Logical Methods in Computer Science
Volume 16, Issue 2, 2020, pp. 4:1–4:39
https://lmcs.episciences.org/

Submitted Jun. 25, 2019
Published May 13, 2020

TIGHT POLYNOMIAL WORST-CASE BOUNDS FOR LOOP PROGRAMS *

AMIR M. BEN-AMRAM [a] AND GEOFF HAMILTON [b]

[a] Qiryat Ono, Israel
e-mail address: amirben@mta.ac.il

[b] School of Computing, Dublin City University, Ireland
e-mail address: 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 move from answering the decision problem to giving a quantitative result, namely, a tight polynomial upper bound. This paper shows how to obtain *asymptotically-tight*, *multivariate*, *disjunctive* polynomial bounds for this class of programs. This is a complete solution: whenever a polynomial bound exists it will be found.

A pleasant surprise is that the algorithm is quite simple; but it relies on some subtle reasoning. An important ingredient in the proof is the *forest factorization theorem*, a strong structural result on homomorphisms into a finite monoid.

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. [Weg75, Ros89, LM88, AAG+12, GAB+17, LDK+18, CHS15, SZV17]); 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

Key words and phrases: asymptotically-tight, multivariate, disjunctive, worst-case, polynomial bounds.
* A preliminary version of this paper appeared in FoSSaCS 2019, LNCS 11425:80–97.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.23638/LMCS-16(2:4)2020

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 [BJK08]. 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 an imperative language, including bounded loops, non-deterministic branches and restricted arithmetic expressions; the syntax is shown in Figure 1. The semantics is explained and motivated below, but is largely intuitive; see also the illustrative example in Figure 2. In 2008, it was proved [BJK08] 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 [BJK08]. 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.

Our philosophy is that research on complete solutions to static analysis questions regarding weak languages is desirable for several reasons. First, it is theoretically satisfying—it establishes a clear and definite result. The algorithm can be possibly employed later in more complex situations, and we will at least have an assurance that it does its part; arguments about the value of relying on decidable problems in program analysis have recently been given in [MP18], and [K18] gives a methodology for incorporating them as parts in a bigger system (albeit for safety problems). Knowing that a problem is solvable for a certain weak language gives us a point of reference for future research, and makes it meaningful to further discuss questions of computational complexity. As pointed out in [MP18], when an algorithm to prove a property has a completeness proof, it usually means that it is possible to furnish a justification for a negative answer, which is of value to the user. In our case, if the algorithm returns a bound which is higher than what you wanted, you can obtain from it an *execution pattern* which shows how the result arose. Finally, the quest for complete solutions drives research forward by setting challenges which invite new insights and ideas.

Next, we explain the definition of the language in more detail. We will comment about the motivation for the definitions, in particular vis-à-vis the analysis of fuller programming

languages. The main argument is that choices in the definition of the language are driven by the idea of using it as a *conservative abstraction*.

1.1. **The core language.**

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

Figure 1: Syntax of the core language.

Data. The only type of data in the core language is non-negative integers.[1] In a practical setting, a program may include statements that manipulate non-integer data that can, however, be abstracted away without losing the information critical to loop control —hence to a complexity analysis—as loops are often controlled by integer variables. 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. These considerations motivate the study of weak languages that handle integers.

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. Note that what we ignore is branches within a loop body and not branches that implement the loop control, as loops are represented 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 [PR04, BHZ08, ADFG10, BAG14, BG17]. Note that the arithmetic in our language is too restricted to allow for the maintenance of counters and the management 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 [BAP16, Sect. 2]. The semantics of the loop is non-deterministic, so that the loop is allowed to actually perform fewer iterations than indicated by the bound expression; this is useful both for modeling loops that can "break," as well as for using the results of auxiliary analyses, as those usually provide just a bound, not a precise number of iterations.

An interesting observation has been made by Jones and Kristiansen [JK09]: typically, algorithms whose goal is to prove loop termination and establish loop bounds do so by focusing

[1] This could be modified to all integers, as explained later.

on values that *decrease* (examples are the Size-Change Termination principle [LJB01] and numerous methods that discover *ranking functions*). In contrast, by abstracting to our core language ([JK09] uses a very similar language), we focus on values that *grow*. Recent work in static analysis [BEF+16] describes an analysis system which combines a subsystem for loop-bound analysis (via ranking functions) with a subsystem for growth-rate analysis, which establishes symbolic bounds on data that grow along a loop. Our definition of the core language separates the concerns and concentrates on the problem of value growth, for a program (or program fragment) where loop bounds are already known. Note however that the loop bound is to be given as a function of the state in which the loop is started, and may well depend on values that are the result of previous computations, as our example program in Figure 2 illustrates.

```
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 } }
```

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

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 weak version of Meyer and Ritchie's LOOP language [MR67], which computes all the primitive recursive functions, and where behavioral questions like "is the result linearly bounded" are undecidable.

1.2. **The algorithm.** Consider the program in Figure 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 $\max(x_4(x_4 + x_1^2), x_4(x_2 + x_3 + x_1^2))$.

Actually, the algorithm produces information in a more precise form, as *a disjunction of simultaneous bounds*. This means that it generates a set of tuples of polynomials, called *multi-polynomials*. Each such tuple provides simultaneous bounds on all variables in a subset of possible executions; for example, with the program in Figure 2, one such multi-polynomial is $\langle x_1, x_2, x_3, x_2+x_3\rangle$, which means that, starting with the valuation $\mathtt{X}_1 \mapsto x_1$, $\mathtt{X}_2 \mapsto x_2$, $\mathtt{X}_3 \mapsto x_3, \mathtt{X}_4 \mapsto x_4$ the value of $\mathtt{X}_4$ at the end of some possible execution (specifically, one in which the second and the third loop commands both exit immediately) is tightly (in this case, exactly) described by $x_2 + x_3$; while the other three variable retain their initial values. Other multi-polynomials will represent other sets of possible executions, their union covering all executions. This *disjunctive* 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 are 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 abstract state-transitions. 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 transitions.

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 give the correctness statement for our algorithm, and in Section 7 we give the correctness proofs for this. In Section 8 we consider the computational complexity of our algorithm, and in Section 9 we consider extensions to our algorithm and open problems. Section 10 describes related work, and Section 11 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 is just a command.

Polynomials and multi-polynomials. We work throughout this article with multivariate polynomials in $x_1, \ldots, x_n$ that have non-negative integer coefficients and no variables other than $x_1, \ldots, x_n$; 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. Our algorithm sometimes deals with monomials, and the reader may assume that a polynomial is always represented as a non-redundant set of monomials (i.e., $p(x) = 2x$ is never represented as $x + x$ or $2x + 0x^2$).

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.

The parameter n always refers to the number of variables in the subject program. The set $[n]$ is $\{1,\ldots,n\}$. 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 are used, 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 2.1. A function of the form $\langle \mathbf{p}[1],\ldots,\mathbf{p}[n]\rangle$, i.e., an n-tuple of polynomials, is called *a multi-polynomial (MP)*. We denote by $\mathtt{MPol}$ the set of multi-polynomials, namely $(\mathbb{N}[\mathbf{x}])^n$,where the number of variables n is fixed by context.

Definition 2.2. A *polynomial transition (PT)* is a computation that transforms an initial state $\mathbf{x} = \langle x_1,\ldots,x_n\rangle$ to a new state $\mathbf{x}' = \langle x'_1,\ldots,x'_n\rangle = \mathbf{p}(\mathbf{x})$ where $\mathbf{p} \in \mathtt{MPol}$.

The distinction between a MP and a PT is perhaps a bit philosophical: An MP is a function: a mathematical object that exists independently of computation. A PT is a computation whose effect is described by such a function. We can say, for example, that the command $\mathtt{X}_2\mathtt{:=}\ \mathtt{X}_2+\mathtt{X}_1$ performs a PT. Its associated MP is $\langle x_1, x_2+x_1\rangle$.

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 $\mathbf{q}$, and we have: $(\mathbf{p}\circ\mathbf{q})[i] = \mathbf{p}[i]\circ\mathbf{q}$. 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 $\mathbf{x}[\![\mathtt{C}]\!]\mathbf{y}$, vector $\mathbf{x}$ (respectively $\mathbf{y}$) is the state 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 state $\mathbf{x}$ a new state $\mathbf{y}$ obtained by replacing the component x_i by the value of the expression $\mathtt{E}$ when evaluated over state $\mathbf{x}$. This is defined in the natural way (details omitted), and is denoted by $[\![\mathtt{E}]\!]\mathbf{x}$. Composite commands are described by the straightforward equations:

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

where $[\![\mathtt{C}]\!]^i$ represents $[\![\mathtt{C}]\!]\circ\cdots\circ[\![\mathtt{C}]\!]$ (with 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 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`." The reason that this change does not affect our results is that we work exclusively with monotone increasing functions. This assignment semantics is called *a lossy assignment* in [BAK12], due to an analogy with lossy counter machines [May03].

(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 $|x + y| \leq |x| + |y|$. 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 [BJK08]. We rely on that algorithm, which is polynomial-time, for doing this classification (as further explained in Section 4). Our main result is thus stated as follows.

Theorem 2.3. *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* $\mathtt{PB}$ *is the set of indices* i *of variables* $\mathtt{X}_i$ *whose final value under* $[\![\mathtt{C}]\!]$ *is polynomially bounded.*

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

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

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

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

A remark regarding the decision to provide bounds only up to constant factors: based on the proof, the algorithm could be extended to explicitly generate the coefficients $c_{\mathbf{p}}$ and $d_{\mathbf{p}}$ in the above statement; but ignoring constant factors simplifies the algorithm significantly. In essence, all the considerations regarding constants are removed from the algorithm to the proofs. Moreover, methodically, our goal was to show that in the sense that we consider (i.e., "big Theta" bounds), the problem could be solved *completely*. We do not attempt to completely solve the problem of computing bounds with the utmost precise constants. We assume that readers with a Computer Science background are aware of the ubiquity of "big Theta" bounds in the analysis of algorithms and dispense with a discussion of justifications for using them.

A comment on counters. As discussed in the introduction, various *countable resource problems*, such as bounding the number of program steps, may be reduced to the bound analysis problem. We'd like to point out that the reduction, by introducing a variable to count the events of interest, may be carried out in our core language, despite the lack of explicit constants. It suffices to reserve a dedicated variable, say `U`, to represent a unit of the resource. Then, we advance the counter by adding `U`, and its worst-case bound will equal the worst-case bound of the resource, times `U`. As an example, Figure 3 shows how the program from Figure 2 could be instrumented for computing its time complexity. We use a fresh variable $\mathtt{X}_5$ as a "unit" and another one $\mathtt{X}_6$ to count the steps. The algorithm will deduce

that the final value of $\mathtt{X}_6$, x'_6, is tightly bounded by $(x_2 + x_3 + x_1^3 + x_1x_4)x_5$. Erasing the x_5 factor we obtain a tight bound on the unit-cost time.

```
loop X1
   X6 := X6 + X5;
   loop X2 + X3 { X6 := X6 + X5;
                  choose { X3:= X1; X2:= X4 } or { X3:= X4; X2:= X1 }
                };
   X4:= X2 + X3
};
loop X4 { X6 := X6 + X5;
          choose  { X3:= X1 + X2 + X3 } or { X3:= X2;  X2:= X1 }
        }
```

Figure 3: An instrumented core-language program.

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 3.1. A *simple disjunctive loop (SDL)* is a finite set $\mathcal{S}$ of polynomial transitions.

The loop is "disjunctive" because its meaning is that in every iteration, any of the given transitions may be chosen. A SDL does not specify the number of iterations; our analysis generates polynomials that 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 3.2. τ-polynomials are polynomials in $x_1, \ldots, x_n$ and τ.

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 $\mathtt{X}_2\mathtt{:=}\ \mathtt{X}_2 + \mathtt{X}_1\mathtt{;}\ \mathtt{X}_3\mathtt{:=}\ \mathtt{X}_3 + \mathtt{X}_2$ is 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 $\tau\mathtt{MPol}$. Note that τ has a special status: as it does not represent a program variable, there is no component in the multi-polynomial for giving its value. We should note that composition $\mathbf{q} \circ \mathbf{p}$ over $\tau\mathtt{MPol}$ 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).

4. Reduction to Simple Disjunctive Loops

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

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

Example 4.1. Consider the following command:

$$\mathtt{X}_2\mathtt{:=}\ \mathtt{X}_1\mathtt{;}\ \mathtt{X}_4\mathtt{:=}\ \mathtt{X}_2\ \mathtt{+}\ \mathtt{X}_3\mathtt{;}\ \mathtt{X}_1\mathtt{:=}\ \mathtt{X}_2\ \mathtt{*}\ \mathtt{X}_3$$

This is precisely represented by the transition $\langle x_1, x_2, x_3, x_4\rangle' = \langle x_1x_3,\ x_1,\ x_3,\ x_1 + x_3\rangle$.

4.2. **Evaluation of non-deterministic choice.** Evaluation of the command

$$\mathtt{choose}\ \mathtt{C}_1\ \mathtt{or}\ \mathtt{C}_2$$

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 $\wp(\mathtt{MPol})$. Composition is naturally extended to sets, and the semantics of a choice command is now simply set union, so we have:

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

Example 4.2. Consider the following command:

$$\mathtt{X}_2\mathtt{:=}\ \mathtt{X}_1\mathtt{;\ choose\ \{\ X}_4\mathtt{:=}\ \mathtt{X}_2\ \mathtt{+}\ \mathtt{X}_3\ \mathtt{\}\ or\ \{\ X}_1\mathtt{:=}\ \mathtt{X}_2\mathtt{*}\ \mathtt{X}_3\ \mathtt{\}}$$

This 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 MPs, that is, its semantics $[\![\mathtt{C}]\!]$ equals the union of a finite number of PTs, which we can compute by symbolic evaluation. Consequently, the problem of analyzing any loop command is reduced to the analysis of simple disjunctive loops, by first analyzing the loop body. The goal of the loop analysis is to produce a set of multi-polynomials that provide tight worst-case bounds on the results of any number of iterations of the loop body—"tight" in the sense of Theorem 2.3 (see also Example 4.3 below). We will refer to this, concisely, as *solving* the loop.

Suppose that we have an algorithm Solve that takes a simple disjunctive loop and computes tight bounds for its results, with explicit dependence on the number of iterations using the parameter τ. We use it to complete the analysis of any program by the following definition:

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

Thus, the whole solution is constructed as an ordinary abstract interpretation, following the semantics of the language, except for what happens inside Solve, which is the subtle

part of the algorithm; this procedure accepts a set of multi-polynomials, describing the body of the loop, and generates a set of τ-MPs that finitely describes the result of all execution sequences of the loop, thanks to a *generalization* procedure that captures the dependence of results on the iteration count by means of the parameter τ. After completing this procedure, we replace τ with the expression $[\![\mathtt{E}]\!]$, which gives the actual loop bound as a polynomial in $x_1, \ldots, x_n$.

