TIGHT POLYNOMIAL BOUNDS FOR LOOP PROGRAMS IN POLYNOMIAL SPACE

AMIR M. BEN-AMRAM[1] AND GEOFF HAMILTON[2]

[1]Department of Computer Science, Holon Institute of Technology, Israel
e-mail address: benamram.amir@gmail.com

[2]School of Computing, Dublin City University, Ireland
e-mail address: geoffrey.hamilton@dcu.ie

ABSTRACT. We consider the following problem: given a program, find tight asymptotic bounds on the values of some variables at the end of the computation (or at any given program point) in terms of its input values. We focus on the case of polynomially-bounded variables, and on a weak programming language for which we have recently shown that tight bounds for polynomially-bounded variables are computable. These bounds are sets of multivariate polynomials. While their computability has been settled, the complexity of this program-analysis problem remained open. In this paper, we show the problem to be PSPACE-complete. The main contribution is a new, space-efficient analysis algorithm. This algorithm is obtained in a few steps. First, we develop an algorithm for univariate bounds, a sub-problem which is already PSPACE-hard. Then, a decision procedure for multivariate bounds is achieved by reducing this problem to the univariate case; this reduction is orthogonal to the solution of the univariate problem and uses observations on the geometry of a set of vectors that represent multivariate bounds. Finally, we transform the univariate-bound algorithm to produce multivariate bounds.

1. INTRODUCTION

A static analysis algorithm takes, as input, program code and answers a question about its possible behaviours. A standard example is to find the set of values that a variable can assume (at a given program point). The algorithm seeks a set of a particular kind (for example, intervals $[a, b]$) and is *complete* if it can always find the tightest such result (e.g., the smallest interval that contains all reachable values). A more complex analysis may establish a *relation* among the values of several variables. While such results cannot be fully computable for programs in general, there are algorithms that guarantee a sound and complete solution for a class of *weak* programs, defined by removing or weakening some features of full programming languages. For example, [14] shows an algorithm that, for a particular weak language, finds any *algebraic relation* that exists among variable values. Such an algorithm can establish facts like: two variables x and y always satisfy, at the conclusion of the program, the equation $y = x^2$. By retaining a copy of the input, this allows

---

Key words and phrases: asymptotically-tight, multivariate, worst-case, polynomial bounds.

us to answer a question like: does the program square its input, or in general, can an output value be expressed as a polynomial in the input values and what is this polynomial?

In this paper we are interested in final values that are not necessarily expressible by polynomials, but are *polynomially bounded*, and we are seeking a *tight asymptotic upper bound*. A prototypical context where such a question is asked is *complexity analysis*: the quantity to be analyzed may represent a counter of steps performed by the program and we wish to establish a result such as "the program's worst case is $\Theta(n^2)$ steps" (where n is the size of an input parameter).

In 2008, Ben-Amram et al. [4] considered a simple imperative programming language with explicitly-bounded loops, and proved that it is decidable whether computed values are polynomially bounded. This work left open the problem of finding an explicit *tight* bound for such values. Recently, in [3], we solved this *bound analysis problem* (the class of programs remained as in [4]). However, the solution was very inefficient (doubly-exponential space) and did not settle the complexity of this problem. In this paper we answer this question and prove that tight polynomial bounds (for the same language) can be computed in polynomial space. Our approach is to first develop an analysis for *univariate* bounds only. This simplifies the task and makes the problem of solving it efficiently more tractable. However the full problem is to compute multivariate bounds (such as $O(xy + yz^2)$, equivalently $O(\max(xy, yz^2))$). In Sections 11 and 12 we show how a solution to the multivariate problem can be evolved from our solution to the univariate problem.

We now discuss, somewhat informally, some key features of the approach. We have already noted that bounds may be multivariate, i.e., depend on multiple variables. In [3] we argued that it is also necessary to compute *simultaneous bounds* on multiple variables; moreover, we find sets of simultaneous bounds. The capability of computing sets of simultaneous bounds is important even if we are ultimately interested in bounding the value of a single variable. Consider, for example, the following piece of code, where `choose` represents non-deterministic choice:

```
choose Y := X*X or { Y := X;  X := X*X }
```

For it we have two incomparable simultaneous bounds on X, Y: $\langle x, x^2 \rangle$ and $\langle x^2, x \rangle$ (where x stands for the initial value of X). This allows us to deduce that if the above command is followed by:

```
X := X*Y
```

The final value of X is $O(x^3)$. If, in the previous phase, we had only computed separate worst-case bounds, they would have been x^2 for each of the variables, and led to a loose bound of x^4.

A remarkable corollary of the analysis in [3] is that for our class of programs, when there is a polynomial (simultaneous) upper bound on a set of variables, it can always be matched by lower bounds; in other words, a *tight* worst-case bound can always be found. This means that the problem of computing asymptotic upper bounds and that of computing worst-case lower bounds are the same problem. For technical reasons, we formulate our results in terms of worst-case lower bounds. More precisely, we seek functions such that, for almost all inputs, for some execution, the output reaches (or surpasses) this function (up to a constant factor). We call such a function *attainable* (for a full, precise definition see Section 3). Thus, the program property that our analyzer computes is *the set of all attainable polynomial bounds for all polynomially-bounded program variables*. These bounds are expressed by *abstract polynomials*, which are ordinary polynomials stripped of their

coefficients. This agrees, of course, with the intention of providing asymptotic ("big-Oh" or "big-Omega") bounds, where coefficients do not matter. It is not hard to see that due to this abstraction, the set of attainable bounds (for polynomially-bounded variables) becomes finite.

Even if finite, the size of this set, and the expression of its members, may be a concern; indeed, in [3] we argued that the size as well as the number of maximal attainable bounds may be exponential in the program size. Here, we overcome this problem by showing that it is always possible to do with expressions of polynomial bit-size, even if the *size of the set of attainable bounds* may be exponential. This is achieved by redefining the task of our analyzer to either *verify* a given bound, or *generate* bounds—one at a time. We also design the algorithm in a new way for achieving the desired complexity. In fact, at first we could only see how to achieve such a result for univariate bounds. We later saw that it is possible to extend the solution to multivariate bounds, by exploring a reduction of the multivariate-bound problem to the univariate-bound problem. We present our algorithms in the same order, so that the justification for each step may be understood.

2. Preliminaries

In this section we make some technical definitions and present the programming language that we study.

2.1. **Notations.** 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 n-tuples is denoted by S^n. Natural liftings of operators to collections of objects are used, e.g., if t is a tuple of integers then $t + 1$ is formed by adding 1 to each component. The intention should usually be clear from context. 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).

A binary relation on a set S is a subset $R \subseteq S \times S$ and we use the customary notation xRy instead of $(x, y) \in R$. Functions are a subclass of relations, and we liberally switch between functional notation $y = f(x)$ and relational notation.

