Tight Worst-Case Bounds for Polynomial Loop Programs

Amir M. Ben-Amram[1] and G.W. Hamilton[2][0000−0001−5954−6444]✉

[1] School of Computer Science, Tel-Aviv Academic College, Israel
amirben@mta.ac.il
[2] School of Computing, Dublin City University, Dublin 9, Ireland
hamilton@computing.dcu.ie

Abstract. In 2008, Ben-Amram, Jones and Kristiansen showed that for a simple programming language—representing non-deterministic imperative programs with bounded loops, and arithmetics limited to addition and multiplication—it is possible to decide precisely whether a program has certain growth-rate properties, in particular whether a computed value, or the program's running time, has a polynomial growth rate.
A natural and intriguing problem was to improve the precision of the information obtained. This paper shows how to obtain asymptotically-tight *multivariate* polynomial bounds for this class of programs. This is a complete solution: whenever a polynomial bound exists it will be found.

1 Introduction

One of the most important properties we would like to know about programs is their *resource usage*, i.e., the amount of resources (such as time, memory and energy) required for their execution. This information is useful during development, when performance bugs and security vulnerabilities exploiting performance issues can be avoided. It is also particularly relevant for mobile applications, where resources are limited, and for cloud services, where resource usage is a major cost factor.

In the literature, a lot of different "cost analysis" problems (also called "resource bound analysis," etc.) have been studied (e.g. [27, 24, 18, 1, 13, 19, 11, 26]); several of them may be grouped under the following general definition. The *countable resource problem* asks about the maximum usage of a "resource" that accumulates during execution, and which one can explicitly count, by instrumenting the program with an accumulator variable and instructions to increment it where necessary. For example, we can estimate the *execution time* of a program by counting certain "basic steps". Another example is counting the number of visits to designated program locations. Realistic problems of this type include bounding the number of calls to specific functions, perhaps to system services; the number of I/O operations; number of accesses to memory, etc. The consumption of resources such as *energy* suits our problem formulation as long as such explicit bookkeeping is possible (we have to assume that the increments, if not constant, are given by a monotone polynomial expression).

In this paper we solve the *bound analysis problem* for a particular class of programs, defined in [7]. The bound analysis problem is to find symbolic bounds on the maximal possible value of an integer variable at the end of the program, in terms of some integer-valued variables that appear in the initial state of a computation. Thus, a solution to this problem might be used for any of the resource-bound analyses above. In this work we focus on values that grow polynomially (in the sense of being bounded by a polynomial), and our goal is to find polynomial bounds that are tight, in the sense of being precise up to a constant factor.

The programs we study are expressed by the so-called *core language*. It is imperative, including bounded loops, non-deterministic branches and restricted arithmetic expressions; the syntax is shown in Fig. 1. Semantics is explained and motivated below, but is largely intuitive; see also the illustrative example in Fig. 2. In 2008, it was proved [7] that for this language it is decidable whether a computed result is polynomially bounded or not. This makes the language an attractive target for work on the problem of computing tight bounds. However, for the past ten years there has been no improvement on [7]. We now present an algorithm to compute, for every program in the language, and every variable in the program which has a polynomial upper bound (in terms of input values), a tight polynomial bound on its largest attainable value (informally, "the worst-case value") as a function of the input values. The bound is guaranteed to be tight up to a multiplicative constant factor but constants are left implicit (for example a bound quadratic in n will always be represented as n^2). The algorithm could be extended to compute upper and lower bounds with explicit constant factors, but choosing to ignore coefficients simplifies the algorithm considerably. In fact, we have striven for a simple, comprehensible algorithm, and we believe that the algorithm we present is sufficiently simple that, beyond being comprehensible, offers insight into the structure of computations in this model.

1.1 The core language

$$\begin{array}{rcl}
\mathtt{X} \in \text{Variable} & ::= & \mathtt{X}_1 \mid \mathtt{X}_2 \mid \mathtt{X}_3 \mid \ldots \mid \mathtt{X}_n \\
\mathtt{E} \in \text{Expression} & ::= & \mathtt{X} \mid \mathtt{E}\ \mathtt{+}\ \mathtt{E} \mid \mathtt{E}\ \mathtt{*}\ \mathtt{E} \\
\mathtt{C} \in \text{Command} & ::= & \mathtt{skip} \mid \mathtt{X}\mathtt{:=}\mathtt{E} \mid \mathtt{C}_1\mathtt{;}\mathtt{C}_2 \mid \mathtt{loop}\ \mathtt{E}\ \{\mathtt{C}\} \\
 & \mid & \mathtt{choose}\ \mathtt{C}_1\ \mathtt{or}\ \mathtt{C}_2
\end{array}$$

Fig. 1. Syntax of the core language.

Data. It is convenient to assume (without loss of generality) that the only type of data is non-negative integers. Note that a realistic (not "core") program may include many statements that manipulate non-integer data that are not relevant to loop control—so in a complexity analysis, we may be able to abstract these parts away and still analyze the variables of interest. In other cases, it is possible to preprocess a program to replace complex data values with their size (or "norm"), which is the quantity of importance for loop control. Methods for this process have been widely studied in conjunction with termination and cost analysis.

Command semantics. The core language is inherently non-deterministic. The `choose` command represents a non-deterministic choice, and can be used to abstract any concrete conditional command by simply ignoring the condition; this is necessary to ensure that our analysis problem is decidable. Note that what we ignore is branches within a loop body and not branches that implement the loop control, which we represent by a dedicated loop command. The command `loop E {C}` repeats `C` a (non-deterministic) number of times bounded by the value of `E`, which is evaluated just before the loop is entered. Thus, as a conservative abstraction, it may be used to model different forms of loops (for-loops, while-loops) as long as a bound on the number of iterations, as a function of the program state on loop initiation, can be determined and expressed in the language. There is an ample body of research on analysing programs to find such bounds where they are not explicitly given by the programmer; in particular, bounds can be obtained from a *ranking function* for the loop[23, 3, 2, 5, 6]. Note that the arithmetic in our language is too restricted to allow for the maintenance of counters and the creation of *while* loops, as there is no subtraction, no explicit constants and no tests. Thus, for realistic "concrete" programs which use such devices, loop-bound analysis is supposed to be performed *on the concrete program* as part of the process of abstracting it to the core language. This process is illustrated in [9, Sect. 2].

From a computability viewpoint, the use of bounded loops restricts the programs that can be represented to such that compute primitive recursive functions; this is a rich enough class to cover a lot of useful algorithms and make the analysis problem challenging. In fact, our language resembles a weakened version of Meyer and Ritchie's LOOP language [20], which computes all the primitive recursive functions, and where behavioral questions like "is the result linearly bounded" are undecidable.

```
loop X1 {
   loop X2 + X3 { choose { X3:= X1; X2:= X4 } or { X3:= X4; X2:= X1 } };
   X4:= X2 + X3
};
loop X4 { choose  { X3:= X1 + X2 + X3 } or { X3:= X2;  X2:= X1 } }
```

Fig. 2. A core-language program. `loop` n `C` means "do `C` at most n times."