Example 4.3. Consider the following command:

$$\mathtt{X_4:= X_1; \ loop \ X_4 \ \{ \ X_2:= X_1 \ + \ X_2; \ X_3:= X_2\}}$$

Solving just the loop yields the 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:

$$\begin{aligned}\mathcal{L}[x_4/\tau] \circ [\![\mathtt{X_4 \ := \ X_1}]\!]^S &= \mathcal{L}[x_4/\tau] \circ \{\langle x_1, x_2, x_3, x_1\rangle\} \\ &= \{\langle x_1, x_2, x_3, x_1\rangle, \langle x_1, x_2 + x_1^2, x_2 + x_1^2, x_1\rangle\}.\end{aligned}$$

We can now demonstrate that the analysis of a loop may involve approximation. Suppose that we slightly modify the code as follows:

$$\mathtt{X_4:= X_1; \ loop \ X_4 \ \{ \ X_3:= X_2; \ X_2:= X_1 \ + \ X_2\}}$$

Our algorithm will produce the same MPs, although a careful examination shows that the precise result of the loop (when it executes $x_1 > 0$ iterations) is now given by $\langle x_1, x_2 + x_1^2, x_2 + x_1(x_1 - 1), x_1\rangle$. This expression is not a MP (since we only consider polynomials with non-negative coefficients), but is tightly approximated by the MP we produced, as $x_1(x_1 - 1) = \Theta(x_1^2)$. The precise statement of what it means to tightly approximate all possible results of a program by a finite set of MPs is the essence of Theorem 2.3.

The next section describes the 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 [BJK08] that 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 (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-polynomial quantity (given the restricted arithmetics in our language). In reporting the results of the algorithm, we should display "`super-polynomial`" instead of any expression that includes 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

$$Q_i = \{Id\} \cup P \cup (P \circ P) \cup \cdots \cup P^{(i)} \tag{5.1}$$

where $P = [\![\mathtt{C}]\!]^S \in \wp(\mathtt{MPol})$. The question now is whether the sequence (5.1) 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 (5.1) 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 5.1. For a SDL P that does not use addition, the sequence Q_i as in (5.1) reaches a fixed point, and the fixed point provides tight bounds for all the polynomially-bounded variables.

Example 5.2. The following loop is in the multiplicative fragment, and has no exponential behaviour:

$$\texttt{loop X}_3\ \texttt{\{ X}_1\texttt{:= X}_2\ \texttt{* X}_2\texttt{;}\ \ \texttt{X}_2\texttt{:= X}_3\ \texttt{\}}$$

The effect of an iteration is given by the multi-polynomial $\mathbf{p} = \langle x_2^2,\ x_3,\ x_3\rangle$. Let $P = \{\mathbf{p}\}$. The accumulating effect of the loop is given by the union of:

$$\begin{aligned} \{Id\} \\ P &= \{\langle x_2^2,\ x_3,\ x_3\rangle\} \\ P \circ P &= \{\langle x_3^2,\ x_3,\ x_3\rangle\} \\ P^{(3)} &= P^{(2)} \\ &\dots \end{aligned}$$

That is, the results are completely characterized by the three MPs above.

When we have addition, we find that knowing that all variables are polynomially bounded does not imply convergence of the sequence (5.1). An example is $\texttt{loop X}_3\ \texttt{\{ X}_1\texttt{:= X}_1\ \texttt{+ X}_2\ \texttt{\}}$ yielding the infinite sequence of MPs $\langle x_1, x_2, x_3\rangle$, $\langle x_1 + x_2, x_2, x_3\rangle$, $\langle x_1 + 2x_2, x_2, x_3\rangle$, ... 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 5.3. $\texttt{APol}$, the set of abstract polynomials, consists of formal sums of distinct monomials over $x_1, \dots, x_n$, where the coefficient of every monomial included is 1. We extend the definition to an abstraction of τ-polynomials, denoted $\tau\texttt{APol}$.

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 $\tau\texttt{APol}$ are 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 coefficient 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 to tuples of (abstract) polynomials in the natural way.
(5) The set of abstract multi-polynomials $\texttt{AMPol}$ and their extension with τ ($\tau\texttt{AMPol}$) are defined as n-tuples over $\texttt{APol}$ (respectively, $\tau\texttt{APol}$). 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.

An abstract polynomial can be reduced by deleting dominated monomials (e.g., $x^2 + x \to x^2$); which clearly preserves its meaning when viewed in "big Theta" terms. We expect that the combinatorial explosion in the algorithm can be attenuated by reducing every abstract polynomial, but in this paper, we ignore this optimization to alleviate the description and proof of the algorithm. Another useful optimization (which we also ignore in our analysis) is to delete dominated *multi-polynomials* in a set of MPs.

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 $\tau\texttt{AMPol}$. 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 5.4 (iterated composition)**.** Let $\mathbf{t}$ be any abstract τ-MP. We define $\mathbf{t}^{\bullet(k)}$, for $k \geq 0$, by:

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

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

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

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

Example 5.5. Let $\mathbf{p} = \langle x_1, x_1 + x_2\rangle$; clearly its kth iterate (for $k \geq 1$) is $\mathbf{p}^{(k)} = \langle x_1, x_1 + kx_2\rangle$. However, with abstract polynomials, we get $\alpha(\mathbf{p})^{\bullet(k)} = \langle x_1, x_1 + x_2\rangle$; the growth in the coefficient is abstracted away. Similarly, we have $\{\alpha(\mathbf{p})\}^{\bullet(k)} = \{Id, \alpha(\mathbf{p})\}$ for all $k \geq 1$.

Definition 5.6 (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 for finiteness 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.

Example 5.7. Consider the following loop:

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

We have omitted the loop bound in order to emphasize that we are now analyzing the *growth of values as the loop progresses* rather than its final result. The generalization construction below introduces τ's into the AMPs, obtained from the loop body, in order to express this growth in terms of the number of iterations. Informally, consider the assignment to $\mathtt{X}_1$: it adds x_3 (the initial value of variable $\mathtt{X}_3$), and does so on every iteration: so we have an arithmetic series with an increment of x_3. After τ iterations, $\mathtt{X}_1$ will have grown by τx_3. The assignment to $\mathtt{X}_2$ similarly adds the values of two other variables, $\mathtt{X}_3$ and $\mathtt{X}_4$, but as $\mathtt{X}_4$ is rewritten with x_3 already at the end of the first iteration, subsequent iterations add $2x_3$ (not $x_3 + x_4$). This will be expressed by introducing the term τx_3 (the coefficient 2 is abstracted away). We return to this example, and the issue it illustrates, further below.

The definition of $\mathbf{p}^\tau$ will is greatly simplified by concentrating on idempotent AMPs, defined next. We will later give an example that shows that our definition does not work for AMPs which are not idempotent; while the fact that a complete solution can be obtained by generalizing only idempotent elements comes out of a Ramsey-like property of finite monoids [Sim90], as will be seen in Section 7.2.

Definition 5.8. $\mathbf{p} \in \tau\mathtt{AMPol}$ 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 $\langle x_1 x_2, x_2\rangle$ and $\langle x_1 + x_2, x_1\rangle$ are not.

Definition 5.9. 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 also say that the entry $\mathbf{p}[i]$ is self-dependent; the choice of term depends on context and the meaning should be clear either way. We call a monomial self-dependent if all the variables appearing in it are. We denote by $\mathrm{SD}(\mathbf{p})$ the set $\{i : x_i \text{ is self-dependent in } \mathbf{p}\}$.

We later show that in polynomially-bounded loops, if x_i is self-dependent then $\mathbf{p}[i]$ must include the monomial x_i. To illustrate the significance of self-dependent monomials, let us consider an example where $\mathbf{p}[1] = x_1 + \ldots$ and $\mathbf{p}[2] = x_2 + \ldots$, and further $\mathbf{p}[3] = x_1 x_2 + \ldots$. Then the monomial $x_1 x_2$ reappears in every iterate $\mathbf{p}^{\bullet(k)}$. This is because all the variables in it are self-dependent.

Definition 5.10. We define a notational convention for τ-MPs, specifically for self-dependent entries of the MP. Assuming that x_i appears in $\mathbf{p}[i]$, 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 5.11. Let $\mathbf{p} = \langle x_1 + \tau x_2 + \tau x_3 + x_3 x_4,\ x_3,\ x_3,\ x_4\rangle$. Then $\mathrm{SD}(\mathbf{p}) = \{1, 3, 4\}$. Since x_1 is self-dependent, we will apply the above definition to $\mathbf{p}[1]$, so that $\mathbf{p}[1]' = x_3$, $\mathbf{p}[1]'' = x_3 x_4$ 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 5.12 (generalization)**.** Let $\mathbf{p}$ be idempotent in $\tau\mathtt{AMPol}$; define $\mathbf{p}^\tau$ by:

$$\mathbf{p}^\tau[i] = \begin{cases} x_i + \tau\mathbf{p}[i]' + \tau\mathbf{p}[i]'' + \mathbf{p}[i]''' & \text{if } i \in \mathrm{SD}(\mathbf{p}) \\ \mathbf{p}[i] & \text{otherwise.} \end{cases}$$

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 will treat them differently, which is why the notation separates them. We next give a couple of examples, just to illustrate the definition; we later (Page 15) motivate the definition in the context of the loop analysis.

Example 5.13. Let $\mathbf{p} = \langle x_1 + x_3,\ x_2 + x_3 + x_4,\ x_3,\ x_3\rangle$. Note that it corresponds to the loop body from Example 5.7. Further note that $\mathbf{p} \bullet \mathbf{p} = \mathbf{p}$, i.e., $\mathbf{p}$ is idempotent. We have $\mathbf{p}^\tau = \langle x_1 + \tau x_3,\ x_2 + \tau x_3 + x_4,\ x_3,\ x_3\rangle$.

Example 5.14. Let $\mathbf{p} = \langle x_1 + \tau x_2 + \tau x_3 + \tau x_3 x_4,\ x_3,\ x_3,\ x_4\rangle$. Note that $\mathbf{p} \bullet \mathbf{p} = \mathbf{p}$. The self-dependent variables are all but x_2. We have $\mathbf{p}^\tau = \langle x_1 + \tau x_2 + \tau x_3 + \tau x_3 x_4,\ x_3,\ x_3,\ x_4\rangle = \mathbf{p}$.

Finally we can present the analysis of the loop command.

Algorithm 5.15. Solve($\mathcal{S}$)
Input: $\mathcal{S}$, a polynomially-bounded disjunctive simple loop.
Output: a set of τ-MPs that tightly approximates the effect of τ iterations of loop $\mathcal{S}$.

(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 5.16. Consider the following loop:

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

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`,

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

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

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

where the first one simplifies to $\mathbf{q}$ by deleting dominated terms, and the second to $\langle x_1 + \tau x_2 + \tau^2 x_3,\ x_2 + \tau x_3,\ x_3,\ x_3\rangle$. 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.

Note that when a program contains nested loops, innermost loops will be processed first and result in a set of abstract polynomials, so we might actually analyze any enclosing commands entirely in the abstract domain. This means rephrasing our definition of SDL by defining the input set as a set of AMPs rather than concrete polynomials; then the initial

step above, where $\mathcal{S}$ is abstracted, is skipped. In the forthcoming sections we ignore this issue and continue to treat the input as being a set of concrete polynomials; this is for convenience only. It is easy enough to restate the results such that the input is understood as abstract, but known to tightly describe the effect of some concrete piece of code.

Comments on generalization. The precise definition of the generalization operator has been one of the key steps in the development of this algorithm (operators which are similar—but insufficient for our purpose—appear in related work [KA80, KN04, NW06, JK09, BJK08, BEF$^+$16]). Let us attempt to give some intuition for its definition (not a correctness proof, yet). Say x_i is self-dependent. An important point is the partition of the terms added to x_i into $\tau\mathbf{p}[i]'$, $\mathbf{p}[i]''$ and $\mathbf{p}[i]'''$. To see why the non-self-dependent monomials $\mathbf{p}[i]'''$ are not multiplied by τ, consider Example 5.13. Had we been too eager to insert τ's and do this to non-self-dependent monomials as well, we would have got:

$$\langle x_1 + \tau x_3,\ x_2 + \tau x_3 + \tau x_4,\ x_3,\ x_3\rangle$$

(differing from the correct result in the second component). This would give a sound, but loose (overestimated) upper bound: as already discussed, when the transition represented by $\mathbf{p}$ is iterated, copies of the initial value x_4 do *not* accumulate. Next, the distinction of $\mathbf{p}[i]''$ from $\tau\mathbf{p}[i]'$ prevents τ's from being added where they already appear; this is important because we reapply generalization to results of previous steps while analyzing a given loop. Finally, an important aspect particular to our algorithm (compared to the above-cited) is the application of the generalization operator only to idempotent elements. Again, we can illustrate with an example that applying it too eagerly (to all AMPs in the closure) would be incorrect. Consider the following AMP:

$$\mathbf{p} = \langle x_1 + x_2 + x_3 + x_4,\ x_3,\ x_4,\ x_4\rangle$$

Generalization (incorrectly applied to $\mathbf{p}$, since it is not idempotent) would yield:

$$\langle x_1 + x_2 + x_3 + \tau x_4,\ x_3,\ x_4,\ x_4\rangle$$

This is bad because if $\mathbf{p}$ is iterated to accumulate copies of x_4 in $\mathtt{X}_1$ (a behaviour represented by the term τx_4), the value of $\mathtt{X}_2$ will no longer be x_3 (but x_4). Hence, this AMP describes a result that is not realizable (and when we substitute the loop bound for τ, we will get a loose upper bound).

A comment on linear bounds. When values accumulate in a loop, a non-linear result is obtained. Stated contrapositively: linear results do not involve accumulation. Indeed, it is not hard to verify that if for linear polynomials (only) we maintain precise numeric coefficients, our algorithm would still converge. Therefore, if we wish, we can modify our algorithm so that *for linear results we have bounds in which the coefficients are explicit and precise*. This is not a strong result as might seem at first, since it is due to the weakness of our language: linear results can only be built up by a finite number of additions. Still, if the algorithm is employed as a brick in a larger system, this property might possibly be useful.

6. Correctness Statement for Simple Disjunctive Loops

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 a "big O" 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. The main, non-trivial part of the algorithm is of course the solution of a simple disjunctive loop, procedure Solve. Completing this to show correctness for an arbitrary program is not difficult.

In this section we formulate the *correctness statement for Simple Disjunctive Loops*, in other words, we state the contract for procedure Solve, which, when satisfied, justifies its proper functioning within the whole algorithm. This formulation forces us to step up the level of detail; specifically, we introduce *traces*, to give a more concrete semantics to loops (compared to Section 2.2).

