $n_j \leq m_j \leq n_j + d_j$ and $n_i \leq m_i \leq n_i + d_i + 1$ if $i \neq j$, we also have $g(m_1, \ldots, m_k) \in \mathbb{Z}$ for these $m_1, \ldots, m_k$. Thus, by the induction hypothesis, we obtain $\text{img}(g) \subseteq \mathbb{Z}$. This means that we have $f(x_1, \ldots, x_{j-1}, x_j + 1, x_{j+1}, \ldots, x_k) - f(x_1, \ldots, x_k) \in \mathbb{Z}$ for all $x_1, \ldots, x_k \in \mathbb{Z}$. Since this construction can be done for every $1 \leq j \leq k$ with $d_j > 0$, we have $f(x_1, \ldots, x_{j-1}, x_j + 1, x_{j+1}, \ldots, x_k) - f(x_1, \ldots, x_k) \in \mathbb{Z}$ *for all* $1 \leq j \leq k$ and all $x_1, \ldots, x_k \in \mathbb{Z}$. Thus, we also have $f(x_1, \ldots, x_{j-1}, x_j - 1, x_{j+1}, \ldots, x_k) - f(x_1, \ldots, x_k) \in \mathbb{Z}$ for all $1 \leq j \leq k$ and all $x_1, \ldots, x_k \in \mathbb{Z}$. Since we also have $f(n_1, \ldots, n_k) \in \mathbb{Z}$ for some numbers $n_1, \ldots, n_k \in \mathbb{Z}$, this proves $\text{img}(f) \subseteq \mathbb{Z}$. □

So if b is a polynomial, then it suffices to check if instantiating the variables in b by finitely many integers always results in an integer. More precisely, if the polynomial b contains the variables $x_1, \ldots, x_k$ of degrees $d_1, \ldots, d_k$, respectively, then we only check if the polynomial maps all arguments from $\{0, \ldots, d_1 + 1\} \times \ldots \times \{0, \ldots, d_k + 1\}$ to integers. So we choose $n_i = 0$ for each n_i from Lemma 3.15. This is not a restriction, because if there is some other n_i' such that $f(m_1, \ldots, m_k) \in \mathbb{Z}$ for all $m_1, \ldots, m_k \in \mathbb{Z}$ with $n_i' \leq m_i \leq n_i' + d_i + 1$, then by Lemma 3.15 we have $\text{img}(f) \subseteq \mathbb{Z}$, which implies $f(m_1, \ldots, m_k) \in \mathbb{Z}$ for all $m_1, \ldots, m_k \in \mathbb{Z}$ with $n_i = 0 \leq m_i \leq d_i + 1 = n_i + d_i + 1$.

For instance, to check that the polynomial $\frac{1}{2}x^2 + \frac{1}{2}x$ maps *all* $x \in \mathbb{Z}$ to integers, it suffices to check this for just $x \in \{0, 1, 2, 3\}$. In contrast, for the polynomial $\frac{x}{2}$, we would check its value for $x \in \{0, 1, 2\}$ and determine that it does not always yield an integer.

Thus, one can implement Theorem 3.12 using Lemma 3.15 (but, of course, one may also incorporate further sufficient criteria). Similarly, as mentioned before, the criterion of Lemma 3.15 can also be used to check well-formedness of the integer program if we permit non-integer constants in the initial program.

To simplify the program, we delete the original rules after instantiating or accelerating them. If acceleration of a rule α still fails after eliminating all temporary variables by instantiating α repeatedly, then α is removed completely. So in the end, we just keep simple loops that have been accelerated. The following theorem shows that deleting rules is always sound.

Theorem 3.16 (Deletion). *Let $\mathcal{P}$ be a well-formed integer program, let $\alpha \in \mathcal{P}$, and let $\mathcal{P}' = \mathcal{P} \setminus \{\alpha\}$. Then $\mathcal{P}'$ is well formed and the processor mapping $\mathcal{P}$ to $\mathcal{P}'$ is sound.*

Proof. Since $\mathcal{P}$ is well formed, $\mathcal{P}'$ is trivially well formed, too. The processor is sound since every evaluation with $\mathcal{P} \setminus \{\alpha\}$ is also an evaluation with $\mathcal{P}$. □

Example 3.17 (Ex. 3.13 and 3.14 Continued). After accelerating and instantiating all simple loops in the program from Figure 1b, we delete the original loops α_1 and α_4, resulting in the following

integer program:

$$\begin{array}{lll}
\alpha_0\colon\ \mathrm{f}_0(x,y,z,u) & \xrightarrow{1} & \mathrm{f}_1(x,0,z,u) \\
\alpha_{\overline{1}}\colon\ \mathrm{f}_1(x,y,z,u) & \xrightarrow{x} & \mathrm{f}_1(0,y+\frac{1}{2}x^2+\frac{1}{2}x,z,u) \quad [x>0] \\
\alpha_2\colon\ \mathrm{f}_1(x,y,z,u) & \xrightarrow{1} & \mathrm{f}_2(x,y,y,u) \quad [x\leq 0] \\
\alpha_3\colon\ \mathrm{f}_2(x,y,z,u) & \xrightarrow{1} & \mathrm{f}_3(x,y,z,z-1) \quad [z>0] \\
\alpha_{\overline{4}}\colon\ \mathrm{f}_3(x,y,z,u) & \xrightarrow{u} & \mathrm{f}_3(x,y,z,0) \quad [u>0] \\
\alpha_5\colon\ \mathrm{f}_3(x,y,z,u) & \xrightarrow{1} & \mathrm{f}_2(x,y,z-1,u) \quad [u\leq 0]
\end{array}$$

3.3 Chaining Rules

After trying to accelerate all simple loops of a program, we can *chain* subsequent rules α_1, α_2 by adding a new rule $\alpha_{1.2}$ that represents their combination. Our notion of *chaining* corresponds to the standard notion of *unfolding* [16], adapted to our program model. Afterwards, the rules α_1 and α_2 can (but need not) be deleted with Theorem 3.16.

Theorem 3.18 (Chaining for Tail-Recursive Integer Programs). *Let $\mathcal{P}$ be a well-formed tail-recursive integer program and let $\alpha_1, \alpha_2 \in \mathcal{P}$ where*

$$\begin{array}{lll}
\alpha_1: & f_1(\overline{x}) \xrightarrow{c_1} f_2(\overline{x})\,\mu & [\varphi_1] \text{ and} \\
\alpha_2: & f_2(\overline{x}) \xrightarrow{c_2} t & [\varphi_2]\,.
\end{array}$$

W.l.o.g., let $\mathcal{TV}(\alpha_1) \cap \mathcal{TV}(\alpha_2) = \emptyset$ (otherwise, the temporary variables in α_2 can be renamed accordingly). Moreover, let $\alpha_{1.2}$ be the rule

$$f_1(\overline{x}) \xrightarrow{c_1 + c_2\mu} t\mu \ [\varphi_1 \wedge \varphi_2\mu]$$

and let $\mathcal{P}' = \mathcal{P} \cup \{\alpha_{1.2}\}$. Then $\mathcal{P}'$ is well formed and the processor that maps $\mathcal{P}$ to $\mathcal{P}'$ is sound.

Proof. We prove the more general Theorem 4.11 in Section 4. □

One goal of chaining is to eliminate all accelerated simple loops. Therefore, after accelerating all simple loops, we chain all subsequent rules α', α where α is a simple loop and α' is not a simple loop. Afterwards, we delete α. Moreover, once α' has been chained with all subsequent simple loops, then we also remove α', since its effect is now (mostly[11]) covered by the newly introduced chained rules.

Example 3.19 (Ex. 3.17 Continued). We continue the transformation of the program from Figure 1b. Now we chain α_0 with the accelerated simple loop $\alpha_{\overline{1}}$, and we chain α_3 with the accelerated simple loop $\alpha_{\overline{4}}$. This yields the following integer program:

$$\begin{array}{lll}
\alpha_{0.\overline{1}}\colon\ \mathrm{f}_0(x,y,z,u) & \xrightarrow{1+x} & \mathrm{f}_1(0,\frac{1}{2}x^2+\frac{1}{2}x,z,u) \quad [x>0] \\
\alpha_2\colon\ \mathrm{f}_1(x,y,z,u) & \xrightarrow{1} & \mathrm{f}_2(x,y,y,u) \quad [x\leq 0] \\
\alpha_{3.\overline{4}}\colon\ \mathrm{f}_2(x,y,z,u) & \xrightarrow{z} & \mathrm{f}_3(x,y,z,0) \quad [z>1] \\
\alpha_5\colon\ \mathrm{f}_3(x,y,z,u) & \xrightarrow{1} & \mathrm{f}_2(x,y,z-1,u) \quad [u\leq 0]
\end{array}$$

In Rule $\alpha_{3.\overline{4}}$, we simplified the guard $z > 0 \wedge z - 1 > 0$ to $z > 1$.

[11] Since accelerated rules α_{it} do not cover the case where α is not executed at all (see Footnote 10), chaining α' with α_{it} and deleting these rules afterwards does not cover those original evaluations where α' was not followed by any subsequent application of α. However, since we are only interested in witnesses for maximal evaluations, this does not affect the soundness of our approach.

Algorithm 1 Program Simplification for Tail-Recursive Integer Programs

While there is a rule α with $\mathrm{root}(\alpha) \neq \mathrm{f}_0$:

1. Apply *Deletion* to rules whose guard is proved unsatisfiable or whose root symbol is unreachable from f_0.
2. While there is a non-accelerated simple loop α:
   2.1. Try to *accelerate* α.
   2.2. If 2.1 succeeded, resulting in $\overline{\alpha}$:
      2.2.1. Try to *instantiate* $\overline{\alpha}$ to eliminate the temporary variable introduced in Step 2.1.
      2.2.2. If 2.2.1 succeeded, apply *Deletion* to $\overline{\alpha}$.
   2.3. If 2.1 failed and α uses temporary variables:
      Try to *instantiate* α to eliminate a temporary variable.
   2.4. Apply *Deletion* to α.
3. Let $S = \emptyset$.
4. While there is an accelerated rule α:
   4.1. For each α' with $\mathrm{root}(\alpha') \neq \mathrm{target}(\alpha') = \mathrm{root}(\alpha)$:
      Apply *Chaining* to α' and α and add α' to S.
   4.2. Apply *Deletion* to α.
5. Apply *Deletion* to each rule in S.
6. While there is a function symbol f without simple loops but with incoming and outgoing rules (starting with symbols f with just one incoming rule):
   6.1. Apply *Chaining* to each pair α', α where $\mathrm{target}(\alpha') = \mathrm{root}(\alpha) = f$.
   6.2. Apply *Deletion* to each α where $\mathrm{root}(\alpha) = f$ or $\mathrm{target}(\alpha) = f$.

Chaining also allows us to eliminate function symbols from the program by chaining all pairs of rules α' and α where $\mathrm{target}(\alpha') = \mathrm{root}(\alpha)$ and removing them afterwards. In this way we can transform loops consisting of several transitions into simple loops. It is advantageous to eliminate symbols which are the target of just one single rule first. This heuristic avoids eliminating the entry points of loops, if possible.

Example 3.20 (Ex. 3.19 Continued). So for the program in Example 3.19, it would avoid chaining α_5 and $\alpha_{3.\overline{4}}$ where $\mathrm{target}(\alpha_5) = \mathrm{root}(\alpha_{3.\overline{4}}) = \mathrm{f}_2$, because f_2 is also the target of the rule α_2. In this way, we avoid constructing chained rules that correspond to a run from the "middle" of a loop to the "middle" of the next loop iteration.

Instead, we chain $\alpha_{0.\overline{1}}$ and α_2 as well as $\alpha_{3.\overline{4}}$ and α_5 to eliminate the function symbols f_1 and f_3. This leads to the following program.

$$\begin{array}{lll} \alpha_{0.\overline{1}.2}\colon\ \mathrm{f}_0(x,y,z,u) & \xrightarrow{2+x} & \mathrm{f}_2(0, \frac{1}{2}x^2+\frac{1}{2}x, \frac{1}{2}x^2+\frac{1}{2}x, u) \quad [x > 0] \\ \alpha_{3.\overline{4}.5}\colon\ \mathrm{f}_2(x,y,z,u) & \xrightarrow{1+z} & \mathrm{f}_2(x, y, z-1, 0) \quad [z > 1] \end{array}$$

Our overall approach for program simplification is shown in Algorithm 1. It transforms any tail-recursive integer program into a *simplified* program. Section 5 will show how to analyze the (asymptotic) complexity of simplified programs. Of course, other strategies for the application of the processors would be possible, too. Recall that the application of *Acceleration*, *Instantiation*, and *Chaining* always generates new rules (so, e.g., in Step 2.2.1, instantiating $\overline{\alpha}$ generates a new rule $\widetilde{\alpha}$ and the original non-instantiated rule $\overline{\alpha}$ is *deleted* in Step 2.2.2). The set S in the Steps 3 – 5 is needed to handle function symbols f with multiple simple loops. The reason is that each rule α' with $\mathrm{target}(\alpha') = f$ should be chained with *each* of f's simple loops before removing α'.

Algorithm 1 terminates: The loop in Step 2 terminates since each iteration either decreases the number of temporary variables in α or reduces the number of non-accelerated simple loops. In Step 4, the number of accelerated rules is decreasing and for the loop in Step 6, the number of function symbols decreases. The overall loop of Algorithm 1 terminates as it reduces the number of function symbols. The reason is that the program does not have simple loops anymore when the algorithm reaches Step 6 (as simple loops where acceleration fails are deleted in Step 2.4 and accelerated rules are eliminated in Step 4). Thus, at this point there is either a function symbol f which can be eliminated or the program does not have a path of length 2, i.e., all rules have the root f_0.

Example 3.21 (Ex. 3.20 Continued). According to Algorithm 1, in our example we go back to Step 1 and 2 and apply *Loop Acceleration* to the rule $\alpha_{3.\overline{4}.5}$. This rule has the metering function $z-1$ and its iterated update sets u to 0 and z to $z-tv$ for a fresh temporary variable tv. To compute $\alpha_{3.\overline{4}.5}$'s iterated cost, we have to find an under-approximation for the solution of the recurrence equations $c^{(1)} = 1+z$ and $c^{(tv+1)} = c^{(tv)} + 1 + z^{(tv)}$. After computing the closed form $z-tv$ of $z^{(tv)}$, the second equation simplifies to $c^{(tv+1)} = c^{(tv)} + 1 + z - tv$, which results in the closed form $c_{\mathrm{it}} = c^{(tv)} = tv \cdot z - \frac{1}{2}tv^2 + \frac{3}{2}tv$. Thus, we obtain the accelerated rule

$$\mathrm{f}_2(x,y,z,u) \xrightarrow{tv\cdot z-\frac{1}{2}tv^2+\frac{3}{2}tv} \mathrm{f}_2(x,y,z-tv,0)\ [0<tv<z]\,.$$

By instantiating tv with $z-1$ in Step 2.2.1 and removing $\alpha_{3.\overline{4}.5}$ in Step 2.4, we obtain the following program.

$$\begin{array}{lll} \alpha_{0.\overline{1}.2}\colon\ \mathrm{f}_0(x,y,z,u) & \xrightarrow{2+x} & \mathrm{f}_2(0,\tfrac{1}{2}x^2+\tfrac{1}{2}x,\tfrac{1}{2}x^2+\tfrac{1}{2}x,u) \quad [x>0] \\ \alpha_{\overline{3.\overline{4}.5}}\colon\ \mathrm{f}_2(x,y,z,u) & \xrightarrow{\frac{1}{2}z^2+\frac{3}{2}z-2} & \mathrm{f}_2(x,y,1,0) \quad [z>1] \end{array}$$

A final chaining step and deletion of $\alpha_{0.\overline{1}.2}$ and $\alpha_{\overline{3.\overline{4}.5}}$ yields the simplified program with the following single rule.

$$\alpha_{0.\overline{1}.2.\overline{3.\overline{4}.5}}\colon\ \mathrm{f}_0(x,y,z,u) \xrightarrow{\frac{1}{8}x^4+\frac{1}{4}x^3+\frac{7}{8}x^2+\frac{7}{4}x} \mathrm{f}_2(0,\tfrac{1}{2}x^2+\tfrac{1}{2}x,1,0) \quad \left[\tfrac{1}{2}x^2+\tfrac{1}{2}x>1\right] \tag{19}$$

4 SIMPLIFYING ARBITRARY RECURSIVE INTEGER PROGRAMS

So far, we only considered tail-recursive programs, i.e., programs where all rules have the form $f(\overline{x}) \to g(\overline{t})\ [\varphi]$. We now extend our technique to non-tail-recursive programs, i.e., we now also consider rules where the right-hand side is a multiset of several terms.

Theorems 3.12 and 3.16 are trivially applicable to non-tail-recursive programs as well, i.e., we can still instantiate temporary variables and we can still delete rules. However, Theorems 3.8 and 3.18 have to be adapted. In Section 4.1 we extend our notion of metering functions to (non-tail-)recursive rules in order to adapt Theorem 3.8. Similar to the case of tail-recursive programs where we started with considering simple loops, we first focus on *simple recursions*, i.e., rules $f(\overline{x}) \xrightarrow{c} T\ [\varphi]$ whose degree $|T|$ is greater than 1 and where all terms in T have the root symbol f.[12] Our extended notion of metering functions then allows us to accelerate simple recursions in Section 4.2. Afterwards, in Section 4.3 we show how to transform more complex recursions into simple recursions or simple loops via Chaining and *Partial Deletion*, a new technique which is specific to non-tail-recursive integer programs. Based on these techniques, we extend Algorithm 1 to a procedure which transforms any integer program into a simplified program.

[12]The reason for excluding the case $|T| = 1$ is that in this way, simple recursions give rise to *exponential* lower bounds, i.e., we can formulate the corresponding Theorem 4.3 which does not hold if $|T| = 1$.

4.1 Under-Estimating the Depth of Recursions

To understand the idea of metering functions for simple recursions, note that repeatedly applying a simple recursive rule $f(\overline{x}) \xrightarrow{c} T\ [\varphi]$ essentially yields an *evaluation tree* of terms where each inner node has $|T|$ successors. While metering functions for simple loops under-estimate the length of evaluations, metering functions for simple recursions under-estimate the height up to which such evaluation trees are complete. Hence, if b is a metering function for a simple recursion α of degree d, then the maximal number of consecutive applications of α is in $\Omega(d^b)$.