1.2 The algorithm

Consider the program in Fig. 2. Suppose that it is started with the values of the variables $\mathtt{X}_1, \mathtt{X}_2, \ldots$ being $x_1, x_2, \ldots$. Our purpose is to bound the values of all variables at the conclusion of the program in terms of those initial values. Indeed, they are all polynomially bounded, and our algorithm provides tight bounds. For instance, it establishes that the final value of $\mathtt{X}_3$ is tightly bounded (up to a constant factor) by $\max(x_4(x_4 + x_1^2), x_4(x_2 + x_3 + x_1^2))$.

In fact, it produces information in a more precise form, as *a disjunction of simultaneous bounds*. This means that it generates vectors, called *multi-polynomials*, that give simultaneous bounds on all variables; for example, with the program in Fig. 2, one such multi-polynomial is $\langle x_1, x_2, x_3, x_4 \rangle$ (this is the result of all loops taking a very early exit). This form is important in the context of a compositional analysis. To see why, suppose that we provide, for a command with variables $\mathtt{X}, \mathtt{Y}$, the bounds $\langle x, y \rangle$ and $\langle y, x \rangle$. Then we know that the *sum* of their values is always bounded by $x + y$, a result that would have not been deduced had we given the bound $\max(x, y)$ on each of the variables. The difference may be critical for the success of analyzing an enclosing or subsequent command.

Multivariate bounds are often of interest, and perhaps require no justification, but let us point out that multivariate polynomials are necessary even if we're ultimately interested in a univariate bound, in terms of some single initial value, say n. This is, again, due to the analysis being compositional. When we analyze an internal command that uses variables $\mathtt{X}, \mathtt{Y}, \ldots$ we do not know in what possible contexts the command will be executed and how the values of these variables will be related to n.

Some highlights of our solution are as follows.

– We reduce the problem of analyzing any core-language program to the problem of analyzing a single loop, whose body is already processed, and therefore presented as a collection of multi-polynomials. This is typical of algorithms that analyze a structured imperative language and do so compositionally.
– Since we are computing bounds only up to a constant factor, we work with *abstract* polynomials, that have no numeric coefficients.
– We further introduce *τ-polynomials*, to describe the evolution of values in a loop. These have an additional parameter τ (for "time"; more precisely, number of iterations). Introducing τ-polynomials was a key step in the solution.
– The analysis of a loop is simply a closure computation under two operations: ordinary composition, and *generalization* which is the operation that predicts the evolution of values by judiciously adding τ's to *idempotent* abstract multi-polynomials.

The remainder of this paper is structured as follows. In Section 2 we give some definitions and state our main result. In Sections 3—5 we present our algorithm. In Section 6, we outline the correctness proofs. Section 7 considers related work, and Section 8 concludes and discusses ideas for further work.

2 Preliminaries

In this section, we give some basic definitions, complete the presentation of our programming language and precisely state the main result.

2.1 Some notation and terminology

The language We remark that in our language syntax there is no special form for a "program unit"; in the text we sometimes use "program" for the subject of our analysis, yet syntactically it's just a command.

Polynomials and multi-polynomials We work throughout this article with multivariate polynomials in $x_1, \dots, x_n$ that have non-negative integer coefficients and no variables other than $x_1, \dots, x_n$; when we speak of a polynomial we always mean one of this kind. Note that over the non-negative integers, such polynomials are monotonically (weakly) increasing in all variables.

The post-fix substitution operator $[a/b]$ may be applied to any sort of expression containing a variable b, to substitute a instead; e.g., $(x^2 + yx + y)[2z/y] = x^2 + 2zx + 2z$.

When discussing a command, state-transition, or program trace, with a variable $\mathtt{X}_i$, x_i will denote, as a rule, the initial value of this variable, and x'_i its final value. Thus we distinguish the syntactic entity by the typewriter font. We write the polynomials manipulated by our algorithms using the variable names x_i. We presume that an implementation of the algorithm represents polynomials concretely so that ordinary operations such as composition can be applied, but otherwise we do not concern ourselves much with representation.

The parameter n always refers to the number of variables in the subject program. The set $[n]$ is $\{1, \dots, n\}$. For a set S an n-tuple over S is a mapping from $[n]$ to S. The set of these tuples is denoted by S^n. Throughout the paper, various natural liftings of operators to collections of objects is tacitly assumed, e.g., if S is a set of integers then $S + 1$ is the set $\{s + 1 \mid s \in S\}$ and $S + S$ is $\{s + t \mid s, t \in S\}$. We use such lifting with sets as well as with tuples. If S is ordered, we extend the ordering to S^n by comparing tuples element-wise (this leads to a partial order, in general, e.g., with natural numbers, $\langle 1, 3\rangle$ and $\langle 2, 2\rangle$ are incomparable).

Definition 1. *A* polynomial transition (PT) *represents a mapping of an "input" state* $\mathbf{x} = \langle x_1, \dots, x_n\rangle$ *to a "result" state* $\mathbf{x}' = \langle x'_1, \dots, x'_n\rangle = \mathbf{p}(\boldsymbol{x})$ *where* $\mathbf{p} = \langle \mathbf{p}[1], \dots, \mathbf{p}[n]\rangle$ *is an n-tuple of polynomials. Such a* $\mathbf{p}$ *is called a* a multi-polynomial (MP)*; we denote by* `MPol` *the set of multi-polynomials, where the number of variables n is fixed by context.*

Multi-polynomials are used in this work to represent the effect of a command. Various operations will be applied to MPs, mostly obvious—in particular, composition (which corresponds to sequential application of the transitions). Note that composition of multi-polynomials, $\mathbf{q} \circ \mathbf{p}$, is naturally defined since $\mathbf{p}$ supplies

n values for the n variables of $\mathbf{q}$ (in other words, they are composed as functions in $\mathbb{N}^n \to \mathbb{N}^n$). We define *Id* to be the identity transformation, $\mathbf{x}' = \mathbf{x}$ (in MP notation: $\mathbf{p}[i] = x_i$ for $i = 1, \ldots, n$).

2.2 Formal semantics of the core language

The semantics associates with every command $\mathtt{C}$ over variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$ a relation $[\![\mathtt{C}]\!] \subseteq \mathbb{N}^n \times \mathbb{N}^n$. In the expression $\boldsymbol{x}[\![\mathtt{C}]\!]\boldsymbol{y}$, vector $\boldsymbol{x}$ (respectively $\boldsymbol{y}$) is the store before (after) the execution of $\mathtt{C}$.