Definition 6.1. Let $\mathcal{S}$ be a set of polynomial transitions. An *(abstract) trace* over $\mathcal{S}$ is a finite sequence $\sigma = \mathbf{p}_1; \ldots; \mathbf{p}_{|\sigma|}$ 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 $[\![\varepsilon]\!] = Id$).

Using the following definition we will be able to give the desired correctness statement for Solve.

Definition 6.2. Let $p(\mathbf{x})$ be a (concrete or abstract) τ-polynomial. We write $\dot{p}$ for the *linear monomials* of p, namely any one of the form ax_i for a constant coefficient a. We write $\ddot{p}$ for the rest. Thus $p = \dot{p} + \ddot{p}$.

Theorem 6.3 (Solution of disjunctive loop problem)**.** *Given a polynomially-bounded SDL represented as a set $\mathcal{S}$ of MPs, procedure* Solve *finds a finite set $\mathcal{B}$ of τ-MPs which* tightly bound *all traces over $\mathcal{S}$. More precisely, it guarantees:*

(1) *(Bounding) There is a constant $c_{\mathbf{p}} > 0$ associated with each $\mathbf{p} \in \mathcal{B}$, such that*

$$\forall \mathbf{x}, \mathbf{y}, \sigma \ . \ \mathbf{x}[\![\sigma]\!]\mathbf{y} \Longrightarrow \exists \mathbf{p} \in \mathcal{B} \,.\, \mathbf{y} \leq c_{\mathbf{p}}\mathbf{p}(\mathbf{x}, |\sigma|)$$

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

$$\mathbf{x}[\![\sigma]\!]\mathbf{y} \ \wedge \ \mathbf{y} \geq \dot{\mathbf{p}}(\mathbf{x}, |\sigma|) + d_{\mathbf{p}}\ddot{\mathbf{p}}(\mathbf{x}, |\sigma|) \,.$$

Note that in the lower-bound clause (2), the linear monomials of p are not multiplied by the constant $d_{\mathbf{p}}$; 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. We remark 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 $d_{\mathbf{p}}$ to be smaller than one; in the upper bounds, it is possible to stick to integers). Note also that polynomial boundedness is ensured by our algorithm before applying the procedure (Section 4.3), so the precondition of the correctness theorem is satisfied.

7. Correctness Proofs

In this section we prove the correctness of the main, non-trivial part of our algorithm, namely the solution of a simple disjunctive loop, showing that it satisfies the requirements set forth in the last section.

Overview of the proof. 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 set of traces by a *pattern*. A pattern is constructed like a regular expression with concatenation and Kleene-star. However, they allow no nested iteration constructs, and when expanding a pattern into a concrete trace, 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 that it is realizable. It is interesting that such simple patterns suffice to establish tight lower bounds for all loops in our class.
- In the upper-bound proof, we describe all traces by a finite set of *well-typed regular expressions* [Boj09]. This elegant tool channels the power of the Factorization Forest Theorem [Sim90]; this theorem exposes 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.

Recall that traces are just sequences of polynomial transitions (Definition 6.1). Next, we define *concrete traces*, which represent executions on concrete data. The following definition assumes that the steps of the trace are specified by τ-free multi-polynomials; it suffices for the first part of the proof (Section 7.1).

Definition 7.1 (Concrete traces (unweighted))**.** A *concrete trace* corresponding to $\sigma \in \mathcal{S}^*$ is a path of labeled arcs in the state space $\mathbb{N}^n$:

$$\widetilde{\sigma} = s_0 \xrightarrow{\mathbf{p}_1} s_1 \dots s_{t-1} \xrightarrow{\mathbf{p}_t} s_t,$$

where $s_{i+1} = \mathbf{p}_{i+1}(s_i)$. We write $s_0 \overset{\sigma}{\rightsquigarrow} s_t$ to indicate just the initial and final states of a trace (as a special case, the empty trace ε corresponds to a path with no arcs: $s_0 \overset{\varepsilon}{\rightsquigarrow} s_0$). The set of all concrete traces is denoted $\widetilde{\mathcal{S}^*}$.

Note that the semantics of polynomial transitions never block a state. That is, given σ and s_0 there always is a $\widetilde{\sigma}$ starting with s_0.

Notations.

- Concatenation of traces σ, ρ is written as $\sigma\rho$. For concrete traces, concatenation requires the final state of σ to be the initial state of ρ, assuming both are non-empty.
- In the proofs, we handle the abstract polynomials computed by the algorithm as if they were concrete polynomials. This should be understood as an "implicit cast," applying the γ conversion function (see Page 11). In fact, it is useful to bear in mind that the correctness claim—namely, Theorem 6.3—is a claim about concrete MPs, relating their numeric values to values generated by program executions. We mostly omit α's and γ's and the reader should interpret a MP as concrete or abstract according to context. For example, comparison of value $p \geq q$ applies to concrete polynomials, while the statement that $\mathbf{p}$ is idempotent indicates that $\mathbf{p}$ is (or should be "cast" into) an abstract MP.
- We call a τ-MP $\mathbf{p}$ τ-closed if $\mathbf{p}^\tau = \mathbf{p}$.

7.1. **Lower-Bound Correctness for Simple Loops.** The key notion in proving that the upper bounds that we compute are tight—equivalently, that they provide lower bounds (up to constant factors) on the worst-case results—is that of *realizability*. Intuitively, when we come up with a bound, we want to show that there are traces which achieve (realize) this bound. Importantly, with asymptotic analysis, a bound is not justified, in general, by showing a *single* trace; what we need is a *pattern* that generates arbitrarily long traces.

Formally, we define the class of *patterns* (over a given set of polynomial transitions, $\mathcal{S}$) and their associated languages (sets of traces). The following statements define patterns π along with corresponding sets of languages, $L(\pi, t)$. The role of t is related to loop counts; it tells how many times to repeat $\mathbf{a}$ in a pattern $\mathbf{a}^*$.

- The empty string ε is also a pattern. It generates the language $L(\varepsilon, t) = \{\varepsilon\}$, consisting of the empty trace.
- A single MP, $\mathbf{p} \in \mathcal{S}$, is a pattern. It generates the language $L(\mathbf{p}, t) = \{\mathbf{p}\}$, consisting of a single trace.
- A concatenation of patterns is a pattern, and $L(\pi_1\pi_2, t) \stackrel{def}{=} L(\pi_1, t)L(\pi_2, t)$, where the right-hand side applies concatenation to traces.
  We define concatenation of patterns to be associative, so $\mathbf{p}(\mathbf{qr})$ and $(\mathbf{pq})\mathbf{r}$ are the same pattern, which may be written $\mathbf{pqr}$ (this works because concatenation of traces is also associative).
- If π is a pattern, π^* is a pattern. However, nested application of the star is not allowed; in other words, π is required to be star-free. We define $L(\pi^*, t) = L(\pi)^t$, where $L^0 = \{\varepsilon\}$ and $L^{t+1} = L^t \cdot L$.
  Parentheses are used for syntactic disambiguation of the operand of the star, e.g., $\mathbf{p}(\mathbf{qr})^*$.

We use the notation π^n as a shorthand for a concatenation of n copies of π. The set of traces corresponding to pattern π, denoted $L(\pi)$, is defined by $L(\pi) = \bigcup_{t\geq 0} L(\pi, t)$. E.g., $L(a^*b^*) = \{a^n b^n \mid n \geq 1\}$. We denote by $\pi(n)$ the result of substituting n for all the stars in π (obtaining a star-free pattern). In particular, $\pi(1)$ is the result of substituting 1 for all the stars in π. Since nesting of iterative expressions is not allowed, $|\pi(t)| = \Theta(t)$.

Definition 7.2 (Realizability)**.** A polynomial transition (PT) $\mathbf{p} \in \tau\texttt{MPol}$ is said to be *realizable* over the given set $\mathcal{S} \subset \texttt{MPol}$ if there is a pattern π and a constant $0 < c \leq 1$, such that for all $t \geq 1$, for all $\sigma \in L(\pi(t))$, if $\mathbf{x} \stackrel{\sigma}{\rightsquigarrow} \mathbf{y}$, then

$$\mathbf{y} \geq \dot{\mathbf{p}}(\mathbf{x}, t) + c\ddot{\mathbf{p}}(\mathbf{x}, t)\,. \tag{7.1}$$

We say that $\mathbf{p}$ is realized by $\pi; c$, or that it is c-realizable. For τ-free MPs, we use the same definition but π should not include a star, so t may be omitted. Thus a τ-free MP is to be realized by a single abstract trace. A set of MPs is called realizable if all its members are.

Example 7.3. We reconsider the loop in Example 5.7. In Example 5.13 we have computed from its body the τ-MP $\mathbf{q} = \mathbf{p}^\tau = \langle x_1 + \tau x_3,\ x_2 + \tau x_3 + x_4,\ x_3,\ x_3\rangle$. We claim that it is realizable. We use the simple pattern $\pi = \mathbf{p}^*$. The final state after $t \geq 1$ iterations (a trace in $L(\pi(t))$) is

$$\mathbf{y} = \langle x_1 + tx_3,\ x_2 + (2t-1)x_3 + x_4,\ x_3,\ x_3\rangle,$$

which is easy to verify by inspection of the program. We now verify that

$$\mathbf{y} \geq \dot{\mathbf{q}}(\mathbf{x}, t) + c\ddot{\mathbf{q}}(\mathbf{x}, t),$$

where $c = 1$.

Note that realizability is a monotone property in the sense that if $\mathbf{p} \geq \mathbf{q}$ and $\mathbf{p}$ is realizable, then $\mathbf{q}$ is. This is natural, since we are arguing that $\mathbf{p}$ is a *lower bound*. Note also that in contrast to Clause 2 of Theorem 6.3, we pass t rather than $|\sigma|$ as the τ parameter to the polynomial bound. However, since $|\pi(t)| = \Theta(t)$, the inequality in the theorem will be satisfied, just with a different constant factor.[2]
We introduce a short form that makes inequalities such as (7.1) easier to manipulate:

Definition 7.4. For any τ-polynomial p, and real number $c \leq 1$,

$$c{:}p(\mathbf{x}, t) \stackrel{def}{=} \dot{p}(\mathbf{x}, t) + c\ddot{p}(\mathbf{x}, t)\,.$$

We also write $c{:}\mathbf{p}$ for component-wise application of "c:" to a multi-polynomial $\mathbf{p}$.

It is useful to be aware of properties of this operation. We leave the verification of the next lemma to the reader.

Lemma 7.5. *Let $r \in \tau\texttt{Pol}$, $\mathbf{p} \in \tau\texttt{MPol}$, and $c \leq 1$. Then we have*

$$(c{:}r) \circ \mathbf{p} \geq c{:}(r \circ \mathbf{p})$$
$$r \circ (c{:}\mathbf{p}) \geq c^{\deg r}{:}(r \circ \mathbf{p})\,.$$

Lemma 7.6. *Let $\mathbf{p}, \mathbf{q} \in \tau\texttt{AMPol}$. Then $\mathbf{p} \circ \mathbf{q} \geq \mathbf{p} \bullet \mathbf{q}$.*

Note that on the left-hand side of the inequality, we compose in $\tau\texttt{MPol}$, while on the right-hand side, we compose in $\tau\texttt{AMPol}$. A more explicit expression of this claim would be:

$$\gamma(\mathbf{p}) \circ \gamma(\mathbf{q}) \geq \gamma(\mathbf{p} \bullet \mathbf{q}).$$

Now this becomes obvious, since both sides of the equality have the same monomials (they are abstractly the same), and the coefficients on the right-hand side are all 1, while on the left-hand side they are integers. The fact that both sides have corresponding monomials allows us to strengthen the statement to the following:

Lemma 7.7. *Let $\mathbf{p}, \mathbf{q} \in \tau\texttt{AMPol}$ and $c \leq 1$. Then $c{:}(\mathbf{p} \circ \mathbf{q}) \geq c{:}(\mathbf{p} \bullet \mathbf{q})$.*

Recall that we aim to prove that all PTs computed by our algorithm are realizable. We do this by a series of lemmas which shows that realizability is preserved by the various constructions in our algorithm. Note that our algorithm produces abstract MPs; so, let us

[2]This consideration only applies to the non-linear part $\ddot{\mathbf{p}}$, since $\dot{\mathbf{p}}$ is τ-free anyway.

clarify that when we say that $\mathbf{p} \in \texttt{AMPol}$ is realizable, Definition 7.2 is actually applied to $\gamma(\mathbf{p})$.

7.1.1. *Realizability without generalization.* The realizability of PTs computed in the closure step will follow quite easily using a few simple lemmas.

Lemma 7.8. *Id is realizable.*

Proof. *Id* is realized by the empty pattern. □

Lemma 7.9. *Every member of $\mathcal{S}$ is realizable.*

Proof. $\mathbf{p} \in \mathcal{S}$ is realized by the pattern $\mathbf{p}$. □

Lemma 7.10. *Let $\mathbf{p}$, $\mathbf{q}$ be realizable τ-MPs. Then $\mathbf{q} \circ \mathbf{p}$ is realizable.*

Proof. Suppose that $\mathbf{p}$ is realized by $\pi_1; c_1$ and $\mathbf{q}$ by $\pi_2; c_2$. We claim that $\mathbf{q} \circ \mathbf{p}$ is realized by $\pi_1\pi_2; c'$ for some c'. Consider a concrete transition sequence $\widetilde{\sigma}$ with $\sigma \in L((\pi_1\pi_2)(t))$, it has the form $\mathbf{x} \overset{\sigma_1}{\rightsquigarrow} \mathbf{y} \overset{\sigma_2}{\rightsquigarrow} \mathbf{z}$ with $\sigma_i \in L(\pi_i(t))$. We have:

$$\mathbf{y} \geq c_1{:}\mathbf{p}(\mathbf{x}, t)$$
$$\mathbf{z} \geq c_2{:}\mathbf{q}(\mathbf{y}, t),$$

consequently, using Lemma 7.5:

$$\mathbf{z} \geq c_1^d c_2{:}(\mathbf{q} \circ \mathbf{p})(\mathbf{x}, t)$$

where d is the highest degree of any component of $\mathbf{q}$. □

Lemma 7.11. *Let $\mathbf{p}, \mathbf{q} \in \tau\texttt{AMPol}$ be realizable. Then $\mathbf{q} \bullet \mathbf{p}$ is realizable.*

Proof. This follows from the previous lemma and Lemma 7.7. □

Corollary 7.12. *Let T be a set of realizable abstract τ-MPs. Then every member of $Cl(T)$ is realizable.*

7.1.2. *Dependence graphs and neat MPs.* We are still missing a realizability lemma for the generalization operation. As a preparation for this proof, we introduce the dependence graph of a PT and state an important structural property of dependency graphs associated with idempotent transitions. This leads to the definition of the class we call *neat MPs*.