Definition 4.1 (Metering Function for Simple Recursions). Let α be a rule of the form

$$f(\overline{x}) \xrightarrow{c} T\ [\varphi]$$

such that T contains no function symbol from Σ except f. We call an arithmetic expression b a *metering function* for α if the following conditions are valid:

$$\neg\text{guard}(\alpha) \implies b \leq 0 \tag{20}$$

$$\text{guard}(\alpha) \implies b\{\overline{x}/\overline{t}\} \geq b - 1 \text{ for all } f(\overline{t}) \in T \tag{21}$$

Definition 4.1 is a generalization of Definition 3.2, i.e., if α has degree 1, then Definition 4.1 and Definition 3.2 coincide. Moreover, note that conditional metering functions of the form $[\![\psi]\!] \cdot b$ for simple recursions can be inferred analogously to conditional metering functions for simple loops (see Theorem 3.4): If $\text{guard}(\alpha)$ is $\varphi \wedge \psi$ and $\text{guard}(\alpha)$ implies $\psi\{\overline{x}/\overline{t}\}$ for all $f(\overline{t}) \in T$, then it suffices to check $\neg\varphi \wedge \psi \implies b \leq 0$ and $\varphi \wedge \psi \implies b\{\overline{x}/\overline{t}\} \geq b - 1$ for all $f(\overline{t}) \in T$ in order to prove that $[\![\psi]\!] \cdot b$ is a metering function for α.

Example 4.2 (Metering Function for Fibonacci). According to Definition 4.1, $\frac{1}{2}x - 1$ is a metering function for the recursive Fibonacci rule (2) from Example 2.1. It satisfies (20), as we have $\neg(x > 1) \implies \frac{1}{2}x - 1 \leq 0$. The recursive call $\text{fib}(x-1)$ satisfies (21), since we have

$$x > 1 \implies \tfrac{1}{2}(x-1) - 1 = \tfrac{1}{2}x - \tfrac{3}{2} \geq \tfrac{1}{2}x - 2.$$

Finally, the recursive call $\text{fib}(x-2)$ also satisfies (21), as we have

$$x > 1 \implies \tfrac{1}{2}(x-2) - 1 = \tfrac{1}{2}x - 2 \geq \tfrac{1}{2}x - 2.$$

For a simple loop α, we directly use its metering function as a lower bound for the number of consecutive applications of α. But for simple recursions we can infer a lower bound that is higher than its metering function. The reason is that when computing a metering function b for a simple recursion $f(\overline{x}) \to \{f(\overline{t_1}), \ldots, f(\overline{t_d})\}\ [\varphi]$, we proceed as if we had d separate tail-recursive rules $f(\overline{x}) \to f(\overline{t_1})\ [\varphi], \ldots, f(\overline{x}) \to f(\overline{t_d})\ [\varphi]$. However, as the original rule initiates d new evaluations in each step, we obtain a lower bound on the length of evaluations that is exponential in b. In other words, since the metering function b under-estimates the height of complete evaluation trees (where every non-leaf node has d children), the number of edges of the tree is in $\Omega(d^b)$.

Theorem 4.3 (Metering Functions Under-Estimate Simple Recursions). *Let b be a metering function for a well-formed simple recursion α with degree d. Then for all integer substitutions σ with $\mathcal{V}(\alpha) \subseteq \text{dom}(\sigma)$, there is an evaluation $\text{lhs}(\alpha)\,\sigma \to_\alpha^m S$ for some configuration S with $m \geq \frac{d^{b\sigma}-1}{d-1}$.*

Proof. First note that we have $d > 1$ since α is a simple recursion and hence $\frac{d^{b\sigma}-1}{d-1}$ is well defined. As in the proof of Theorem 3.3, let $m_\sigma \in \mathbb{N} \cup \{\omega\}$ be the length of the longest evaluation which starts with $\text{lhs}(\alpha)\,\sigma$ and only applies the rule α repeatedly. We prove that $m_\sigma \geq \frac{d^{b\sigma}-1}{d-1}$.

The case $m_\sigma = \omega$ is trivial. For $m_\sigma \neq \omega$, we use induction on m_σ. In the base case $m_\sigma = 0$, we have $\sigma \not\models \text{guard}(\alpha)$ and thus, (20) implies $b\sigma \leq 0$. Hence, $\frac{d^{b\sigma}-1}{d-1} \leq 0 = m_\sigma$.

For the induction step $m_\sigma \geq 1$, we must have $\sigma \models \text{guard}(\alpha)$. Let $\text{lhs}(\alpha) = f(\overline{x})$. Then we obtain

$$b\,\{\overline{x}/\overline{t}\}\,\sigma \quad \geq \quad b\sigma - 1 \qquad \text{for all } f(\overline{t}) \in \text{rhs}(\alpha), \quad \text{by (21)} \tag{22}$$

$$\text{lhs}(\alpha)\,\sigma = f(\overline{x})\sigma \quad \to_\alpha \quad \text{rhs}(\alpha)\,\sigma \tag{23}$$

Let $\text{rhs}(\alpha) = \{f(\overline{t_1}), \ldots, f(\overline{t_d})\}$. Then due to (23), the longest evaluation $\text{lhs}(\alpha)\,\sigma \to_\alpha^{m_\sigma} S$ has the form

$$\text{lhs}(\alpha)\,\sigma \;\to_\alpha\; \text{rhs}(\alpha)\,\sigma \;=\; \{f(\overline{t_1})\sigma, \ldots, f(\overline{t_d})\sigma\} \;\to_\alpha^{m_{\{\overline{x}/\overline{t_1}\}\circ\sigma}+\ldots+m_{\{\overline{x}/\overline{t_d}\}\circ\sigma}} S,$$

i.e., $m_\sigma = m_{\{\overline{x}/\overline{t_1}\}\circ\sigma} + \ldots + m_{\{\overline{x}/\overline{t_d}\}\circ\sigma} + 1$. Since $\mathcal{V}(\alpha) \subseteq \text{dom}(\{\overline{x}/\overline{t_i}\}\circ\sigma)$ and $\{\overline{x}/\overline{t_i}\}\circ\sigma$ is an integer substitution (since α is well formed), the induction hypothesis implies $m_{\{\overline{x}/\overline{t_i}\}\circ\sigma} \geq \frac{d^{b\,\{\overline{x}/\overline{t_i}\}\,\sigma}-1}{d-1} \geq \frac{d^{b\sigma-1}-1}{d-1}$ by (22) for all $1 \leq i \leq d$. Hence, we have $m_\sigma = m_{\{\overline{x}/\overline{t_1}\}\circ\sigma} + \ldots + m_{\{\overline{x}/\overline{t_d}\}\circ\sigma} + 1 \geq d \cdot \frac{d^{b\sigma-1}-1}{d-1} + 1 = \frac{d^{b\sigma}-d}{d-1} + 1 = \frac{d^{b\sigma}-1}{d-1}$. □

Example 4.4 (Under-Estimating Fibonacci). Since $\frac{1}{2}x - 1$ is a metering function for the recursive Fibonacci rule (2), the term $\text{fib}(x)\,\sigma$ starts an evaluation of at least length $2^{\frac{1}{2}\cdot x\sigma - 1} - 1$ for each integer substitution σ with $x \in \text{dom}(\sigma)$.

4.2 Accelerating Simple Recursions

Using Definition 4.1 and Theorem 4.3, we can now accelerate simple recursions. In contrast to the acceleration of simple loops in Theorem 3.8, here we disregard the result of the accelerated rule and replace its result with $\emptyset$. The reason is that otherwise the degree of the accelerated rule (i.e., the number of elements in its right-hand side) would depend on the instantiation of the variable tv that represents the height of the evaluation tree. This cannot be expressed in our program model. Moreover, we cannot compute iterated updates, as a simple recursion α has *several* updates which may be applied in arbitrary order when α is applied repeatedly. Since computing the iterated cost would require the iterated updates, we do not compute the iterated cost anymore, but simply under-estimate each evaluation step with cost 1. This suffices to infer a lower bound for the cost of the accelerated rule which is exponential in the metering function.

Theorem 4.5 (Recursion Acceleration). *Let $\mathcal{P}$ be a well-formed integer program, let $\alpha \in \mathcal{P}$ be a simple recursion of degree d such that*

$$\text{guard}(\alpha) \implies \text{cost}(\alpha) \geq 1 \tag{24}$$

is valid,[13] *and let $[\![\psi]\!] \cdot b$ be a metering function for α. Moreover, let α' be the rule*

$$\text{lhs}(\alpha) \xrightarrow{c} \emptyset \;\; [\text{guard}(\alpha) \wedge \psi] \;\; \textit{where } c = \frac{d^b - 1}{d-1},$$

and let $\mathcal{P}' = \mathcal{P} \cup \{\alpha'\}$. Then $\mathcal{P}'$ is well formed and the processor that maps $\mathcal{P}$ to $\mathcal{P}'$ is sound.

Proof. Well-formedness is trivial due to the empty right-hand side $\emptyset$ of the rule α'. To prove soundness, note that by Theorem 4.3, $\sigma \models \alpha'$ implies $\text{lhs}(\alpha)\,\sigma \xrightarrow{k}_\alpha^m S$ with $m \geq \frac{d^{b\sigma}-1}{d-1}$ for some cost k and some configuration S (since $\sigma \models \text{guard}(\alpha')$ implies $\sigma \models \psi$ which in turn implies $([\![\psi]\!] \cdot b)\,\sigma = b\sigma$). Since $\text{guard}(\alpha) \implies \text{cost}(\alpha) \geq 1$ is valid, we get $k \geq m \geq \frac{d^{b\sigma}-1}{d-1} = c\sigma$. Thus, for every evaluation with α', there is an evaluation with α which has at least the same cost. □

[13] If (24) is not valid, then one can simply add the constraint $\text{cost}(\alpha) \geq 1$ to the guard of the rule, since adding constraints to the guard is always sound as it can only decrease the derivation height (by disallowing some evaluations).

Example 4.6 (Accelerating Fibonacci). Since the rule (2) from Example 2.1 has cost 1, (24) is trivially valid. Thus, accelerating the rule (2) yields

$$\mathrm{fib}(x) \xrightarrow{2^{\frac{1}{2}x-1}-1} \emptyset \ [x > 1].$$

Afterwards, chaining this rule with (1) and deleting all other rules yields the simplified program with the rule

$$\mathrm{f}_0(x) \xrightarrow{2^{\frac{1}{2}x-1}-1} \emptyset \ [x > 1]. \tag{25}$$

4.3 Simplifying Recursive Rules

Section 4.2 showed how to accelerate simple recursions. If our implementation fails in applying Theorem 4.5 to a simple recursion, then we again try to eliminate temporary variables via *Instantiation*, as in the case of simple loops. If all temporary variables were eliminated and we still fail to accelerate a simple recursion, then further simplifications are possible. In particular, we can remove parts of the right-hand side via *Partial Deletion*.

Theorem 4.7 (Partial Deletion). *Let $\mathcal{P}$ be a well-formed integer program and let $\alpha \in \mathcal{P}$. Moreover, let α' be like α, but $\mathrm{rhs}(\alpha') \subset \mathrm{rhs}(\alpha)$, and let $\mathcal{P}' = \mathcal{P} \cup \{\alpha'\}$. Then $\mathcal{P}'$ is well formed and the processor that maps $\mathcal{P}$ to $\mathcal{P}'$ is sound.*

Proof. Since $\mathcal{P}$ is well formed, $\mathcal{P}'$ is trivially well formed, too. To prove soundness, we define

$$S \xrightarrow{k}_{\mathcal{P}\circ\supseteq} T \quad \text{if} \quad S \xrightarrow{k}_{\mathcal{P}} \circ \supseteq T.$$

So $S \xrightarrow{k}_{\mathcal{P}\circ\supseteq} T$ holds if S evaluates to some configuration S' with cost k (i.e., $S \xrightarrow{k}_{\mathcal{P}} S'$) and the configuration T results from S' by deleting some terms from T (i.e., $S' \supseteq T$). Then we clearly have $\mathrm{dh}_{\mathcal{P}'}(S) \leq \sup\{k \in \mathbb{R} \mid S \xrightarrow{k}{}^*_{\mathcal{P}\circ\supseteq} T \text{ for some } T \in C\}$, as each $\rightarrow_{\mathcal{P}'}$-sequence is also a $\rightarrow_{\mathcal{P}\circ\supseteq}$-sequence. To finish the proof, we show

$$\sup\{k \in \mathbb{R} \mid S \xrightarrow{k}{}^*_{\mathcal{P}\circ\supseteq} T \text{ for some } T \in C\} \leq \mathrm{dh}_{\mathcal{P}}(S).$$

This clearly implies $\mathrm{rc}_{\mathcal{P}'}(n) \leq \mathrm{rc}_{\mathcal{P}}(n)$, i.e., it implies that the processor is sound.

Consider an evaluation $S_0 \xrightarrow{k_1}{}^*_{\mathcal{P}\circ\supseteq} \ldots \xrightarrow{k_m}{}^*_{\mathcal{P}\circ\supseteq} S_m$. We prove $S_0 \xrightarrow{k_1}{}^*_{\mathcal{P}} \ldots \xrightarrow{k_m}{}^*_{\mathcal{P}} S'_m \supseteq S_m$ for some $S'_m \in C$ by induction on m. The case $m = 0$ is trivial. If $m > 0$, the induction hypothesis implies

$$S_0 \xrightarrow{k_1}{}^*_{\mathcal{P}} \ldots \xrightarrow{k_{m-1}}{}^*_{\mathcal{P}} S'_{m-1} \supseteq S_{m-1} \text{ for some } S'_{m-1} \in C.$$

Moreover, $S_{m-1} \xrightarrow{k_m}_{\mathcal{P}\circ\supseteq} S_m$ implies $S_{m-1} \xrightarrow{k_m}_{\mathcal{P}} \widetilde{S_m} \supseteq S_m$ for some $\widetilde{S_m} \in C$, i.e., we have $s \xrightarrow{k_m}_{\mathcal{P}} Q$ and $\widetilde{S_m} = (S_{m-1} \setminus \{s\}) \cup Q$ for some $s \in S_{m-1}$ and some $Q \in C$. Since $S_{m-1} \subseteq S'_{m-1}$, we get

$$S'_{m-1} \xrightarrow{k_m}_{\mathcal{P}} (S'_{m-1} \setminus \{s\}) \cup Q = S'_m$$

and thus $S_0 \xrightarrow{k_1}{}^*_{\mathcal{P}} \ldots \xrightarrow{k_m}{}^*_{\mathcal{P}} S'_m$ with $S'_m \supseteq \widetilde{S_m} \supseteq S_m$, as desired. □

Example 4.8 (Partial Deletion to Enable Recursion Acceleration). Consider the simple recursion $\mathrm{f}(x, y) \rightarrow \{\mathrm{f}(x-1, y), \mathrm{f}(x-y, y)\}\ [x > 0 \wedge y > x]$. As $\mathrm{f}(x-y, y)$ cannot be reduced any further if $y > x$, we cannot find a useful metering function for this rule and hence, *Recursion Acceleration* fails. By applying *Partial Deletion*, we obtain the simple loop $\mathrm{f}(x, y) \rightarrow \mathrm{f}(x-1, y)\ [x > 0 \wedge y > x]$, which can easily be accelerated via Theorem 3.8. In this way, we can infer that the original non-tail-recursive rule can be applied at least linearly often.

As shown above, *Recursion Acceleration* is useful to handle programs with non-linear recursion like the Fibonacci program, where the result is composed of two recursive calls. However, non-tail-recursion also occurs when composing a recursive call with the call of an auxiliary function.

Example 4.9 (Non-Tail-Recursive facSum *Program [14]).* Consider the following imperative program.

```
int fac(int x) {
  if (x > 1) return x * fac(x-1);
  else return 1;
}

int facSum(int x) {
  if (x > 0) return fac(x) + facSum(x-1);
  else return 1;
}
```

Here, fac(x) computes x! and facSum(x) computes 0! + ... + x!. The program is not tail-recursive, because the last action of facSum is not the recursive call, but an addition. The integer program below represents its recursive structure, i.e., it can be obtained from the above program by a suitable abstraction.

$$\mathsf{f}_0(x) \xrightarrow{0} \mathsf{facSum}(x) \tag{26}$$

$$\mathsf{facSum}(x) \xrightarrow{1} \{\mathsf{fac}(x), \mathsf{facSum}(x-1)\} \quad [x > 0] \tag{27}$$

$$\mathsf{facSum}(x) \xrightarrow{1} \emptyset \quad [x \leq 0]$$

$$\mathsf{fac}(x) \xrightarrow{1} \mathsf{fac}(x-1) \quad [x > 1] \tag{28}$$

$$\mathsf{fac}(x) \xrightarrow{1} \emptyset \quad [x \leq 1] \tag{29}$$

To analyze this integer program, we first accelerate and chain the recursive rule (28) as in Theorem 3.8 and Theorem 3.18.

Example 4.10 (Accelerating fac*).* Clearly, $x - 1$ is a metering function for the rule (28). Accelerating it using this metering function yields

$$\mathsf{fac}(x) \xrightarrow{tv} \mathsf{fac}(x - tv) \ [x > 1 \wedge 0 < tv < x] .$$

Instantiating tv with $x - 1$ via Theorem 3.12 results in

$$\mathsf{fac}(x) \xrightarrow{x-1} \mathsf{fac}(1) \ [x > 1] . \tag{30}$$

At this point, we would like to chain the recursive facSum-rule (27) with the fac-rule (30). However, Theorem 3.18 is only applicable to tail-recursive rules. Hence, we now generalize Theorem 3.18 to arbitrary rules.

Theorem 4.11 (Chaining for Arbitrary Integer Programs). *Let $\mathcal{P}$ be a well-formed integer program and let $\alpha_1, \alpha_2 \in \mathcal{P}$ where*

$$\begin{array}{lllll} \alpha_1 : & f_1(\overline{x}) & \xrightarrow{c_1} & S & [\varphi_1] \text{ with } f_2(\overline{x})\,\mu \in S \text{ and} \\ \alpha_2 : & f_2(\overline{x}) & \xrightarrow{c_2} & T & [\varphi_2] . \end{array}$$

W.l.o.g., let $\mathcal{TV}(\alpha_1) \cap \mathcal{TV}(\alpha_2) = \emptyset$ (otherwise, the temporary variables in α_2 can be renamed accordingly). Moreover, let $\alpha_{1.2}$ be the rule

$$f_1(\overline{x}) \xrightarrow{c_1 + c_2\mu} (S \setminus \{f_2(\overline{x})\,\mu\}) \cup T\,\mu \ [\varphi_1 \wedge \varphi_2\mu]$$

and let $\mathcal{P}' = \mathcal{P} \cup \{\alpha_{1.2}\}$. Then $\mathcal{P}'$ is well formed and the processor that maps $\mathcal{P}$ to $\mathcal{P}'$ is sound.

Proof. To prove the soundness of the processor, we show that every evaluation step with $\alpha_{1.2}$ can be simulated by two evaluation steps with the rules α_1, α_2 of the same cost. Let σ be an integer substitution with $\sigma \models \alpha_{1.2}$. Then we have

$$f_1(\overline{x})\,\sigma \xrightarrow{c_1\sigma + c_2\mu\sigma}_{\alpha_{1.2}} (S\sigma \setminus \{f_2(\overline{x})\,\mu\sigma\}) \cup T\mu\sigma.$$

Since $\sigma \models \varphi_1$, we have

$$f_1(\overline{x})\,\sigma \xrightarrow{c_1\sigma}_{\alpha_1} S\sigma.$$

Since $\sigma \models \varphi_2\mu$ implies $\mu \circ \sigma \models \varphi_2$ we have

$$f_2(\overline{x})\,\mu\sigma \xrightarrow{c_2\mu\sigma}_{\alpha_2} T\mu\sigma.$$

As $f_2(\overline{x})\mu \in S$, this implies

$$S\sigma \xrightarrow{c_2\mu\sigma}_{\alpha_2} (S\sigma \setminus \{f_2(\overline{x})\,\mu\sigma\}) \cup T\mu\sigma.$$

Thus, we have $f_1(\overline{x})\sigma \xrightarrow{c_1\sigma + c_2\mu\sigma}{}^{2}_{\mathcal{P}} (S\sigma \setminus \{f_2(\overline{x})\,\mu\sigma\}) \cup T\mu\sigma$, as desired. As α_1 and α_2 are well formed, this also proves that $\mathcal{P}'$ is well formed. □

Note that Theorem 4.11 coincides with Theorem 3.18 if the degree of α_1 and α_2 is 1. Theorem 4.11 allows us to continue the transformation of the program in Example 4.10.