The semantics of `skip` is the identity. The semantics of an assignment $\mathtt{X}_i\mathtt{:=E}$ associates to each store $\boldsymbol{x}$ a new store $\boldsymbol{y}$ obtained by replacing the component x_i by the value of the expression $\mathtt{E}$ when evaluated over store $\boldsymbol{x}$. This is defined in the natural way (details omitted), and is denoted by $[\![\mathtt{E}]\!]\boldsymbol{x}$. Composite commands are described by the straight-forward equations:

$$\begin{aligned}
[\![\mathtt{C}_1;\mathtt{C}_2]\!] &= [\![\mathtt{C}_2]\!] \circ [\![\mathtt{C}_1]\!] \\
[\![\mathtt{choose}\ \mathtt{C}_1\ \mathtt{or}\ \mathtt{C}_2]\!] &= [\![\mathtt{C}_1]\!] \cup [\![\mathtt{C}_2]\!] \\
[\![\mathtt{loop\ E\ \{C\}}]\!] &= \{(\boldsymbol{x}, \boldsymbol{y}) \mid \exists i \le [\![\mathtt{E}]\!]\boldsymbol{x} : \boldsymbol{x}[\![\mathtt{C}]\!]^i\boldsymbol{y}\}
\end{aligned}$$

where $[\![\mathtt{C}]\!]^i$ represents $[\![\mathtt{C}]\!] \circ \cdots \circ [\![\mathtt{C}]\!]$ (i occurrences of $[\![\mathtt{C}]\!]$); and $[\![\mathtt{C}]\!]^0 = Id$.

Remarks The following two changes may enhance the applicability of the core language for simulating certain concrete programs; we include them as "options" because they do not affect the validity of our proofs.

1. The semantics of an assignment operation may be non-deterministic: `X:=E` assigns to `X` some non-negative value *bounded* by `E`. This is useful to abstract expressions which are not in the core language, and also to use the results of size analysis of subprograms. Such an analysis may determine invariants such as "the value of `f(X,Y)` is at most the sum of `X` and `Y`."
2. The domain of the integer variables may be extended to $\mathbb{Z}$. In this case the bounds that we seek are on the absolute value of the output in terms of absolute values of the inputs. This change does not affect our conclusions because of the facts $|xy| = |x| \cdot |y|$ and $|x + y| \le |x| + |y|$. The semantics of the loop command may be defined either as doing nothing if the loop bound is not positive, or using the absolute value as a bound.

2.3 Detailed statement of the main result

The *polynomial-bound analysis problem* is to find, for any given command, which output variables are bounded by a polynomial in the input values (which are simply the values of all variables upon commencement of the program), and to bound these output values tightly (up to constant factors). The problem of *identifying* the polynomially-bounded variables is completely solved by [7]. We rely on that algorithm, which is polynomial-time, to do this for us (as further explained below).

Our main result is thus stated as follows.

Theorem 1. *There is an algorithm which, for a command* $\mathtt{C}$*, over variables* $\mathtt{X}_1$ *through* $\mathtt{X}_n$*, outputs a set* $\mathcal{B}$ *of multi-polynomials, such that the following hold, where* PB *is the set of indices* i *of variables* $\mathtt{X}_i$ *which are polynomially bounded under* $[\![\mathtt{C}]\!]$*.*

1. *(Bounding) There is a constant* $c_{\mathbf{p}}$ *associated with each* $\mathbf{p} \in \mathcal{B}$*, such that*

$$\forall \boldsymbol{x}, \boldsymbol{y} \ . \ \boldsymbol{x}[\![\mathtt{C}]\!]\boldsymbol{y} \Longrightarrow \exists \mathbf{p} \in \mathcal{B} \,. \forall i \in PB \,.\, y_i \leq c_{\mathbf{p}}\mathbf{p}[i](\boldsymbol{x})$$

2. *(Tightness) For every* $\mathbf{p} \in \mathcal{B}$ *there are constants* $d_{\mathbf{p}} > 0$*,* $\boldsymbol{x}_0$ *such that for all* $\boldsymbol{x} \geq \boldsymbol{x}_0$ *there is a* $\boldsymbol{y}$ *such that*

$$\boldsymbol{x}[\![\mathtt{C}]\!]\boldsymbol{y} \text{ and } \forall i \in PB \,.\, y_i \geq d_{\mathbf{p}}\mathbf{p}[i](\boldsymbol{x}).$$

3 Analysis Algorithm: First Concepts

The following sections describe our analysis algorithm. Naturally, the most intricate part of the analysis concerns loops. In fact we break the description into stages: first we reduce the problem of analyzing any program to that of analyzing *simple disjunctive loops*, defined next. Then, we approach the analysis of such loops, which is the main effort in this work.

Definition 2. *A* simple disjunctive loop (SDL) *is a finite set of PTs.*

The loop is "disjunctive" because its meaning is that in every iteration, any of the given transitions may be applied. The semantics is formalized by *traces* (Definition 4). A SDL does not specify the number of iterations; our analysis generates polynomials which depend on the number of iterations as well as the initial state. For this purpose, we now introduce τ-polynomials where τ represents the number of iterations.

Definition 3. τ*-polynomials are polynomials in* $x_1, \ldots, x_n$ *and* τ*.*

τ has a special status and does not have a separate component in the polynomial giving its value. If p is a τ-polynomial, then $p(v_1, \ldots, v_n)$ is the result of substituting each v_i for the respective x_i; and we also write $p(v_1, \ldots, v_n, t)$ for the result of substituting t for τ as well. The set of τ-polynomials in n variables (n known from context) is denoted $\tau\mathtt{Pol}$.

Multi-polynomials and polynomial transitions are formed from τ-polynomials just as previously defined and are used to represent the effect of a variable number of iterations. For example, the τ-polynomial transition $\langle x_1', x_2' \rangle = \langle x_1,\ x_2 + \tau x_1 \rangle$ represents the effect of repeating (τ times) the assignment $\mathtt{X}_2\mathtt{:=}\ \mathtt{X}_2 + \mathtt{X}_1$. The effect of iterating the composite command: $\mathtt{X}_2\mathtt{:=}\ \mathtt{X}_2 + \mathtt{X}_1\mathtt{;}\ \mathtt{X}_3\mathtt{:=}\ \mathtt{X}_3 + \mathtt{X}_2$ has an effect described by $\mathbf{x}' = \langle x_1,\ x_2 + \tau x_1,\ x_3 + \tau x_2 + \tau^2 x_1 \rangle$ (here we already have an upper bound which is not reached precisely, but is correct up to a constant factor). We denote the set of τ-polynomial transitions by $\tau\mathbf{MPol}$. We should note that composition $\mathbf{q} \circ \mathbf{p}$ over $\tau\mathbf{MPol}$ is performed by substituting $\mathbf{p}[i]$ for each occurrence of x_i in $\mathbf{q}$. Occurrences of τ are unaffected (since τ is not part of the state). We make a couple of preliminary definitions before reaching our goal which is the definition of the *simple disjunctive loop problem* (Definition 6).