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

When we write "polynomials," we are referring, unless stated otherwise, to multivariate polynomials in $x_1, \ldots, x_n$ that have non-negative integer coefficients. We presume that an implementation of the algorithm represents a polynomial as a set of monomials (which may be empty—for the zero polynomial), and that numbers are stored in binary, but otherwise we do not concern ourselves with low-level implementation details. According to context, a polynomial may refer to the function it expresses rather than the syntactic entity.

2.2. **The core language.** Following previous publications [4, 2, 3], we study the bound analysis problem for programs in a weak programming language, a "core language." This is an imperative language, including bounded loops, non-deterministic branches and restricted arithmetic expressions; the syntax is shown in Figure 1.

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

Figure 1: Syntax of the core language (from Ben-Amram, Kristiansen and Jones, 2008).

A program has a finite set of integer variables, $\mathtt{X}_1, \ldots, \mathtt{X}_n$; by convention, n will denote the number of variables throughout this article. In code fragments we may use other identifiers ($\mathtt{X}$, $\mathtt{Y}$...) for convenience. Variables store non-negative integers[1].

The core language is inherently non-deterministic, a property motivated by the origin of its language as a *weakening* of a fuller imperative language. This type of weakening is related to the approach of *conservative abstraction* often employed in program analysis: the $\mathtt{choose}$ command represents a non-deterministic choice, and replaces deterministic conditionals found in real programs. This reflects the idea that the analysis conservatively abstracts any conditional command by simply ignoring the condition. Weakening the language is also necessary if we wish our analysis problem to be solvable; otherwise we would clearly be blocked by undecidability. The command $\mathtt{loop}\ \mathtt{X}\ \{\mathtt{C}\}$ repeats $\mathtt{C}$ a (non-deterministic) number of times bounded by the value of the variable $\mathtt{X}$ upon entry to the loop. 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. For convenience, we stipulate that the loop-bound variable $\mathtt{X}$ may not be assigned a new value in the loop body (this is easily seen to be non-restrictive, since we could add a dedicated variable for the loop bound).

An *unstructured* language in which loops are implemented by conditional branching and not by structured commands may also be amenable to transformation into a form that our algorithms can handle, using some front-end analysis; we will not elaborate on this but refer the reader to [6].

2.3. **Semantics, more formally.** We define a *program state* to be an n-tuple of non-negative integers, $\mathbf{x} = \langle x_1, \ldots, x_n \rangle$, representing the values of the variables $\mathtt{X}_1, \ldots, \mathtt{X}_n$. Note the different fonts used for program variables, vectors and scalars.

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 $\mathbf{x}[\![\mathtt{C}]\!]\mathbf{x}'$, vector $\mathbf{x}$ (respectively $\mathbf{x}'$) is the state before (after) the execution of $\mathtt{C}$.

The semantics of $\mathtt{skip}$ is the identity. The semantics of an assignment $\mathtt{X}_i\!:\!=\!\mathtt{E}$ associates to each store $\mathbf{x}$ a new store $\mathbf{x}'$ obtained by replacing the component x_i by the value of the expression $\mathtt{E}$ when evaluated over state $\mathbf{x}$, namely with x_j, $x_j + x_k$ or $x_j x_k$, corresponding to expressions $\mathtt{X}_j$, $\mathtt{X}_j + \mathtt{X}_k$ and $\mathtt{X}_j * \mathtt{X}_k$. Composite commands are described by the equations:

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

---

[1]An alternative definition uses signed integers, see [3, remark on p. 7]; this makes little difference because of the restricted arithmetics in the language.

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

2.4. **Multi-polynomials.** In this section we define some concepts and notations used for expressing the analysis results (including intermediate results, internal to the algorithm).

Definition 2.1. A *multi-polynomial* $\mathbf{p} = \langle \mathbf{p}[1], \ldots, \mathbf{p}[n] \rangle$ is an n-tuple of polynomials (with non-negative coefficients); it represents a mapping of $\mathbb{N}^n$ to $\mathbb{N}^n$. We denote by $\mathtt{MPol}$ the set of multi-polynomials, where the number of variables n is fixed by context.

A multi-polynomial $\mathbf{p}$ expresses simultaneous bounds on the variables $\mathtt{X}_1 \ldots \mathtt{X}_n$ in terms of their respective initial values $x_1 \ldots x_n$, where each component $\mathbf{p}[i]$ represents the bound on variable $\mathtt{X}_i$. For example, the command $\mathtt{X}_4 \mathtt{:=} \mathtt{X}_2 + \mathtt{X}_3$ is represented by the multi-polynomial $\mathbf{p} = \langle x_1, x_2, x_3, x_2 + x_3 \rangle$.

Composition of multi-polynomials, $\mathbf{p} \circ \mathbf{q}$, 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 $i = 1, \ldots, n$).

Definition 2.2 (Univariate MP). A univariate MP is defined as an n-tuple $\mathbf{p}$ where each component $\mathbf{p}[i]$ is a polynomial in the single variable x.

Definition 2.3. $\mathtt{APol}$, the set of *abstract polynomials*, consists of formal sums of monomials over $x_1, \ldots, x_n$, where the set of coefficients is reduced by making all non-zero coefficients equivalent to 1. In other words, these are polynomials where the coefficients vary over the Boolean ring $\{0, 1\}$.

As an example, the sum of the *abstract* polynomial $x + y$ and the *abstract* polynomial $y + z$ is $x + y + z$, not $x + 2y + z$, as in the ring of coefficients, $1 + 1 = 1$ (also $1 \cdot 1 = 1$). The intention is that an abstract polynomial represents all the polynomials which are formed by varying its non-zero coefficients, thus $x + y$ represents $ax + by$ for all $a, b > 0$. We can also view an abstract polynomial as a set of monomials (whose coefficients are, implicitly, 1). We use AMP as an abbreviation for abstract multi-polynomial. We use $\alpha(\mathbf{p})$ for the abstraction of $\mathbf{p}$ (obtained by modifying all non-zero coefficients to 1). Conversely, if $\mathbf{p}$ is abstract we can "coerce" it to a concrete polynomial by considering the 1-valued coefficients as being literally 1; this is how one should read any statement that uses an abstract polynomial where a concrete function is needed. Abstract polynomials are used to ensure that the analysis is finite, since there is a finite number of different abstract polynomials of any given degree. They correspond to the use of big-O (or Θ) notation in typical algorithm analysis.

The set of abstract multi-polynomials is denoted by $\mathtt{AMPol}$; this has a composition operation $\mathbf{p} \bullet \mathbf{q}$, which relates to the standard composition $\mathbf{p} \circ \mathbf{q}$ by $\alpha(\mathbf{p}) \bullet \alpha(\mathbf{q}) = \alpha(\mathbf{p} \circ \mathbf{q})$; the different operator symbol ("$\bullet$" versus "$\circ$") helps in disambiguating the meaning of an expression (referring to abstract polynomials versus concrete ones).