Example 4.12 (Chaining facSum *and* fac*).* Chaining the recursive facSum-rule (27) of Example 4.10 with the accelerated fac-rule (30) yields

$$\mathsf{facSum}(x) \xrightarrow{x} \{\mathsf{facSum}(x-1), \mathsf{fac}(1)\}\ [x > 1]\,.$$

Chaining this rule with the non-recursive fac-rule (29) results in

$$\mathsf{facSum}(x) \xrightarrow{x+1} \mathsf{facSum}(x-1)\ [x > 1]\,.$$

The iterated update and cost of this rule are $x\mu^{tv} = x - tv$ and

$$\textstyle\sum_{i=0}^{tv-1}(1 + x)\,\mu^i \quad = \quad \sum_{i=0}^{tv-1}(1 + x - i) \quad = \quad x \cdot tv - \frac{1}{2}tv^2 + \frac{3}{2}tv.$$

Thus, accelerating it via Theorem 3.8 with the metering function $x - 1$ results in

$$\mathsf{facSum}(x) \xrightarrow{x\cdot tv - \frac{1}{2}tv^2 + \frac{3}{2}tv} \mathsf{facSum}(x - tv)\ [0 < tv < x] \tag{31}$$

since $0 < tv < x$ implies $x > 1$. By instantiating tv with $x - 1$ and chaining (26) with the resulting rule, we obtain

$$\mathsf{f}_0(x) \xrightarrow{\frac{1}{2}x^2 + \frac{3}{2}x - 2} \mathsf{facSum}(1)\ [1 < x]\,. \tag{32}$$

Finally, by deleting all other rules, we obtain a simplified program.

Algorithm 2 shows how Algorithm 1 can be adapted in order to handle non-tail-recursive programs as well. The first additional step is Step 2, which deletes sinks (i.e., function symbols without any rules) from right-hand sides of non-tail-recursive rules α. This simplifies the rules and possibly even transforms them into tail-recursive rules.[14] Note that *Partial Deletion* adds a new rule α' with fewer terms in its right-hand side (thus, we then *delete* the original rule α afterwards).

The second change is that we apply *Partial Deletion* in Step 3.3 if we failed to accelerate a simple recursion that does not contain any temporary variables anymore. In this way, the degree of the simple recursion is reduced, which may simplify its acceleration as in Example 4.8. In our implementation, we first try all partial deletions which result in rules of degree 2 (if any). If we fail to

[14] If the rule is already tail-recursive, i.e., the right-hand side only contains a single term, and this term is a sink, then deleting this term does not help much to simplify the program. As mentioned in Footnote 3, we transform rules with empty right-hand side into rules with the right-hand side "sink" to simplify the formalization.

Algorithm 2 Program Simplification for Arbitrary Integer Programs

While there is a rule α with $\text{root}(\alpha) \neq \mathsf{f}_0$:

1. Apply *Deletion* to rules whose guard is proved unsatisfiable or whose root symbol is unreachable from f_0.
2. While there is a non-tail-recursive rule α whose right-hand side contains a symbol f without outgoing rules:
   2.1. Apply *Partial Deletion* to an occurrence of f in $\text{rhs}(\alpha)$ and apply *Deletion* to α afterwards.
3. While there is a simple recursion or a non-accelerated simple loop α:
   3.1. Try to *accelerate* α.
   3.2. If 3.1 succeeded, resulting in $\overline{\alpha}$, and α is a simple loop:
      3.2.1. Try to *instantiate* $\overline{\alpha}$ to eliminate the temporary variable introduced in Step 3.1.
      3.2.2. If 3.2.1 succeeded, apply *Deletion* to $\overline{\alpha}$.
   3.3. If 3.1 failed:
      If $\mathcal{TV}(\alpha) \neq \emptyset$, then try to *instantiate* α to eliminate a temporary variable.
      Otherwise, if the degree of α is greater than 1, then apply *Partial Deletion* to α.
   3.4. Apply *Deletion* to α.
4. Let $S = \emptyset$.
5. While there is an accelerated rule α:
   5.1. For each α' where $\text{rhs}(\alpha')$ contains $\text{root}(\alpha)$ and where $\text{root}(\alpha') \neq \text{root}(\alpha)$:
      Apply *Chaining* to α' and α and add α' to S.
   5.2. Apply *Deletion* to α.
6. Apply *Deletion* to each rule in S.
7. While there is a function symbol f without simple recursions or simple loops but with incoming and outgoing rules (starting with symbols f with just one incoming rule):
   7.1. Apply *Chaining* to each pair α', α where $\text{root}(\alpha) = f$ occurs in $\text{rhs}(\alpha')$.
   7.2. Apply *Deletion* to each α where $\text{root}(\alpha) = f$ or where $\text{rhs}(\alpha)$ contains no function symbol from Σ except f.

accelerate any of the resulting rules, then we also try all partial deletions which result in rules of degree 1.

Algorithm 2 terminates and thus it transforms any integer program into a simplified program: The loop in Step 2 reduces the degree of some rule in each iteration. The loop in Step 3 either reduces the number of non-accelerated loops or recursions, or it reduces the number of temporary variables or the degree of some rule in each iteration. In Step 5, the number of accelerated rules is decreasing. The loop in Step 7 terminates as it reduces the number of function symbols with outgoing rules in each iteration. Finally, the overall loop of Algorithm 2 terminates as well, because after having finished Step 7 the first time, the number of function symbols decreases in each further iteration of the algorithm. To see this, note that there is no simple loop or simple recursion anymore when the loop of Step 5 terminates. Thus, after finishing the loop of Step 7, the only function symbol is either f_0 (and hence the overall loop terminates) or there is at least one function symbol $f \neq \mathsf{f}_0$ without incoming or without outgoing rules. Thus, in the next iteration, this function symbol is removed. The reason is that if f has no incoming rules, then all rules α with $\text{root}(\alpha) = f$ are deleted in Step 1. If f has no outgoing rules, then all occurrences of f in right-hand sides are deleted in Step 2.

5 ASYMPTOTIC LOWER BOUNDS

After applying Algorithm 2, all programs are simplified and thus, we assume that $\mathcal{P}$ is a simplified program throughout this section. So all rules $\alpha \in \mathcal{P}$ have the same left-hand side $\mathrm{f}_0(\overline{x})$. Now for any integer substitution σ, the derivation height of $\mathrm{f}_0(\overline{x})\,\sigma$ in the simplified program $\mathcal{P}$ is

$$\mathrm{dh}_{\mathcal{P}}(\mathrm{f}_0(\overline{x})\,\sigma) \;=\; \max\{\mathrm{cost}(\alpha)\,\sigma \mid \alpha \in \mathcal{P}, \sigma \models \alpha\}, \tag{33}$$

i.e., (33) is the maximal cost of those rules whose guard is satisfied by σ. Thus, if $\mathcal{P}$ results from the transformation of an integer program $\widetilde{\mathcal{P}}$, then (33) is a lower bound on $\mathrm{dh}_{\widetilde{\mathcal{P}}}(\mathrm{f}_0(\overline{x})\,\sigma)$. So for the program in Figure 1b which was transformed into the simplified program with the only rule

$$\alpha_{0.\overline{1}.2.3.\overline{\overline{4}.5}}\colon\ \mathrm{f}_0(x,y,z,u) \xrightarrow{\frac{1}{8}x^4+\frac{1}{4}x^3+\frac{7}{8}x^2+\frac{7}{4}x} \mathrm{f}_2(0,\tfrac{1}{2}x^2+\tfrac{1}{2}x,1,0) \quad \left[\tfrac{1}{2}x^2+\tfrac{1}{2}x > 1\right], \tag{19}$$

we obtain the lower bound

$$\tfrac{1}{8}x^4 + \tfrac{1}{4}x^3 + \tfrac{7}{8}x^2 + \tfrac{7}{4}x \tag{34}$$

for all integer substitutions with $\sigma \models \frac{1}{2}x^2 + \frac{1}{2}x > 1$. However, in general such bounds do not provide an intuitive understanding of the program's complexity and they are also not suitable to detect possible attacks. The reason is that both $\mathrm{cost}(\alpha)$ and $\mathrm{guard}(\alpha)$ may be complicated and, even more importantly, they may be interdependent. Then, it is not sufficient to only regard the cost of a rule in order to draw conclusions on the resulting complexity. To see this, consider a simplified rule with cost tv and guard φ. Its complexity can, e.g., be unbounded if φ does not impose any bound on tv, exponential if φ only implies $tv \leq b$ for arithmetic expressions b that are exponential in the program variables, or constant if φ implies $tv \leq e$ for some $e \in \mathbb{N}$. But even without temporary variables, there can be subtle interdependencies between the cost and the guard of a transition, as the following example illustrates.

Example 5.1 (Sub-Linear Bounds). Let

$$\mathcal{P} = \left\{\mathrm{f}_0(x,y) \xrightarrow{y} \mathrm{f}(x,y) \;\left[x > y^2\right]\right\}.$$

The runtime complexity of this program is sub-linear, even though the cost function y is linear. The reason is that to achieve a linear increase of the cost y, a quadratic increase of x and thus of the input size is required. So in general, it is not correct to simply take the cost of a rule as a lower bound on its runtime (e.g., in this example we have $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(\sqrt{n})$, whereas "$\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n)$" would be incorrect).

Hence, we now show how to derive *asymptotic* lower bounds for simplified programs. These asymptotic bounds can easily be understood (i.e., a high lower bound can help programmers to improve their program to make it more efficient) and they identify potential attacks.

To derive asymptotic bounds, we use so-called *limit problems*, which are introduced in Section 5.1. A limit problem is an abstraction of the guard φ of a rule which allows us to analyze how to satisfy φ, presuming that all variables are instantiated with "large enough" values. More precisely, a solution of a limit problem is a *family* of substitutions σ_n which is parameterized by a variable n. This family of substitutions satisfies φ for large enough n and can be found using the calculus presented in Section 5.2. Thus, applying σ_n to the cost of a rule yields an expression that only contains the single variable n, even if the rule has a multivariate cost function. Hence, this allows us to deduce an asymptotic bound. In Section 5.3 we present an alternative approach to find solutions of limit problems via SMT solving which can be combined with the calculus of Section 5.2 in order to improve efficiency.

5.1 Limit Problems

While the derivation height $\mathrm{dh}_{\mathcal{P}}$ is defined on configurations like $\mathrm{f}_0(\overline{x})\,\sigma$, asymptotic bounds are usually defined for functions on $\mathbb{N}$ like the runtime complexity $\mathrm{rc}_{\mathcal{P}}$. Recall that according to Definition 2.8, $\mathrm{rc}_{\mathcal{P}}(n)$ is the maximal cost of any evaluation starting with a configuration $\mathrm{f}_0(\overline{n})$ where the size $|\overline{n}|$ of the input is at most n. Thus, our goal is to derive an asymptotic lower bound for $\mathrm{rc}_{\mathcal{P}}$ from a concrete bound on $\mathrm{dh}_{\mathcal{P}}$ like (34). So for the program $\mathcal{P}$ in (19), we would like to derive $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n^4)$. However, as discussed above, in general the cost (34) of a rule does not directly give rise to the desired asymptotic lower bound.

To infer an asymptotic lower bound from a rule $\alpha \in \mathcal{P}$, we try to find an infinite family of integer substitutions σ_n with $\mathcal{V}(\alpha) \subseteq \mathrm{dom}(\sigma_n)$ (parameterized by $n \in \mathbb{N}$) such that there is an $n_0 \in \mathbb{N}$ with $\sigma_n \models \mathrm{guard}(\alpha)$ for all $n \geq n_0$. Note that both $|\overline{x}\sigma_n|$ and $\mathrm{cost}(\alpha)\,\sigma_n$ are arithmetic expressions that only contain the single variable n, and we have $\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \in \Omega(\mathrm{cost}(\alpha)\,\sigma_n)$, since for all $n \geq n_0$ we obtain

$$\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \geq \mathrm{dh}_{\mathcal{P}}(\mathrm{f}_0(\overline{x})\,\sigma_n) \geq \mathrm{cost}(\alpha)\,\sigma_n.$$

For the program $\mathcal{P}$ containing only the rule

$$\alpha_{0.\overline{1}.2.3.\overline{\overline{4}.5}}\colon\ \mathrm{f}_0(x,y,z,u) \xrightarrow{\frac{1}{8}x^4+\frac{1}{4}x^3+\frac{7}{8}x^2+\frac{7}{4}x} \mathrm{f}_2(0,\tfrac{1}{2}x^2+\tfrac{1}{2}x,1,0) \quad \left[\tfrac{1}{2}x^2+\tfrac{1}{2}x > 1\right], \tag{19}$$

our approach will infer the family σ_n with

$$x\sigma_n = n \quad \text{and} \quad y\sigma_n = z\sigma_n = u\sigma_n = 0. \tag{35}$$

So for $\overline{x}$ consisting of x, y, z, u, this implies

$$\begin{array}{rcl} \mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) & = & \mathrm{rc}_{\mathcal{P}}(|x\sigma_n| + |y\sigma_n| + |z\sigma_n| + |u\sigma_n|) \\ & = & \mathrm{rc}_{\mathcal{P}}(|n|) \\ & \geq & \mathrm{cost}(\alpha_{0.\overline{1}.2.3.\overline{\overline{4}.5}})\,\sigma_n \\ & = & (34)\,\sigma_n \\ & = & \frac{1}{8}n^4 + \frac{1}{4}n^3 + \frac{7}{8}n^2 + \frac{7}{4}n. \end{array}$$

To find such a family of substitutions σ_n, we first normalize all constraints in $\mathrm{guard}(\alpha)$ such that they have the form $a > 0$ or $a \geq 0$.[15] Now we search for substitutions σ_n such that for large enough $n \in \mathbb{N}$, σ_n is a model for a formula of the form "$\bigwedge_{i=1}^{k}(a_i \circ_i 0)$" where $\circ_i \in \{>, \geq\}$. Obviously, such a formula is satisfied for large enough n if all expressions $a_i\sigma_n$ are positive constants or increase infinitely towards ω. Thus, we introduce a technique which tries to find out whether fixing the valuations of some variables and increasing or decreasing the valuations of others results in positive resp. increasing valuations of $a_1, \ldots, a_k$. Our technique operates on *limit problems* of the form $\{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}\}$ where a_i is an arithmetic expression and $\bullet_i \in \{+, -, +_!, -_!\}$ for all $1 \leq i \leq k$. Here, a^+ (resp. a^-) means that a has to grow towards ω (resp. $-\omega$) and $a^{+_!}$ (resp. $a^{-_!}$) means that a has to be a positive (resp. negative) constant. So we represent $\mathrm{guard}(\alpha)$ by an *initial limit problem* $\{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}\}$ where $\bullet_i \in \{+, +_!\}$ for all $1 \leq i \leq k$. By choosing $\bullet_i$ to be $+$ or $+_!$, $a_i^{\bullet_i}$ means that a_i grows towards ω or it is a positive constant, i.e., this ensures that $a_i > 0$ holds for large enough n. Our implementation leaves the choice of the $\bullet_i \in \{+, +_!\}$ in the initial limit problem open as long as possible and only fixes it when this is needed in order to *solve* the limit problem (see the end of Section 5.2 for further details). To solve a limit problem L, we search for a *solution* σ_n of L, where this concept is defined in terms of *limits* of functions.

[15] Note that while the variables range over $\mathbb{Z}$, there may be non-integer expressions in $\mathrm{guard}(\alpha)$ which result from non-integer metering functions. Thus, we allow both constraints of the form $a > 0$ and $a \geq 0$ in normalized guards, since transforming $a > 0$ to $a - 1 \geq 0$ would be incorrect in general.

Definition 5.2 (Limit). For each $f : \mathbb{N} \to \mathbb{R}$ we have $\lim_{n\mapsto\omega} f(n) = \omega$ (resp. $\lim_{n\mapsto\omega} f(n) = -\omega$) if for every $m \in \mathbb{R}$ there is an $n_0 \in \mathbb{N}$ such that $f(n) \geq m$ (resp. $f(n) \leq m$) holds for all $n \geq n_0$. Similarly, we have $\lim_{n\mapsto\omega} f(n) = m$ if for every $\varepsilon \in \mathbb{R}$ with $\varepsilon > 0$ there is an $n_0 \in \mathbb{N}$ such that $|f(n) - m| < \varepsilon$ holds for all $n \geq n_0$.

Now a family of substitutions σ_n is a solution for a limit problem $\{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}\}$ if $\lim_{n\mapsto\omega} a_i\sigma_n$ complies with $\bullet_i$ for each $1 \leq i \leq k$.

Definition 5.3 (Solutions of Limit Problems). For any function $f : \mathbb{N} \to \mathbb{R}$ and any $\bullet \in \{+, -, +_!, -_!\}$, we say that f *satisfies* $\bullet$ if:

$$\begin{array}{lll} \lim_{n\mapsto\omega} f(n) = \ \ \omega, & \text{if } \bullet = + & \exists m \in \mathbb{R}.\ \lim_{n\mapsto\omega} f(n) = m > 0, \quad \text{if } \bullet = +_! \\ \lim_{n\mapsto\omega} f(n) = -\omega, & \text{if } \bullet = - & \exists m \in \mathbb{R}.\ \lim_{n\mapsto\omega} f(n) = m < 0, \quad \text{if } \bullet = -_! \end{array}$$

A family σ_n of integer substitutions with $\mathcal{V}(L) \subseteq \mathrm{dom}(\sigma_n)$ is a *solution* of a limit problem L if for every $a^\bullet \in L$, the function $\lambda n.\ a\sigma_n$ satisfies $\bullet$. For any arithmetic expression a with $\mathcal{V}(a) \subseteq \{n\}$, "$\lambda n.\ a$" is the function from $\mathbb{N} \to \mathbb{R}$ that maps any $n \in \mathbb{N}$ to the value of a.