Definition 4. *Let $\mathcal{S}$ be a set of polynomial transitions. An* (abstract) trace *over $\mathcal{S}$ is a finite sequence $\mathbf{p}_1; \ldots; \mathbf{p}_{|\sigma|}$ of elements of $\mathcal{S}$. Thus $|\sigma|$ denotes the* length *of the trace. The set of all traces is denoted $\mathcal{S}^*$. We write $[\![\sigma]\!]$ for the composed relation $\mathbf{p}_{|\sigma|} \circ \cdots \circ \mathbf{p}_1$ (for the empty trace, ε, we have $[\![\varepsilon]\!] = Id$).*

Definition 5. *Let $p(\mathbf{x})$ be a (concrete or abstract) τ-polynomial. We write $\dot{p}$ for the sum of* linear monomials *of p, namely any one of the form ax_i with constant coefficient a. We write $\ddot{p}$ for the rest. Thus $p = \dot{p} + \ddot{p}$.*

Definition 6 (Simple disjunctive loop problem). *The* simple disjunctive loop problem *is: given the set $\mathcal{S}$, find (if possible) a finite set $\mathcal{B}$ of τ-polynomial transitions which* tightly bound *all traces over $\mathcal{S}$. More precisely, we require:*

1. *(Bounding) There is a constant $c_{\mathbf{p}} > 0$ associated with each $\mathbf{p} \in \mathcal{B}$, such that*
$$\forall \boldsymbol{x}, \boldsymbol{y}, \sigma \;.\; \boldsymbol{x}[\![\sigma]\!]\boldsymbol{y} \Longrightarrow \exists \mathbf{p} \in \mathcal{B} \,.\, \boldsymbol{y} \leq c_{\mathbf{p}}\mathbf{p}(\boldsymbol{x}, |\sigma|)$$
2. *(Tightness) For every $\mathbf{p} \in \mathcal{B}$ there are constants $d_{\mathbf{p}} > 0$, $\boldsymbol{x}_0$ such that for all $\boldsymbol{x} \geq \boldsymbol{x}_0$ there are a trace σ and a state vector $\boldsymbol{y}$ such that*
$$\boldsymbol{x}[\![\sigma]\!]\boldsymbol{y} \wedge \boldsymbol{y} \geq \dot{\mathbf{p}}(\boldsymbol{x}, |\sigma|) + d_{\mathbf{p}}\ddot{\mathbf{p}}(\boldsymbol{x}, |\sigma|)\,.$$

Note that in the lower-bound clause (2), the linear monomials of p are not multiplied, in the left-hand side, by the coefficient $d_{\mathbf{p}}$; this sets, in a sense, a stricter requirement for them: if the trace maps x to x^2 then the bound $2x^2$ is acceptable, but if it maps x to x, the bound $2x$ is not accepted. The reader may understand this technicality by considering the effect of iteration: it is important to distinguish the transition $x_1' = x_1$, which can be iterated ad libitum, from the transition $x_1' = 2x_1$, which produces exponential growth on iteration. Distinguishing $x_1' = x_1^2$ from $x_1' = 2x_1^2$ is not as important. The result set $\mathcal{B}$ above is sometimes called a *loop summary*. We remark that Definition 6 implies that the **max** of all these polynomials provides a "big Theta" bound for the worst-case (namely biggest) results of the loop's computation. We prefer, however, to work with sets of polynomials. Another technical remark is that $c_{\mathbf{p}}, d_{\mathbf{p}}$ range over real numbers. However, our data and the coefficients of polynomials remain integers, it is only such comparisons that are performed with real numbers (specifically, to allow $c_{\mathbf{p}}$ to be smaller than one).

4 Reduction to Simple Disjunctive Loops

We show how to reduce the problem of analysing core-language programs to the analysis of polynomially-bounded simple disjunctive loops.

4.1 Symbolic evaluation of straight-line code

Straight-line code consists of atomic commands—namely assignments (or `skip`, equivalent to `X`$_1$`:=X`$_1$), composed sequentially. It is obvious that symbolic evaluation of such code leads to polynomial transitions.

Example 1. `X`$_2$`:= X`$_1$`; X`$_4$`:= X`$_2$` + X`$_3$`; X`$_1$`:= X`$_2$` * X`$_3$
is precisely represented by the transition $\langle x_1, x_2, x_3 \rangle' = \langle x_1x_3,\ x_1,\ x_3,\ x_1 + x_3 \rangle$.

4.2 Evaluation of non-deterministic choice

Evaluation of the command `choose` $\mathtt{C}_1$ `or` $\mathtt{C}_2$ yields a set of possible outcomes. Hence, the result of analyzing a command will be a *set* of multi-polynomial transitions. We express this in the common notation of abstract semantics:

$$[\![\mathtt{C}]\!]^S \in \wp(\mathtt{MPol})\,.$$

For uniformity, we consider $[\![\mathtt{C}]\!]^S$ for an atomic command to be a singleton in $\wp(\mathtt{MPol})$ (this means that we represent a transition $\boldsymbol{x}' = \mathbf{p}(\boldsymbol{x})$ by $\{\mathbf{p}\}$). Composition is naturally extended to sets, and the semantics of a choice command is now simply set union, so we have:

$$\begin{aligned}
[\![\mathtt{C}_1;\mathtt{C}_2]\!]^S &= [\![\mathtt{C}_2]\!]^S \circ [\![\mathtt{C}_1]\!]^S \\
[\![\mathtt{choose}\ \mathtt{C}_1\ \mathtt{or}\ \mathtt{C}_2]\!]^S &= [\![\mathtt{C}_1]\!]^S \cup [\![\mathtt{C}_2]\!]^S
\end{aligned}$$

Example 2. $\mathtt{X}_2$`:=` $\mathtt{X}_1$`; choose {` $\mathtt{X}_4$`:=` $\mathtt{X}_2$ `+` $\mathtt{X}_3$ `} or {` $\mathtt{X}_1$`:=` $\mathtt{X}_2$ `*` $\mathtt{X}_3$ `}`
is represented by the set $\{\langle x_1, x_1, x_3, x_1 + x_3\rangle,\ \langle x_1x_3, x_1, x_3, x_4\rangle\}$.

4.3 Handling loops

The above shows that any loop-free command in our language can be precisely represented by a finite set of PTs. Consequently, the problem of analyzing *any* command is reduced to the analysis of simple disjunctive loops.