Definition 7.13. Let $\mathbf{p} \in \tau\texttt{MPol}$. Its *dependency graph* $G(\mathbf{p})$ is a directed graph with node set $[n]$. The graph includes an arc $i \to j$ if and only if $\mathbf{p}[j]$ depends on x_i.

Intuitively, $G(\mathbf{p})$ shows the data flow in the transition $\mathbf{p}$. It is easy to see that paths in the graph correspond to data-flow effected by a sequence of transitions. For instance, if we have a path $i \to j \to k$ in $G(\mathbf{p})$, then x_i will appear in the expression $(\mathbf{p} \circ \mathbf{p})[k]$.

Lemma 7.14. *Suppose that the SDL $\mathcal{S}$ is polynomially bounded. Then for all $\mathbf{p} \in \tau\texttt{MPol}$ which is realizable over $\mathcal{S}$, $\mathbf{p}[i]$ has no monomial divisible by x_i other than x_i.*

Proof. Suppose that $\mathbf{p}$ has such a monomial, then it has the form $\mathfrak{m}x_i$. If $\mathfrak{m}$ is τ-free, then starting with a state in which all variables have values greater than 1, and repeatedly executing a trace that realizes $\mathbf{p}$, we clearly obtain an exponential growth of the value in $\mathtt{X}_i$, contradicting the assumption. In the case that τ occurs in $\mathfrak{m}$, suppose that $\mathbf{p}$ is realized by $\pi; c$. Choose $t \geq \max(c^{-1}, 2)$. Then repeating the trace $\pi(t)$ creates an exponential growth in $\mathtt{X}_i$. □

Lemma 7.15. *Suppose that $\mathbf{p} \in \tau\mathtt{MPol}$ is realizable, and $\alpha(\mathbf{p})$ is idempotent. Assuming that the loop under analysis is polynomially bounded, $G(\mathbf{p})$ does not have any simple cycle longer than one arc. In other words, it consists of a directed acyclic graph (DAG) plus some self-loops.*

Proof. Assume, to the contrary, that $G(\mathbf{p})$ does have a cycle $i \to \cdots \to k \to i$, where $k \neq i$. Let r be the length of the cycle. Then

(1) $\mathbf{p}^{(r)}[i]$ depends on x_i; we can write $\mathbf{p}^{(r)}[i] = x_i + p(\mathbf{x}, \tau)$ (the occurrence of x_i must be in a linear monomial with a concrete coefficient of 1, otherwise iteration would cause exponential growth). By the remark after Definition 5.4, $\alpha(\mathbf{p}^{(r)})[i] = ((\alpha(\mathbf{p}))^{\bullet(r)})[i]$, which, by idempotence, equals $\alpha(\mathbf{p})[i]$. Reading backwards, $\alpha(\mathbf{p}[i]) = \alpha(\mathbf{p}^{(r)})[i] = \alpha(x_i) + p(\mathbf{x}, \tau)$, so $\mathbf{p}[i]$ also has the form $x_i + q(x, \tau)$ (where q may differ from p).
(2) By assumption, $\mathbf{p}[i]$ depends on x_k. Since $k \neq i$, we conclude that q depends on x_k. Also by the assumption of the cycle in $G(\mathbf{p})$, $\mathbf{p}^{(r-1)}[k]$ depends on x_i. This shows that $q \circ \mathbf{p}^{(r-1)}$ depends on x_i.
(3) Now,

$$\mathbf{p}^{(r)}[i] = (\mathbf{p} \circ \mathbf{p}^{(r-1)})[i] = \mathbf{p}[i] \circ \mathbf{p}^{(r-1)} = (x_i + q) \circ \mathbf{p}^{(r-1)} = \mathbf{p}^{(r-1)}[i] + (q \circ \mathbf{p}^{(r-1)}) .$$

We argue that the last expression has at least *two* occurrences of x_i. First, by the same token as (1), $\mathbf{p}^{(r-1)}[i]$ has an occurrence of x_i. Secondly, $q \circ \mathbf{p}^{(r-1)}$ also has one.
(4) Thus $\mathbf{p}^{(r)}$ generates exponential growth when iterated, after all.

We conclude that a cycle as assumed cannot exist. □

The realizability lemmas are intended to be applied under the assumption of Theorem 6.3, namely that our loop is polynomially bounded; therefore we can rely on the properties guaranteed by Lemmas 7.14 and 7.15. We focus on a particular idempotent realizable AMP $\mathbf{p}$. Then $G(\mathbf{p})$, with self-loops removed, is a DAG. We assume, w.l.o.g., that the variables are indexed in an order consistent with $G(\mathbf{p})$, so that if x_i depends on x_j then $j \leq i$. We shall refer to an (abstract) MP satisfying the properties in Lemma 7.14, Lemma 7.15 and with variables so re-indexed (if necessary) as being *neat*.[3]

Definition 7.16. $\mathbf{p} \in \tau\mathtt{MPol}$ is called *neat* if (a) $G(\mathbf{p})$, with self-loops removed, is a DAG; (b) for all i, $\mathbf{p}[i]$ has no monomial divisible by x_i except (possibly) x_i.

Note that $\mathbf{p}$ is not required to be idempotent; neat MPs that are not necessarily idempotent come up in the upper-bound proof. In the rest of this subsection we study neat idempotent AMPs, establishing (as Corollary 7.19 below) a property important for the subsequent realizability lemma. This property involves the nth iterate $\mathbf{p}^{(n)}$. It is not hard to prove that $(\mathbf{p}^{(n)})(s') \geq \mathbf{p}(s)$ if $s'[i] \geq s[i]$ for all i. We skip the proof, however, since we

[3]Readers who like linear algebra may draw some intuition about neat MPs from thinking about triangular matrices whose diagonal elements are in $\{0, 1\}$.

can establish a sharper result, for states s, s' where $s'[i] \geq s[i]$ is only asserted for $i \in \mathrm{SD}(\mathbf{p})$ (i.e., x_i is self-dependent in $\mathbf{p}$).

We introduce the following notation: for $\mathbf{p} \in \tau\texttt{AMPol}$, $\chi_{\mathbf{p}}(i)$ is 1 if $i \in \mathrm{SD}(\mathbf{p})$ and 0 otherwise.

Lemma 7.17. *Let $\mathbf{p} \in \tau\texttt{AMPol}$ be idempotent and neat. Let $s, s' \in \mathbb{N}^n$ be any state vectors such that for $i \in \mathrm{SD}(\mathbf{p})$, $s'[i] \geq s[i]$ (while for $i \notin \mathrm{SD}(\mathbf{p})$ no relation is asserted). Then for all $\ell \leq n$, $\mathbf{q}_\ell \stackrel{def}{=} (c{:}\mathbf{p})^{(\ell)}$ has the following property: for all $i \leq \ell$, for all $t \geq 0$,*

$$\mathbf{q}_\ell[i](s', t) \geq (c^{(d+1)^i}{:}\mathbf{p})[i](s, t) + (s'[i] - s[i]) \cdot \chi_{\mathbf{p}}(i),$$

where d is the maximum degree in $\mathbf{p}$.

The following fact is useful for proving the lemma; it follows from Lemma 7.5.

Fact 7.18. Let $\mathbf{p} \in \tau\texttt{MPol}$, $\mathbf{q} \in \tau\texttt{MPol}$. Suppose that for all $i < \ell$, we have $\mathbf{q}[i] \geq c^{(d+1)^i}{:}\mathbf{p}[i]$, were d is the maximum degree in $\mathbf{p}$. And suppose that r is a τ-polynomial depending only on $x_1, \ldots, x_{\ell-1}$, also of degree at most d. Then $(c{:}r) \circ \mathbf{q} \geq c^{(d+1)^\ell}{:}(r \circ \mathbf{p})$.

Proof of Lemma 7.17. We use induction on ℓ. Since $\mathbf{p}$ is neat we know that $\mathbf{p}[1] = x_1$. Thus

$$(c{:}\mathbf{p})[1](s', t) = s'[1] = s[1] + (s'[1] - s[1]) \cdot \chi_{\mathbf{p}}(1) = c^{d+1}{:}\mathbf{p}[1](s, t) + (s'[1] - s[1]) \cdot \chi_{\mathbf{p}}(1).$$

Next, for $\ell > 1$, consider $\mathbf{q}_\ell = (c{:}\mathbf{p}) \circ \mathbf{q}_{\ell-1}$. If $i \leq \ell$ and x_i is not self-dependent in $\mathbf{p}$, then $\mathbf{p}[i]$ only depends on variables x_j with $j < i$, so:

$$\begin{aligned}
\mathbf{q}_\ell[i](s', t) &= (c{:}\mathbf{p}[i] \circ \mathbf{q}_{\ell-1})(s', t) \\
&\geq (c{:}\mathbf{p}[i] \circ c^{(d+1)^{i-1}}{:}\mathbf{p})(s, t) && \text{by IH} \\
&\geq (c^{(d+1)^i}{:}(\mathbf{p}[i] \circ \mathbf{p}))(s, t) && \text{by Fact 7.18} \\
&\geq (c^{(d+1)^i}{:}\mathbf{p}[i])(s, t) && \text{by idempotence} \\
&= (c^{(d+1)^i}{:}\mathbf{p}[i])(s, t) + (s'[i] - s[i]) \cdot \chi_{\mathbf{p}}(i)
\end{aligned}$$

where the next-to-last step uses the idempotence of $\mathbf{p}$ in $\tau\texttt{AMPol}$ and Lemma 7.7.

If x_i is self-dependent in $\mathbf{p}$, then $\mathbf{p}[i] = x_i + r(\mathbf{x}, \tau)$ where r only depends on variables x_j with $j < i$. Now by IH, and Fact 7.18:

$$\begin{aligned}
\mathbf{q}_\ell[i](s', t) &= (c{:}\mathbf{p}[i] \circ \mathbf{q}_{\ell-1})(s', t) \\
&= \mathbf{q}_{\ell-1}[i](s', t) + (c{:}r \circ \mathbf{q}_{\ell-1})(s', t) \geq s'[i] + (c{:}r \circ (c^{(d+1)^{i-1}}{:}\mathbf{p}))(s, t) \\
&= s[i] + (s'[i] - s[i]) \cdot \chi_{\mathbf{p}}(i) + (c{:}r \circ (c^{(d+1)^{i-1}}{:}\mathbf{p}))(s, t) \\
&= (c{:}\mathbf{p}[i] \circ (c^{(d+1)^{i-1}}{:}\mathbf{p}))(s, t) + (s'[i] - s[i]) \cdot \chi_{\mathbf{p}}(i) \\
&\geq c^{(d+1)^i}{:}(\mathbf{p}[i] \circ \mathbf{p})(s, t) + (s'[i] - s[i]) \cdot \chi_{\mathbf{p}}(i) \\
&\geq (c^{(d+1)^i}{:}\mathbf{p}[i])(s, t) + (s'[i] - s[i]) \cdot \chi_{\mathbf{p}}(i)\,.
\end{aligned}$$

□

Letting $\ell = n$ in this lemma we obtain:

Corollary 7.19. *Let $\mathbf{p} \in \tau\texttt{AMPol}$ be idempotent and neat. Let $s, s' \in \mathbb{N}^n$ be any state vectors such that for $i \in \mathrm{SD}(\mathbf{p})$, $s'[i] \geq s[i]$. Then, for all $i \leq n$ and all t, $(c{:}\mathbf{p})^{(n)}[i](s', t) \geq (c^{(d+1)^n}{:}\mathbf{p})[i](s, t) + (s'[i] - s[i]) \cdot \chi_{\mathbf{p}}(i)$.*

7.1.3. *Realizability lemma for generalization.* This is what we have been preparing for.

Lemma 7.20. *Let* $\mathbf{p}$ *be a realizable, idempotent and neat abstract* τ*-MP. Then* $\mathbf{p}^\tau$ *is realizable.*

Note that if $\mathbf{p}$ is τ-closed, then $\mathbf{p}^\tau = \mathbf{p}$ and the statement is trivial. But if $\mathbf{p}$ is not τ-closed, then we have to construct a pattern that causes additive terms to accumulate in the self-dependent variables in order to justify the replacement of sums $x_i + \mathbf{p}[i]''$ by $x_i + \tau\mathbf{p}[i]''$. The interesting case is when $\mathbf{p}$ is not τ-free: then π already has stars, however nested stars are not allowed in a pattern, so we cannot iterate π. The solution is to use $(\pi(1))^*$. However, in reducing π to the star-free $\pi(1)$ we pull the rug under τ-monomials already present in $\mathbf{p}$. Therefore we use a pattern that includes both $(\pi(1))^*$ and π; actually, not π but π^n, in order to use Corollary 7.19. Note that realizability of $\mathbf{p}$ by π means that $\pi(1)$ realizes $\mathbf{p}[1/\tau]$ (which is $\mathbf{p}$ with all τ's erased). The following observation is useful:

Observation 7.21. If $\mathbf{p} \in \tau\texttt{AMPol}$ is idempotent, so is $\mathbf{p}[1/\tau]$; and similarly for neat.