Example 5.4 (Solution of the Limit Problem of the Program (19)*).* The program (19) has the guard $\frac{1}{2}x^2 + \frac{1}{2}x > 1$ which normalizes to $\frac{1}{2}x^2 + \frac{1}{2}x - 1 > 0$. Hence, the resulting initial limit problem could be $\{(\frac{1}{2}x^2 + \frac{1}{2}x - 1)^+\}$. It is solved by the family of substitutions σ_n from (35). The reason is that $\lim_{n\mapsto\omega}(\lambda n.\ (\frac{1}{2}x^2 + \frac{1}{2}x - 1)\,\sigma_n) = \omega$, i.e., the function $\lambda n.\ (\frac{1}{2}x^2 + \frac{1}{2}x - 1)\,\sigma_n$ satisfies $+$. Thus, there is an n_0 such that $\sigma_n \models \mathrm{guard}(\alpha_{0.\overline{1}.2.3.\overline{4}.5})$ holds for all $n \geq n_0$.

In Sections 5.2 and 5.3 we will show how to infer such solutions of limit problems automatically. The following theorem clarifies how to deduce an asymptotic lower bound from a solution of a limit problem.

Theorem 5.5 (Asymptotic Bounds for Simplified Programs). *Given a rule α of a simplified program $\mathcal{P}$ with the program variables $\overline{x}$ and* $\mathrm{guard}(\alpha) = (a_1 \circ_1 0) \wedge \cdots \wedge (a_k \circ_k 0)$ *where* $\circ_1, \ldots, \circ_k \in \{>, \geq\}$*, let the family σ_n be a solution of an initial limit problem $\{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}\}$ with* $\bullet_1, \ldots, \bullet_k \in \{+, +_!\}$*. Then* $\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \in \Omega(\mathrm{cost}(\alpha)\,\sigma_n)$.

Proof. Since σ_n is a solution of $\{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}\}$, there is an $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, we have $\sigma_n \models a_1 > 0 \wedge \cdots \wedge a_k > 0$, which implies $\sigma_n \models \mathrm{guard}(\alpha)$. Hence, for all $n \geq n_0$, we obtain:

$$\begin{array}{lll} \mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) & \geq & \mathrm{dh}_{\mathcal{P}}(\mathrm{lhs}(\alpha)\,\sigma_n) \\ & \geq & \mathrm{cost}(\alpha)\,\sigma_n \qquad\qquad \text{as } \sigma_n \models \mathrm{guard}(\alpha) \end{array}$$

This implies $\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \in \Omega(\mathrm{cost}(\alpha)\,\sigma_n)$. □

Of course, if $\mathcal{P}$ has several rules, then we try to take the one which results in the highest lower bound.

Example 5.6 (Asymptotic Bound for the Program (19)*).* We continue Example 5.4 with the program $\mathcal{P} = \{(19)\}$. For $\overline{x} = (x, y, z, u)$, according to Theorem 5.5, we get the asymptotic lower bound

$$\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \ \in \ \Omega(\mathrm{cost}((19))\,\sigma_n). \tag{36}$$

Note that $\mathrm{cost}((19))\,\sigma_n = \frac{1}{8}n^4 + \frac{1}{4}n^3 + \frac{7}{8}n^2 + \frac{7}{4}n$. Hence, (36) is equivalent to

$$\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \ \in \ \Omega(n^4).$$

Up to now, we only took the guard $\bigwedge_{i=1}^{k}(a_i \circ_i 0)$ of a rule α into account in the initial limit problem $\{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}\}$. This has the disadvantage that solutions of this limit problem do not necessarily try to maximize the cost of the rule. For example, for the rule

$$f_0(x, y) \xrightarrow{x \cdot y} f(0, y)\ [x > 0]\,,$$

we would obtain the initial limit problem $\{x^+\}$ which is solved by the family of substitutions $\sigma_n = \{x/n, y/0\}$. According to Theorem 5.5, this only allows us to infer $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(\mathrm{cost}(\alpha)\sigma_n)$, where $\mathrm{cost}(\alpha)\sigma_n = 0$, i.e., it only allows us to infer a constant lower bound. To obtain non-trivial lower bounds instead, one should extend the initial limit problem $\{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}\}$ of a rule α by $\mathrm{cost}(\alpha)^+$. In this way, one searches for families of substitutions σ_n where $\mathrm{cost}(\alpha)\,\sigma_n$ grows towards ω, i.e., where $\mathrm{cost}(\alpha)\,\sigma_n$ depends on n and is not constant. So in our example, we should start with the initial limit problem $\{x^+, (x \cdot y)^+\}$ which has the solution $\sigma_n = \{x/n, y/n\}$. By Theorem 5.5, one now obtains the quadratic lower bound $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n^2)$, since $\mathrm{cost}(\alpha)\sigma_n = n^2$.

The costs are *unbounded* (i.e., they depend on temporary variables) if the initial limit problem $\{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}, \mathrm{cost}(\alpha)^+\}$ has a solution σ_n where $x\sigma_n$ is constant for all program variables x. Then we can even infer $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(\omega)$.

Example 5.7 (Unbounded Loops Continued). By chaining the initial rule $f_0(x, y) \xrightarrow{0} f(x, y)$ of the program from Example 3.6 with the accelerated rule

$$f(x, y) \xrightarrow{tv_1 \cdot y} f(x + tv_1, y)\ [0 < x \wedge 0 < tv_1] \tag{18}$$

from Example 3.11, we obtain

$$f_0(x, y) \xrightarrow{tv_1 \cdot y} f(x + tv_1, y)\ [0 < x \wedge 0 < tv_1]\,.$$

The resulting initial limit problem $\{x^{+_1}, tv_1^+, (tv_1 \cdot y)^+\}$ has the solution $\sigma_n = \{x/1,\ y/1,\ tv_1/n\}$, which implies $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(\omega)$.

Theorem 5.5 results in bounds "$\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \in \Omega(\mathrm{cost}(\alpha)\,\sigma_n)$" which depend on the sizes $|\overline{x}\sigma_n|$. Let $f(n) = \mathrm{rc}_{\mathcal{P}}(n)$, $g(n) = |\overline{x}\sigma_n|$, and let $\Omega(\mathrm{cost}(\alpha)\,\sigma_n)$ have the form $\Omega(n^k)$ or $\Omega(k^n)$ for some $k \in \mathbb{N}$. Moreover for all $x \in \overline{x}$, let $x\sigma_n$ be a polynomial of at most degree d, i.e., let $g(n) \in \mathcal{O}(n^d)$. Then, based on an observation from [30],[16] we can infer a lower bound for $f(n) = \mathrm{rc}_{\mathcal{P}}(n)$ instead of $f(g(n)) = \mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|)$. Moreover, if $g(n) = |\overline{x}\sigma_n|$ is constant whereas $\Omega(\mathrm{cost}(\alpha)\,\sigma_n)$ is not constant, then the lemma allows us to infer that $f(n) = \mathrm{rc}_{\mathcal{P}}(n) \in \Omega(\omega)$, as in Example 5.7.

Lemma 5.8 (Bounds for Function Composition). *Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0} \cup \{\omega\}$ and $g : \mathbb{N} \to \mathbb{N}$ where $g(n) \in \mathcal{O}(n^d)$ for some $d \in \mathbb{N}$ with $d > 0$. Moreover, let $f(n)$ be weakly monotonically increasing for large enough n.*

- *(a) If $g(n)$ is strictly monotonically increasing for large enough n and $f(g(n)) \in \Omega(n^k)$ with $k \in \mathbb{N}$, then $f(n) \in \Omega(n^{\frac{k}{d}})$.*
- *(b) If $g(n)$ is strictly monotonically increasing for large enough n and $f(g(n)) \in \Omega(k^n)$ with $k > 1$, then $f(n) \in \Omega(e^{\sqrt[d]{n}})$ for some number $e \in \mathbb{R}$ with $e > 1$.*
- *(c) If $g(n) \in \mathcal{O}(1)$ and $f(g(n)) \notin \mathcal{O}(1)$, then $f(n) \in \Omega(\omega)$.*

Example 5.9 (Asymptotic Bound for Program (19) Continued). In Example 5.6, for $\overline{x} = (x, y, z, u)$, we inferred $\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \in \Omega(n^4)$ where $x\sigma_n = n$ and $y\sigma_n = z\sigma_n = u\sigma_n = 0$. Hence, we have

[16] In the second case of Lemma 5.8, we fix a small inaccuracy from [29] where we inadvertently wrote $f(n) \in \Omega(k^{\sqrt[d]{n}})$. Since Lemma 5.8 is very similar to Lemma 24 from our paper [30], we omit its proof here. The proof can be found in Appendix A.

$|\overline{x}\sigma_n| = |n| = n \in \mathcal{O}(n^1)$. By Lemma 5.8(a), we obtain $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n^{\frac{4}{1}}) = \Omega(n^4)$ for the program $\mathcal{P} = \{(19)\}$. Due to the soundness of the processors for program simplification in Section 3, we also have $\mathrm{rc}_{\widetilde{\mathcal{P}}}(n) \in \Omega(n^4)$ for the program $\widetilde{\mathcal{P}}$ in Figure 1b.

In cases like Example 5.1, Lemma 5.8 even allows us to infer sub-linear bounds.

Example 5.10 (Example 5.1 Continued). Reconsider the program from Example 5.1. By Definition 5.3, the family σ_n with $x\sigma_n = n^2 + 1$ and $y\sigma_n = n$ is a solution of the initial limit problem $\{(x - y^2)^{+_!}, y^+\}$. (Our implementation chooses $(x - y^2)^{+_!}$ instead of $(x - y^2)^+$ in the initial limit problem, because in this way, the limit problem can be solved by our technique, see Section 5.2.) Due to Theorem 5.5, this proves $\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \in \Omega(n)$ for the program variables $\overline{x} = (x, y)$. As $|\overline{x}\sigma_n| = n^2 + 1 + n \in \mathcal{O}(n^2)$, Lemma 5.8(a) results in $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n^{\frac{1}{2}}) = \Omega(\sqrt{n})$.

If the cost of the rule from Example 5.1 was 2^y, then σ_n would still be a solution of the initial limit problem $\{(x - y^2)^{+_!}, (2^y)^+\}$. So we would obtain $\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \in \Omega(2^n)$ due to Theorem 5.5 and thus $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(e^{\sqrt{n}})$ for an $e > 1$ due to Lemma 5.8(b). Intuitively, the exponent $\sqrt{n}$ expresses that the cost grows exponentially w.r.t. y, where the guard $x > y^2$ implies $|y| \in \mathcal{O}(\sqrt{|\overline{x}|})$, i.e., y is bounded by the square root of the input size.

The reason why we cannot specify e in Lemma 5.8(b) is that it depends on the coefficients of $g(n) = |\overline{x}\sigma_n|$, but Lemma 5.8 only requires $g(n) \in \mathcal{O}(n^d)$. Thus, a variant of Lemma 5.8 where the polynomial g is known would allow us to compute e. Our implementation simply reports that the runtime is at least exponential if Lemma 5.8(b) applies and $d = 1$.

5.2 Transforming Limit Problems

In order to use Theorem 5.5 (and Lemma 5.8) for the automatic inference of lower bounds, we still have to show how to find a family of substitutions σ_n automatically that solves the initial limit problem of a program's rule.

A limit problem L is *trivial* if all expressions in L are variables and there is no variable x with $x^{\bullet_1}, x^{\bullet_2} \in L$ and $\bullet_1 \neq \bullet_2$. For trivial limit problems L we can immediately obtain a particular solution σ_n^L which instantiates variables "according to L".

Lemma 5.11 (Solving Trivial Limit Problems). *Let L be a trivial limit problem. Then σ_n^L is a solution of L where for all $n \in \mathbb{N}$, σ_n^L is defined as follows:*

$$x\sigma_n^L = \begin{cases} n & \text{if } x^+ \in L \\ -n & \text{if } x^- \in L \\ 1 & \text{if } x^{+_!} \in L \\ -1 & \text{if } x^{-_!} \in L \\ 0 & \text{otherwise} \end{cases}$$

Proof. If $x^+ \in L$ (resp. $x^- \in L$), then $x\sigma_n^L = n$ (resp. $x\sigma_n^L = -n$) and thus, $\lim_{n\mapsto\omega} x\sigma_n = \lim_{n\mapsto\omega} n = \omega$ (resp. $\lim_{n\mapsto\omega} x\sigma_n = \lim_{n\mapsto\omega} -n = -\omega$), i.e., $\lambda n.\, x\sigma_n$ satisfies + (resp. −). If $x^{+_!} \in L$ (resp. $x^{-_!} \in L$), then $x\sigma_n^L = 1$ (resp. $x\sigma_n^L = -1$). Thus, $\lim_{n\mapsto\omega} x\sigma_n = 1$ (resp. $\lim_{n\mapsto\omega} x\sigma_n = -1$), i.e., $\lambda n.\, x\sigma_n$ satisfies $+_!$ (resp. $-_!$). Hence, σ_n^L is a solution of L. □

For instance, if $\mathcal{V}(\alpha) = \{x, y, tv\}$ and $L = \{x^+, y^{-_!}\}$, then L is a trivial limit problem and σ_n^L with $x\sigma_n^L = n, y\sigma_n^L = -1$, and $tv\,\sigma_n^L = 0$ is a solution for L.

However, in general the initial limit problem $L = \{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}, \text{cost}(\alpha)^+\}$ is not trivial. Therefore, we now define a transformation $\rightsquigarrow$ to simplify limit problems until one reaches a trivial problem. With our transformation, $L \rightsquigarrow L'$ ensures that each solution of L' also gives rise to a solution of L.

If L contains $f(a_1, a_2)^\bullet$ for some standard arithmetic operation f like addition, subtraction, multiplication, division, or exponentiation, we use a so-called *limit vector* $(\bullet_1, \bullet_2)$ with $\bullet_i \in \{+, -, +_!, -_!\}$ to characterize for which kinds of arguments the operation f is increasing (if $\bullet = +$), decreasing (if $\bullet = -$), or a positive or negative constant (if $\bullet = +_!$ or $\bullet = -_!$).[17] Then L can be transformed into the new limit problem $(L \setminus \{f(a_1, a_2)^\bullet\}) \cup \{a_1^{\bullet_1}, a_2^{\bullet_2}\}$.

For example, $(+, +_!)$ and $(+, -_!)$ are increasing limit vectors for subtraction. The reason is that $a_1 - a_2$ is increasing if a_1 is increasing and a_2 is a constant. Hence, our transformation $\rightsquigarrow$ allows us to replace $(a_1 - a_2)^+$ by a_1^+ and $a_2^{+_!}$ or by a_1^+ and $a_2^{-_!}$.

Definition 5.12 (Limit Vectors). Let $f : \mathbb{R} \to \mathbb{R}$ be a function and let $\bullet_1, \bullet_2 \in \{+, -, +_!, -_!\}$. We say that $(\bullet_1, \bullet_2)$ is an *increasing* (resp. *decreasing*) *limit vector* for f if the function $\lambda n.\ f(g(n), h(n))$ satisfies $+$ (resp. $-$) for any functions g and h that satisfy $\bullet_1$ and $\bullet_2$, respectively. Similarly, $(\bullet_1, \bullet_2)$ is a *positive* (resp. *negative*) *limit vector* for f if $\lambda n.\ f(g(n), h(n))$ satisfies $+_!$ (resp. $-_!$) for any functions g and h that satisfy $\bullet_1$ and $\bullet_2$, respectively.

With this definition, $(+, +_!)$ and $(+, -_!)$ are indeed an increasing limit vectors for subtraction, since $\lim_{n\mapsto\omega} g(n) = \omega$ and $\lim_{n\mapsto\omega} h(n) = m$ with $m > 0$ or $m < 0$ implies $\lim_{n\mapsto\omega}(g(n) - h(n)) = \omega$. In other words, if $g(n)$ satisfies $+$ and $h(n)$ satisfies $+_!$ or $-_!$, then $g(n) - h(n)$ satisfies $+$ as well. In contrast, $(+, +)$ is not an increasing limit vector for subtraction. To see this, consider the functions $g(n) = h(n) = n$. Both $g(n)$ and $h(n)$ satisfy $+$, whereas $g(n) - h(n) = 0$ does not satisfy $+$. Similarly, $(+_!, +_!)$ is not a positive limit vector for subtraction, since for $g(n) = 1$ and $h(n) = 2$, both $g(n)$ and $h(n)$ satisfy $+_!$, but $g(n) - h(n) = -1$ does not satisfy $+_!$.

Limit vectors can be used to simplify limit problems, as in (A) in the following definition. Moreover, for numbers $m \in \mathbb{R}$, one can easily simplify constraints of the form $m^{+_!}$ and $m^{-_!}$ (e.g., $2^{+_!}$ is obviously satisfied since $2 > 0$), as in (B).

Definition 5.13 ($\rightsquigarrow$). Let L be a limit problem. We have:

(A) $L \cup \{f(a_1, a_2)^\bullet\} \rightsquigarrow L \cup \{a_1^{\bullet_1}, a_2^{\bullet_2}\}$ if $\bullet$ is $+$ (resp. $-$, $+_!$, $-_!$) and $(\bullet_1, \bullet_2)$ is an increasing (resp. decreasing, positive, negative) limit vector for f

(B) $L \cup \{m^{+_!}\} \rightsquigarrow L$ if $m \in \mathbb{R}$ and $m > 0$, $L \cup \{m^{-_!}\} \rightsquigarrow L$ if $m \in \mathbb{R}$ and $m < 0$

However, transforming a limit problem with $\rightsquigarrow$ may also result in *contradictory* limit problems that contain $x^{\bullet_1}$ and $x^{\bullet_2}$ where $\bullet_1 \neq \bullet_2$, as the following example illustrates.

Example 5.14 (Contradictory Limit Problems – Example 5.10 Continued). The initial limit problem $\{(x - y^2)^{+_!}, y^+\}$ from Example 5.10 cannot be solved with the current transformation rules. While $(+, +_!)$ and $(+, -_!)$ are *increasing* limit vector for subtraction, the only *positive* limit vector for subtraction is $(+_!, -_!)$. Thus, by (A) one obtains $\{(x - y^2)^{+_!}, y^+\} \rightsquigarrow \{x^{+_!}, (y^2)^{-_!}, y^+\}$ which contains the unsolvable requirement $(y^2)^{-_!}$.

[17] To ease the presentation, we restrict ourselves to binary operations f. For operations of arity k, one would need limit vectors of the form $(\bullet_1, \ldots, \bullet_k)$.

As an alternative, in Example 5.10 one could regard the initial limit problem $\{(x - y^2)^+, y^+\}$ instead. However, here the transformation rules fail as well. We have

$$\begin{array}{ll} & \{(x - y^2)^+, y^+\} \\ \rightsquigarrow & \{x^+, (y^2)^{+_!}, y^+\} \quad \text{by (A) with the increasing limit vector } (+, +_!) \text{ for subtraction} \\ \rightsquigarrow & \{x^{+_!}, y^{+_!}, y^+\} \quad \text{by (A) with the positive limit vector } (+_!, +_!) \text{ for multiplication} \end{array}$$

However, the resulting problem is contradictory, as it contains both $y^{+_!}$ and y^+.

Recall that the guard of the rule from Example 5.1 implies $x \geq y^2 + 1$. If we substitute x with its upper bound $y^2 + 1$ in the beginning, then we can reduce the initial limit problem $\{(x - y^2)^{+_!}, y^+\}$ to a trivial one. Hence, we now extend $\rightsquigarrow$ by allowing to apply substitutions.