Suppose that we have an algorithm SOLVE that takes a simple disjunctive loop and computes tight bounds for it (see Definition 6). We use it to complete the analysis of any program by the following definition:

$$[\![\mathtt{loop\ E\ \{C\}}]\!]^S = (\textsc{Solve}([\![\mathtt{C}]\!]^S))[E/\tau]\ .$$

Thus, the whole solution is constructed as an ordinary abstract interpretation, following the semantics of the language, except for procedure SOLVE, described below.

Example 3. $\mathtt{X}_4$`:=` $\mathtt{X}_1$`; loop` $\mathtt{X}_4$ `{` $\mathtt{X}_2$`:=` $\mathtt{X}_1$ `+` $\mathtt{X}_2$`;` $\mathtt{X}_3$`:=` $\mathtt{X}_2$ `}`.
The loop includes just one PT. Solving the loop yields a set $\mathcal{L} = \{\langle x_1, x_2, x_3, x_4\rangle,$ $\langle x_1, x_2 + \tau x_1, x_2 + \tau x_1, x_4\rangle\}$ (the first MP accounts for zero iterations, the second covers any positive number of iterations). We can now compute the effect of the given command as

$$\begin{aligned}
\mathcal{L}[x_4/\tau] \circ [\![\mathtt{X}_4\ \mathtt{:=}\ \mathtt{X}_1]\!]^S &= \mathcal{L}[x_4/\tau] \circ \{\langle x_1, x_2, x_3, x_1\rangle\} \\
&= \{\langle x_1, x_2, x_3, x_1\rangle, \langle x_1, x_2 + x_1^2, x_2 + x_1^2, x_1\rangle\}.
\end{aligned}$$

The next section describes procedure SOLVE, and operates under the assumption that all variables are polynomially bounded in the loop. However, a loop can generate exponential growth. To cover this eventuality, we first apply the algorithm of [7] which identifies which variables are polynomially bounded. If some $\mathtt{X}_i$ is *not* polynomially bounded we replace the ith component of all the loop transitions with x_n (recall that we assume x_n to be a dedicated, unmodified variable). Clearly, after this change, all variables are polynomially bounded; moreover, variables which are genuinely polynomial are unaffected, because they cannot depend on a super-exponential quantity (given the restricted arithmetics in our language). In reporting the results of the algorithm, we should display "`super-polynomial`" instead of all bounds that depend on x_n.

5 Simple Disjunctive Loop Analysis Algorithm

Intuitively, evaluating `loop E {C}` abstractly consists of simulating any finite number of iterations, i.e., computing

$$Q_i = \{Id\} \cup P \cup (P \circ P) \cup \cdots \cup P^{(i)} \tag{1}$$

where $P = [\![\mathtt{C}]\!]^S \in \wp(\mathtt{MPol})$. The question now is whether the sequence (1) reaches a fixed point. In fact, it often doesn't. However, it is quite easy to see that in the *multiplicative fragment* of the language, that is, where the addition operator is not used, such non-convergence is associated with exponential growth. Indeed, since there is no addition, all our polynomials are monomials with a leading coefficient of 1 (*monic monomials*)—this is easy to verify. It follows that if the sequence (1) does not converge, higher and higher exponents must appear, which indicates that some variable cannot be bounded polynomially. Taking the contrapositive, we conclude that if all variables are known to be polynomially bounded the sequence will converge. Thus we have the following easy (and not so satisfying) result:

Observation 2 *For a SDL that does not use addition, the sequence Q_i as in* (1) *reaches a fixed point, and the fixed point provides tight bounds for all the polynomially-bounded variables.*

When we have addition, we find that knowing that all variables are polynomially bounded does not imply convergence of the sequence (1). An example is: `loop` $\mathtt{X}_3$ `{` $\mathtt{X}_1$`:=` $\mathtt{X}_1$ `+` $\mathtt{X}_2$ `}` yielding the infinite sequence of MPs $\langle x_1, x_2, x_3\rangle$, $\langle x_1 + x_2, x_2, x_3\rangle$, $\langle x_1 + 2x_2, x_2, x_3\rangle$, $\ldots$ Our solution employs two means. One is the introduction of τ-polynomials, already presented. The other is a kind of *abstraction*—intuitively, ignoring the concrete values of (non-zero) coefficients. Let us first define this abstraction:

Definition 7. `APol`*, the set of abstract polynomials, consists of formal sums of distinct monomials over $x_1, \ldots, x_n$, where the coefficient of every monomial included is* 1*. We extend the definition to an abstraction of τ-polynomials, denoted* τ`APol`*.*

The meaning of abstract polynomials is given by the following rules:

1. The abstraction of a polynomial p, $\alpha(p)$, is obtained by modifying all (non-zero) coefficients to 1.

2. Addition and multiplication in τ`APol` is defined in a natural way so that $\alpha(p) + \alpha(q) = \alpha(p + q)$ and $\alpha(p) \cdot \alpha(q) = \alpha(p \cdot q)$ (to carry these operations out, you just go through the motions of adding or multiplying ordinary polynomials, ignoring the coeffcient values).
3. The *canonical concretization* of an abstract polynomial, $\gamma(\mathbf{p})$ is obtained by simply regarding it as an ordinary polynomial.
4. These definitions extend naturally to tuples of (abstract) polynomials.
5. The set of abstract multi-polynomials `AMPol` and their extension with τ (τ`AMPol`) are defined as n-tuples over `APol` (respectively, τ`APol`). We use AMP as an abbreviation for abstract multi-polynomial.
6. Composition $\mathbf{p} \bullet \mathbf{q}$, for $\mathbf{p}, \mathbf{q} \in \texttt{AMPol}$ (or $\tau\texttt{AMPol}$) is defined as $\alpha(\gamma(\mathbf{p}) \circ \gamma(\mathbf{q}))$; it is easy to see that one can perform the calculation without the detour through polynomials with coefficients. The different operator symbol ("$\bullet$" versus "$\circ$") helps in disambiguating expressions.

Analysing a SDL. To analyse a SDL specified by a set of MPs $\mathcal{S}$, we start by computing $\alpha(\mathcal{S})$. The rest of the algorithm computes within τ`AMPol`. We define two operations that are combined in the analysis of loops. The first, which we call *closure*, is simply the fixed point of accumulated iterations as in the multiplicative case. It is introduced by the following two definitions.