Proof of Lemma 7.20. Let $\pi; c$ realize $\mathbf{p}$. Note that π will include stars iff $\mathbf{p}$ includes τ's. We are handling the general case so we treat $\mathbf{p}$ as a τ-MP; the arguments also apply to the case that it is τ-free. So, our assumption is that:

$$\text{if } \mathbf{x} \overset{\pi(t)}{\rightsquigarrow} \mathbf{y} \text{ then } \mathbf{y} \geq c{:}\mathbf{p}(\mathbf{x}, t)\,. \tag{7.2}$$

We will show that $\mathbf{p}^\tau$ is realized by the pattern $(\pi(1))^*\pi^n$. Now, traces in $L((\pi(1))^*\pi^n)$ have the form $\sigma_t = (\pi(1))^t\pi^n(t)$. We look at concrete traces

$$s_0 \overset{\pi(1)}{\rightsquigarrow} s_1 \cdots \overset{\pi(1)}{\rightsquigarrow} s_t \overset{\pi^n(t)}{\rightsquigarrow} s_{t+1}\,.$$

We first prove by induction on t, for x_i self-dependent in $\mathbf{p}$, and $t \geq 0$,

$$s_t[i] \geq s_0[i] + t(c{:}\mathbf{p}[i]''(s_0)) \tag{7.3}$$

(note that the omission of the τ parameter in $\mathbf{p}[i]''(s_0)$ reminds us that $\mathbf{p}[i]''$ is τ-free). For $t = 0$, the statement is trivial. For the inductive case,

$$\begin{aligned}
s_t[i] &\geq c{:}\mathbf{p}[i](s_{t-1}, 1) \\
&\geq s_{t-1}[i] + c{:}(\mathbf{p}[i]'(s_{t-1}, 1) + \mathbf{p}[i]''(s_{t-1}) + \mathbf{p}[i]'''(s_{t-1}, 1)) && \text{by structure of } \mathbf{p}[i] \\
&\geq s_{t-1}[i] + c{:}\mathbf{p}[i]''(s_{t-1}) \\
&\geq s_0[i] + (t-1)(c{:}\mathbf{p}[i]''(s_0)) + c{:}\mathbf{p}[i]''(s_{t-1}) && \text{by IH} \\
&\geq s_0[i] + (t-1)(c{:}\mathbf{p}[i]''(s_0)) + c{:}\mathbf{p}[i]''(s_0) && \text{(s.d. variables are non-decreasing)} \\
&\geq s_0[i] + t\mathbf{p}[i]''(s_0)
\end{aligned}$$

Now, for $i \in \mathrm{SD}(\mathbf{p})$,

$$\begin{aligned}
s_{t+1}[i] &\geq (c{:}\mathbf{p})^{(n)}[i](s_t, t) \\
&\geq (c^{(d+1)^n}{:}\mathbf{p})[i](s_0, t) + (s_t[i] - s_0[i]) && \text{by Cor. 7.19} \\
&= s_0[i] + c^{(d+1)^n}{:}(t\mathbf{p}[i]'(s_0, t) + \mathbf{p}[i]''(s_0) + \mathbf{p}[i]'''(s_0, t)) + (s_t[i] - s_0[i]) && \text{by Def. 5.10} \\
&\geq s_0[i] + c^{(d+1)^n}{:}(t\mathbf{p}[i]'(s_0, t) + \mathbf{p}[i]''(s_0) + \mathbf{p}[i]'''(s_0, t)) + t\mathbf{p}[i]''(s_0) && \text{by (7.3)} \\
&\geq s_0[i] + c^{(d+1)^n}{:}(t\mathbf{p}[i]'(s_0, t) + t\mathbf{p}[i]''(s_0) + \mathbf{p}[i]'''(s_0, t)) \\
&= c^{(d+1)^n}{:}\mathbf{p}^\tau[i](s_0, t)\,.
\end{aligned}$$

For $i \notin \mathrm{SD}(\mathbf{p})$,

$$\begin{aligned} s_{t+1}[i] &\geq (c{:}\mathbf{p})^{(n)}[i](s_t, t) \\ &\geq (c^{(d+1)^n}{:}\mathbf{p})[i](s_0, t) + (s_t[i] - s_0[i]) \cdot \chi_{\mathbf{p}}(i) \quad \text{by Cor. 7.19} \\ &= (c^{(d+1)^n}{:}\mathbf{p})[i](s_0, t)\,. \end{aligned}$$

We conclude that the pattern $(\pi(1))^*\pi^n$ realizes $\mathbf{p}^\tau$. □

We can now complete our lower-bound correctness proof:

Theorem 7.22. *Consider any set of MPs $\mathcal{S}$. Then the set of τ-MPs returned by* Solve($\mathcal{S}$) *is realizable over $\mathcal{S}$.*

Proof. This follows by induction on the number of operations used to construct each resulting τ-MP, being either composition steps or generalization steps (justified, respectively, by Lemma 7.10 or Lemma 7.20). □

This has a crucial corollary:

Corollary 7.23. *Consider any set of MPs $\mathcal{S}$, that represent a SDL in which all variables are polynomially bounded. Then the set of τ-MPs generated by* Solve($\mathcal{S}$) *is finite, and therefore obtained in a finite number of steps.*

Proof. Since we are working within τ`AMPol`, it is clear that if the set of τ-MPs computed by the algorithm is infinite, unbounded exponents must appear. Since all these MPs are realizable, this indicates that some variable cannot be bounded polynomially over all executions of the loop. Taking the contrapositive, we conclude that if all variables are known to be polynomially bounded, the set is finite. □

7.2. **Upper-Bound Correctness.** The upper-bound correctness establishes a correspondence between the set of AMPs computed by our algorithm and a set of concrete polynomials that actually provide upper bounds for all loop executions (as in Theorem 6.3, Clause 1). Implicitly, this proof provides *an algorithm to compute concrete upper bounds*, while ensuring that the abstractions of these bounds remain within our set of AMPs. But we do not attempt to explicate the algorithm as such, since our main contribution is the algorithm already given. Firstly, we have to provide a finer definition of "upper bound" in terms of *weighted traces*. The main technical part of the section defines a special class of τ-MPs, called *iterative*, and shows some properties of the class. Finally, the main part of the proof uses a corollary of Simon's *Forest Factorization Theorem* to construct the desired upper bounds. The application of this theorem is justified by Corollary 7.23. Before diving into upper bounds, we give a useful definition and some related lemmas.

Definition 7.24. For multiivariate polynomials p, q we define their *join* $p \sqcup q$ as the polynomial obtained by setting the coefficient of every monomial to the largest of its coefficients in p and in q.

For example: $(2x_1 + x_1x_2) \sqcup (x_1 + x_2 + 3x_1x_2) = 2x_1 + x_2 + 3x_1x_2\,.$

It is easy to see that $\sqcup$ is the join (least upper bound) operation in a natural partial order on polynomials, which we denote by $\sqsubset$. Note that this order is "syntactic" and is a part of the "semantic" relation $p \leq q$ defined by treating them as functions over $\mathbb{N}^n$. To see that the

relations do not coincide, consider xy versus $x^2 + y^2$. With multi-polynomials, we apply the relation component-wise.

Lemma 7.25. *Suppose that* $\mathbf{p} \sqsubset \mathbf{q}$ *and* $\mathbf{r} \sqsubset \mathbf{s}$. *Then* $\mathbf{p} \circ \mathbf{r} \sqsubset \mathbf{q} \circ \mathbf{s}$.

Proof. Every monomial $\mathfrak{m}$ of $\mathbf{p} \circ \mathbf{r}$ is the result of substituting a certain monomial of $\mathbf{r}[i]$ for every occurrence of x_i in $\mathbf{p}$ (where x_i^d counts as d occurrences). As the same monomials (up to coefficients) appear in, respectively, $\mathbf{s}[i]$ and $\mathbf{q}$, we conclude that $\mathfrak{m}$ (times some constant) appears in $\mathbf{q} \circ \mathbf{s}$. □

Using the join operator, we thus have: $\mathbf{p} \circ (\mathbf{r} \sqcup \mathbf{s}) \sqsupset (\mathbf{p} \circ \mathbf{r}) \sqcup (\mathbf{p} \circ \mathbf{s})$.

Lemma 7.26. *For any pair of polynomials* f, g, *and numbers* $a, b \geq 0$,

$$af + bg \leq (a + b)(f \sqcup g)\,.$$

Proof. Partition the monomials of $f \sqcup g$ into three groups: those that appear only in f; they are multiplied by a on the LHS and by $(a + b)$ on the RHS. Those that appear only in g; they are multiplied by b on the LHS and by $(a + b)$ on the RHS. For a monomial $\mathfrak{m}$ that appears in both polynomials—possibly with different coefficients, say c in f and d in g, we find $ac\mathfrak{m} + bd\mathfrak{m}$ in the LHS, and $(a + b)\max(c, d)\mathfrak{m}$ in the RHS. □

Weighted traces and upper bounds. We extend our notion of traces by defining *weighted traces*. The need for this definition arises because, in the proof, we argue about a decomposition of an actual trace into segments, and we need to abstract from the intermediate states within a segment. The number of actual transitions in the segment matters, however, and will be represented by its weight. Thus a weighted step, denoted $s \xrightarrow{\mathbf{p}|w} s'$, indicates that state s has evolved into step $s' = \mathbf{p}(s, w)$ in w actual transitions, for an integer $w \geq 1$. A weighted concrete trace σ is composed of such steps in the same way as an ordinary trace is; a weighted abstract trace specifies the bounds and weights but leaves the states unspecified, e.g., $(\mathbf{p}_1|w_1)(\mathbf{p}_2|w_2)(\mathbf{p}_3|w_3)$. The total weight of a trace σ is denoted by $\|\sigma\|$ and calculated by adding up the weights of the steps. Thus we have $\|\sigma\rho\| = \|\sigma\| + \|\rho\|$. Note that since the MPs that label a weighted step are, in general, upper bounds obtained in previous analysis steps, the value $s' = \mathbf{p}(s, w)$ represents, in general, a bound and not a state that is actually reachable; however, we argue that due to the monotonicity of the functions computable in our language, we do not lose soundness by considering s' instead of a set of values bounded by s'.

We denote the set of weighted traces over $\mathcal{S}$ by $\mathcal{WS}^*$. Note that weights are associated with abstract transitions and all concretizations of a weighted trace have the same weight.

Definition 7.27 (bound for a trace)**.** Let $\sigma \in \widetilde{\mathcal{WS}^*}$. We say that σ admits a τ-polynomial p as an upper bound on variable j, or that p bounds variable j in σ, if the following holds:

$$\mathbf{x} \overset{\sigma}{\rightsquigarrow} \mathbf{y} \wedge t \geq \|\sigma\| \implies y_j \leq p(\mathbf{x}, t)\,. \tag{7.4}$$

If $\mathbf{p}$ bounds *all* variables in a concrete weighted trace $\mathbf{x} \overset{\sigma}{\rightsquigarrow} \mathbf{y}$, we say that $\mathbf{p}$ bounds this trace. If $\mathbf{p}$ bounds all concretizations of σ, we say it bounds σ.

The following lemma follows from the monotonicity of all our polynomials.

Lemma 7.28. *If* $\mathbf{p}$ *bounds* σ *and* $\mathbf{q}$ *bounds* ρ, *then* $\mathbf{q} \circ \mathbf{p}$ *bounds* $\sigma\rho$.

Proof. Let $t_s = \|\sigma\|$, $t_r = \|\rho\|$. Consider a concrete run of $\sigma\rho$ and name the states thus: $\mathbf{x} \overset{\sigma}{\rightsquigarrow} \mathbf{y} \overset{\rho}{\rightsquigarrow} \mathbf{z}$. Then $z_i \le \mathbf{q}[i](\mathbf{y}, t_r)$ (by assumption) and for any j, $y_j \le \mathbf{p}[j](\mathbf{x}, t_s)$. It clearly follows that $z_i \le (\mathbf{q}[i] \circ \mathbf{p})(\mathbf{x}, t_s + t_r)$. Note that $t_s + t_r$ is precisely $\|\sigma\rho\|$. □

Iterative MPs and upper bounds for sequences of similar MPs.

Definition 7.29. $\mathbf{p} \in \tau\texttt{MPol}$ is called *iterative* if all its entries depend only on self-dependent variables. Moreover, each $\mathbf{p}[i]$ depends only on variables with indices $j \le i$.

Example 7.30. $\langle x_1, x_2 + x_1, x_2 x_1 \rangle$ is iterative, while $\langle x_1, x_1, x_3 + x_2 + x_1 \rangle$ is not (note that the last MP is idempotent when abstracted to `AMPol`, which demonstrates that this property does not imply iterativity; the converse does not hold, either).

We abbreviate "iterative multi-polynomial" to IMP. We next give some technical definitions and results regarding IMPs. The upshot of this part is the computation of tight upper bounds for the end-state of certain traces composed of IMPs. We now attempt to give some intuition. Consider a sequence of MPs which are idempotent in their abstract form. Thus if we compose their abstractions in `AMPol` we just get the same AMP again. This is a nice situation which suggests that we can extrapolate the bounds for a single step to bounds for any number of steps. But there may be some values that grow and we will have to account for this in our upper bounds. Consider the following MP:

$$\langle x_1 + x_2 + x_3,\ x_3,\ x_3 \rangle$$

Note that it is idempotent in `AMPol`. However, in concrete computation, the increments accumulate. But hastily changing the first component to $x_1 + \tau(x_2 + x_3)$ overshoots the upper bound. The correct result is $x_1 + x_2 + x_3 + \tau(2x_3)$. We find this result by extracting an iterative MP from the given set of MPs. In the above example, the extracted IMP will be $\langle x_1 + 2x_3,\ x_3,\ x_3 \rangle$; we will see later why.

Moving to more precise details, we are going to consider sequences of *similar* MPs, where similarity means that we get the same MP if we ignore the coefficients as well as τ symbols, as expressed by the following set of definitions.

Definition 7.31. Let $\mathbf{p} \in \tau\texttt{MPol}$. We define $\alpha!(\mathbf{p})$ to be $\alpha(\mathbf{p}[1/\tau])$.

For example, $\alpha!(\langle x_1 + \tau x_2, x_1^2 + \tau x_1^2 \rangle) = \langle x_1 + x_2, x_1^2 \rangle$.

Definition 7.32. We call $\mathbf{p}$, $\mathbf{q}$ *similar* if $\alpha!(\mathbf{p}) = \alpha!(\mathbf{q})$.

Observation 7.33. Suppose that $\alpha!(\mathbf{p}) = \alpha!(\mathbf{q})$. Then for any $\mathbf{t}$, $\alpha!(\mathbf{p} \circ \mathbf{t}) = \alpha!(\mathbf{q} \circ \mathbf{t}) = \alpha!(\mathbf{p}) \bullet \alpha!(\mathbf{t})$.

Whenever an idempotent MP is considered, we also assume it is realizable, which in turn allows us to assume that its variables are indexed in topological order, as in the previous section, making it neat. One could worry that the assumptions of topological order might contradict each other in an argument that involves several different idempotents, but this is not a problem since all the development below only concerns a set of similar MPs, so they agree on the topological indexing. In such a discussion we may say, for example, that some x_i is self-dependent, without specifying which MP of the set is concerned.

Definition 7.34. Let $\mathbf{p} \in \tau\texttt{MPol}$ be such that $\alpha!(\mathbf{p})$ is neat. Define its *self-dependent cut* $\lfloor \mathbf{p} \rfloor$ as follows: for all i self-dependent in $\mathbf{p}$, $\lfloor \mathbf{p} \rfloor[i] = x_i$. For all other i, $\lfloor \mathbf{p} \rfloor[i] = \mathbf{p}[i]$.