Definition 5.15 ($\rightsquigarrow$ Continued). Let L be a limit problem and let θ be a substitution such that $x \notin \mathcal{V}(x\theta)$ for all $x \in \mathrm{dom}(\theta)$ and $\theta \circ \sigma$ is an integer substitution for each integer substitution σ whose domain includes all variables occurring in the range of θ. Then we have:[18]

(C) $L \overset{\theta}{\rightsquigarrow} L\theta$

Example 5.16 (Applying Substitutions to Limit Problems – Example 5.14 Continued). For the initial limit problem $\{(x - y^2)^{+_!}, y^+\}$ from Example 5.10, we now have

$$\begin{array}{lll} \{(x - y^2)^{+_!}, y^+\} & \overset{\{x/y^2+1\}}{\rightsquigarrow} & \{1^{+_!}, y^+\} \\ & \rightsquigarrow & \{y^+\} \end{array}$$

i.e., we obtain the trivial limit problem $\{y^+\}$. Note that, given an integer substitution σ with $y \in \mathrm{dom}(\sigma)$, $\{x/y^2 + 1\} \circ \sigma$ is an integer substitution as well. By Lemma 5.11, the family $\sigma_n^{\{y^+\}}$ with $y\sigma_n^{\{y^+\}} = n$ solves the resulting trivial limit problem $\{y^+\}$. To obtain a solution for the initial limit problem $\{(x - y^2)^{+_!}, y^+\}$ one has to take the substitution $\{x/y^2 + 1\}$ into account that was used in its transformation. In this way, we get the solution $\sigma_n = \{x/y^2 + 1\} \circ \sigma_n^{\{y^+\}}$ for the initial limit problem where $x\,\sigma_n = n^2 + 1$ and $y\,\sigma_n = n$. Thus, we obtain $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n^{\frac{1}{2}}) = \Omega(\sqrt{n})$, as in Example 5.10.

Although Definition 5.15 requires that $\theta \circ \sigma$ is an integer substitution whenever σ is an integer substitution, it is also useful to handle limit problems which contain expressions that do not evaluate to integer numbers.

Example 5.17 (Non-Integer Metering Functions Continued). By chaining[19] the only initial rule of the program in Example 3.10 with the accelerated rule (17), we obtain

$$\mathrm{f}_0(x) \quad \xrightarrow{tv} \quad \mathrm{f}(x - 2tv) \quad \left[0 < tv < \tfrac{1}{2}x + 1\right]. \tag{37}$$

For the initial limit problem $\{tv^+, (\frac{1}{2}x + 1 - tv)^{+_!}\}$ we get

$$\begin{array}{lll} \{tv^+, (\frac{1}{2}x + 1 - tv)^{+_!}\} & \overset{\{x/2\,tv-1\}}{\rightsquigarrow} & \{tv^+, \frac{1}{2}^{+_!}\} \\ & \rightsquigarrow & \{tv^+\} \end{array}$$

by (C) and (B). (Our implementation first leaves it open whether to choose $(\frac{1}{2}x + 1 - tv)^+$ or $(\frac{1}{2}x + 1 - tv)^{+_!}$, but when transforming the arithmetic expression to $\frac{1}{2}$ by (C), it finds out that one should use $(\frac{1}{2}x + 1 - tv)^{+_!}$ in order to solve the limit problem. We describe the strategy used

[18] The other rules for $\rightsquigarrow$ are implicitly labeled with the identical substitution $\emptyset$.

[19] Note that we cannot instantiate tv with the metering function that was used to accelerate the loop from Example 3.10, as it does not map to the integers, i.e., the prerequisites of Theorem 3.12 are not satisfied.

by our implementation and its heuristic to find suitable substitutions θ for the application of rule (C) at the end of this subsection.) By Lemma 5.11, the family $\sigma_n^{\{tv^+\}}$ with $tv\,\sigma_n^{\{tv^+\}} = n$ solves the resulting trivial limit problem $\{tv^+\}$. Again, to obtain a solution for the original initial limit problem $\{tv^+, (\frac{1}{2}x + 1 - tv)^{+_!}\}$ one has to take the substitution $\{x/2\,tv - 1\}$ into account, resulting in σ_n with $x\sigma_n = 2\,n - 1$ and $tv\,\sigma_n = n$. Thus, by Theorem 5.5 we have $\mathrm{rc}_{\{(37)\}}(|x\sigma_n|) = \mathrm{rc}_{\{(37)\}}(2\,n-1) \in \Omega(\mathrm{cost}((37))\,\sigma_n) = \Omega(tv\,\sigma_n) = \Omega(n)$. As $2\,n - 1 \in O(n)$, Lemma 5.8(a) implies $\mathrm{rc}_{\{(37)\}}(n) \in \Omega(n)$. By the soundness of the processors for program simplification in Section 3, we also have $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n)$ for the original program in Example 3.10.

Definition 5.15 requires that $\theta \circ \sigma$ is an integer substitution for every integer substitution σ whose domain includes all variables occurring in the range of θ. To check this side-condition automatically, one can again use Lemma 3.15: If the range of θ consists of polynomials, then for every $x \in \mathrm{dom}(\theta)$ we only have to check if instantiating the polynomial $x\,\theta$ by finitely many suitable integers again results in an integer. More precisely, if $x\,\theta$ contains the variables $x_1, \ldots, x_k$ of degrees $d_1, \ldots, d_k$, respectively, we check if $x\,\theta$ maps all arguments from $\{0, \ldots, d_1 + 1\} \times \ldots \times \{0, \ldots, d_k + 1\}$ to integers.

However, up to now we cannot prove that, e.g., a rule α with $\mathrm{guard}(\alpha) = x^2 - x > 0$ and $\mathrm{cost}(\alpha) = x$ has a linear lower bound, since $(+, +)$ is not an increasing limit vector for subtraction. To handle such cases, the following transformation rules allow us to neglect polynomial sub-expressions if they are "dominated" by other polynomials of higher degree or by exponential sub-expressions.

Definition 5.18 ($\rightsquigarrow$ Continued). Let L be a limit problem, let $\pm \in \{+, -\}$, and let a, b, c be univariate polynomials over the same variable. Then we have:

(D) $L \cup \{(a \pm b)^+\} \rightsquigarrow L \cup \{a^+\}$ and $L \cup \{(a \pm b)^-\} \rightsquigarrow L \cup \{a^-\}$ if the degree of a is greater than the degree of b

(E) $L \cup \{(a^c \pm b)^+\} \rightsquigarrow L \cup \{(a-1)^+, c^+\}$ and $L \cup \{(a^c \pm b)^+\} \rightsquigarrow L \cup \{(a-1)^+, c^{+_!}\}$

Thus, $\{(x^2 - x)^+\} \rightsquigarrow \{(x^2)^+\} = \{(x \cdot x)^+\} \rightsquigarrow \{x^+\}$ by (D) and (A) with the increasing limit vector $(+, +)$ for multiplication. The intuition for (E) is that any exponential expression a^c dominates any polynomial expression b provided that the base a is greater than 1 and the exponent c grows towards ω.

Example 5.19 (Example 4.6 Continued). We continue Example 4.6, where the Fibonacci program was simplified to the program consisting just of the rule

$$\mathrm{f}_0(x) \xrightarrow{2^{\frac{1}{2}x-1}-1} \emptyset \;\; [x > 1]\,. \tag{25}$$

Here, we obtain the initial limit problem $\{(x-1)^+, (2^{\frac{1}{2}x-1} - 1)^+\}$. We get:

$$\begin{array}{lll}
 & \{(x-1)^+, (2^{\frac{1}{2}x-1} - 1)^+\} & \\
\rightsquigarrow & \{x^+, (2^{\frac{1}{2}x-1} - 1)^+\} & \text{by (D)} \\
\rightsquigarrow & \{x^+, 1^{+_!}, (\frac{1}{2}x - 1)^+\} & \text{by (E)} \\
\rightsquigarrow & \{x^+, 1^{+_!}, (\frac{1}{2}x)^+\} & \text{by (A) with the increasing limit vector } (+, +_!) \text{ for subtraction} \\
\rightsquigarrow & \{x^+, 1^{+_!}, (\frac{1}{2})^{+_!}\} & \text{by (A) with the increasing limit vector } (+_!, +) \text{ for multiplication} \\
\rightsquigarrow^2 & \{x^+\} & \text{by (B) (twice)}
\end{array}$$

By Lemma 5.11, the family $\sigma_n^{\{x^+\}}$ with $x\sigma_n^{\{x^+\}} = n$ solves the resulting trivial limit problem $\{x^+\}$ and hence, it also solves the initial limit problem of Rule (25). Thus, Theorem 5.5 implies $\mathrm{rc}_{\{(25)\}}(|x\sigma_n|) = \mathrm{rc}_{\{(25)\}}(n) \in \Omega(\mathrm{cost}((25))\,\sigma_n) = \Omega(2^{\frac{1}{2}n-1} - 1) = \Omega(2^{\frac{1}{2}n}) = \Omega(\sqrt{2}^n) \subset \Omega(1.4^n)$. Due

(A) $L \cup \{f(a_1, a_2)^\bullet\} \rightsquigarrow L \cup \{a_1^{\bullet_1}, a_2^{\bullet_2}\}$ if $\bullet$ is $+$ (resp. $-, +_!, -_!$) and $(\bullet_1, \bullet_2)$ is an increasing (resp. decreasing, positive, negative) limit vector for f
(B) $L \cup \{m^{+_!}\} \rightsquigarrow L$ if $m \in \mathbb{R}$ and $m > 0$, $L \cup \{m^{-_!}\} \rightsquigarrow L$ if $m \in \mathbb{R}$ and $m < 0$
(C) $L \overset{\theta}{\rightsquigarrow} L\theta$ if $x \notin \mathcal{V}(x\theta)$ for all $x \in \mathrm{dom}(\theta)$, and if σ is an integer substitution with $\mathcal{V}(\mathrm{range}(\theta)) \subseteq \mathrm{dom}(\sigma)$, then $\theta \circ \sigma$ is also an integer substitution
(D) $L \cup \{(a \pm b)^\bullet\} \rightsquigarrow L \cup \{a^\bullet\}$ if $\bullet \in \{+, -\}$ and $\mathrm{degree}(a) > \mathrm{degree}(b)$ for the univariate polynomials a, b
(E) $L \cup \{(a^c \pm b)^+\} \rightsquigarrow L \cup \{(a-1)^\bullet, c^+\}$ if $\bullet \in \{+, +_!\}$ for the univariate polynomials a, b, c

Fig. 2. Definition of our Transformation for a Limit Problem L

to the soundness of the program simplification in Section 3 and Section 4, this also implies that the runtime complexity of the original Fibonacci program $\mathcal{P}$ from Example 2.1 is exponential, i.e., $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(\sqrt{2}^n)$.

Note that (E) can also be used to handle limit problems like $(a^c)^+$ (by choosing $b = 0$). We summarize the full definition of our transformation $\rightsquigarrow$ of limit problems in Figure 2. Theorem 5.20 states that our transformation $\rightsquigarrow$ is indeed correct. As already illustrated in Examples 5.16 and 5.17, when constructing the solution from the resulting trivial limit problem, one has to take the substitutions into account which were used in the derivation.

Theorem 5.20 (Correctness of $\rightsquigarrow$). *If $L \overset{\theta}{\rightsquigarrow} L'$ and the family σ_n is a solution of L', then $\theta \circ \sigma_n$ is a solution of L.*

Proof. First assume that the step from L to L' was done by Definition 5.13 (A). Since σ_n is a solution for L', it is a solution for $a_1^{\bullet_1}$ and $a_2^{\bullet_2}$, where $(\bullet_1, \bullet_2)$ is an increasing (resp. decreasing, positive, or negative) limit vector for f. As σ_n is a solution for both $a_i^{\bullet_i}$, the function $\lambda n.\ a_i\sigma_n$ satisfies $\bullet_i$. By the definition of limit vectors, this implies that $\lambda n.\ f(a_1\sigma_n, a_2\sigma_n) = \lambda n.\ f(a_1, a_2)\,\sigma_n$ satisfies $\bullet$. Thus, σ_n is a solution for $f(a_1, a_2)^\bullet$.

If the step from L to L' was done by Definition 5.13 (B), then every solution σ_n for L' is also a solution for L, since $m\,\sigma_n = m$ holds for any $m \in \mathbb{R}$.

If the step from L to L' was done by Definition 5.15 (C), then let σ_n be a solution for $L' = L\theta$. Then for every $(a\,\theta)^\bullet \in L\theta$, $\lambda n.\ a\,\theta\sigma_n$ satisfies $\bullet$ and hence $\theta \circ \sigma_n$ is a solution for $a^\bullet$. Thus, $\theta \circ \sigma_n$ is a solution for L.

If the step from L to L' was done by Definition 5.18 (D), then let σ_n be a solution for $a^\bullet$. Since the polynomial a only contains a single variable (say, x), we must have $\lim_{n\mapsto\omega} x\sigma_n = \omega$ or $\lim_{n\mapsto\omega} x\sigma_n = -\omega$. W.l.o.g, let $\lim_{n\mapsto\omega} x\sigma_n = \omega$ and $\bullet = +$ (the other cases work analogously). Then $\lim_{n\mapsto\omega} a\,\sigma_n = \omega$ implies $\lim_{x\mapsto\omega} a = \omega$. Since the degree of a is greater than the degree of b, this means $\lim_{x\mapsto\omega}(a \pm b) = \omega$ and hence $\lim_{n\mapsto\omega}(a \pm b)\,\sigma_n = \omega$.

For Definition 5.18 (E), the proof is analogous. Here for large enough n, $a^c\sigma_n$ is an exponential function with a base > 1. Since σ_n is a solution for c^+, we again have $\lim_{n\mapsto\omega} x\sigma_n = \omega$ or $\lim_{n\mapsto\omega} x\sigma_n = -\omega$. Thus $a^c\sigma_n$ is an exponential function which grows faster than $b\,\sigma_n$ for $n \mapsto \omega$. Hence, we obtain $\lim_{n\mapsto\omega}(a^c \pm b)\,\sigma_n = \omega$. □

So to find an asymptotic lower bound for the runtime of a simplified program with a rule α, we start with an initial limit problem $L = \{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}, \mathrm{cost}(\alpha)^+\}$ that represents guard(α) and requires non-constant cost, and transform L with $\rightsquigarrow$ into a trivial limit problem L', i.e., $L \overset{\theta_1}{\rightsquigarrow}$

$\ldots \overset{\theta_m}{\rightsquigarrow} L'$. As mentioned before, for automation one should leave the $\bullet_i$ in the initial problem L open, and only instantiate them by a value from $\{+, +_!\}$ when this is needed to apply a particular rule of the transformation $\rightsquigarrow$. Then by Lemma 5.11 and Theorem 5.20, the resulting family $\sigma_n^{L'}$ of substitutions gives rise to a solution $\sigma_n = \theta_1 \circ \ldots \circ \theta_m \circ \sigma_n^{L'}$ of L. Thus by Theorem 5.5, we have $\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \in \Omega(\mathrm{cost}(\alpha)\,\sigma_n)$, which leads to a lower bound for $\mathrm{rc}_{\mathcal{P}}(n)$ with Lemma 5.8.

Our implementation uses the following strategy to apply the rules from Definition 5.13, 5.15, and 5.18 for the transformation $\rightsquigarrow$. Initially, we reduce the number of variables by propagating bounds implied by the guard of the rule α. For example, if an arithmetic expression a with $x \notin \mathcal{V}(a)$ is a minimal upper or a maximal lower bound on x (i.e., guard(α) implies $x \leq a$ but not $x \leq a-1$, or guard(α) implies $x \geq a$ but not $x \geq a+1$), then we may apply the substitution $\{x/a\}$ to the initial limit problem by the rule (C). Thus, we can, e.g., simplify the limit problem from Example 5.10 by instantiating x with y^2+1, see Example 5.16. In the same way, the substitution which is applied in Example 5.17 can be found automatically. Afterwards, we use (B) and (D) with highest and (E) with second highest priority. The third priority is trying to apply (A) to univariate expressions (since processing univariate expressions helps to guide the search). As fourth priority, we apply (C) with a suitable substitution $\{x/m\}$ if $x^{+_!}$ or $x^{-_!}$ occurs in the current limit problem. Otherwise, we apply (A) to multivariate expressions. Since $\rightsquigarrow$ is well founded and, except for (C), finitely branching, one may also backtrack and explore alternative applications of $\rightsquigarrow$. In particular, we backtrack if we obtain a contradictory limit problem. Moreover, if we obtain a trivial limit problem L' where $\mathrm{cost}(\alpha)\,\sigma_n^{L'}$ is a polynomial, but $\mathrm{cost}(\alpha)$ is a polynomial of higher degree or an exponential function, then we backtrack to search for other solutions which might lead to a higher lower bound. However, our implementation can of course fail, since solvability of limit problems is undecidable (due to Hilbert's Tenth Problem).

Example 5.21 (Solving the Limit Problem of the Program (19)*).* For the program $\mathcal{P} = \{(19)\}$ that results from the simplification of the program $\widetilde{\mathcal{P}}$ in Figure 1b, we obtain the initial limit problem $\{(\frac{1}{2}x^2+\frac{1}{2}x-1)^+, (\frac{1}{8}x^4+\frac{1}{4}x^3+\frac{7}{8}x^2+\frac{7}{4}x)^+\}$. Here, we have:

$$\{(\tfrac{1}{2}x^2+\tfrac{1}{2}x-1)^+, (\tfrac{1}{8}x^4+\tfrac{1}{4}x^3+\tfrac{7}{8}x^2+\tfrac{7}{4}x)^+\}$$

$\overset{(D)}{\rightsquigarrow}{}^2$ $\{(\frac{1}{2}x^2)^+, (\frac{1}{8}x^4)^+\}$ as the degree of $\frac{1}{2}x^2$ is greater than the degree of $\frac{1}{2}x-1$ and the degree of $\frac{1}{8}x^4$ is greater than the degree of $\frac{1}{4}x^3+\frac{7}{8}x^2+\frac{7}{4}x$

$\overset{(A)}{\rightsquigarrow}{}^2$ $\{\frac{1}{2}^{+_!}, (x^2)^+, \frac{1}{8}^{+_!}, (x^4)^+\}$ using the increasing limit vector $(+_!, +)$ for multiplication

$\overset{(B)}{\rightsquigarrow}{}^2$ $\{(x^2)^+, (x^4)^+\}$

$\overset{(A)}{\rightsquigarrow}{}^2$ $\{x^+\}$ using the increasing limit vector $(+, +)$ for multiplication

The resulting trivial limit problem $\{x^+\}$ gives rise to the solution (35) and hence proves $\mathrm{rc}_{\mathcal{P}}(|\overline{x}\sigma_n|) \in \Omega(n^4)$ for $\overline{x} = (x, y, z, u)$, see Example 5.6. As $|\overline{x}\sigma_n| \in \mathcal{O}(n)$, as in Example 5.9 we get $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n^4)$ and thus also $\mathrm{rc}_{\widetilde{\mathcal{P}}}(n) \in \Omega(n^4)$.