In the special case of *abstract univariate* polynomials, it is useful to restrict them to be monomials x^d; this can be done since in "big Θ" bounds, a univariate polynomial reduces to its leading monomial. We allow the monomial x^0, as well as 0, and thus we obtain a semiring isomorphic to $\langle \mathbb{N} \cup \{-\infty\}, \max, + \rangle$.

Definition 2.4. A *univariate state* is an n-tuple of univariate monomials. A univariate state is called positive if it does not include a zero.

Based on this definition, a univariate state can be written as $\langle x^{d_1}, \ldots, x^{d_n} \rangle$ and internally represented by either a vector of integers $\langle d_1, \ldots, d_n \rangle$ (for a non-zero monomial) or the constant 0.

3. FORMAL STATEMENT OF THE PROBLEM AND OUR RESULTS

Our goal is to prove polynomial-space complexity bounds for the tight-bound problem. In [3], our algorithm was encumbered by creating and calculating with exponentially-large objects. In particular, we argued that there are commands for which a set of multi-polynomials that provides tight bounds must include exponentially many elements. In order to reduce the size of objects *internal* to the algorithm, we have to rewrite the algorithm, but first we circumvent the issue of exponential *output size* by adjusting the specification of the algorithm. This can be done in two ways. First, as one commonly does in Complexity Theory, one can reduce the problem to a decision problem:

Problem 1. *Multivariate decision problem*: Given a core-language program P and a monomial $m(x_1, \ldots, x_n) = x_1^{d_1} \ldots x_n^{d_n}$, report whether m constitutes a "big-Omega" lower bound on the highest final value of $\mathtt{X}_n$ for initial state $\langle x_1, \ldots, x_n \rangle$.

There is, of course, no loss of generality in fixing the queried variable to be the highest-numbered; neither do we lose generality by restricting the bounds to monomials (a polynomial is an asymptotic lower bound if and only if all its monomials are). Note that we have chosen to focus on worst-case lower bounds, unlike most publications in this area, which focus on upper bounds for worst-case results. Our reader should keep in mind, though, that [3] shows that *for our core language, whenever a variable is polynomially bounded, there is a set of polynomials which provides tight bounds for all executions.* This means that if we form the function "$\mathtt{max}$ of (all the lower bounds)," we have a tight worst-case upper bound. Thus solving for lower bounds also solves the upper bound problem. Furthermore, we note that if we are looking at a polynomially-bounded variable, the set of abstract monomials (or even polynomials) that *lower-bound the final value* is finite. This is convenient (for instance, the expression "$\mathtt{max}$ of (all the lower bounds)" can be explicitly written out, if desired). We refer to a polynomial in this set as *attainable* (since the program can compute values that reach, or surpass, this polynomial, asymptotically).

We call a function *attainable* (by a program) if its values can be reached or surpassed.

Definition 3.1 (Attainable). Consider a command $\mathtt{C}$, and a function $f : \mathbb{N}^n \to \mathbb{N}^n$ we say that f is *attainable* by $\mathtt{C}$ if there are constants $d > 0$ and $\mathbf{x}_0 \in \mathbb{N}^n$ such that for all $\mathbf{x} \geq \mathbf{x}_0$ there is a $\mathbf{y}$ such that

$$\mathbf{x}[\![\mathtt{C}]\!]\mathbf{y} \text{ and } \mathbf{y} \geq df(\mathbf{x}).$$

Given a function $f' : \mathbb{N}^n \to \mathbb{N}$, extend the definition to f' by applying the above condition to $f(\mathbf{x}) = \langle 0, \ldots, 0, f'(\mathbf{x}) \rangle$, in other words we use f' as a lower bound on a single variable ($\mathtt{X}_n$).

We remark that in the above inequation ,d ranges over real numbers, and may be smaller than one.

Thus, for a program that starts with data x_1 and x_2 in its two variables, and non-deterministically chooses to exchange them, the bounds $\langle x_1, x_2 \rangle$ and $\langle x_2, x_1 \rangle$ are both attainable, while $\langle x_1, x_1 \rangle$ is not.

In the current work, we are attempting to find tight bounds only for variables that are polynomially bounded in terms of the input (the initial contents of the variables). The

problem of identifying which variables are polynomially bounded is completely solved in [4], by a polynomial-time algorithm. We will tacitly rely on that algorithm. More precisely, we assume that the algorithm is invoked as a preprocessing step for the analysis of each loop in the program, and allows for excluding variables that may grow super-polynomially in the given loop. This reduces the problem of handling *any* loop in our language to that of handling loops in which all variables are polynomially bounded; the key observation here is that, due to the restricted arithmetics in our language, which only include addition and product, any value derived from a super-polynomial value is also super-polynomial. Hence, *ignoring* variables which are not polynomially bounded does not lose any information necessary for analyzing the polynomially bounded ones.

Note that in the formulation as a decision problem, the user queries about a given polynomial; we think that it is a natural use-case, but one can also ask the analyzer to furnish a bound:

Problem 2. *Multivariate bound generation problem*: Given a core-language program P, and an integer d, find a polynomial $p(x_1, \ldots, x_n)$ with the degree in every variable bounded by d, such that p constitutes a "big-Omega" lower bound on the highest final value of $\mathtt{X}_n$ as a function of $x_1, \ldots, x_n$.

The function of the input parameter d is to (possibly) reduce the complexity of the search—not looking further than what the user is wishing to find. There is, of course, no loss of generality in fixing the queried variable to be the highest-numbered.

For this problem we claim a solution whose space complexity is polynomial in the input size, namely the program size plus the bit-size of d. Recall that if we have a polynomial-space non-deterministic algorithm, we can also determinize it in polynomial space (this is the essence of the famous equality NPSPACE=PSPACE, which follows from Savitch's Theorem [15]). This means that it is possible to transform the non-deterministic solution to Problem 2 (as long as it is complete) into a deterministic generator that writes out one monomial at a time. Moreover, by exhaustive search it is possible to generate all the monomials needed to bound all program executions, even when their number is exponential.

It should thus be clear that in a similar manner we can also search for a bound of highest degree d_{max}. The complexity of our solution to the multivariate-bound problem (Sections 11 and 12) is *output-sensitive* in the sense that the bound is polynomial in the program size and the bit-size of d_{max}.