Lemma 7.35. *Let $\mathbf{p}_1, \ldots, \mathbf{p}_n$ be similar τ-MPs such that $\alpha!(\mathbf{p}_1) = \alpha!(\mathbf{p}_2) = \ldots$ is neat. Then for all $\ell \leq n$, $\mathbf{q}_\ell \stackrel{def}{=} \mathbf{p}_1 \circ \lfloor \mathbf{p}_2 \rfloor \cdots \circ \lfloor \mathbf{p}_\ell \rfloor$ has the following property: for all i, $\mathbf{q}_\ell[i]$ only depends on variables x_j which are either self-dependent or else $j \leq i - \ell$.*

Proof. We use induction on ℓ. For $\ell = 1$, the result is immediate, by virtue of the topological indexing. Next, for $\ell > 1$, consider $\mathbf{q}_{\ell-1} = \mathbf{p}_1 \circ \cdots \circ \lfloor \mathbf{p}_{\ell-1} \rfloor$. Now $\mathbf{q}_\ell = \mathbf{q}_{\ell-1} \circ \lfloor \mathbf{p}_\ell \rfloor$. By IH, $\mathbf{q}_{\ell-1}[i]$ only depends on variables x_j which are either self-dependent or have $j \leq i - \ell + 1$. In the case that x_j is self-dependent, then $\lfloor \mathbf{p}_\ell \rfloor[j] = x_j$, so $\mathbf{q}_\ell[i]$ also depends on x_j which agrees with the lemma. In the case that x_j is not self-dependent, $\lfloor \mathbf{p}_\ell \rfloor[j]$ must depend on variables x_k with $k \leq j - 1 \leq i - \ell$. □

This lemma is actually formulated for induction; what we are really interested in is the MP denoted by $\mathbf{q}_n$, for which we introduce a special notation while enunciating the following corollary:

Corollary 7.36. *Let $\mathbf{p}_1, \ldots, \mathbf{p}_n$ be similar τ-MPs such that $\alpha!(\mathbf{p}_1) = \alpha!(\mathbf{p}_2) = \ldots$ is neat. Then $\lfloor\!\lfloor \mathbf{p}_1 \cdots \mathbf{p}_n \rfloor\!\rfloor \stackrel{def}{=} \mathbf{p}_1 \circ \lfloor \mathbf{p}_2 \rfloor \cdots \circ \lfloor \mathbf{p}_n \rfloor$ is iterative.*

Notation: if S is a set of neat, similar τ-MPs as above, we let

$$\lfloor\!\lfloor S \rfloor\!\rfloor \stackrel{def}{=} \{ \lfloor\!\lfloor \mathbf{p}_1 \ \ldots \ \mathbf{p}_n \rfloor\!\rfloor \mid \mathbf{p}_1, \ldots, \mathbf{p}_n \in S \}.$$

Note that the set $\lfloor\!\lfloor S \rfloor\!\rfloor$ consists of IMPs (by the above corollary) and it is easy to see that they are neat and similar to each other. Thus, we have a way to extract a set of IMPs from a certain set of similar MPs. We will next move to the way in which we compute upper bounds for sequences of similar MPs using these IMPs. Recall that a τ-MP is *τ-closed* if $\mathbf{p}^\tau = \mathbf{p}$. In particular, τ-MPs that result directly from generalization (i.e., $\mathbf{q}^\tau$ for some $\mathbf{q}$) are τ-closed. We next define a special composition operator for τ-closed IMPs, and show an interesting property that it has. Note that, using the notation of Definition 5.10 (Page 13), if $\mathbf{p}$ is a τ-closed IMP then its self-dependent entries have the form $\mathbf{p}[i] = x_i + \tau\mathbf{p}[i]'$.

Definition 7.37 (τ-absorbing composition)**.** We introduce a non-standard composition operation on similar, τ-closed IMPs, denoted by the operator $\star$ (differing from both ordinary composition $\circ$ and abstract composition $\bullet$). Specifically, $(\mathbf{q} \star \mathbf{p})[i]$ is defined as follows:

(1) If $i \in \mathrm{SD}(\mathbf{q}) = \mathrm{SD}(\mathbf{p})$, then $(\mathbf{q} \star \mathbf{p})[i] \stackrel{def}{=} \mathbf{p}[i] \sqcup \tau(\mathbf{q}[i]' \circ \mathbf{p}) = x_i + \tau(\mathbf{p}[i]' \sqcup (\mathbf{q}[i]' \circ \mathbf{p}))$.

(2) Otherwise, $(\mathbf{q} \star \mathbf{p})[i] \stackrel{def}{=} (\mathbf{q}[i] \circ \mathbf{p})$.

Observe that the difference between $\mathbf{q} \star \mathbf{p}$ and $\mathbf{q} \circ \mathbf{p}$ is only in the numeric coefficient of some monomials. In particular, they have the same abstraction: $\alpha(\mathbf{q} \star \mathbf{p}) = \alpha(\mathbf{q} \circ \mathbf{p}) = \alpha(\mathbf{q}) \bullet \alpha(\mathbf{p})$.

Example 7.38. Consider the following τ-polynomial:

$$\mathbf{p} = \langle \quad x_1, \quad x_2 + \tau x_1, \quad x_3 + \tau x_2 + \tau^2 x_1, \quad x_4 \rangle$$

Then:

$$\mathbf{p} \star \mathbf{p} = \langle \quad x_1, \quad x_2 + \tau x_1, \quad x_3 + \tau x_2 + \tau^2 x_1, \quad x_4 \rangle \quad = \mathbf{p}$$

while:

$$\mathbf{p} \circ \mathbf{p} = \langle \quad x_1, \quad x_2 + 2\tau x_1, \quad x_3 + 2\tau x_2 + 3\tau^2 x_1, \quad x_4 \rangle.$$

Thus, the operation keeps (some) coefficients from growing when we compose τ-MPs. We will see next that this has the desirable result that a sequence of "$\star$ powers" $\mathcal{T}^{\star(i)}$ (defined analogously to Definition 5.4 on Page 12) only includes a finite number of MPs. On the other hand we will prove that it gives a sound upper bound for a sequence of transitions.

Lemma 7.39. *Let* $\mathbf{p}$, $\mathbf{q}$ *be* τ*-closed, similar IMPs and let* σ, ρ *be weighted traces such that* $\mathbf{p}$ *bounds* σ *and* $\mathbf{q}$ *bounds* ρ*. Then* $\mathbf{q} \star \mathbf{p}$ *bounds* $\sigma\rho$*.*

Compared to Lemma 7.28, this lemma poses stronger conditions, but has a stronger conclusion since $\mathbf{q} \star \mathbf{p}$ is, in general, lower than $\mathbf{q} \circ \mathbf{p}$.

Proof. The definition of $\mathbf{q} \star \mathbf{p}$ makes the statement trivial for variables that are not self-dependent. So, let x_i be self-dependent. Suppose that $\mathbf{x} \overset{\sigma}{\rightsquigarrow} \mathbf{y} \overset{\rho}{\rightsquigarrow} \mathbf{z}$. Let $t_s = \|\sigma\|$, $t_r = \|\rho\|$. By assumption,

$$\begin{aligned}
z_i &\le \mathbf{q}(\mathbf{y}, t_r) \\
&= y_i + t_r \mathbf{q}[i]'(\mathbf{y}, t_r) \\
&\le \big(x_i + t_s \mathbf{p}[i]'(\mathbf{x}, t_s)\big) + t_r \mathbf{q}[i]'(\mathbf{y}, t_r) \\
&\le \big(x_i + t_s \mathbf{p}[i]'(\mathbf{x}, t_s + t_r)\big) + t_r (\mathbf{q}[i]' \circ \mathbf{p})(\mathbf{x}, t_s + t_r) \\
&\le x_i + (t_s + t_r)(\mathbf{p}[i]' \sqcup (\mathbf{q}[i]' \circ \mathbf{p}))(\mathbf{x}, t_s + t_r) && \text{(by Lemma 7.26)} \\
&= (\mathbf{q} \star \mathbf{p})[i](t_s + t_r)\,. && \square
\end{aligned}$$

Lemma 7.40 (The Finite Closure Lemma)**.** *Let* $\mathcal{T}$ *be a finite set of* τ*-closed, similar IMPs. Then the set*

$$\mathcal{T}^{\bigstar} = \bigcup_{k=1}^{\infty} \mathcal{T}^{\star(k)}$$

is finite. Hence, there is a number ℓ *such that* $\mathcal{T}^{\bigstar} = \mathcal{T}^{\star(\ell)}$*.*

Example 7.41. Consider $\mathcal{T} = \{\mathbf{p}, \mathbf{q}\}$ with $\mathbf{p}$ from Example 7.38 and

$$\mathbf{q} = \langle \quad x_1 + \tau x_4, \quad x_2, \quad x_3 + \tau x_4, \quad x_4 \rangle.$$

Then

$$(\mathcal{T})^{\star(2)} = \quad \mathcal{T} \star \mathcal{T} = \quad \begin{matrix} \{\mathbf{p}\star\mathbf{p}, & \mathbf{q}\star\mathbf{p}, & \mathbf{q}\star\mathbf{q}, & \mathbf{p}\star\mathbf{q}\} = \\ \{\mathbf{p}, & \mathbf{r}, & \mathbf{q}, & \mathbf{r}\} \end{matrix}$$

Where

$$\mathbf{r} = \langle \quad x_1 + \tau x_4, \quad x_2 + \tau x_1, \quad x_3 + \tau x_2 + \tau^2 x_1 + \tau x_4, \quad x_4 \rangle.$$

Next, $(\mathcal{T})^{\star(3)}$ consists of 8 products, which contribute only one new element to the accumulated union, namely

$$\mathbf{p} \star \mathbf{r} = \langle \quad x_1 + \tau x_4, \quad x_2 + \tau x_1, \quad x_3 + \tau x_2 + 2\tau^2 x_1 + \tau x_1 + \tau x_4, \quad x_4 \rangle.$$

Adding $(\mathcal{T})^{\star(4)}$ to the union, we only have to check $\mathbf{p} \star \mathbf{p} \star \mathbf{r}$ and $\mathbf{q} \star \mathbf{p} \star \mathbf{r}$. It turns out that these τ-MPs have appeared already. Therefore, we have reached $\mathcal{T}^{\bigstar}$.

Proof. Define the following sequence:

$$\begin{aligned}
T_{(0)} &= Id \\
T_{(k+1)} &= T_{(k)} \sqcup \big(\bigsqcup \mathcal{T}\big) \circ T_{(k)}\,.
\end{aligned}$$

We prove the following statement by complete induction on k:

$$\mathbf{r} \in \mathcal{T}^{\bigstar},\ i \le k \ \Rightarrow \mathbf{r}[i] \sqsubset T_{(k)}[i]\,. \tag{7.5}$$

This implies the desired result because it shows that for any $i \le n$, the set of values assumed by $\mathbf{r}[i]$, when $\mathbf{r}$ ranges over $\mathcal{T}^{\bigstar}$, is finite, because it is contained in the down-set of a single

polynomial, namely $T_{(i)}[i]$, and such a down-set is always finite. Hence, $\mathcal{T}^{\bigstar}$ is finite. It is worthwhile to observe that since the members of $\mathcal{T}$ are similar IMPs, then, whenever i is a self-dependent variable (in any of them—and then in all), $\bigsqcup\mathcal{T}$ also has the form $x_i +$ (function of lower variables) and, consequently, so do the $T_{(k)}$.

To prove (7.5), fix k and $i \leq k$. We now employ induction on the number t of compositions used to construct $\mathbf{r}$ (following the definition of $\mathcal{T}^{\bigstar}$). The base case is $t = 1$, i.e., $\mathbf{r} \in \mathcal{T}$ where the result is immediate. In the general case,

$$\mathbf{r} = \mathbf{q} \star \mathbf{f}$$

where $\mathbf{q} \in \mathcal{T}$ and $\mathbf{f} \in \mathcal{T}^{\star(t-1)}$.

Case 1: we consider $\mathbf{r}[i]$ when $i \in \mathrm{SD}(\mathbf{q})$. Write

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

then

$$\mathbf{r}[i] = (x_i + \tau\mathbf{f}[i]') \sqcup \tau(\mathbf{q}[i]' \circ \mathbf{f})$$

Note that $\mathbf{q}[i]'$ only depends on variables x_j with $j < i \leq k$. By the induction hypothesis (of the induction on k) we have

$$\mathbf{f}[j] \sqsubset T_{(k-1)}[j]$$

Consequently

$$\tau(\mathbf{q}[i]' \circ \mathbf{f}) \sqsubset \left(\mathbf{q} \circ T_{(k-1)}\right)[i] \sqsubset T_{(k)}[i]\,.$$

By the induction hypothesis on t,

$$\mathbf{f}[i] = x_i + \tau\mathbf{f}[i]' \sqsubset T_{(k)}[i]$$

and we conclude that

$$(x_i + \tau\mathbf{f}[i]') \sqcup \tau(\mathbf{q}[i]' \circ \mathbf{f}) \sqsubset T_{(k)}[i] \sqcup T_{(k)}[i] = T_{(k)}[i]\,.$$

Case 2: consider $i \notin \mathrm{SD}(\mathbf{q})$. Then $\mathbf{q}[i]$ only depends on self-dependent variables lower than i, hence lower than k. By the induction hypothesis on k, when $j < k$ we have

$$\mathbf{f}[j] \sqsubset T_{(k-1)}[j]$$

Consequently

$$\mathbf{r}[i] = \mathbf{q}[i] \circ \mathbf{f} \sqsubset T_{(k)}[i]\,. \qquad \square$$

We are heading towards a central part of the proof, where we construct upper bounds for sequences of MPs taken from a set $\mathcal{S}$ of similar τ-MPs. Moreover, we restrict attention to MPs $\mathbf{p}$ such that $\alpha!(\mathbf{p})$ is idempotent. Lemma 7.40 is used to show that a finite set of upper bounds covers such traces of any length. However, it requires the MPs to be τ-closed IMPs. We close this gap by demonstrating that we can obtain upper bounds for such traces using $\llbracket\mathcal{S}\rrbracket^{\tau}$ instead of $\mathcal{S}$ (the τ superscript over a set name means that we apply generalization to all members of the set).

Lemma 7.42. *Let $\mathcal{S}$ be a set of similar τ-MPs such that $\alpha!(\mathbf{p})$, for $\mathbf{p} \in \mathcal{S}$, is neat. For any $t > n$, consider a concrete trace*

$$\widetilde{\sigma} = s_0 \xrightarrow{\mathbf{p}_1|w_1} s_1 \ldots s_{t-1} \xrightarrow{\mathbf{p}_t|w_t} s_t\,.$$

Let $\mathbf{r} = \llbracket\mathbf{p}_t \ \ldots \ \mathbf{p}_{t-n+1}\rrbracket$. Then $s_t \leq \mathbf{r}(s_{t-1}, w')$ where $w' = \sum_{z=t-n+1}^{t} w_z$. In other words, $\mathbf{r}$ bounds the last step of σ provided that its weight is suitably increased.

The point is to replace $\mathbf{p}_t$ by $\mathbf{r}$ as a bound on the last step, the gain being the convenient form of $\mathbf{r}$, namely an IMP.

Proof. We use induction on d to prove the following claim for all $0 < d \leq n$:

Let $\mathbf{r}_d = \lfloor\!\lfloor \mathbf{p}_{t-n+d} \ \cdots \ \mathbf{p}_{t-n+1} \rfloor\!\rfloor$ and let $w'_d = \sum_{z=t-n+1}^{t-n+d} w_z$.

Then for $i \leq d$, $s_{t-n+d}[i] \leq \mathbf{r}_d(s_{t-n+d-1}, w'_d)$.

Note that this claim implies the lemma's statement (setting $d = n$). We now move to its proof.