5.3 Solving Limit Problems via SMT

While the calculus presented in Section 5.2 permits a precise analysis of simplified programs, it can also be quite expensive in practice. The reason is that the next $\rightsquigarrow$-step is rarely unique and thus backtracking is often unavoidable in order to find a good lower bound. We now show how limit problems can be encoded as conjunctions of polynomial inequations. In many cases, this allows us to use SMT solvers to solve limit problems more efficiently.

Essentially, the idea is to search for a solution σ_n (i.e., a suitable family of substitutions) that instantiates each variable x in the limit problem by a linear polynomial $x\sigma_n = m_x \cdot n + k_x$. Here, we leave the integers m_x and k_x open (i.e., they are *abstract coefficients*) and we use SMT solving to find an instantiation of the abstract coefficients by integer numbers such that σ_n becomes a solution for the limit problem.

Thus, if a is a polynomial arithmetic expression, then $a\sigma_n = a\ \{x/(m_x \cdot n + k_x) \mid x \in \mathcal{V}(a)\}$ is a univariate polynomial over n with abstract coefficients. If a is of degree d, then $a\sigma_n$ can be rearranged to the form $a_d \cdot n^d + \ldots + a_0 \cdot n^0$ where the a_i are arithmetic expressions over the abstract coefficients $\{m_x, k_x \mid x \in \mathcal{V}(a)\}$ that do not contain n.

Example 5.22 (Encoding the Initial Limit Problem for Program (19)*).* Consider the initial limit problem $\{(\frac{1}{2}x^2 + \frac{1}{2}x - 1)^+\}$ for the program with the rule (19), see Example 5.4. We use σ_n with $x\sigma_n = m_x \cdot n + k_x$. Therefore, we obtain

$$\begin{array}{rcl}(\frac{1}{2}x^2 + \frac{1}{2}x - 1)\ \sigma_n & = & \frac{1}{2} \cdot (m_x \cdot n + k_x)^2 + \frac{1}{2} \cdot (m_x \cdot n + k_x) - 1 \\ & = & a_2 \cdot n^2 + a_1 \cdot n + a_0\end{array}$$

where $a_2 = \frac{1}{2}m_x^2,\ a_1 = m_x \cdot k_x + \frac{1}{2}m_x,$ and $a_0 = \frac{1}{2}k_x^2 + \frac{1}{2}k_x - 1$.

Clearly, we have $\lim_{n\mapsto\omega} a\sigma_n = \omega$ (resp. $-\omega$) if and only if $a_i > 0$ (resp. $a_i < 0$) for some $i > 0$ and $a_j = 0$ for all $i + 1 \leq j \leq d$. Similarly, $\lim_{n\mapsto\omega} a\sigma_n$ is a positive (resp. negative) constant if and only if $a_i = 0$ for all $1 \leq i \leq d$ and $a_0 > 0$ (resp. $a_0 < 0$). This allows us to translate the solvability of a limit problem into the satisfiability of an arithmetic formula.

Definition 5.23 (SMT Encoding of Limit Problems). Let a be a polynomial arithmetic expression of degree d and let σ_n instantiate each occurring variable x by $m_x \cdot n + k_x$ where m_x, k_x are abstract coefficients. Let $a\sigma_n = a_d \cdot n^d + \ldots + a_0 \cdot n^0$ where $a_0, \ldots, a_d$ do not contain n. We define

$$\mathrm{smt}(a^\bullet) = \begin{cases} \bigvee_{i=1}^{d} \left(a_i > 0 \wedge \bigwedge_{j=i+1}^{d} a_j = 0\right) & \text{if } \bullet = + \\ \bigvee_{i=1}^{d} \left(a_i < 0 \wedge \bigwedge_{j=i+1}^{d} a_j = 0\right) & \text{if } \bullet = - \\ \bigwedge_{j=1}^{d} a_j = 0 \wedge a_0 > 0 & \text{if } \bullet = +_! \\ \bigwedge_{j=1}^{d} a_j = 0 \wedge a_0 < 0 & \text{if } \bullet = -_! \end{cases}$$

We lift smt to limit problems L where a is a polynomial for each $a^\bullet \in L$ by defining $\mathrm{smt}(L) = \bigwedge_{a^\bullet \in L} \mathrm{smt}(a^\bullet)$.

Furthermore, given a polynomial cost c of degree d with $c\sigma_n = c_d \cdot n^d + \ldots + c_0 \cdot n^0$ where $c_0, \ldots, c_d$ do not contain n, we define $\mathrm{smt}_{c,i}(L) = \mathrm{smt}(L) \wedge c_i > 0$ for each $1 \leq i \leq d$. Finally, for the program variables $\overline{x}$, we define $\mathrm{smt}_\omega(L) = \mathrm{smt}(L) \wedge \bigwedge_{x \in \overline{x}} m_x = 0$.

To solve a limit problem L, it suffices to find a solution for $\mathrm{smt}(L)$, because then the substitution that results from instantiating the abstract coefficients of σ_n accordingly is a solution for the limit problem L. However, to maximize the cost c, one should try to find a solution for $\mathrm{smt}_\omega(L)$ or $\mathrm{smt}_{c,i}(L)$ where i is as large as possible. The reason is that a solution for $\mathrm{smt}_\omega(L)$ allows us to deduce unbounded costs (provided that the initial limit problem contained c^+, i.e., the cost is non-constant for each solution of L), as the corresponding substitution σ_n maps all program variables x to constants k_x that do not depend on n. A solution for $\mathrm{smt}_{c,i}(S)$ allows us to prove a polynomial lower bound whose degree is at least i via Theorem 5.5 and Lemma 5.8 (since $|\overline{x}\sigma_n| \in O(n)$).

Example 5.24 (Encoding the Initial Limit Problem for Program (19) *Continued).* We now show how to encode the initial limit problem $\{(\frac{1}{2}x^2 + \frac{1}{2}x - 1)^+\}$ from Example 5.4.[20] Since $x\,\sigma_n = m_x \cdot n + k_x$,

[20] For reasons of simplicity, we do not include the cost of the rule in the initial limit problem.

we have $(\frac{1}{2}x^2 + \frac{1}{2}x - 1)\,\sigma_n = a_2 \cdot n^2 + a_1 \cdot n + a_0$ with a_2, a_1, a_0 as in Example 5.22. Thus,

$$\text{smt}\left(\left\{\left(\tfrac{1}{2}x^2 + \tfrac{1}{2}x - 1\right)^+\right\}\right) \quad = \quad (a_2 > 0 \vee (a_1 > 0 \wedge a_2 = 0)).$$

Now SMT solvers can easily find a solution like, e.g., $\{m_x/1, k_x/0\}$.

Example 5.25 (Encoding the Initial Limit Problem for Example 5.1). Next we encode the initial limit problem $\{(x - y^2)^{+!}, y^+\}$ for Example 5.1. Since $x\,\sigma_n = m_x \cdot n + k_x$ and $y\,\sigma_n = m_y \cdot n + k_y$, we have $(x - y^2)\,\sigma_n = m_x \cdot n + k_x - (m_y \cdot n + k_y)^2 = -m_y^2 \cdot n^2 + (m_x - 2 \cdot m_y \cdot k_y) \cdot n + k_x - k_y^2$. Thus,

$$\text{smt}\left(\{(x - y^2)^{+!}, y^+\}\right) = (-m_y^2 = 0 \wedge m_x - 2 \cdot m_y \cdot k_y = 0 \wedge k_x - k_y^2 > 0 \wedge m_y > 0).$$

Here, the first three (in)equations are the encoding of $(x - y^2)^{+!}$ and the last inequation is the encoding of y^+. As $-m_y^2 = 0$ implies $m_y = 0$, the overall formula is unsatisfiable. State-of-the-art SMT solvers can prove unsatisfiability of $\text{smt}\left(\{(x - y^2)^{+!}, y^+\}\right)$ within milliseconds. This is not surprising, since we instantiated x with a non-linear expression in Example 5.16 in order to find a solution, but Definition 5.23 instantiates x with a *linear* template polynomial.

Thus, Example 5.25 shows that even if all arithmetic expressions in the analyzed limit problem are polynomials, $\rightsquigarrow$ is still required, i.e., our SMT-based technique does not subsume the calculus of Section 5.2.[21] Note that the new SMT-based technique can be integrated into the calculus from Section 5.2 seamlessly. In other words, one can first simplify a limit problem with the transformation $\rightsquigarrow$ for a few steps and then apply the SMT-based technique to find a solution for the obtained limit problem. For instance, the initial limit problem $\{(x - y^2)^{+!}, y^+\}$ in Example 5.25 can easily be solved via the SMT encoding from Definition 5.23 after applying the substitution $\{x/y^2 + 1\}$ as in Example 5.16.

The following theorem shows how a solution for the SMT problem $\text{smt}(L)$ can be used to obtain a solution for the limit problem L.

Theorem 5.26 (Solving Limit Problems via SMT). *Let L be a limit problem such that each expression in L is a polynomial and let σ be an integer substitution such that $\sigma \models \text{smt}(L)$. Then*

$$\sigma_n \circ \sigma = \{x/(m_x\sigma \cdot n + k_x\sigma) \mid x \in \mathcal{V}(L)\}$$

is a solution for L.

Proof. First note that $\sigma_n \circ \sigma$ is clearly an integer substitution for each $n \in \mathbb{N}$. We have to show that $\lambda n.\ a\sigma_n\sigma$ satisfies $\bullet$ for any $a^\bullet \in L$. Let d be the degree of a. Then we have $a\sigma_n = a_d \cdot n^d + \ldots + a_0 \cdot n^0$ for suitable expressions $a_0, \ldots, a_d$ over $\{m_x, k_x \mid x \in \mathcal{V}(a)\}$ that do not contain n.

We first consider the case $\bullet = +$ (the case $\bullet = -$ works analogously). Then $\sigma \models \text{smt}(L)$ implies $\sigma \models \text{smt}(a^+)$, i.e., $\sigma \models \bigvee_{i=1}^{d}(a_i > 0 \wedge \bigwedge_{j=i+1}^{d} a_j = 0)$. Hence, there exists an $1 \leq i \leq d$ with $a_i\sigma > 0$ and we have

$$a\sigma_n\sigma = a_i\sigma \cdot n^i + \ldots + a_0\sigma \cdot n^0.$$

Thus, we obtain $\lim_{n\mapsto\omega} a\sigma_n\sigma = \omega$, i.e., $\lambda n.\ a\sigma_n\sigma$ satisfies $+$.

Now we consider the case $\bullet = +_!$ (the case $\bullet = -_!$ works analogously). Then $\sigma \models \text{smt}(L)$ implies $\sigma \models \text{smt}(a^{+!})$, i.e., $\sigma \models \bigwedge_{j=1}^{d} a_j = 0 \wedge a_0 > 0$. Hence, we have

$$\lim_{n\mapsto\omega} a_n\sigma_n\sigma = \lim_{n\mapsto\omega} a_0\sigma = a_0\sigma > 0,$$

[21] We could also use non-linear template polynomials for σ_n, but in any case the degree of the template polynomials has to be fixed in advance and thus, it may be insufficient for the problem at hand.

i.e., $\lambda n.\ a\sigma_n\sigma$ satisfies $+_!$. □

Example 5.27 (Solving the Initial Limit Problem for Program (19)). In Example 5.24, we saw that $\sigma = \{m_x/1, k_x/0\} \models \mathrm{smt}(\{(\frac{1}{2}x^2 + \frac{1}{2}x - 1)^+\})$. Hence, according to Theorem 5.26, $\sigma_n \circ \sigma = \{x/(m_x \cdot n + k_x)\} \circ \{m_x/1, k_x/0\}$ with $x\sigma_n\sigma = n$ solves the limit problem $\{(\frac{1}{2}x^2 + \frac{1}{2}x - 1)^+\}$. This corresponds to the solution in (35). As explained in Example 5.9, this proves $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n^4)$ for the program $\mathcal{P} = \{(19)\}$ and hence, also for the program in Figure 1b.

To integrate Theorem 5.26 into the calculus of Section 5.2, we proceed as follows. Whenever the current limit problem L for a rule α only contains polynomial arithmetic expressions, one tries to find a solution for $\mathrm{smt}_\omega(L)$ or $\mathrm{smt}_{c,i}(L)$ where i is initially set to the degree of the cost c of the rule α and decremented until the SMT solver finds a solution. As soon as a solution is found, one can either return the resulting family of substitutions that solves L or keep searching for a better solution. To this end, one can either backtrack or continue simplifying L via $\rightsquigarrow$. Similarly, if the SMT solver does not find a solution one can either backtrack or keep simplifying L via $\rightsquigarrow$.

Note that the intention of Theorem 5.26 and its integration into $\rightsquigarrow$ is not to add more power to $\rightsquigarrow$. Instead, as often as possible one should delegate the search for a solution to SMT solvers (which are very efficient in solving search problems) instead of relying on heuristics. This may, of course, also lead to better results in cases where the heuristics used for $\rightsquigarrow$ are not ideal. For example, consider a simplified rule with cost $x \cdot (1 - tv^2)$ and guard $-2 < tv < 2$. The heuristic discussed at the end of Section 5.2 would instantiate tv with the bounds implied by the guard, i.e., it would apply the substitution $\theta = \{tv/1\}$ or $\theta = \{tv/-1\}$, resulting in the unsolvable limit problem $\{(x \cdot (1 - tv^2))^+\}\,\theta = \{0^+\}$. In contrast, our SMT encoding allows us to find the solution $\{tv/0, x/n\}$ which results in a linear lower bound.

Example 5.28 (Example 4.12 Continued). For the simplified facSum program with the rule (32), we obtain the initial limit problem

$$\{(x-1)^+, \left(\tfrac{1}{2}x^2 + \tfrac{3}{2}x - 2\right)^+\}.$$

Using $\sigma_n = \{x/(m_x \cdot n + k_x)\}$, its SMT encoding is

$$m_x > 0 \wedge \left(\left(m_x \cdot k_x + \tfrac{3}{2}m_x > 0 \wedge \tfrac{1}{2}m_x^2 = 0\right) \vee \tfrac{1}{2}m_x^2 > 0\right).$$

An SMT solver can find a model like $\sigma = \{m_x/1,\ k_x/0\}$, for example. This results in the solution

$$\sigma_n \circ \sigma = \{x/(m_x\sigma \cdot n + k_x\sigma)\} = \{x/n\}.$$

Applying it to the cost $\frac{1}{2}x^2 + \frac{3}{2}x - 2$ yields $\frac{1}{2}n^2 + \frac{3}{2}n - 2 \in \Omega(n^2)$. By Theorem 5.5, this proves $\mathrm{rc}_{\{(32)\}}(|x\sigma_n\sigma|) = \mathrm{rc}_{\{(32)\}}(n) \in \Omega(n^2)$. By the soundness of the program simplification in Sections 3 and 4, we obtain $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(n^2)$ for the original integer program $\mathcal{P}$ from Example 4.9.

6 EXPERIMENTS

To evaluate the performance of our approach, we implemented it in the tool LoAT ("Loop Acceleration Tool")[22] using the recurrence solver PURRS [8] and the SMT solver Z3 [21]. We evaluated LoAT on the 689 benchmark integer programs from the evaluation [49] of the tool KoAT [14] which infers *upper* runtime bounds for integer programs. On average, the ITSs in the collection [49] have a size of 23.4 rules. In addition, the results of running LoAT on the examples from this paper can be found at [33].

[22]Initially, LoAT stood for "Lower Bounds Analysis Tool", but we renamed it to reflect that LoAT's loop acceleration techniques can be used for several purposes, see [32].

		LoAT							
Best Upper Bound	$\mathrm{rc}_{\mathcal{P}}(n)$	$\Omega(1)$	$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^3)$	$\Omega(n^4)$	$\Omega(n^5)$	EXP	$\Omega(\omega)$
	$O(1)$	135	–	–	–	–	–	–	–
	$O(n)$	41	146	–	–	–	–	–	–
	$O(n^2)$	7	14	54	–	–	–	–	–
	$O(n^3)$	1	1	–	9	–	–	–	–
	$O(n^4)$	–	–	–	–	2	–	–	–
	$O(n^5)$	–	–	–	1	–	–	–	–
	EXP	–	–	–	–	–	–	13	–
	INF	32	18	2	–	–	–	–	213

Table 1. Best Upper Bound vs. LoAT

Comparison with Upper Bound Provers. As we are not aware of any other tool to compute worst-case lower bounds for integer programs, we compared our results with the asymptotically smallest results of the tools KoAT, CoFloCo [26, 27], Loopus [57], and RanK [6], that compute upper runtime bounds for integer programs.

The results of running all tools on an Azure F4s v2 instance with a timeout of 60 seconds per example are shown in Table 1. LoAT inferred non-constant lower bounds for 473 examples. For 135 additional examples, the upper bound $\mathrm{rc}_{\mathcal{P}}(n) \in O(1)$ was proved and thus, the lower bound $\Omega(1)$ inferred by LoAT is optimal. Thus, LoAT finds non-trivial or optimal bounds on 473 + 135 = 608 (88.2 %) of all examples. For 572 examples (83.0 %), the inferred bounds are asymptotically tight (e.g., lower and upper bounds coincide). Whenever an exponential upper bound was proved, LoAT also proved an exponential lower bound (i.e., $\mathrm{rc}_{\mathcal{P}}(n) \in \Omega(k^n)$ for some $k > 1$). It proved the existence of executions with unbounded length in 213 cases (this includes both non-terminating examples and examples whose runtime depends on temporary variables). On average, LoAT required 2 seconds per example. For 10 of the 689 examples, LoAT could not finish its analysis due to the timeout. The reason for most timeouts is that the number of rules gets too large during the program simplification and hence, LoAT fails to compute a simplified program in time.

One reason why the bounds inferred by LoAT do not always coincide with the upper bounds obtained by other tools may of course be that these upper bounds are not necessarily tight. If the lower bound is too small, then in our experience the most common reasons for imprecision are the following: For some loops, LoAT fails to find (non-trivial) metering functions. But if LoAT finds a metering function, then the precision of the metering function is usually very good. Another reason for imprecision are unsuitable instantiations of temporary variables. Finally, LoAT heuristically deletes rules from the analyzed program if the number of rules becomes too large, which is another common source of imprecision. Nevertheless, our experiments show that optimal lower bounds are inferred for the large majority of examples.