Definition 8 (iterated composition). *Let* $\mathbf{t}$ *be any abstract* τ*-MP. We define* $\mathbf{t}^{\bullet(n)}$*, for* $n \geq 0$*, by:*

$$\mathbf{t}^{\bullet(0)} = Id$$
$$\mathbf{t}^{\bullet(n+1)} = \mathbf{t} \bullet \mathbf{t}^{\bullet(n)}.$$

For a set $\mathcal{T}$ *of abstract* τ*-MPs, we define, for* $n \geq 0$*:*

$$\mathcal{T}^{\bullet(0)} = \{Id\}$$
$$\mathcal{T}^{\bullet(n+1)} = \mathcal{T}^{\bullet(n)} \cup \bigcup_{\mathbf{q} \in \mathcal{T},\ \mathbf{p} \in \mathcal{T}^{\bullet(n)}} \mathbf{q} \bullet \mathbf{p}\,.$$

Note that $\mathbf{t}^{\bullet(n)} = \alpha(\gamma(\mathbf{t})^{(n)})$, where $\mathbf{p}^{(n)}$ is defined using ordinary composition.

Definition 9 (abstract closure). *For finite* $P \subset \tau\texttt{AMPol}$*, we define:*

$$Cl(P) = \bigcup_{i=0}^{\infty} P^{\bullet(i)}\,.$$

In the correctness proof, we argue that when all variables are polynomially bounded in a loop $\mathcal{S}$, the closure of $\alpha(\mathcal{S})$ can be computed in finite time; equivalently, it equals $\bigcup_{i=0}^{k}(\alpha(\mathcal{S}))^{\bullet(i)}$ for some k. The argument is essentially the same as in the multiplicative case.

The second operation is called *generalization* and its role is to capture the behaviour of accumulator variables, meaning variables that grow by accumulating increments in the loop, and make explicit the dependence on the number of iterations. The identification of which additive terms in a MP should be considered as increments that accumulate is at the heart of our problem, and is greatly simplified by concentrating on idempotent AMPs.

Definition 10. $\mathbf{p} \in \tau\texttt{AMPol}$ *is called* idempotent *if* $\mathbf{p} \bullet \mathbf{p} = \mathbf{p}$.

Note that this is composition in the abstract domain. So, for instance, $\langle x_1, x_2\rangle$ is idempotent, and so is $\langle x_1 + x_2, x_2\rangle$, while $\langle x_1x_2, x_2\rangle$ and $\langle x_1 + x_2, x_1\rangle$ are not.

Definition 11. *For* $\mathbf{p}$ *an (abstract) multi-polynomial, we say that* x_i *is* self-dependent *in* $\mathbf{p}$ *if* $\mathbf{p}[i]$ *depends on* x_i*. We call a monomial self-dependent if all the variables appearing in it are.*

Definition 12. *We define a notational convention for* τ*-MPs. Assuming that* $\mathbf{p}[i]$ *depends on* x_i*, we write*

$$\mathbf{p}[i] = x_i + \tau\mathbf{p}[i]' + \mathbf{p}[i]'' + \mathbf{p}[i]''' \,,$$

where $\mathbf{p}[i]'''$ *includes all the non-self-dependent monomials of* $\mathbf{p}[i]$*, while the self-dependent monomials (other than* x_i*) are grouped into two sums:* $\tau\mathbf{p}[i]'$*, including all monomials with a positive degree of* τ*, and* $\mathbf{p}[i]''$ *which includes all the* τ*-free monomials.*

Example 4. Let $\mathbf{p} = \langle x_1 + \tau x_2 + \tau x_3 + x_3x_4,\ x_3,\ x_3,\ x_4\rangle$. The self-dependent variables are all but x_2. Since x_1 is self-dependent, we will apply the above definition to $\mathbf{p}[1]$, so that $\mathbf{p}[1]' = x_3$, $\mathbf{p}[1]'' = x_3x_4$ and $\mathbf{p}[1]''' = \tau x_2$. Note that a factor of τ is stripped in $\mathbf{p}[1]'$. Had the monomial been $\tau^2 x_3$, we would have $\mathbf{p}[1]' = \tau x_3$.

Definition 13 (generalization). *Let* $\mathbf{p}$ *be idempotent in* $\tau\texttt{AMPol}$*; define* $\mathbf{p}^\tau$ *by*

$$\mathbf{p}^\tau[i] = \begin{cases} x_i + \tau\mathbf{p}[i]' + \tau\mathbf{p}[i]'' + \mathbf{p}[i]''' & \textit{if } \mathbf{p}[i] \textit{ depends on } x_i \\ \mathbf{p}[i] & \textit{otherwise.} \end{cases}$$

Note that the arithmetic here is abstract (see examples below). Note also that in the term $\tau\mathbf{p}[i]'$ the τ is already present in $\mathbf{p}$, while in $\tau\mathbf{p}[i]''$ it is added to existing monomials. In this definition, the monomials of $\mathbf{p}[i]'''$ are treated like those of $\tau\mathbf{p}[i]'$; however, in certain steps of the proofs we treat them differently, which is why the notation separates them.

Example 5. Let $\mathbf{p} = \langle x_1 + x_3,\ x_2 + x_3 + x_4,\ x_3,\ x_3\rangle$

Note that $\mathbf{p} \bullet \mathbf{p} = \mathbf{p}$. We have $\mathbf{p}^\tau = \langle x_1 + \tau x_3,\ x_2 + \tau x_3 + x_4,\ x_3,\ x_3\rangle$.

Example 6. Let $\mathbf{p} = \langle x_1 + \tau x_2 + \tau x_3 + \tau x_3x_4,\ x_3,\ x_3,\ x_4\rangle$

Note that $\mathbf{p} \bullet \mathbf{p} = \mathbf{p}$. The self-dependent variables are all but x_2.

We have $\mathbf{p}^\tau = \langle x_1 + \tau x_2 + \tau x_3 + \tau x_3x_4,\ x_3,\ x_3,\ x_4\rangle = \mathbf{p}$.

Finally we can present the analysis of the loop command.

```
Algorithm  Solve($\mathcal{S}$)
Input: $\mathcal{S}$, a polynomially-bounded disjunctive simple loop
Output: a set of $\tau$-MPs which tightly approximates the effect of all $\mathcal{S}$-traces.
 1. Set $T = \alpha(\mathcal{S})$.
 2. Repeat the following steps until $T$ remains fixed:
    (a) Closure: Set $T$ to $Cl(T)$.
    (b) Generalization: For all $\mathbf{p} \in T$ such that $\mathbf{p} \bullet \mathbf{p} = \mathbf{p}$, add $\mathbf{p}^\tau$ to $T$.
```

Example 7. `loop` $\mathtt{X}_3$ `{` $\mathtt{X}_1$`:=` $\mathtt{X}_1$ `+` $\mathtt{X}_2$`;` $\mathtt{X}_2$`:=` $\mathtt{X}_2$ `+` $\mathtt{X}_3$`;` $\mathtt{X}_4$`:=` $\mathtt{X}_3$ `}`
The body of the loop is evaluated symbolically and yields the multi-polynomial:

$$\mathbf{p} = \langle x_1 + x_2,\ x_2 + x_3,\ x_3,\ x_3 \rangle$$

Now, computing within `AMPol`,

$$\begin{aligned}
\alpha(\mathbf{p})^{\bullet(2)} &= \alpha(\mathbf{p}) \bullet \alpha(\mathbf{p}) = \langle x_1 + x_2 + x_3,\ x_2 + x_3,\ x_3,\ x_3 \rangle; \\
\alpha(\mathbf{p})^{\bullet(3)} &= \alpha(\mathbf{p})^{\bullet(2)} .
\end{aligned}$$