Base case: $d = 1$. Hence $\mathbf{r}_d = \mathbf{p}_{t-n+1}$. The claim only concerns $i = 1$, and $s_{t-n+1}[i] \leq \mathbf{p}_{t-n}(s_{t-n+1}, w'_1)[i]$ by definition.

Inductive case: $d > 1$. Then

$$\mathbf{r}_d = \mathbf{p}_{t-n+d} \circ \lfloor \mathbf{p}_{t-n+d-1} \rfloor \circ \cdots \circ \lfloor \mathbf{p}_{t-n+1} \rfloor \,.$$

Consider $s_{t-n+d}[i]$, for some $i \leq d$. By assumption,

$$s_{t-n+d}[i] = \mathbf{p}_{t-n+d}[i](s_{t-n+d-1}, w_{t-n+d}) \,. \tag{7.6}$$

Let us consider the entries $s_{t-n+d-1}[j]$ on which the above expression may depend; thus $j \leq i$. Let us focus first on the case that x_j is self-dependent. We observe that

$$s_{t-n+d-1}[j] = (\lfloor \mathbf{p}_{t-n+d-1} \rfloor \circ \cdots \circ \lfloor \mathbf{p}_{t-n+1} \rfloor)[j](s_{t-n+d-1}, w) \tag{7.7}$$

for any w, since $\lfloor \mathbf{p} \rfloor[j] = x_j$ for all $\mathbf{p} \in \mathcal{S}$ (note that both sides of the equation refer to $s_{t-n+d-1}$).

When x_j is not self-dependent, we have (using the induction hypothesis, and the definition of self-dependent cut)

$$\begin{aligned} s_{t-n+d-1}[j] &\leq (\mathbf{p}_{t-n+d-1} \circ \lfloor \mathbf{p}_{t-n+d-2} \rfloor \circ \cdots \circ \lfloor \mathbf{p}_{t-n+1} \rfloor)[j](s_{t-n+d-2}, w'_{d-1}) \\ &= (\lfloor \mathbf{p}_{t-n+d-1} \rfloor \circ \cdots \circ \lfloor \mathbf{p}_{t-n+1} \rfloor)[j](s_{t-n+d-2}, w'_{d-1}) \end{aligned} \tag{7.8}$$

We now wish to replace $s_{t-n+d-2}$ in the last expression by $s_{t-n+d-1}$. To this end we apply Lemma 7.35, establishing that the entries $s_{t-n+d-2}[e]$ that influence the last expression are either self-dependent or have index $e \leq j - d + 1 \leq 1$. But in the latter case $e = 1$ and x_1 is certainly self-dependent. We conclude that $\mathbf{p}_{t-n+d-1}[e] \geq x_e$, hence $s_{t-n+d-1}[e] \geq s_{t-n+d-2}[e]$. We now obtain from (7.8) that

$$\begin{aligned} s_{t-n+d-1}[j] &\leq (\lfloor \mathbf{p}_{t-n+d-1} \rfloor \circ \cdots \circ \lfloor \mathbf{p}_{t-n+1} \rfloor)[j](s_{t-n+d-2}, w'_{d-1}) \\ &\leq (\lfloor \mathbf{p}_{t-n+d-1} \rfloor \circ \cdots \circ \lfloor \mathbf{p}_{t-n+1} \rfloor)[j](s_{t-n+d-1}, w'_{d-1}) \,. \end{aligned}$$

We substitute this in (7.6) to obtain

$$\begin{aligned} s_{t-n+d}[i] &\leq (\mathbf{p}_{t-n+d}[i] \circ (\lfloor \mathbf{p}_{t-n+d-1} \rfloor \circ \cdots \circ \lfloor \mathbf{p}_{t-n+1} \rfloor))(s_{t-n+d-1}, w'_{d-1} + w_{t-n+d}) \\ &= \mathbf{r}_d(s_{t-n+d-1}, w'_d) \,. \end{aligned}$$

□

Lemma 7.43. *Let $\mathcal{S}$ be a set of similar τ-MPs such that $\alpha!(\mathbf{p})$, for all $\mathbf{p} \in \mathcal{S}$, is neat. Then every weighted trace over $\mathcal{S}$ of length greater than n has an upper bound in the set $\left((\lfloor\!\lfloor \mathcal{S} \rfloor\!\rfloor^{\tau})^{\bigstar} \circ \mathcal{S}^{(n)} \right) [n\tau/\tau]$.*

Proof. Write such a trace as $\sigma = s_0 \xrightarrow{\mathbf{p}_1|w_1} s_1 \dots s_{t-1} \xrightarrow{\mathbf{p}_t|w_t} s_t$. Let σ_i be the ith step, namely $s_{i-1} \xrightarrow{\mathbf{p}_i|w_i} s_i$. For each $i > n$, by the last lemma, the ith transition is bounded by a τ-MP $\mathbf{r} \in \lfloor\!\lfloor \mathcal{S} \rfloor\!\rfloor$, with modified weight. If $\mathbf{r}$ bounds a certain transition, then $\mathbf{r}^\tau$ certainly does. Consequently, by Lemmas 7.28 and 7.39, $\sigma_1 \dots \sigma_n \sigma'_{n+1} \dots \sigma'_t$ (where the primed transitions use modified weights) is bounded by a τ-MP $\mathbf{q} \in (\lfloor\!\lfloor \mathcal{S} \rfloor\!\rfloor^\tau)^\bigstar \circ \mathcal{S}^n$. In order to get rid of the modified weight, we note that

$$\begin{aligned}\|\sigma_1 \dots \sigma_n \sigma'_{n+1} \dots \sigma'_t\| &= \|\sigma_1 \dots \sigma_n\| + \sum_{i=n+1}^{t} \|\sigma'_i\| \\ &= \sum_{i=1}^{n} w_i + \sum_{i=n+1}^{t} \sum_{z=i-n+1}^{i} w_z \\ &\leq n\|\sigma\|\end{aligned}$$

Thus we see that $\mathbf{q}[n\tau/\tau]$ bounds σ using its original weight. □

Let $\mathcal{S}$ be the SDL under analysis, and $\alpha(\mathcal{S})$ map it into `AMPol`. The latter is a monoid with respect to the composition operation, and we obtain a monoid homomorphism $\alpha : \mathcal{S}^* \to$ `AMPol` from abstract traces to `AMPol`. The *Factorization Forest Theorem* of Imre Simon [Sim90] shows that such a homomorphism induces a useful structure on the traces, provided that the codomain is a *finite* monoid. Now, `AMPol` is, of course, infinite; but if we assume that the loop under analysis is polynomially bounded, then, as argued in the proof of termination (Corollary 7.23), the closure $Cl(\alpha(\mathcal{S}))$ is finite. Note that it is a sub-monoid of `AMPol`. Thus, Simon's theorem can be applied. Instead of the original formulation, we can use a convenient corollary of Simon's theorem, stated by Bojańczyk [Boj09] (actually we will not use his formulation but a simplified one, as we do not need its full power).

In the statement of this result, we refer to $\alpha(\sigma)$, with $\sigma \in \mathcal{S}^*$, as the *type* of σ. We consider regular expressions constructed using the operators: concatenation, union and Kleene-plus (where E^+ generates the union of all languages generated by E^i for $i > 0$). A regular expression E over the alphabet $\mathcal{S}$ is *well-typed* if for each of its sub-expressions F (including E), all words (traces) generated by F have the same type, which is then the type of the expression.

Theorem 7.44 (Bojańczyk)**.** *The existence of a homomorphism $\alpha : \mathcal{S}^* \to M$, where M is a finite monoid, implies that $\mathcal{S}^*$ can be generated by a finite union of well-typed regular expressions.*

Now, all we have to do is prove that the set of AMPs returned by our algorithm provides upper bounds for all the traces generated by each of these regular expressions. Note that this theorem highlights the role of *idempotence* in `AMPol`, since for an expression E^+ to be well-typed, $\alpha(\sigma)$ has to be the same for all words σ generated by E^+, which implies that it is an idempotent element.

Theorem 7.45. *Let $\mathcal{S}$ be a polynomially-bounded SDL. Let $\mathcal{A}$ be the set of abstract τ-MPs returned by Algorithm* Solve*. Then there is a finite set $\mathcal{B} \subset \tau$`MPol` such that $\alpha(\mathcal{B}) \subseteq \mathcal{A}$ and for any trace $\sigma \in \mathcal{S}^*$ there exists $\mathbf{p} \in \mathcal{B}$ that bounds σ.*

Proof. We consider the regular expressions established by Theorem 7.44 and all their sub-expressions: this is a finite set. We construct a set of bounds $\mathcal{B}_E$, with $\alpha(\mathcal{B}_E) \subseteq \mathcal{A}$, for each

such sub-expression E, by structural induction on the expressions. Clearly, this proves the theorem. Importantly, the construction maintains these properties:

- If expression E has type $\mathbf{a} \in \texttt{AMPol}$ then every $\mathbf{p} \in \mathcal{B}_E$ has $\alpha!(\mathbf{p}) = \mathbf{a}$ (intuitively, the algorithm does not change the shape of the multi-polynomials except by adding τ's).
- Moreover, $\mathbf{p}$ is realizable (this is not hard but requires a bit of attention since Theorem 7.22 does not refer to the bounds constructed in the current proof, but only to $\mathcal{A}$).

If E is a single transition $\mathbf{p}$ we set $\mathcal{B} = \{\mathbf{p}\}$.

If E is FG we compose $\mathcal{B}_F$ with $\mathcal{B}_G$. Note that $\alpha(\mathcal{B}_G \circ \mathcal{B}_F) = \alpha(\mathcal{B}_G) \bullet \alpha(\mathcal{B}_F) \subseteq \mathcal{A}$ thanks to the closure computation. Realizability follows from Lemma 7.10.

If E is $F + G$ we unite $\mathcal{B}_F$ with $\mathcal{B}_G$. Again, the abstraction of the result is in $\mathcal{A}$.

It remains to consider an expression of the form F^+. Consider $\mathcal{B}_F$. By IH, it consists of realizable τ-MPs and are all similar to a single idempotent AMP, so the requirements of Lemma 7.43 are satisfied. Let Φ be the set of traces generated by F. Then every $\sigma \in \Phi$ has a bound in $\mathcal{B}_F$. For a concatenation of $i \le n$ such traces we have a bound in $(\mathcal{B}_F)^{(i)}$. For a concatenation of more than n traces, consider each of these traces as a weighted transition where the weight represents the trace's length. By Lemma 7.43, the concatenation of the traces has a bound in

$$\left((\llbracket \mathcal{B}_F \rrbracket^\tau)^\bigstar \circ {\mathcal{B}_F}^{(n)} \right) [n\tau/\tau] \,. \tag{7.9}$$

By Lemma 7.40, we can replace $(\llbracket \mathcal{B}_F \rrbracket^\tau)^\bigstar$ with $(\llbracket \mathcal{B}_F \rrbracket^\tau)^{\star(\ell)}$ for some $\ell > 0$. Moreover, it is sound to relax the upper bound to $({\mathcal{B}_F}^\tau)^{\star(\ell)}$. Now (7.9) becomes

$$\left(({\mathcal{B}_F}^\tau)^{\star(\ell)} \circ {\mathcal{B}_F}^{(n)} \right) [n\tau/\tau] \,.$$

We claim that this set of upper bounds satisfies all our requirements. For realizability, all we need is the observation that substituting $n\tau$ for τ does not affect realizability; plus Lemma 7.10 and the fact $\mathbf{q} \star \mathbf{p} \le \mathbf{q} \circ \mathbf{p}$.

Finally we look at the abstractions of these τ-MPs, namely the set

$$\alpha\left(\left(({\mathcal{B}_F}^\tau)^{\star(\ell)} \circ {\mathcal{B}_F}^{(n)} \right) [n\tau/\tau] \right) = (\alpha(\mathcal{B}_F)^\tau)^{\bullet(\ell)} \bullet (\alpha(\mathcal{B}_F))^{\bullet(n)} \,.$$

These AMPs are included in the result of Solve, since they they are produced by closure, generalization and closure again. Note also that the bounds conform with the type of F (when τ's are ignored). □

8. On the Computational Complexity of our Problem

Our main goal in this research was to establish that the problem of computing tight bounds is solvable. However, once proved solvable, the question of its complexity arises. For simplicity we assume that we are only dealing with programs where all variables are polynomially bounded. We consider the complexity in terms of three parameters: $|P|$, the size of the program; n, the number of variables; and d, the highest degree reached. We note that if the user wishes to verify a desired degree bound d, say check that a program is at most of cubic

time complexity, it is possible to use an abstraction that truncates exponents higher than d and the complexity will be reduced accordingly. First, we give an upper bound.[4]

Theorem 8.1. *Our algorithm runs in time polynomial in* $|P| \cdot 2^{n^{d+1}}$.

Proof. We estimate the complexity of our algorithm, based on bounding the number of different AMPs that may be encountered. The number of monomials over n variables is bounded (for $n > 2$) by n^d (think of a monomial as a product of at most d variables. For uniqueness list the indices in descending order. The number of descending lists of length at most d is bounded by n^d). Since a MP is an n-tuple of sets of monomials, the number of possible AMPs is less than $\left(2^{n^d}\right)^n = 2^{n^{d+1}}$.

Next, we consider the running time of the analysis of a loop (independent of the position of the loop in the program, so we can later just multiply this time bound by the number of loops in the program). In procedure Solve, a set of AMPs is maintained and repeatedly enlarged (by applying closure and generalization), until stable. The time to perform each round of enlargement is polynomial in the size of the resulting set (more precisely its representation, but with any reasonable implementation this does not change much) and the number of rounds is clearly bounded by the size of the final set. So we deduce that the total time is polynomial in $2^{n^{d+1}}$.

Finally we should add the time to represent non-looping code as a set of AMPs, and other "book-keeping" operations, but clearly they contribute at most a polynomial in $|P|$ and the number of AMPs. □

Is this a satisfactory upper bound? It seems high, and is probably not tight. We know, however, that a solution to our problem must use at least exponential time in the worst case, because it has a potentially high *output size*.

Claim 8.2. There is a command, of size polynomial in n and d, which requires

$$(\lfloor n/(2d) \rfloor)^{\lfloor nd/2 \rfloor}$$

AMPs to describe its result.