Evaluating Individual Contributions and Comparison with Best-Case Lower Bounds. In a second experiment, we performed an individual evaluation of the main contributions of this paper that are new compared to our earlier paper [29]. As baseline, we took LoAT-Basic, the version of LoAT

Tool	$\Omega(1)$	$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^3)$	$\Omega(n^4)$	EXP	$\Omega(\omega)$	time (s)
LoAT-Basic	269	159	42	2	2	5	210	2.29
LoAT-Cond	268	159	43	3	2	5	209	2.32
LoAT-Rec	227	175	55	6	2	13	211	2.44
LoAT-SMT	263	161	43	2	2	5	213	1.68
LoAT-Just-SMT	263	161	43	2	2	5	213	1.64
LoAT	216	179	56	10	2	13	213	1.91
CoFloCo	454	189	40	6	0	0	0	2.00

Table 2. Evaluating Individual Contributions in LoAT and Comparison with Best-Case Lower Bounds

implementing the techniques in [29].[23] Using LoAT-Basic as a starting point, we considered the following four variants:

- LoAT-Cond adds support for conditional metering functions, as introduced in Theorem 3.4.
- LoAT-Rec is LoAT-Basic extended by the handling of non-tail-recursive programs (described in Section 4).
- LoAT-SMT is like LoAT-Basic, but it applies the SMT encoding of limit problems from Section 5.3 in addition to the calculus for the transformation of limit problems that is used in LoAT-Basic. For this combination, we use a strategy which first simplifies limit problems with the calculus if the guard or the cost of the analyzed simplified rule contains non-polynomial arithmetic. However, the SMT encoding is used whenever it is applicable. Whenever there was a choice during the application of the calculus, we backtrack afterwards and apply the calculus again in order to examine the remaining possibilities. The analysis of the rule terminates as soon as LoAT proves a bound which is asymptotically equal to its cost function, when the timeout specified by the user expires, or when there are no further possibilities to backtrack (where we use suitable heuristics to ensure that case (C) of our calculus is only applied with finitely many substitutions). Then the largest bound found so far is returned as the result.
- LoAT-JUST-SMT is like LoAT-SMT, but in contrast to LoAT-SMT, LoAT-JUST-SMT never applies the calculus again once the SMT encoding is applicable.

Note that each of these variants only adds one single new contribution to LoAT-Basic, whereas the other new contributions are disabled. The intention of the last variant is to compare the power and performance of the calculus from Section 5.2 with the SMT encoding of Section 5.3 (whereas LoAT-SMT represents the *combination* of both techniques). However, as the novel SMT encoding only applies to polynomial limit problems, LoAT-JUST-SMT still uses the calculus from Section 5.2 for non-polynomial limit problems.

The results of our experiments are summarized in Table 2: LoAT-Basic already uses an optimization from [29] which is similar to (but weaker than) the conditional metering functions of Section 3.1. Conditional metering functions do not only improve the formalization and presentation of our approach (by integrating the optimization of [29] into our concept of metering functions), but they also lead to a minimal change in power: The detailed experimental results on our website [33] show that LoAT-Cond deduces better (i.e., larger) asymptotic bounds in eight cases,

[23] As we refactored large parts of the code and improved some heuristics since publishing that paper, LoAT-Basic is already more powerful than the version of LoAT from 2016 that we used in [29].

whereas LoAT-Basic deduces better asymptotic bounds in four cases. Adding support for arbitrary recursion allows LoAT to infer non-constant bounds for 41 of the 50 non-tail-recursive examples in the collection (where a constant upper bound was proved for three of these examples). The full version of LoAT even obtains non-constant bounds for 44 of these examples. The SMT encoding of limit problems improves the performance compared to LoAT-Basic by 26%. Moreover, LoAT-SMT deduces a better asymptotic bound than LoAT-Basic in eight cases, whereas LoAT-Basic infers a better asymptotic bound in one case. The configurations LoAT-SMT and LoAT-Just-SMT yield identical results. So when disregarding limit problems with non-polynomial arithmetic, the calculus from Section 5.2 is outperformed by the novel SMT encoding from Section 5.3 in our experiments on the examples from [49]. However, this is not true in general, as shown by Example 5.25. Moreover, the calculus from Section 5.2 is still required for the analysis of limit problems with non-polynomial arithmetic, i.e., for all examples where LoAT proves exponential lower bounds. In LoAT (i.e., in the last line of Table 2), we extended LoAT-Basic by all new contributions. This results in a significant improvement in both power and runtime.

Finally, in the last row of Table 2 we used the tool CoFloCo [26, 27] to compute *best-case* lower bounds. While the asymptotic bounds obtained from LoAT and CoFloCo coincide for 335 of the 689 examples, the results of the two tools are of course not directly comparable, since LoAT infers *worst-case* lower bounds, but in general, a worst-case lower bound is not a valid best-case lower bound. Moreover, CoFloCo analyzes different program paths (so-called *chains*) separately and infers individual lower bounds for them. However, similar to the experimental evaluation of CoFloCo in [27], the results in the last row of Table 2 do not take the preconditions of the different chains into account, but they are simply the maximum lower bound of all chains. Thus, they are not always asymptotic best-case lower bounds for the whole program. So the purpose of the last row is only to indicate that LoAT's performance is also convincing when comparing it to the performance of other tools for the inference of (other forms of) lower bounds.

For a detailed experimental evaluation of our implementation as well as a pre-compiled binary of LoAT we refer to [33]. The source code of LoAT is freely available at [52].

7 RELATED WORK

While there are many techniques to infer *upper bounds* on the worst-case complexity of integer programs (e.g., [1–8, 10, 14, 19, 20, 26, 27, 39, 42, 45–47, 57, 58]), there is little work on *lower bounds*. In [7], it is briefly mentioned that their technique could also be adapted to infer lower instead of upper bounds for *abstract cost rules*, i.e., integer procedures with (possibly multiple) outputs. However, this only considers *best-case* lower bounds instead of worst-case lower bounds as in our technique. Upper and lower bounds for *cost relations* are inferred in [4, 27]. Cost relations extend recurrence equations such that, e.g., non-determinism can be modeled. However, this technique also considers best-case lower bounds only. A method for best-case lower bounds for logic programs is presented in [22].

Note that techniques to infer lower bounds on the best-case complexity differ fundamentally from techniques for the inference of worst-case lower bounds. To deduce best-case lower bounds, one has to prove that a certain bound holds for *every* program run. Thus, as in the case of worst-case upper bounds, *over-approximating* techniques are used to ensure that the proven bound covers all program runs, i.e., even though such techniques under-approximate the runtime of the program, they over-approximate the set of all program runs.

In contrast, techniques to infer lower bounds on the worst-case complexity have to identify families of inputs (i.e., witnesses) that result in expensive program runs. Thus, for the inference

of worst-case lower bounds, over-approximations are usually unsound, since one has to ensure that the witness of the proven lower bound corresponds to "real" program runs. Thus, *under-approximating* techniques have to be used in order to infer lower bounds on the worst-case complexity.

Nevertheless, our approach has certain aspects in common with the technique in [4], since [4] also uses recurrence solving to compute a closed form for the costs of several consecutive applications of a cost equation with direct recursion, which corresponds to a simple loop or simple recursion in our setting. However, as mentioned above, the analyses for best-case lower bounds from [4, 27] have to reason about all program runs. Thus, there the handling of non-determinism is challenging as all possible non-deterministic choices have to be taken into account. In contrast, we can treat temporary variables (which we use to model non-determinism) as constants when computing the iterated update and cost. Thus, our iterated update and cost only represent evaluations where temporary variables are instantiated with the same values in each iteration. This restriction is sound in our setting, as we only need to prove the existence of a certain family of program runs, i.e., we do not have to reason about all program runs. To reason about evaluations where the valuation of the temporary variables changes, we can instantiate them with expressions containing program variables via *Instantiation* (Theorem 3.12).

Since our computation of the iterated update relies on the existence of a single deterministic update, it is not applicable to simple recursions, which also prevents us from computing iterated costs when accelerating non-tail-recursive rules in Theorem 4.5. Thus, our handling of simple recursions may be improved by incorporating ideas from [4, 27] for the inference of bounds of cost equations with multiple recursive calls.

In [30], we introduced two techniques to infer worst-case lower bounds for term rewrite systems (TRSs). However, TRSs differ substantially from the programs considered here, since they do not allow integers and have no notion of a "program start". Thus, the techniques from [30] are very different to the present paper.

In contrast to the techniques for the computation of symbolic runtime bounds, [15] presents a technique to generate test-cases that trigger the worst-case execution time of programs. The idea is to execute the program for small inputs, observe the required runtime, and then generalize those inputs that lead to expensive runs. In this way, one obtains *generators* which can be used to construct larger inputs that presumably result in expensive runs as well. In contrast to the technique presented in the current paper, [15] operates on Java, i.e., it also supports data structures. However, [15] does not try to infer symbolic bounds, which is the main purpose of our technique. Nevertheless, ideas from [15] could be integrated into our framework. For example, a similar approach could be used in order to apply *Instantiation* (Theorem 3.12) in a way that leads to expensive runs.

The approach from [60] can synthesize worst-case inputs, but the size of the input needs to be fixed a priori. This approach is fundamentally different from our technique to deduce *symbolic* worst-case bounds. However, the inputs that are synthesized by the technique from [60] are provably optimal, whereas our worst-case lower bounds are correct, but not necessarily tight.

Inferring bounds on the runtime of programs has also been investigated for probabilistic programs. While there exist several approaches to find upper bounds on the expected runtime of such programs, again there are only very few works that consider the inference of lower bounds on the expected runtime of probabilistic programs [34, 36, 40, 54].

To simplify programs, we use *Loop Acceleration* to summarize the effect of applying a simple loop (or a simple recursion) repeatedly. Acceleration is mostly used in over-approximating settings (e.g., [25, 38, 48, 53, 59]), where handling non-determinism is challenging, as loop summaries have to

cover *all* possible non-deterministic choices. However, our technique is under-approximating, i.e., we can instantiate non-deterministic values arbitrarily.

The under-approximating acceleration technique in [50] uses quantifier elimination, whereas our acceleration technique relies on metering functions. In [32], we generalized the loop acceleration technique from [50] in order to prove non-termination of integer programs. In future work, we will examine whether ideas from [32, 50] can also be incorporated into our framework for the inference of lower runtime bounds.

Another related approach (see, e.g., [11, 12, 35]) accelerates loops whose transitive closure can be expressed in Presburger Arithmetic. In particular, this is the case for loops whose transition relation can be described by *octagons*, i.e., conjunctions of inequations of the form $\pm x \pm y \leq c$ where x, y are variables and $c \in \mathbb{Z}$, and for loops with Presburger-definable guards and affine updates $\mu(\overline{x}) = A \cdot \overline{x} + \overline{c}$ with the *finite monoid property*, i.e., where the set $\{A^i \mid i \in \mathbb{N}\}$ is finite. In contrast, our acceleration technique does not necessarily compute the transitive closure of loops exactly. The reason is that our metering functions may be imprecise and that we approximate non-determinism by assuming that the values of temporary variables remain unchanged across loop iterations. On the other hand, our approach can also handle loops where the transitive closure cannot be expressed in Presburger Arithmetic.

The paper [1] presents a technique to infer asymptotic bounds from concrete bounds with a so-called context constraint φ, i.e., bounds of the form $\varphi \implies rt \leq e$ or $\varphi \implies rt \geq e$. Here, rt is the runtime of the program and e is a *cost expression*. These expressions are orthogonal to the expressions that are supported by our technique from Section 5 (e.g., e may contain maximum and logarithm, but negative numbers are only allowed in sub-expressions of the form $\max(0, \ldots)$). The approach in [1] infers multi-variate asymptotic bounds, whereas our technique infers univariate bounds which are only parameterized in the size of the input. Moreover, [1] does not aim to eliminate the context constraint, i.e., the resulting asymptotic bounds are of the form $\varphi \implies rt \in O(b)$ or $\varphi \implies rt \in \Omega(b)$. In contrast, eliminating the context constraints φ is one of the main motivations for our technique to deduce asymptotic bounds from concrete bounds.

Our SMT encoding for limit problems from Section 5.3 inspired parts of the work from [31], where we used a similar encoding to prove that termination is decidable for a certain class of integer loops. As in Section 5.3, the underlying idea of [31] is to abstract the loop guard by focusing on its behavior "for large enough values of n". In the present work, the variable n is the parameter of the family of substitutions that solves the limit problem. In [31], n represents the number of loop iterations. Due to the restricted form of the loops in [31], their SMT encoding only requires *linear* integer arithmetic, such that we obtain a decision procedure for termination.

Finally, in [17], our concept of metering functions has been adapted in order to synthesize invariants. Note that invariant inference and complexity analysis are closely related: Invariant inference techniques can be used to compute complexity bounds by introducing an additional counter that is incremented in each step and deducing an invariant that bounds its value (see, e.g., [57]). Conversely, complexity analysis techniques can be used to bound the value of any arithmetic expression b (i.e., to compute invariants) by choosing the costs of transitions in a way that reflects changes of the value of b (see, e.g., [55, 57]).

8 CONCLUSION AND FUTURE WORK

We introduced the first technique to infer lower bounds on the worst-case runtime complexity of integer programs, based on a modular program simplification framework. The main simplification techniques are *Loop Acceleration* and *Recursion Acceleration*, which rely on *recurrence solving* and

metering functions, an adaptation of classical ranking functions. By eliminating loops and function symbols via *Chaining* and *Partial Deletion*, we eventually obtain *simplified programs*. We presented a technique to infer *asymptotic lower bounds* from simplified programs, which can also be used to find program vulnerabilities. An experimental evaluation with our tool LoAT demonstrates the applicability of our technique in practice, see [33, 52].

In comparison to the preliminary version of our paper from [29], we showed how to deduce *conditional* metering functions, we improved our program simplification by eliminating variables from the program, we extended our approach to non-tail-recursive programs, and we improved our technique to infer asymptotic lower bounds for simplified programs by an SMT encoding. See Section 1 for a full list of the contributions of the current paper compared to [29].

There are several interesting directions for future work. First of all, one could couple LoAT with invariant inference techniques to improve its power. Furthermore, LoAT's heuristics to apply *Instantiation* are relatively simple and should be improved, e.g., by incorporating ideas from [15]. Another interesting question is to what extent LoAT can benefit from more sophisticated techniques to infer metering functions. Possibilities include the inference of logarithmic or super-linear polynomial metering functions, but one could also adapt the *quasi-ranking functions* from [51] to our setting. Apart from that, we plan to investigate if our approach can benefit from alternative loop acceleration techniques [12, 32, 50]. Moreover, as mentioned in Section 7, ideas from [4, 27] could be adapted to under-approximate the costs of repeatedly applying simple recursions more precisely when accelerating them. Finally, one could generalize our program model to improve its expressiveness. In particular, one could consider the return values of auxiliary function calls (by allowing terms with nested occurrences of functions). Moreover, one could combine our technique with ideas from [30] for the inference of lower bounds for term rewrite systems (i.e., programs operating on tree-shaped data structures) to analyze programs whose complexity depends on both integers and data structures.

ACKNOWLEDGMENTS

We thank Samir Genaim, Jan Böker, and Jera Hensel for discussions and comments. We are also very grateful to the anonymous reviewers for many helpful suggestions.

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 389792660 as part of TRR 248 (https://perspicuous-computing.science) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2).

A PROOF OF LEMMA 5.8

Lemma 5.8 is based on Lemma 24 from our paper [30]. However, since the lemmas are slightly different and since the proof for part (b) was omitted from [30], we provide the proof of Lemma 5.8 in this appendix. Moreover, part (c) of the lemma was not present in [30].

Lemma 5.8 (Bounds for Function Composition). *Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0} \cup \{\omega\}$ and $g : \mathbb{N} \to \mathbb{N}$ where $g(n) \in O(n^d)$ for some $d \in \mathbb{N}$ with $d > 0$. Moreover, let $f(n)$ be weakly monotonically increasing for large enough n.*

(a) If $g(n)$ is strictly monotonically increasing for large enough n and $f(g(n)) \in \Omega(n^k)$ with $k \in \mathbb{N}$, then $f(n) \in \Omega(n^{\frac{k}{d}})$.

(b) If $g(n)$ is strictly monotonically increasing for large enough n and $f(g(n)) \in \Omega(k^n)$ with $k > 1$, then $f(n) \in \Omega(e^{\sqrt[d]{n}})$ for some number $e \in \mathbb{R}$ with $e > 1$.

(c) If $g(n) \in O(1)$ and $f(g(n)) \notin O(1)$, then $f(n) \in \Omega(\omega)$.

Proof. For any $n_0 \in \mathbb{N}$, let $\mathbb{N}_{\geq n_0} = \{n \in \mathbb{N} \mid n \geq n_0\}$. For any (total) function $h : M \to \mathbb{N}_{\geq n_0}$ with $M \subseteq \mathbb{N}$ where M is infinite, we define $\lfloor h \rfloor(n) : \mathbb{N}_{\geq \min(M)} \to \mathbb{N}_{\geq n_0}$ and $\lceil h \rceil(n) : \mathbb{N} \to \mathbb{N}_{\geq n_0}$ by:

$$\begin{aligned} \lfloor h \rfloor(n) &= h(\max\{x \in M \mid x \leq n\}) \\ \lceil h \rceil(n) &= h(\min\{x \in M \mid x \geq n\}) \end{aligned}$$

Note that infinity of h's domain M ensures that there is always an $x \in M$ with $x \geq n$. Since $\lfloor h \rfloor$ is only defined on $\mathbb{N}_{\geq \min(M)}$, there is always an $x \in M$ with $x \leq n$ for any $n \in \mathbb{N}_{\geq \min(M)}$.

To prove the lemma, as in the proof of [30, Lemma 24] we first show that if $h : M \to \mathbb{N}_{\geq n_0}$ is strictly monotonically increasing and surjective, then

$$\lfloor h \rfloor(n) \in \{\lceil h \rceil(n), \lceil h \rceil(n) - 1\} \quad \text{for all } n \in \mathbb{N}_{\geq \min(M)} \tag{38}$$

Afterwards, we prove (a) – (c) separately.

Claim 1. (38) holds, i.e., $\lfloor h \rfloor(n) \in \{\lceil h \rceil(n), \lceil h \rceil(n) - 1\}$
To prove (38), let $n \in \mathbb{N}_{\geq \min(M)}$. If $n \in M$, then clearly $\lfloor h \rfloor(n) = \lceil h \rceil(n)$. If $n \notin M$, then let $\check{n} = \max\{x \in M \mid x < n\}$ and $\hat{n} = \min\{x \in M \mid x > n\}$. Thus, $\check{n} < n < \hat{n}$. Strict monotonicity of h implies $h(\check{n}) < h(\hat{n})$. Assume that $h(\hat{n}) - h(\check{n}) > 1$. Then by surjectivity of h, there is an $\overline{n} \in M$ with $h(\overline{n}) = h(\check{n}) + 1$ and thus $h(\check{n}) < h(\overline{n}) < h(\hat{n})$. By strict monotonicity of h, we obtain $\check{n} < \overline{n} < \hat{n}$. Since $n \notin M$ and $\overline{n} \in M$ implies $n \neq \overline{n}$, we either have $\overline{n} < n$ which contradicts $\check{n} = \max\{\check{n} \in M \mid \check{n} < n\}$ or $\overline{n} > n$ which contradicts $\hat{n} = \min\{\hat{n} \in M \mid \hat{n} > n\}$. Hence, $\lfloor h \rfloor(n) = h(\check{n}) = h(\hat{n}) - 1 = \lceil h \rceil(n) - 1$, which proves (38).