Here the closure computation stops. Since $\alpha(\mathbf{p}^{\bullet(2)})$ is idempotent, we compute

$$\mathbf{q} = (\alpha(\mathbf{p})^{\bullet(2)})^\tau = \langle x_1 + \tau x_2 + \tau x_3,\ x_2 + \tau x_3,\ x_3,\ x_3 \rangle$$

and applying closure again, we obtain some additional results:

$$\begin{aligned}
\mathbf{q} \bullet \alpha(\mathbf{p}) &= \langle x_1 + x_2 + x_3 + \tau x_2 + \tau x_3,\ x_2 + x_3 + \tau x_3,\ x_3,\ x_3 \rangle \\
(\mathbf{q})^{\bullet(2)} &= \langle x_1 + \tau x_2 + \tau x_3 + \tau^2 x_3,\ x_2 + \tau x_3,\ x_3,\ x_3 \rangle \\
(\mathbf{q})^{\bullet(2)} \bullet \alpha(\mathbf{p}) &= \langle x_1 + x_2 + x_3 + \tau x_2 + \tau x_3 + \tau^2 x_3,\ x_2 + x_3 + \tau x_3,\ x_3,\ x_3 \rangle
\end{aligned}$$

The last element is idempotent but applying generalization does not generate anything new. Thus the algorithm ends. The reader may reconsider the source code to verify that we have indeed obtained tight bounds for the loop.

6 Correctness

We claim that our algorithm obtains a description of the worst-case results of the program that is precise up to constant factors. That is, we claim that the set of MPs returned provides an upper bound (on all executions) which is also tight; tightness means that every MP returned is also a lower bound (up to a constant factor) on an infinite sequence of possible executions. Unfortunately, due to space constraints, we are not able to give full details of the proofs here; however, we give the main highlights. Intuitively, what we want to prove is that the multi-polynomials we compute cover all "behaviors" of the loop. More precisely, in the upper-bound part of the proof we want to cover all behaviors: upper-bounding is a universal statement. To prove that bounds are tight, we show that each such bound constitutes a *lower bound* on a certain "worst-case behavior": tightness is an existential statement. The main aspects of these proofs are as follows:

- A key notion in our proofs is that of *realizability*. Intuitively, when we come up with a bound, we want to show that there are traces that achieve (realize) this bound for arbitrarily large input values.
- In the lower-bound proof, we describe a "behavior" by a *pattern*. A pattern is constructed like a regular expression with concatenation and Kleene-star. However, they allow no nested iteration constructs, and the starred sub-expressions have to be repeated the same number of times; for example, the pattern $\mathbf{p}^*\mathbf{q}^*$ generates the traces $\{\mathbf{p}^t\mathbf{q}^t,\ t \geq 0\}$. The proof constructs a pattern for every multi-polynomial computed, showing it is realizable. It is interesting that such simple patterns suffice to establish tight lower bounds for all our programs.
- In the upper-bound proof, we describe all "behaviors" by a finite set of *well-typed regular expressions* [10]. This elegant tool channels the power of the Factorization Forest Theorem [25]; this brings out the role of idempotent elements, which is key in our algorithm.
- Interestingly, the lower-bound proof not only justifies the tightness of our upper bounds, it also justifies the termination of the algorithm and the application of the Factorization Forest Theorem in the upper-bound proof, because it shows that our abstract multi-polynomials generate a finite monoid.

7 Related Work

Bound analysis, in the sense of finding symbolic bounds for data values, iteration bounds and related quantities, is a classic field of program analysis [27, 24, 18]. It is also an area of active research, with tools being currently (or recently) developed including COSTA [1], AProVE [13], CiaoPP [19] , C^4B [11], Loopus [26]—all for imperative programs. There is also work on functional and logic programs, term rewriting systems, recurrence relations, etc. which we cannot attempt to survey here. In the rest of this section we survey work which is more directly related to ours, and has even inspired it.

The LOOP language is due to Meyer and Ritchie [20], who note that it computes only primitive recursive functions, but complexity can rise very fast, even for programs with nesting-depth 2. Subsequent work [16, 17, 22, 15] concerning similar languages attempted to analyze such programs more precisely; most of them proposed syntactic criteria, or analysis algorithms, that are sufficient for ensuring that the program lies in a desired class (often, polynomial-time programs), but are not both necessary and sufficient: thus, they do not prove decidability (the exception is [17] which has a decidability result for a weak "core" language). The core language we use in this paper is from Ben-Amram et al. [7], who observed that by introducing weak bounded loops instead of concrete loop commands and non-deterministic branching instead of "`if`", we have weakened the semantics just enough to obtain decidability of polynomial growth-rate. Justifying the necessity of these relaxations, [8] showed undecidability for a language that can only do addition and definite loops (that cannot exit early).

In the vast literature on bound analysis in various forms, there are a few other works that give a complete solution for a weak language. *Size-change programs* are considered by [12, 28]. Size-change programs abstract away nearly everything in the program, leaving a control-flow graph annotated with assertions about variables which decrease (or do not increase) in a transition. Thus, it does not assume structured and explicit loops, and it cannot express information about values which increase. Both works yield tight bounds on the number of transitions until termination.

Dealing with a somewhat different problem, [21, 14] both check, or find, *invariants* in the form of polynomial equations. We find it remarkable that they give complete solutions for weak languages, where the weakness lies in the non-deterministic control-flow, as in our language. If one could give a complete solution for polynomial *inequalities*, this would have implied a solution to our problem as well.

8 Conclusion and Further Work

We have solved an open problem in the area of analyzing programs in a simple language with bounded loops. For our language, it has been previously shown that it is possible to decide whether a variable's value, number of steps in the program, etc. are polynomially bounded or not. Now, we have an algorithm that computes tight polynomial bounds on the final values of variables in terms of initial values. The bounds are tight up to constant factors (suitable constants are also computable). This result improves our understanding of what is computable by, and about, programs of this form. An interesting corollary of our algorithm is that as long as variables are *polynomially bounded*, their worst-case bounds are described tightly by (multivariate) *polynomials*. This is, of course, not true for common Turing-complete languages. Another interesting corollary of the *proofs* is the definition of a simple class of patterns that suffice to realize the worst-case behaviors. This will appear in a planned extended version of this paper.