Proof. We assume that n is even and a multiple of d, so we can avoid the "floor" signs. We use $m = n/2$ variables called $\mathtt{X}_1, \ldots, \mathtt{X}_m$ and m variables called $\mathtt{Y}_1, \ldots, \mathtt{Y}_m$. As usual let x_i denote the initial value in $\mathtt{X}_i$. For each $j \leq m$ we write a piece of code that computes

$$(x_1 \vee x_2 \vee \cdots \vee x_{m/d}) * \cdots * (x_{m-(m/d)+1} \vee \cdots \vee x_m),$$

where the disjunction operator represents a non-deterministic choice. Hence we choose d values and multiply them together. It should be clear that this produces one of $(m/d)^d$ uncomparable monomials, and the result of analysing this command will have to list all of them. We assign the product to $\mathtt{Y}_j$. Since the non-deterministic choices are made independently for each j, to describe the outcome in $\mathtt{Y}_1, \ldots, \mathtt{Y}_m$ we need $\left((m/d)^d\right)^m = (m/d)^{md}$ uncomparable AMPs. □

Thus, as long as we use an explicit AMP representation for the output of our algorithm, worst-case complexity exponential in nd is unavoidable—at least for d smaller than n. This does not necessarily rule out the applicability of the algorithm, as some algorithms susceptible to combinatorial explosion still prove usable in static analysis applications (consider the

[4]To avoid any confusion, we are obliged to point out that in the preliminary version (proceedings of FoSSaCS 2019) a wrong expression for the upper bound was given.

closure-based algorithm for Size-Change Termination). At any rate, we intend to complement the work presented in this article by further research into improving the algorithm, which we consider only a starting point, being the first complete algorithm for this problem. We now list some speculations about this future work. The reader may have noticed that for the "bad" example we used above, the description would be very compact if we were allowed to use the **max** operator and embed it in expressions (for our output bound for each $\mathtt{Y}_j$ is just a product of d "max" expressions). But this does not seem to be a panacea, and we suspect that exponential output size—and running time, for sure—may be necessary even with **max**. We leave this as an open problem; of course, the central problem is to identify the complexity class of the analysis problem. We can also ask what happens if we redefine our problem so that the output size is small—a natural example is the case of a univariate bound; or ask for an output-size dependent complexity function. In these cases the output-size excuse for exponential complexity does not apply. Another open problem that is raised by the above considerations is: what is the best bound on d in terms of $|P|$ and n alone?

9. Algorithm Extensions and Open Problems

In this section we list some ideas about how this research might be extended, specifically in terms of adding features to the subject language (while keeping the completeness of the algorithm!), and whether we believe that our current approach suffices for solving these extensions.

9.1. **"Unknown" value.** When abstracting real-world code into a restricted language, it is common to have cases in which a value has to be treated as "unknown." It might be really unknown (determined by the environment) or the result of computations that we cannot model in the restricted language (note that in our case, if we can *bound* a computation by a polynomial expression we are happy enough). It seems useful to extend our language by a special "unknown" value. Another example of its usage, following [JK09], is to analyze the growth of variables in loops for which no iteration bound is known; in this situation one cannot obtain a time bound for the program, but we can still bound computed values and may be able to draw conclusions regarding other quantities of interest, perhaps space complexity. We simulate such a loop using the "unknown" value as the loop bound. Concretely, this extension can be implemented by using a dedicated variable x_u for anything unknown, and, throughout the algorithm, replacing any expression that includes x_u immediately by x_u. This prevents an explosion of the MP set (or even failure to reach a fixed point) because of expressions including x_u.

9.2. **Resets.** [BA10] extended the decidability result from [BJK08] to a language that contains the *reset* command `X := 0`. This addition may seem trivial at first, however for the problem of deciding polynomial boundedness it caused the complexity of decision to jump from PTIME to PSPACE-complete. The increased complexity arises from the need to recognize the situation that a variable "is definitely zero," subsequent to a particular execution path. In this case, a loop that has this variable as counter will *definitely* not execute. So the algorithm has to deal with tracking these 0's around. However, our algorithm already tracks data-flow rather precisely and the algorithm is exponential anyway. We believe that our algorithm can be extended to handle resets without further raising the complexity of the solution.

9.3. **Flowchart programs.** In [BAP16] the results of [BJK08] (and, implicitly, also [BA10]) are extended from a structured language, that can be analyzed in a compositional manner (as we have done), to a "flowchart" language, where a program is presented as a *control-flow graph*, or flowchart, of arbitrary shape, together with *annotations* that convey information regarding iteration bounds. We argued there that this program form is more general and closer to the form used by several analysis tools for real-world programs. The results of [BJK08] were carried over to this language by transforming flowchart programs into programs in a well-structured language LARE that is slightly more expressive than our core language. It seems that the same development should be doable with precise polynomial-bound analysis, we only have to extend our algorithm to the language LARE. We have not investigated this in detail yet.

9.4. **Deterministic loops.** In [BAK12], Kristiansen and Ben-Amram looked at a variant of our core language where loops are deterministic: the semantics of `loop X {C}` is to perform `C` *precisely* as many times as the value of `X`. It was shown that the decision problem of polynomial boundedness, addressed in [BJK08], becomes undecidable in this case; however the undecidability proof exploits worst-case scenarios where some variables are constant while others grow. It is conjectured that the problem is decidable if one only asks about bounds that are either univariate, or multivariate but asymptotic in all variables. We propose the same conjecture with respect to the problem of tight polynomial bounds.

9.5. **Increments and Decrements.** In our opinion, the feature that most strikingly marks our language as weak is the absence of increments and decrements (and explicit constants in general; but if you have increment and reset you can generate other constants). However, anyone familiar with counter machines will realize that including them would bring our language very close to counter machines and hence to undecidability; still, without deterministic loops, this model falls just *a little* short of counter machines. We pose the decidability of the polynomial bound problem in such a language (and the investigation of the model from a computability viewpoint, in general) as a challenging open problem.

9.6. **Procedures.** Due to the compositional form of our algorithm, we suppose that extension of the language with first-order, non-recursive procedures is not hard, but have not investigated this further. A much greater challenge is to allow recursive procedures (one has to figure out what is a good way to do that since, of course, we cannot allow unbounded recursion). Another one is to include high-order functions. Avery, Kristiansen and Moyen outline in [JAM10] a possible approach for promoting analyses like [BJK08] to higher-order programs, but a definite decidability result is not obtained.

10. 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 [Weg75, Ros89, LM88]. It is also an area of active research, with tools being currently (or recently) developed including COSTA [AAG+12], AProVE [GAB+17], CiaoPP [LDK+18] , C^4B [CHS15], Loopus [SZV17]—and this is just a sample of tools for imperative programs. There is also work on functional and logic programs, term rewriting systems, recurrence relations, etc. that

we cannot attempt to survey here. In the rest of this section we point out work that is more directly related to ours, and has even inspired it.

The LOOP language is due to Meyer and Ritchie [MR67], who note that it computes only primitive recursive functions, but complexity can rise very fast, even for programs with nesting-depth 2. Subsequent work concerning similar languages [KA80, KN04, NW06, JK09] 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 [KN04] which has a decidability result for a weak "core" language). The core language we use in this paper is from Ben-Amram et al. [BJK08], 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. This research was motivated by observing that all the previous algorithms, although they implicitly relax the semantics (since they do not analyze conditionals, etc.), do not provide completeness over the core language. Justifying the necessity of these relaxations, [BAK12] 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 [CDZ14, Zul15]. Size-change programs abstract away nearly everything in the program, leaving a control-flow graph annotated with assertions about variables that decrease (or do not increase) in a transition. Thus, it does not assume structured and explicit loops, and it cannot express information about values that increase. Both works yield tight bounds on the number of transitions until termination.

Dealing with a somewhat different problem, [MOS04, HOPW18] 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 imply a solution to our problem as well.

The *joint spectral radius* problem for semigroups of matrices is related to our work as well (though we have not discovered this until recently, due to the work being done in an entirely different context). Specifically, [JPB08] gives an algorithm that can be expressed in the language of our work as follows: given a Simple Disjunctive Loop in which all polynomials are linear, the algorithm decides if the loop is polynomially bounded and if it is, returns the highest degree of τ in the tight polynomial upper bound (over all variables). A closer inspection shows that it can actually determine the degree in which τ enters the bound for every variable. Thus, it solves a certain aspect of the SDL analysis problem. Their algorithm is polynomial-time and uses an approach similar to [BJK08].

11. 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. 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.

There are a number of possible directions for further work.

- As discussed in Section 8, we have not settled the computational complexity of the problem we have solved.
- We propose to look for decidability results for richer (yet, obviously, sub-recursive) languages. Some possible language extensions include deterministic loops, variable resets (cf. [BA10]), explicit constants, and (recursive) 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 [KN04], 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}.
- Our algorithm computes bounds on the highest possible values of variables. With our restricted arithmetic we can reduce the calculation of a "countable resource" like the number of steps to bounding a variable. However, our weak language seems useless as an abstraction for more advanced resource-analysis problems, e.g., analysis of expected costs (for programs in which such an analysis is interesting). So we pose the problem of designing weak languages that adequately abstract some non-trivial cases of more advanced analyses and obtain computability for them.
- Finally, we hope to see the inclusion of our algorithm (or at least the approach) in a system that handles a real-life programming language. In particular, it would be interesting to see how our method works together with techniques that discover loop bounds, typically via ranking functions.

Acknowledgment. Amir M. Ben-Amram is grateful for the hospitality at the School of Computing, Dublin City University, where part of this work has been done. The authors also thank the referees for valuable comments.

References

[AAA+09] Elvira Albert, Diego Alonso, Puri Arenas, Samir Genaim, and German Puebla. Asymptotic resource usage bounds. In Zhenjiang Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 294–310. Springer, 2009.

[AAG+12] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

[ADFG10] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis Symposium, SAS'10*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010.

[BA10] Amir M. Ben-Amram. On decidable growth-rate properties of imperative programs. In Patrick Baillot, editor, *International Workshop on Developments in Implicit Computational complExity (DICE 2010)*, volume 23 of *EPTCS*, pages 1–14, 2010.

[BAG14] Amir M. Ben-Amram and Samir Genaim. Ranking functions for linear-constraint loops. *Journal of the ACM*, 61(4):26:1–26:55, jul 2014.

[BAK12] Amir M. Ben-Amram and Lars Kristiansen. On the edge of decidability in complexity analysis of loop programs. *International Journal on the Foundations of Computer Science*, 23(7):1451–1464, 2012.

[BAP16] Amir M. Ben-Amram and Aviad Pineles. Flowchart programs, regular expressions, and decidability of polynomial growth-rate. In Geoff Hamilton, Alexei Lisitsa, and Andrei P. Nemytykh, editors, Proceedings of the Fourth International Workshop on *Verification and Program Transformation (VPT),* Eindhoven, The Netherlands, volume 216 of *Electronic Proceedings in Theoretical Computer Science*, pages 24–49. Open Publishing Association, 2016.

[BEF+16] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4):13:1–13:50, August 2016.

[BG17] Amir M. Ben-Amram and Samir Genaim. On multiphase-linear ranking functions. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification, CAV'17*, volume 10427 of *Lecture Notes in Computer Science*, pages 601–620. Springer, 2017.

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 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.

[BJK08] Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *LNCS*, pages 67–76. Springer, 2008.

[Boj09] Mikołaj Bojańczyk. Factorization forests. In Volker Diekert and Dirk Nowotka, editors, *Developments in Language Theory, 13th International Conference, DLT 2009, Stuttgart, Germany, June 30 - July 3, 2009. Proceedings*, volume 5583 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009.

[CDZ14] Thomas Colcombet, Laure Daviaud, and Florian Zuleger. Size-change abstraction and max-plus automata. In Erzsébet Csuhaj-Varjú, Martin Dietzfelbinger, and Zoltán Ésik, editors, *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part I*, volume 8634 of *LNCS*, pages 208–219. Springer, 2014.

[CHS15] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. In *Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2015.

[GAB+17] J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with aprove. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

[HOPW18] Ehud Hrushovski, Joël Ouaknine, Amaury Pouly, and James Worrell. Polynomial invariants for affine programs. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 530–539, New York, NY, USA, 2018. ACM.

[JAM10] Lars Kristiansen James Avery and Jean-Yves Moyen. Static complexity analysis of higher order programs. In *International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA '09, Eindhoven, the Netherlands, Proceedings*, volume 6324 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2010.

[JK09] Neil D. Jones and Lars Kristiansen. A flow calculus of mwp-bounds for complexity analysis. *ACM Trans. Computational Logic*, 10(4):1–41, 2009.

[JPB08] Raphaël M. Jungers, Vladimir Protasov, and Vincent D. Blondel. Efficient algorithms for deciding the type of growth of products of integer matrices. *Linear Algebra and its Applications*, 428:2296–2311, 2008.

[KA80] Takumi Kasai and Akeo Adachi. A characterization of time complexity by simple loop programs. *Journal of Computer and System Sciences*, 20(1):1–17, 1980.

[K18] Zachary Kincaid. Numerical invariants via abstract machines. In A. Podelski, editor, *Static Analysis—25th International Symposium, SAS 2018, Proceedings*, volume 11002 of *LNCS*, pages 24–42. Springer, 2018.

[KN04] Lars Kristiansen and Karl-Heinz Niggl. On the computational complexity of imperative programming languages. *Theor. Comp. Sci.*, 318(1-2):139–161, 2004.

[LDK+18] Pedro López-García, Luthfi Darmawan, Maximiliano Klemen, Umer Liqat, Francisco Bueno, and Manuel V. Hermenegildo. Interval-based resource usage verification by translation into horn clauses and an application to energy consumption. *TPLP*, 18(2):167–223, 2018.

[LM88] Daniel Le Métayer. Ace: an automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10(2):248–266, 1988.

[LJB01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proc. 28th ACM Symp. on Principles of Programming Languages*, 81–92. ACM press, 2001.

[May03] Richard Mayr. Undecidable problems in unreliable computations. *Theor. Comput. Sci*, 297(1-3):337–354, 2003.

[MP18] Kenneth L. McMillan and Oded Padon. Deductive verification in decidable fragments with ivy. In A. Podelski, editor, *Static Analysis—25th International Symposium, SAS 2018, Proceedings*, volume 11002 of *LNCS*, pages 43–55. Springer, 2018.

[MOS04] Markus Müller-Olm and Helmut Seidl. Computing polynomial program invariants. *Information Processing Letters*, 91(5):233–244, 2004.

[MR67] Albert R. Meyer and Dennis M. Ritchie. The complexity of loop programs. In *Proc. 22nd ACM National Conference*, pages 465–469, Washington, DC, 1967.

[NW06] Karl-Heinz Niggl and Henning Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM J. Comput*, 35(5):1122–1147, 2006.

[PR04] Andreas Podelski and Andrey Rybalchenko. A complete method for synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI 2003: Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.

[Ros89] M. Rosendahl. Automatic complexity analysis. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture FPCA'89*, pages 144–156. ACM, 1989.

[Sim90] Imre Simon. Factorization forests of finite height. *Theoretical Computer Science*, 72(1):65–94, 1990.

[SZV17] Moritz Sinn, Florian Zuleger, and Helmut Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 59(1):3–45, 2017.

[Weg75] Ben Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[Zul15] Florian Zuleger. 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*, volume 9139 of *Lecture Notes in Computer Science*, pages 426–442. Springer, 2015.