A main idea in this paper is to reduce Problem 1 to a problem concerning univariate bounds. In this problem, we assume that we are given a *univariate initial state* in terms of a single input variable x. The initial state assigns to every variable $\mathtt{X}_i$ an initial symbolic value of form x^{d_i} for some integer $d_i \geq 0$, e.g., $\mathbf{i}(x) = \langle x, x^3, x^2 \rangle$. The case $d_i = 0$ intuitively represents a variable whose initial value should be treated as an unknown constant (not dependent on x, but possibly large). The univariate form seems to be rather common in textbook examples of algorithm analysis (where the single parameter in terms of which we express the bounds is usually called n), and the facility of stating initial polynomial values for multiple variables may be useful to express questions like: *An algorithm processes a graph. Its running time depends on the number of vertices n and the number of edges m. Bound its worst-case time,* in terms of n, *given that $m \leq n^2$.* In this case we would use an initial state like $\langle n, n^2 \rangle$. Our goal is now to find an attainable univariate state, providing a lower bound on the worst-case results of a computation from the given initial state. Here is the formal definition; note that we are providing simultaneous bounds for all variables.

Definition 3.2. Given a command C, a function $g : \mathbb{N} \to \mathbb{N}^n$ and an initial univariate state **i**, we say that g is attainable (by C, given **i**) if there are constants $d > 0$, x_0 such that for all $x \geq x_0$ there is a **y** such that

$$\mathbf{i}(x)[\![\mathtt{C}]\!]\mathbf{y} \text{ and } \mathbf{y} \geq dg(x).$$

As in Definition 3.1, a function with codomain $\mathbb{N}$ may be used an interpreted as a lower bound for just $\mathtt{X}_n$.

We thus focus on the following problem.

Problem 3. *Univariate decision problem*: Given core-language program P, a positive[2] initial state **i**, the index of a chosen variable $\mathtt{X}_j$ and an integer d, decide whether there is an attainable univariate state whose degree in the jth component is at least d.

We also solve the corresponding bound generation problem—even for simultaneous bounds, namely

Problem 4. *Univariate bound generation problem*: Given a core-language program P, a positive initial state **i**, a variable index j and an integer d, non-deterministically find an attainable univariate state whose degree in the jth component is at least d.

We have included the parameter d in this problem formulation as well, since we will be able to bound the complexity of our algorithm in terms of d (instead of d_{max}).

Here is a summary of the results we are going to prove regarding the above problems:

- In Section 7, we give a polynomial space algorithm to solve Problem 4.
- The algorithm in Section 7, along with the complexity analysis in Section 8, prove that Problem 3 and Problem 4 can be solved in polynomial space. Note that this means that the space complexity is polynomial in the input size, consisting of the size of program P and the bit-size of the numbers in **i** and d.
- In Section 10 we show that this result is essentially optimal since the problem is PSPACE-hard. This clearly means that the more involved problems (for multivariate bounds) are also PSPACE-hard.
- In Section 11 we show a reduction of the multivariate-bound decision problem (Problem 1) to the univariate problem (Problem 3), and obtain in this way a solution with space complexity polynomial in terms of the size of the program and the maximum degree.
- In Section 12 we solve the multivariate-bound generation problem (Problem 2).

We'd like to mention that solving the above problems also solves some variants that can be reduced to tight-bound computation:

- We can find tight bounds on the number of *visits* to a given set of program locations (counting all locations gives the program's time complexity, in the ordinary unit-cost model). The reduction is to instrument the program with a counter that counts the visits to the locations of interest [3, Section 2].
- Sometimes we want to verify a bound for the values of a variable *throughout* the program, not just at its end. This can also be easily reduced to our problem by instrumentation.
- Similarly, one can find a tight bound on the highest value assumed by a variable at a given program location.

---

[2]A zero in the initial state can be accommodated in an extension discussed in Section 13, where an assignment $\mathtt{X} := \mathtt{0}$ is also allowed.

4. Properties of the Core Language

We next present some important properties of the core language that follow from [3] and are required for our result. First, let us cite the main result of [3]:

Theorem 4.1 ([3])**.** *There is an algorithm which, for a command* C*, over variables* X_1 *through* X_n*, outputs a set* $\mathcal{B}$ *of multi-polynomials, such that the following hold, where* PB *is the set of indices i of variables* X_i *which are polynomially bounded under* $[\![C]\!]$*.*

(1) *(Bounding) There is a constant c such that*

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

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

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

We rely on this result because our new algorithm will be proven correct by showing that it matches the bounds of the previous algorithm. But the implied *existence result* is crucial in itself: the fact that a set $\mathcal{B}$ as above exists, which provides both upper bounds (clause "Bounding") and matching worst-case lower bounds (clause "Tightness"). This property is tied to the limitations of our language: in a Turing-complete programming language this clearly does not hold—some times no polynomial tight bound exists, as for a program that inputs x and computes $\log x$, $\sqrt{x}$, etc. In other cases the bound is only "tight" in the sense that it is matched by a lower bound for infinitely many inputs, but not for all inputs (this is sometimes expressed by the notation Ω_∞); e.g., a program that squares a number only if it is odd. These cases cannot happen with our language.

Since a polynomial is a sum of a constant number of monomials, by simple arithmetics we obtain:

Corollary 4.2. *For a command* C*, over variables* X_1 *through* X_n*, assume that* X_n *is polynomially bounded in terms of the variables' initial values. Then there is a set* $\mathcal{S}$ *of monomials such that* $\max \mathcal{S}$ *is a tight asymptotic bound on the highest value obtained by* X_n *at the conclusion of* C*; or in more detail,*

(1) *(Bounding) There is a constant c such that*

$$\forall \mathbf{x}, \mathbf{y} \;.\; \mathbf{x}[\![C]\!]\mathbf{y} \implies \exists m \in \mathcal{S} \,.\, y_n \leq cm(\mathbf{x})$$

(2) *(Tightness) There are constants* d*,* $\mathbf{x}_0$ *such that for every* $m \in \mathcal{S}$*, for all* $\mathbf{x} \geq \mathbf{x}_0$ *there is a* $\mathbf{y}$ *with*

$$\mathbf{x}[\![C]\!]\mathbf{y} \; and \; y_n \geq dm(\mathbf{x}) \,.$$

We refer to monomials satisfying the Tightness condition as attainable; they constitute valid answers to Problem 2 (subject to the degree bound); and the fact that they also provide an upper bound is, again, an important property that depends on the limitations of the programming language. This will be used in Sections 11 and 12 where we check or compute multivariate bounds in the form of monomials.

5. Algorithmic ideas

In this section we give a brief overview and motivation of key elements in our solution. In particular, we relate these elements to prior work. We discuss symbolic evaluation, the motivation for working with univariate bounds, and data-flow matrices. This is followed by an overview of the way we prove correctness of this algorithm, and the way we finally solve the multivariate-bound problem.