There are a number of possible directions for further work. We would like to look for decidability results for richer (yet, obviously, sub-recursive) languages. Some possible language extensions include deterministic loops, variable resets (cf. [4]), explicit constants, and procedures. The inclusion of explicit constants is a particularly challenging open problem.

Rather than extending the language, we could extend the range of bounds that we can compute. In light of the results in [17], it seems plausible that the approach can be extended to classify the Grzegorczyk-degree of the growth rate of variables when they are super-polynomial. There may also be room for progress regarding precise bounds of the form 2^{poly}.

In terms of time complexity, our algorithm is polynomial in the size of the program times n^{nd}, where d is the highest degree of any MP computed. Such exponential behavior is to be expected, since a program can be easily written to compute a multivariate polynomial that is exponentially long to write. But there is still room for finer investigation of this issue.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science **413**(1), 142–159 (2012). https://doi.org/10.1016/j.tcs.2011.07.009
2. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) Static Analysis, Proceedings of the 17th International Symposium, Perpignan, France. Lecture Notes in Computer Science, vol. 6337, pp. 117–133. Springer (2010). https://doi.org/10.1007/978-3-642-15769-1_8
3. Bagnara, R., Hill, P.M., Zaffanella, E.: The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program. **72**(1-2), 3–21 (2008)
4. Ben-Amram, A.M.: On decidable growth-rate properties of imperative programs. In: Baillot, P. (ed.) International Workshop on Developments in Implicit Computational complExity (DICE 2010). EPTCS, vol. 23, pp. 1–14 (2010). https://doi.org/10.4204/EPTCS.23.1
5. Ben-Amram, A.M., Genaim, S.: Ranking functions for linear-constraint loops. Journal of the ACM **61**(4), 26:1–26:55 (jul 2014). https://doi.org/10.1145/2629488
6. Ben-Amram, A.M., Genaim, S.: On multiphase-linear ranking functions. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification, CAV'17. Lecture Notes in Computer Science, vol. 10427, pp. 601–620. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_32
7. Ben-Amram, A.M., Jones, N.D., Kristiansen, L.: Linear, polynomial or exponential? complexity inference in polynomial time. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008. LNCS, vol. 5028, pp. 67–76. Springer (2008). https://doi.org/10.1007/978-3-540-69407-6_7
8. Ben-Amram, A.M., Kristiansen, L.: On the edge of decidability in complexity analysis of loop programs. International Journal on the Foundations of Computer Science **23**(7), 1451–1464 (2012). https://doi.org/10.1142/S0129054112400588
9. Ben-Amram, A.M., Pineles, A.: Flowchart programs, regular expressions, and decidability of polynomial growth-rate. In: Hamilton, G., Lisitsa, A., Nemytykh, A.P. (eds.) Proceedings of the Fourth International Workshop on Verification and Program Transformation (VPT). EPTCS, vol. 216, pp. 24–49 (2016). https://doi.org/10.4204/EPTCS.216.2
10. Bojańczyk, M.: Factorization forests. In: Diekert, V., Nowotka, D. (eds.) Developments in Language Theory, 13th International Conference, DLT 2009, Stuttgart, Germany, June 30 - July 3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5583, pp. 1–17. Springer (2009). https://doi.org/10.1007/978-3-642-02737-6_1
11. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation (PLDI). ACM (2015)
12. Colcombet, T., Daviaud, L., Zuleger, F.: Size-change abstraction and max-plus automata. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.) Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014. Proceedings, Part I. LNCS, vol. 8634, pp. 208–219. Springer (2014). https://doi.org/10.1007/978-3-662-44522-8_18
13. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swider-

ski, S., Thiemann, R.: Analyzing program termination and complexity automatically with aprove. Journal of Automated Reasoning **58**(1), 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y

14. Hrushovski, E., Ouaknine, J., Pouly, A., Worrell, J.: Polynomial invariants for affine programs. In: Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 530–539. LICS '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3209108.3209142
15. Jones, N.D., Kristiansen, L.: A flow calculus of mwp-bounds for complexity analysis. ACM Trans. Computational Logic **10**(4), 1–41 (2009), https://doi.org/10.1145/1555746.1555752
16. Kasai, T., Adachi, A.: A characterization of time complexity by simple loop programs. Journal of Computer and System Sciences **20**(1), 1–17 (1980). https://doi.org/10.1016/0022-0000(80)90001-X
17. Kristiansen, L., Niggl, K.H.: On the computational complexity of imperative programming languages. Theor. Comp. Sci. **318**(1-2), 139–161 (2004). https://doi.org/10.1016/j.tcs.2003.10.016
18. Le Métayer, D.: Ace: an automatic complexity evaluator. ACM Trans. Program. Lang. Syst. **10**(2), 248–266 (1988), https://doi.org/10.1145/42190.42347
19. López-García, P., Darmawan, L., Klemen, M., Liqat, U., Bueno, F., Hermengildo, M.V.: Interval-based resource usage verification by translation into Horn clauses and an application to energy consumption. Theory and Practice of Logic Programming **18**(2), 167–223 (2018)
20. Meyer, A.R., Ritchie, D.M.: The complexity of loop programs. In: Proc. 22nd ACM National Conference. pp. 465–469. Washington, DC (1967)
21. Müller-Olm, M., Seidl, H.: Computing polynomial program invariants. Information Processing Letters **91**(5), 233–244 (2004). https://doi.org/10.1016/j.ipl.2004.05.004
22. Niggl, K.H., Wunderlich, H.: Certifying polynomial time and linear/polynomial space for imperative programs. SIAM J. Comput **35**(5), 1122–1147 (2006), https://doi.org/10.1137/S0097539704445597
23. Podelski, A., Rybalchenko, A.: A complete method for synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2003: Verification, Model Checking, and Abstract Interpretation. LNCS, vol. 2937, pp. 239–251. Springer (2004)
24. Rosendahl, M.: Automatic complexity analysis. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture FPCA'89. pp. 144–156. ACM (1989). https://doi.org/10.1145/99370.99381
25. Simon, I.: Factorization forests of finite height. Theoretical Computer Science **72**(1), 65–94 (1990). https://doi.org/10.1016/0304-3975(90)90047-L, https://doi.org/10.1016/0304-3975(90)90047-L
26. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. Journal of Automated Reasoning **59**(1), 3–45 (2017). https://doi.org/10.1007/s10817-016-9402-4
27. Wegbreit, B.: Mechanical program analysis. Communications of the ACM **18**(9), 528–539 (1975). https://doi.org/10.1145/361002.361016
28. Zuleger, F.: Asymptotically precise ranking functions for deterministic size-change systems. In: Computer Science—Theory and Applications—10th International Computer Science Symposium in Russia, CSR 2015, Listvyanka, Russia, July 13-17, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9139, pp. 426–442. Springer (2015). https://doi.org/10.1007/978-3-319-20297-6_27