Claim 2. Lemma 5.8(a) holds, i.e., $f(g(n)) \in \Omega(n^k)$ implies $f(n) \in \Omega(n^{\frac{k}{d}})$
For the proof of this claim, we slightly adapt the corresponding proof of [30, Lemma 24] to arbitrary functions f and g. Note that $g(n) \in O(n^d)$ and $f(g(n)) \in \Omega(n^k)$ imply

$$\exists n_0, m, m' > 0.\ \forall n \in \mathbb{N}_{\geq n_0}.\ \ g(n) \leq m \cdot n^d \wedge m' \cdot n^k \leq f(g(n)).$$

We can choose n_0 large enough such that $f|_{\mathbb{N}_{\geq n_0}}$ is weakly and $g|_{\mathbb{N}_{\geq n_0}}$ is strictly monotonically increasing, where for any function $h : \mathbb{N} \to \mathbb{N}$ and any $M \subseteq \mathbb{N}$, $h|_M$ denotes the restriction of h to M. Let $M = \{g(n) \mid n \geq n_0\}$ and let $g^{-1} : M \to \mathbb{N}_{\geq n_0}$ be the function such that $g(g^{-1}(n)) = n$. Note that g^{-1} exists, since strict monotonicity of g implies injectivity of g. By instantiating n with $g^{-1}(n)$, we obtain

$$\exists n_0, m, m' > 0.\ \forall n \in M.\ \ g(g^{-1}(n)) \leq m \cdot (g^{-1}(n))^d \wedge m' \cdot (g^{-1}(n))^k \leq f(g(g^{-1}(n)))$$

which simplifies to

$$\exists n_0, m, m' > 0.\ \forall n \in M.\ \ n \leq m \cdot (g^{-1}(n))^d \wedge m' \cdot (g^{-1}(n))^k \leq f(n).$$

When dividing by m and taking the d-th root on both sides of the first inequation, we get

$$\exists n_0, m, m' > 0.\ \forall n \in M.\ \ \sqrt[d]{\tfrac{n}{m}} \leq g^{-1}(n) \wedge m' \cdot (g^{-1}(n))^k \leq f(n).$$

By monotonicity of $\sqrt[d]{\frac{n}{m}}$ and $f(n)$ in n, this implies

$$\exists n_0, m, m' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ \sqrt[d]{\tfrac{n}{m}} \leq \lceil g^{-1} \rceil(n) \wedge m' \cdot (\lfloor g^{-1} \rfloor(n))^k \leq f(n).$$

Note that $g|_{\mathbb{N}_{\geq n_0}}$ is total and hence, $g^{-1} : M \to \mathbb{N}_{\geq n_0}$ is surjective. Moreover, by strict monotonicity of $g|_{\mathbb{N}_{\geq n_0}}$, M is infinite and g^{-1} is also strictly monotonically increasing. Hence, by (38) we get $\lceil g^{-1} \rceil(n) \leq \lfloor g^{-1} \rfloor(n) + 1$ for all $n \in \mathbb{N}_{\geq g(n_0)}$. Thus,

$$\exists n_0, m, m' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ \sqrt[d]{\tfrac{n}{m}} - 1 \leq \lfloor g^{-1} \rfloor(n) \wedge m' \cdot (\lfloor g^{-1} \rfloor(n))^k \leq f(n)$$

which implies

$$\exists n_0, m, m' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ m' \cdot \left(\sqrt[d]{\tfrac{n}{m}} - 1\right)^k \leq f(n).$$

Therefore, $\exists m > 0.\ \ f(n) \in \Omega\left(\left(\sqrt[d]{\tfrac{n}{m}} - 1\right)^k\right)$ and thus, $f(n) \in \Omega\left(n^{\frac{k}{d}}\right)$.

Claim 3. Lemma 5.8(b) holds, i.e., $f(g(n)) \in \Omega(k^n)$ implies $f(n) \in \Omega(e^{\sqrt[d]{n}})$ for some $e > 1$
The proof is analogous to the proof of the case $f(g(n)) \in \Omega(n^k)$, but it was not given in [30]. Here, $g(n) \in O(n^d)$ and $f(g(n)) \in \Omega(k^n)$ imply

$$\exists n_0, m, m' > 0.\ \forall n \in \mathbb{N}_{\geq n_0}.\ \ g(n) \leq m \cdot n^d \wedge m' \cdot k^n \leq f(g(n)).$$

Again, we can choose n_0 large enough such that $f|_{\mathbb{N}_{\geq n_0}}$ is weakly and $g|_{\mathbb{N}_{\geq n_0}}$ is strictly monotonically increasing. As in the proof of the previous claim, let $M = \{g(n) \mid n \geq n_0\}$ and let $g^{-1} : M \to \mathbb{N}_{\geq n_0}$ be the function such that $g(g^{-1}(n)) = n$. By instantiating n with $g^{-1}(n)$, we obtain

$$\exists n_0, m, m' > 0.\ \forall n \in M.\ \ g(g^{-1}(n)) \leq m \cdot (g^{-1}(n))^d \wedge m' \cdot k^{g^{-1}(n)} \leq f(g(g^{-1}(n)))$$

which simplifies to

$$\exists n_0, m, m' > 0.\ \forall n \in M.\ \ n \leq m \cdot (g^{-1}(n))^d \wedge m' \cdot k^{g^{-1}(n)} \leq f(n).$$

When dividing by m and taking the d-th root on both sides of the first inequation, we get

$$\exists n_0, m, m' > 0.\ \forall n \in M.\ \ \sqrt[d]{\tfrac{n}{m}} \leq g^{-1}(n) \wedge m' \cdot k^{g^{-1}(n)} \leq f(n).$$

By monotonicity of $\sqrt[d]{\frac{n}{m}}$ and $f(n)$ in n, this implies

$$\exists n_0, m, m' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ \sqrt[d]{\tfrac{n}{m}} \leq \lceil g^{-1} \rceil(n) \wedge m' \cdot k^{\lfloor g^{-1} \rfloor(n)} \leq f(n).$$

As in the proof of Claim 2, $g|_{\mathbb{N}_{\geq n_0}}$ is total and hence, $g^{-1} : M \to \mathbb{N}_{\geq n_0}$ is surjective. Moreover, by strict monotonicity of $g|_{\mathbb{N}_{\geq n_0}}$, M is infinite and g^{-1} is also strictly monotonically increasing. Hence, by (38) we get $\lceil g^{-1} \rceil(n) \leq \lfloor g^{-1} \rfloor(n) + 1$ for all $n \in \mathbb{N}_{\geq g(n_0)}$. Thus,

$$\exists n_0, m, m' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ \sqrt[d]{\tfrac{n}{m}} - 1 \leq \lfloor g^{-1} \rfloor(n) \wedge m' \cdot k^{\lfloor g^{-1} \rfloor(n)} \leq f(n).$$

Since $k > 1$, this implies

$$\exists n_0, m, m' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ m' \cdot k^{\sqrt[d]{\frac{n}{m}} - 1} \leq f(n)$$

which is equivalent to

$$\exists n_0, m, m' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ \tfrac{m'}{k} \cdot k^{\sqrt[d]{\frac{n}{m}}} \leq f(n)$$

Since $m' > 0$ and $k > 1$, we have $m'' = \frac{m'}{k} > 0$ and thus we get

$$\exists n_0, m, m'' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ m'' \cdot k^{\sqrt[d]{\frac{n}{m}}} \leq f(n)$$

which is equivalent to

$$\exists n_0, m, m'' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ m'' \cdot k^{\frac{\sqrt[d]{n}}{\sqrt[d]{m}}} \leq f(n)$$

Since $m > 0$, we have $r = \sqrt[d]{m} > 0$ and hence:

$$\exists n_0, r, m'' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ m'' \cdot k^{\frac{\sqrt[d]{n}}{r}} \leq f(n)$$

which is equivalent to

$$\exists n_0, r, m'' > 0.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ m'' \cdot \sqrt[r]{k}^{\sqrt[d]{n}} \leq f(n)$$

Finally, $k > 1$ implies $e = \sqrt[r]{k} > 1$ and we obtain

$$\exists n_0, m'' > 0, e > 1.\ \forall n \in \mathbb{N}_{\geq g(n_0)}.\ \ m'' \cdot e^{\sqrt[d]{n}} \leq f(n)$$

which implies

$$\exists e > 1.\ \ f(n) \in \Omega(e^{\sqrt[d]{n}})$$

Claim 4. Lemma 5.8(c) holds, i.e., $g(n) \in \mathcal{O}(1)$ and $f(g(n)) \notin \mathcal{O}(1)$ implies $f(n) \in \Omega(\omega)$
Note that $f(g(n)) \notin \mathcal{O}(1)$ means that

$$\forall m \in \mathbb{N}.\ \exists n \in \mathbb{N}.\ \ f(g(n)) > m$$

By $g(n) \in \mathcal{O}(1)$ we have

$$\exists m' \in \mathbb{N}.\ \forall n \in \mathbb{N}.\ \ g(n) \leq m'$$

As f is weakly monotonic, we have $f(g(n)) \leq f(m')$ for all n. Hence, we get

$$\exists m' \in \mathbb{N}.\ \forall m \in \mathbb{N}.\ \ f(m') > m$$

This implies that $f(m') = \omega$. By weak monotonicity of f, we obtain

$$\exists m' \in \mathbb{N}.\ \forall n \geq m'.\ \ f(n) = \omega$$

which means $f(n) \in \Omega(\omega)$. □

REFERENCES

[1] Elvira Albert, Diego E. Alonso-Blas, Puri Arenas, Samir Genaim, and Germán Puebla. 2009. Asymptotic Resource Usage Bounds. In *APLAS '09 (LNCS 5904)*. 294–310.

[2] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2011. Closed-Form Upper Bounds in Static Cost Analysis. *JAR* 46, 2 (2011), 161–203.

[3] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2012. Cost Analysis of Object-Oriented Bytecode Programs. *TCS* 413, 1 (2012), 142–159.

[4] Elvira Albert, Samir Genaim, and Abu N. Masud. 2013. On the Inference of Resource Usage Upper and Lower Bounds. *ACM TOCL* 14, 3 (2013), 22:1–22:35.

[5] Elvira Albert, Miquel Bofill, Cristina Borralleras, Enrique Martin-Martin, and Albert Rubio. 2019. Resource Analysis Driven by (Conditional) Termination Proofs. *TPLP* 19, 5-6 (2019), 722–739.

[6] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-Dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS '10 (LNCS 6337)*. 117–133.

[7] Diego E. Alonso-Blas and Samir Genaim. 2012. On the Limits of the Classical Approach to Cost Analysis. In *SAS '12 (LNCS 7460)*. 405–421.

[8] Roberto Bagnara, Andrea Pescetti, Alessandro Zaccagnini, and Enea Zaffanella. 2005. PURRS: Towards Computer Algebra Support for Fully Automatic Worst-Case Complexity Analysis. *CoRR* abs/cs/0512056 (2005).

[9] Amir M. Ben-Amram and Samir Genaim. 2014. Ranking Functions for Linear-Constraint Loops. *Journal of the ACM* 61, 4 (2014), 26:1–26:55.

[10] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. 2010. ABC: Algebraic Bound Computation for Loops. In *LPAR '10 (LNCS 6355)*. 103–118.

[11] Marius Bozga, Codruta Gîrlea, and Radu Iosif. 2009. Iterating Octagons. In *TACAS '09 (LNCS 5505)*. 337–351.

[12] Marius Bozga, Radu Iosif, and Filip Konecný. 2010. Fast Acceleration of Ultimately Periodic Relations. In *CAV '10 (LNCS 6174)*. 227–242.

[13] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear Ranking with Reachability. In *CAV '05 (LNCS 3576)*. 491–504.

[14] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2016. Analyzing Runtime and Size Complexity of Integer Programs. *ACM TOPLAS* 38, 4 (2016), 13:1–13:50.

[15] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated Test Generation for Worst-Case Complexity. In *ICSE '09*. 463–473.
[16] Rod M. Burstall and John Darlington. 1977. A Transformation System for Developing Recursive Programs. *Journal of the ACM* 24, 1 (1977), 44–67.
[17] Pavel Cadek, Clemens Danninger, Moritz Sinn, and Florian Zuleger. 2018. Using Loop Bound Analysis For Invariant Generation. In *FMCAD '18*. 1–9.
[18] Peter Cameron. 2017. *Polynomials Taking Integer Values*. https://cameroncounts.wordpress.com/2017/01/31/polynomials-taking-integer-values/
[19] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional Certified Resource Bounds. In *PLDI '15*. 467–478.
[20] Quentin Carbonneaux, Jan Hoffmann, Thomas W. Reps, and Zhong Shao. 2017. Automated Resource Analysis with Coq Proof Objects. In *CAV '17 (LNCS 10427)*. 64–85.
[21] Leonardo de Moura and Nikolay Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS '08 (LNCS 4963)*. 337–340.
[22] Saumya Debray, Pedro López-García, Manuel V. Hermenegildo, and Nai-Wei Lin. 1997. Lower Bound Cost Estimation for Logic Programs. In *ILPS '97*. 291–305.
[23] Complexity Analysis-Based Guaranteed Execution. 2015. https://www.draper.com/news-releases/drapers-cage-could-spot-code-vulnerable-denial-service-attacks.
[24] Stephan Falke, Deepak Kapur, and Carsten Sinz. 2012. Termination Analysis of Imperative Programs Using Bitvector Arithmetic. In *VSTTE '12 (LNCS 7152)*. 261–277.
[25] Azadeh Farzan and Zachary Kincaid. 2015. Compositional Recurrence Analysis. In *FMCAD '15*. 57–64.
[26] Antonio Flores-Montoya and Reiner Hähnle. 2014. Resource Analysis of Complex Programs with Cost Equations. In *APLAS '14 (LNCS 8858)*. 275–295.
[27] Antonio Flores-Montoya. 2016. Upper and Lower Amortized Cost Bounds of Programs Expressed as Cost Relations. In *FM '16 (LNCS 9995)*. 254–273.
[28] Space/Time Analysis for Cybersecurity (STAC). 2015. http://www.darpa.mil/program/space-time-analysis-for-cybersecurity.
[29] Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl. 2016. Lower Runtime Bounds for Integer Programs. In *IJCAR '16 (LNAI 9706)*. 550–567.
[30] Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder. 2017. Lower Bounds for Runtime Complexity of Term Rewriting. *JAR* 59, 1 (2017), 121–163.
[31] Florian Frohn and Jürgen Giesl. 2019. Termination of Triangular Integer Loops is Decidable. In *CAV '19 (LNCS 11562)*. 426–444.
[32] Florian Frohn and Jürgen Giesl. 2019. Proving Non-Termination via Loop Acceleration. In *FMCAD '19*. 221–230.
[33] Florian Frohn, Matthias Naaf, Marc Brockschmidt, and Jürgen Giesl. 2020. *Empirical Evaluation of "Inferring Lower Runtime Bounds for Integer Programs"*. https://aprove-developers.github.io/its-lowerbounds-journal
[34] Hongfei Fu and Krishnendu Chatterjee. 2019. Termination of Nondeterministic Probabilistic Programs. In *VMCAI '19 (LNCS 11388)*. 468–490.
[35] Pierre Ganty, Radu Iosif, and Filip Konecný. 2017. Underapproximation of Procedure Summaries for Integer Pograms. *STTT* 19, 5 (2017), 565–584.
[36] Jürgen Giesl, Peter Giesl, and Marcel Hark. 2019. Computing Expected Runtimes for Constant Probability Programs. In *CADE '19 (LNAI 11716)*. 269–286.
[37] Jürgen Giesl, Albert Rubio, Christian Sternagel, Johannes Waldmann, and Akihisa Yamada. 2019. The Termination and Complexity Competition. In *TACAS '19 (LNCS 11429)*. 156–166.
[38] Laure Gonnord and Nicolas Halbwachs. 2006. Combining Widening and Acceleration in Linear Relation Analysis. In *SAS '06 (LNCS 4134)*. 144–160.
[39] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *POPL '09*. 127–139.
[40] Marcel Hark, Benjamin Lucien Kaminski, Jürgen Giesl, and Joost-Pieter Katoen. 2020. Aiming Low Is Harder – Inductive Proof Rules for Lower Bounds on Weakest Preexpectations in Probabilistic Program Verification. *PACMPL* 4, POPL (2020), 37:1–37:28.
[41] André Heck. 1996. *Introduction to Maple (2. ed.)*. Springer.
[42] Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder. 2018. Termination and Complexity Analysis for Programs with Bitvector Arithmetic by Symbolic Execution. *JLAMP* 97 (2018), 105–130.
[43] Nao Hirokawa and Georg Moser. 2008. Automated Complexity Analysis Based on the Dependency Pair Method. In *IJCAR '08 (LNAI 5195)*. 364–379.
[44] Dieter Hofbauer and Clemens Lautemann. 1989. Termination Proofs and the Length of Derivations. In *RTA '89 (LNCS 355)*. 167–177.

[45] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate Amortized Resource Analysis. *ACM TOPLAS* 34, 3 (2012), 14:1–14:62.
[46] Jan Hoffmann and Zhong Shao. 2015. Type-Based Amortized Resource Analysis with Integers and Arrays. *Journal of Functional Programming* 25 (2015).
[47] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *POPL '17*. 359–373.
[48] Bertrand Jeannet, Peter Schrammel, and Sriram Sankaranarayanan. 2014. Abstract Acceleration of General Linear Loops. In *POPL '14*. 529–540.
[49] KoAT Benchmarks 2014. https://github.com/s-falke/kittel-koat/tree/master/koat-evaluation/examples
[50] Daniel Kroening, Matt Lewis, and Georg Weissenbacher. 2015. Under-Approximating Loops in C Programs for Fast Counterexample Detection. *FMSD* 47, 1 (2015), 75–92.
[51] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2013. Proving Termination of Imperative Programs Using Max-SMT. In *FMCAD '13*. 218–225.
[52] LoAT 2019. https://github.com/aprove-developers/LoAT
[53] Kumar Madhukar, Björn Wachter, Daniel Kroening, Matt Lewis, and Mandayam K. Srivas. 2015. Accelerating Invariant Generation. In *FMCAD '15*. 105–111.
[54] Annabelle McIver and Carroll Morgan. 2005. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer.
[55] Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl. 2017. Complexity Analysis for Term Rewriting by Integer Transition Systems. In *FroCoS '17 (LNAI 10483)*. 132–150.
[56] Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *VMCAI '04 (LNCS 2937)*. 239–251.
[57] Moritz Sinn, Florian Zuleger, and Helmuth Veith. 2017. Complexity and Resource Bound Analysis of Imperative Programs Using Difference Constraints. *JAR* 59, 1 (2017), 3–45.
[58] Akhilesh Srikanth, Burak Sahin, and William R. Harris. 2017. Complexity Verification Using Guided Theorem Enumeration. In *POPL '17*. 639–652.
[59] Jan Strejcek and Marek Trtík. 2012. Abstracting Path Conditions. In *ISSTA '12*. 155–165.
[60] Di Wang and Jan Hoffmann. 2019. Type-Guided Worst-Case Input Generation. *PACMPL* 3, POPL (2019), 13:1–13:30.
[61] Stephen Wolfram. 1992. Mathematica: A System for Doing Mathematics by Computer. *SIAM Rev.* 34, 3 (1992), 519–522.