5.1. **Symbolic evaluation.** This is an old concept. We interpret the program, with states mapping variables not to integers, but to polynomials in $x_1, \ldots, x_n$, where x_i represents the initial value of the corresponding variable $\mathtt{X}_i$. Due to the restricted arithmetics in our language, we can carry out symbolic evaluation precisely, staying within the domain of multivariate polynomials. Thus a program state is a MP. However, when there are different program paths that reach the same point, we can get different polynomials, and therefore we construct *sets* of polynomials. The abstraction to *abstract polynomials* may reduce the size of this set. Here is an example of a piece of program code, along with a symbolic state, as a set of AMPs, before and after each command:

$$\{\langle x_1,\ x_2,\ x_3,\ x_4\rangle\}$$
$$\mathtt{X}_2\mathtt{:=}\mathtt{X}_2 + \mathtt{X}_4;$$
$$\{\langle x_1,\ x_2 + x_4,\ x_3,\ x_4\rangle\}$$
$$\mathtt{X}_4\mathtt{:=}\mathtt{X}_3;$$
$$\{\langle x_1,\ x_2 + x_4,\ x_3,\ x_3\rangle\}$$
$$\mathtt{choose}\ \{\ \mathtt{X}_1\mathtt{:=}\mathtt{X}_1 + \mathtt{X}_3\}\ \mathtt{or}\ \{\ \mathtt{X}_2\mathtt{:=}\mathtt{X}_2 + \mathtt{X}_3\}$$
$$\{\langle x_1 + x_3,\ x_2 + x_4,\ x_3,\ x_3\rangle, \langle x_1,\ x_2 + x_3 + x_4,\ x_3,\ x_3\rangle\}$$

This is all quite straight-forward until one comes to a loop. Since we are analysing the program statically, we have to take into account any number of iterations. Consequently, had we worked with concrete polynomials, they would grow indefinitely.

Example 5.1. Consider this loop:
```
loop X₂ {
    X₁ := X₁ + X₂;
}
```
Evaluating with *concrete* polynomials, the value for $\mathtt{X}_1$ would take the values x_1, $x_1 + x_2$, $x_1 + 2x_2$, $x_1 + 3x_2$, etc. Working with abstract polynomials, the coefficients are ignored, and we have a finite set, which is reached after a finite number of iterations. In our example, the value for $\mathtt{X}_1$ does not rise beyond $x_1 + x_2$. This is good, but unfortunately, we also lose completeness: $x_1 + x_2$ is not a valid asymptotic upper bound on the final value of $\mathtt{X}_1$; a correct upper bound is $x_1 + x_2{}^2$, taking into account the bound x_2 on the number of loop iterations. The passage from $x_1 + x_2$ to $x_1 + x_2{}^2$ is based on correctly *generalizing* the behaviour that we see in one iteration to the long-term behaviour, and deducing that the *increment* $+x_2$ may be applied to this variable at most x_2 times. Making such generalizations precisely is the main challenge addressed in the algorithm of [3].

Our algorithm in the cited work gave a precise set of bounds but suffered from efficiency problems. This is inherent in the approach of generating symbolic polynomials. In fact, it

is easy to write pieces of code that generate exponentially big sets of expressions—try the following:

```
choose  { X₃:= X₁ } or { X₃:= X₂ } ;
choose  { X₄:= X₁ } or { X₄:= X₂ } ;
.  .  .
choose  { Xₙ:= X₁ } or { Xₙ:= X₂ }
```

5.2. **Univariate bounds.** Suppose that we decide to compute only univariate bounds. We have a single input parameter x and we want to express everything in terms of x. Then, if we also ignore coefficients, the only functions we need are x, x^2, x^3,... and the efficiency issue, at least the problem of big expressions, is resolved. An abstract state is just an n-tuple of degrees and takes polynomial space. However, important information is lost. If we start with all variables set to x, and the loop body sets $\mathtt{X_1:=X_1+X_2}$, we get the symbolic value $2x$ in $\mathtt{X_1}$ after the first iteration (or just x if we ignore coefficients). How do we know that this becomes $x + tx$ after t iterations? We cannot deduce it from the final expression. We are compelled to also keep track of how the value was computed—so we know that there is an increment that can accumulate upon iteration. Note that the fact that the value after one iteration is different from the value before does not necessarily mean that the difference will accumulate—consider a loop body that simply sets $\mathtt{X_1:=X_2}$, where the initial value of $\mathtt{X_2}$ is bigger than that of $\mathtt{X_1}$.

5.3. **Data-flow matrices.** The discussion in the last paragraph hints that it may be useful to record data-flow: does x'_1 (value of $\mathtt{X_1}$ after an iteration) depend on x_1 (its initial prior to the iteration)? On x_2? On both? Such information may be represented by a *bipartite graph* with arcs from $x_1, x_2, \ldots, x_n$ leading to $x'_1, x'_2, \ldots, x'_n$ and showing how values propagate, or a *data-flow matrix*, which is the same information in matrix form. This is a very concise representation which does not record the actual expressions computed, only the Boolean property that some x'_j depends, or does not depend, on x_i. Some of the previous results in complexity analysis [17, 18] showed that the existence of polynomial bounds on computed values may sometimes be deduced by examining the dependence graph. Later works [23, 16] showed that by enriching the information in data-flow matrices (allowing for a finite number of "dependence types") one has sufficient information to soundly conclude that a result is polynomially bounded in a larger set of programs. As a basic example, the technique allows us to distinguish a loop body that sets $x'_1 = 2x_1$ (doubling the value of $\mathtt{X_1}$, leading to exponential growth upon iteration) from one that sets $x'_1 = x_1 + x_2$ (which entails polynomial growth). Ben-Amram et al. [4] derived a complete decision procedure for polynomial growth-rates in the Core Language, still by tracking data-flow properties (with an abstraction similar to data-flow matrices) and without explicitly computing any bounds. In the current work, we use data-flow matrices in addition to a symbolic state, to aid the analysis of loops, specifically the precise generalization of iterations that seem to increase certain variables. Suprisingly, it turns out that all we need is Boolean matrices which record only linear data-dependence.

5.4. **Correctness.** We have motivated our work by wishing to solve the efficiency issue with the algorithm of [3]. However that algorithm—the so-called *Closure Algorithm*—remains the foundation of our current contribution. We both motivate the polynomial-space algorithm, and formally prove its correctness, based on the idea that it should match the results of the Closure Algorithm, in the following sense: the Closure Algorithm computes AMPs; for instance, for the loop in Example 5.1, it outputs the AMP $\mathbf{p} = \langle x_1 + x_2{}^2,\ x_2 \rangle$. Our algorithm works with an initial univariate state, e.g., $\mathbf{x} = \langle x^3, x^2 \rangle$. An abstract multi-polynomial can be applied to a univariate state, e.g., $\mathbf{p} \bullet \mathbf{x} = \langle x^4, x^2 \rangle$ (as already mentioned, we reduce abstract univariate polynomials to their leading monomial). The correctness assertion which we will prove for the polynomial-space algorithm is that it can match the result $\mathbf{p} \bullet \mathbf{x}$ (though it never maintains $\mathbf{p}$). Our algorithm will be non-deterministic, and in this example it will also generate additional final abstract states, including $\langle x^3, x^2 \rangle$. In the *soundness* part of the proof, we argue that every such result is attainable, by showing that it is bounded from above by $\mathbf{p} \bullet \mathbf{x}$ for $\mathbf{p}$ computed by the Closure Algorithm; in the *completeness* part, we show that for every $\mathbf{p}$ computed by the Closure Algorithm, we can reach (or surpass) $\mathbf{p} \bullet \mathbf{x}$ (clearly, if $\mathbf{p} \bullet \mathbf{x}$ is a maximal element in the set of attainable bounds, then based on soundness, we will reach but not exceed it). Thus, our proof does not validate the new analysis by relating its results with the concrete semantics of the analysed program, but with the results of the Closure Algorithm, which [3] already related to concrete results.

5.5. **Multivariate bounds.** As the restriction to univariate bounds was a key element in the development and proof of the polynomial-space algorithm, it came as a surprise that the same algorithm (practically the same code)—with a change of domain from univariate AMPs to multivariate ones—can compute tight multivariate bounds. But our correctness proof, as outlined above, *does not* work for the multivariate algorithm (otherwise we could have skipped the univariate stage altogether). We arrive at the result for multivariate bounds by first establishing a *reduction* among decision problems—the attainability of a multivariate bound and that of univariate bounds. We study the degree vectors $(d_1, d_2, \ldots, d_n)$ of monomial bounds $x_1^{d_1} \ldots x_n^{d_n}$, and show that these vectors define a polyhedron in n-space, and that a linear-programming view of the problem allows us to decide membership in this polyhedron using the univariate analysis. This linear-programming approach also shows that in order to obtain a tight upper bound it is not necessary to take a **max** of all the attainable bounds—it suffices to use those that figure as the vertices of the polyhedron. We proceed to use this insight, along with the linear-programming connection to univariate bounds, to prove that we can compute a sound and complete set of multivariate bounds—where by *sound* we mean that the bounds are attainable, i.e., constitute lower bounds on some possible results, and by *complete* we mean that their maximum is an upper bound on *all* possible results.

6. The Closure Algorithm

In this section, we present a version of the algorithm of [3], computing in exponential space and time. We call it "the Closure Algorithm" since an important component of the algorithm (and a cause of combinatorial explosion) is the computation of transitive closure to find the effect of any number of loop iterations. We need the Closure Algorithm because our polynomial-space solution evolved from this one, and moreover our *proof* for the efficient algorithm relies on the correctness of its predecessor.

In fact, the algorithm below is already a step beyond [3]: even if it is still exponential, it is somewhat simplified. Since this is not the main contribution here, we have decided to present it, in this section, concisely and without proofs, to allow the reader to understand and proceed quickly to our new polynomial-space algorithm. For completeness, Appendix A includes proofs that show that the Closure Algorithm below is equivalent to the old one [3].

6.1. **Symbolic semantics.** We present our analysis as symbolic semantics that assigns to every command C a symbolic abstract value $[\![\texttt{C}]\!]^S \in \wp(\texttt{AMPol})$, that is, a set of AMPs. For atomic commands, this set is a singleton:

$$[\![\,\texttt{skip}\,]\!]^S = \{Id\} \qquad\qquad [\![\,\texttt{X}_i\,\texttt{:=}\,\texttt{X}_j\,]\!]^S = \{\mathbf{set}_{ij}\}$$

$$[\![\,\texttt{X}_i\,\texttt{:=}\,\texttt{X}_j + \texttt{X}_k\,]\!]^S = \{\mathbf{add}_{ijk}\} \qquad\qquad [\![\,\texttt{X}_i\,\texttt{:=}\,\texttt{X}_j * \texttt{X}_k\,]\!]^S = \{\mathbf{mul}_{ijk}\}$$

where $\mathbf{set}_{ij}$, $\mathbf{add}_{ijk}$ and $\mathbf{mul}_{ijk}$ are AMPs defined by:

$$\mathbf{set}_{ij}(x_1,\ldots,x_n) = \langle x_1,\ldots,x_{i-1},x_j,x_{i+1},\ldots,x_n\rangle$$
$$\mathbf{add}_{ijk}(x_1,\ldots,x_n) = \langle x_1,\ldots,x_{i-1},x_j+x_k,x_{i+1},\ldots,x_n\rangle$$
$$\mathbf{mul}_{ijk}(x_1,\ldots,x_n) = \langle x_1,\ldots,x_{i-1},x_jx_k,x_{i+1},\ldots,x_n\rangle\,.$$

For composite commands, except the loop, the definition is also straightforward:

$$[\![\texttt{choose } \texttt{C}_1 \texttt{ or } \texttt{C}_2]\!]^S \;=\; [\![\texttt{C}_1]\!]^S \cup [\![\texttt{C}_2]\!]^S$$
$$[\![\texttt{C}_1;\texttt{C}_2]\!]^S \;=\; [\![\texttt{C}_2]\!]^S \bullet [\![\texttt{C}_1]\!]^S\,.$$

To handle a loop command, $\texttt{loop } \texttt{X}_\ell \texttt{ \{C\}}$, we first compute $\mathcal{S} = [\![\texttt{C}]\!]^S$, obtaining a representation of the possible effects of any *single iteration*. Then we have to apply certain operations to calculate, from $\mathcal{S}$, the effect of the entire loop:

$$[\![\texttt{loop } \texttt{X}_\ell \texttt{ \{C\}}]\!]^S = LC([\![\texttt{C}]\!]^S)[x_\ell/\tau]\,.$$

The rest of this section builds up to the definition of the function LC and the explanation of the above expression.

Example 6.1. Consider the following loop (also considered in Section 5):

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

The abstraction of the *loop body* is the following set of AMPs:

$$\mathcal{S} = \{\langle x_1 + x_3,\ x_2 + x_4,\ x_3,\ x_3\rangle,\quad \langle x_1,\ x_2 + x_3 + x_4,\ x_3,\ x_3\rangle\}.$$

We will later show how this is used to compute the effect of the whole loop.

6.2. **Simple Disjunctive Loops, Closure and Generalization.** We cite some definitions from [3] that we use in presenting the inference of the effect of a loop from an abstraction of its body.

Definition 6.2. A *polynomial transition (PT)* represents the passage from a 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 $\mathbf{p} = \langle \mathbf{p}[1], \ldots, \mathbf{p}[n] \rangle$ is a multi-polynomial.

Definition 6.3. A *simple disjunctive loop (SDL)* is a finite set $\mathcal{S}$ of AMPs, representing polynomial transitions.

The loop is "disjunctive" because the meaning is that, in every iteration, any of the given transitions may be chosen. The sense in which *abstract* MPs represent *concrete* transitions calls for a detailed definition but we omit it here, since in our context the intent should be clear: we expect $\mathcal{S}$ to be the result of analyzing the loop body with an analysis that generates asymptotically tight bounds. Importantly, a SDL does not specify the number of iterations; our analysis of a SDL generates results parameterized by the number of iterations as well as the initial state. For this purpose, we now introduce τ-polynomials, where τ (for "time") is a parameter to represent the number of iterations.

Definition 6.4. τ-polynomials are polynomials in $x_1, \ldots, x_n$ and τ.

As τ represents the number of iterations, it is not a component of the state vector, which still has dimension n. 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.

We form multi-polynomials from τ-polynomials 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 $\mathtt{X_2\!:=\!X_2 + X_1}$. The effect of iterating the composite command: $\mathtt{X_2\!:=\!X_2 + X_1;\ X_3\!:=\!X_3 + X_2}$ has an effect described by $\mathbf{x}' = \langle x_1, \ x_2 + \tau x_1, \ x_3 + \tau x_2 + \tau^2 x_1 \rangle$ (note that this is an upper bound which is not reached precisely, but is correct up to a constant factor). We denote the set of τ-multi-polynomials by $\tau\mathbf{MPol}$. We should note that composition $\mathbf{q} \circ \mathbf{p}$ over $\tau\mathbf{MPol}$ is performed by substituting $\mathbf{p}[i]$ for each occurrence of x_i in $\mathbf{q}$. Occurrences of τ are unaffected (since τ is not part of the state).

The notion of *abstract (multi-) polynomials* is extended naturally to abstract τ-(multi-) polynomials. We denote the set of abstract τ-polynomials (respectively, multi-polynomials) by $\tau\mathtt{APol}$ (respectively, $\tau\mathtt{AMPol}$). The *SDL Problem* is to compute, from $\mathcal{S}$, a set of abstract τ-multi-polynomials that represent the effect of any number of loop transitions, where τ is used to express dependence on the number of transitions. As for general programs, we focus on loops which are polynomially bounded, and search for attainable bounds; these notions are defined as follows.

Definition 6.5. A SDL $\mathcal{S}$ is said to be *polynomially bounded* when there exists a τ-MP, $\mathbf{b}$, such that for all $\mathbf{x} \in \mathbb{N}^n$, if we start with $\mathbf{x}$ and consecutively apply t arbitrary transitions from $\mathcal{S}$, the final state $\mathbf{y}$ satisfies $\mathbf{y} \leq \mathbf{b}(\mathbf{x}, t)$.

Note that we require a polynomial bound dependent on the number of iterations t, considering t as an independent variable. This makes the SDL analysis independent of the program context where the SDL is extracted from. Clearly, when we plug the results of SDL analysis back into a context where the number of iterations is also bounded by a polynomial,

we shall obtain a polynomial bound on the final state in terms of the program variables (eliminating t).

Definition 6.6. Given SDL $\mathcal{S}$ and a function $\mathbf{f} : \mathbb{N}^{n+1} \to \mathbb{N}^n$ (in the current context, $\mathbf{f}$ will be a τ-MP) we say that $\mathbf{f}$ is *attainable over* $\mathcal{S}$ if there are constants $d > 0$, $\mathbf{x}_0$ such that for all $\mathbf{x} \geq \mathbf{x}_0$, for infinitely many values $t > 0$, there exist $\mathbf{y}_0, \dots, \mathbf{y}_t$ such that

$$\mathbf{y}_0 = \mathbf{x}; \quad \forall i < t . \exists \mathbf{p} \in \mathcal{S} . \mathbf{y}_{i+1} = \mathbf{p}(\mathbf{y}_i); \quad \text{and } \mathbf{y}_t \geq d\mathbf{f}(\mathbf{x}, t).$$

We remark that SDLs enjoy the properties we pointed out in Section 4. In particular, a complete set of attainable lower bounds provides a tight asymptotic upper bound as well. In [3], we studied the SDL problem, and what we present next is an improved version of our solution from that article; the difference is explained at the end of this section. The solution consists of applying two operators to a set of (τ)-AMPs, which we shall now define. The first, *abstract closure*, naturally represents the limit-set of accumulated iterations.

Definition 6.7 (abstract closure)**.** For finite $P \subset \tau \texttt{AMPol}$, we define $Cl(P)$ to be the smallest set including Id and the elements of P, and closed under AMP composition.

In [3] we proved that when loop $\mathcal{S}$ is polynomially bounded, the abstract closure of $\mathcal{S}$ is finite, and can be computed in a finite number of steps. Briefly, the reason that the closure must be finite, when the loop is polynomially bounded, is that the polynomials in the abstract closure are of bounded degree (since they are valid lower bounds). Moreover, there is a finite number of different abstract polynomials of any given degree.

Example 6.8. In Example 6.1 we have seen that

$$\mathcal{S} = \{\langle x_1 + x_3, \ x_2 + x_4, \ x_3, \ x_3 \rangle, \quad \langle x_1, \ x_2 + x_3 + x_4, \ x_3, \ x_3 \rangle\}.$$

Computing $Cl(\mathcal{S})$ adds Id as well as the composition of the above two AMPs, which equals

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

The reader is invited to verify that this gives a composition-closed set.

The second operation is called *generalization* and its role is to capture the behaviour of variables that grow by accumulating increments in the loop, and make explicit the dependence on the number of iterations. The identification of which additive terms in a MP should be considered as increments that accumulate is at the heart of our problem, and its solution led to the definition of *iterative MPs* and *iterative kernel* below.

Definition 6.9. The *support* $\sup p$ of a polynomial p is the set of variables on which it depends, identified by their indices, e.g., $\sup(x_1 x_3 + x_4) = \{1, 3, 4\}$.

Definition 6.10. For $\mathbf{p}$ an (abstract) multi-polynomial, we say that x_i is *self-dependent* in $\mathbf{p}$ if $i \in \sup \mathbf{p}[i]$. 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 denote by $\mathrm{SD}(\mathbf{p})$ the set $\{i : i \in \sup \mathbf{p}[i]\}$, i.e., the self-dependent variables of $\mathbf{p}$.

It is easy to see that, given that variable $\mathtt{X}_i$ is polynomially bounded in a loop, if x_i is self-dependent, then $\mathbf{p}[i]$ must be of the form $x_i + q$, where q is a polynomial that does not depend on x_i (we say that x_i is *linearly self-dependent* in $\mathbf{p}$). Otherwise, $\mathbf{p}$ represents a transition that multiplies $\mathtt{X}_i$ by a non-constant quantity and iterating $\mathbf{p}$ will cause exponential growth.

Definition 6.11. We call an (abstract) MP $\mathbf{p}$ *doubling-free* if for any i, if entry $\mathbf{p}[i]$ has a monomial $\mathfrak{m} \cdot x_i$ then $\mathfrak{m} = 1$.

Assuming that we have made sure that a loop under analysis is polynomially bounded (as briefly discussed in Section 3 and more fully in [3]), then all the loop transitions must be doubling-free. Hence if $\mathbf{p}[i]$ depends on x_i, it must include the monomial x_i.

Definition 6.12. Let $\mathbf{p} \in \mathtt{AMPol}$. A monomial $x_{i_1} \ldots x_{i_n}$ is called *iterative* with respect to $\mathbf{p}$ if all i_k belong to $\mathrm{SD}(\mathbf{p})$. We call $\mathbf{p}$ *iterative* if all its monomials are iterative with respect to $\mathbf{p}$.

Iterative AMPs are crucial in the analysis of loops as, intuitively, they represent a transformation that can happen multiple times. To see what we mean compare the following assignment statements:

- $\mathtt{X_1:=X_2+X_3}$, represented by the AMP $\mathbf{p} = \langle x_2 + x_3, x_2, x_3 \rangle$. Both x_2 and x_3 are self-dependent and therefore $\mathbf{p}$ is iterative. In fact, if iterated any number of times, the resulting AMP is the same (in this case even the concrete MP remains the same, since there is no growth in the loop).
- $\mathtt{X_1:=X_1+X_3}$, represented by the AMP $\mathbf{p} = \langle x_1 + x_3, x_2, x_3 \rangle$. All three variables are self-dependent and therefore $\mathbf{p}$ is iterative. In fact, if iterated any number of times, the resulting AMP is the same. Importantly, the concrete MP will not be the same: increments of x_3 *accumulate* in variable $\mathtt{X_1}$. The algorithm will have to correctly express this growth, generalizing $\mathbf{p}$ to $\langle x_1 + \tau x_3, x_2, x_3 \rangle$.
- $\mathtt{X_1:=X_1+X_3; X_3:=X_2}$, represented by the AMP $\mathbf{q} = \langle x_1 + x_3, x_2, x_2 \rangle$. Here x_1 and x_2 are self-dependent, but x_3 (used in $\mathbf{q}[1]$) is not, therefore $\mathbf{q}$ is not iterative. In fact, if iterated twice, we get $x_1' = x_1 + x_3 + x_2$; informally, the action of the first application is to add x_3 to $\mathtt{X_1}$ while the action of the second is to add x_2. It would have been incorrect to generalize from the first step and assume that increments of x_3 will be accumulated on iteration.

Definition 6.13 (ordering of multi-polynomials)**.** For $\mathbf{p}, \mathbf{q} \in \mathtt{AMPol}$ we define $\mathbf{p} \sqsubseteq \mathbf{q}$ to hold if every monomial of $\mathbf{p}[i]$ also appears in $\mathbf{q}[i]$ for $i = 1, \ldots, n$. We then say that $\mathbf{p}$ is a *fragment* of $\mathbf{q}$.

Lemma 6.14. *Let* $\mathbf{p} \in \mathtt{AMPol}$*. Then* $\mathbf{p}$ *has a unique maximal iterative fragment, which shall be denoted by* $(\!|\mathbf{p}|\!)$ *("the iterative kernel of* $\mathbf{p}$*").*

Proof. Every monomial which is not iterative in $\mathbf{p}$ will not be iterative in any fragment of $\mathbf{p}$, so form $\mathbf{q}$ by deleting all these monomials. Then we must have $\mathbf{i} \sqsubseteq \mathbf{q}$ for any iterative $\mathbf{i} \sqsubseteq \mathbf{p}$. On the other hand, $\mathbf{q}$ is clearly iterative, so $\mathbf{q}$ is the maximal iterative fragment. $\qquad\square$

Note that even if all entries of $\mathbf{p}$ are non-zero, $(\!|\mathbf{p}|\!)$ may include zeros, e.g., for $\mathbf{p} = \langle x_3, x_2, x_1 + x_2 \rangle$, where $(\!|\mathbf{p}|\!) = \langle 0, x_2, x_2 \rangle$.

Consider a loop that has a transition $\mathbf{p}$ with $\mathbf{p}[3] = x_3 + x_1 x_2$, where x_1 and x_2 are also self-dependent in $\mathbf{p}$. Then the initial contents of variables $\mathtt{X_1}$ and $\mathtt{X_2}$ are preserved (or may even grow) when we iterate $\mathbf{p}$; and $\mathtt{X_3}$ accumulates an increment of (at least) $x_1 x_2$ on each iteration. This motivates the following definition.

Definition 6.15 (generalization)**.** Let $\mathbf{p} \in \mathtt{AMPol}$ be iterative; we define the *generalization* of $\mathbf{p}$ to be

$$\mathbf{p}^\tau[i] = \begin{cases} x_i + \tau q & \text{if } \mathbf{p}[i] = x_i + q \\ \mathbf{p}[i] & \text{otherwise.} \end{cases}$$

To continue the example presented above the definition, let

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

then

$$\mathbf{p}^\tau = \langle x_1, \; x_2 + \tau x_1, \; x_3 + \tau x_1 x_2 \rangle \,.$$

We now define the operators used in analysing a loop command:

$$Gen(\mathcal{S}) = \mathcal{S} \cup \{ (\!|\mathbf{p}|\!)^\tau \mid \mathbf{p} \in \mathcal{S} \}, \tag{6.1}$$

$$LC(\mathcal{S}) = Cl(Gen(Cl(\mathcal{S}))). \tag{6.2}$$

Recall that we have defined

$$[\![\texttt{loop } \texttt{X}_\ell \; \texttt{\{C\}}]\!]^S = LC([\![\texttt{C}]\!]^S)[x_\ell / \tau] \,.$$

This means that one takes the set $\mathcal{S}$ representing the analysis of the loop body, applies the function LC, which generates τ-AMPs, and concludes by substituting x_ℓ, the maximum number of iterations, for τ.

Example 6.16. In Example 6.1 we have seen that

$$\mathcal{S} = \{ \langle x_1 + x_3, \; x_2 + x_4, \; x_3, \; x_3 \rangle, \quad \langle x_1, \; x_2 + x_3 + x_4, \; x_3, \; x_3 \rangle \}.$$

and in Example 6.8 we have seen that $Cl(\mathcal{S}) = \mathcal{S} \cup \{Id, \mathbf{p}\}$ where

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

Next, we apply generalization. We will take a shortcut and apply it only to $\mathbf{p}$, as it subsumes the two previous AMPs. We have:

$$(\!|\mathbf{p}|\!) \; = \langle x_1 + x_3, \; x_2 + x_3, \; x_3, \; x_3 \rangle, \tag{6.3}$$

$$(\!|\mathbf{p}|\!)^\tau = \langle x_1 + \tau x_3, \; x_2 + \tau x_3, \; x_3, \; x_3 \rangle, \tag{6.4}$$

Then, in the final closure computation, we construct the composition

$$(\!|\mathbf{p}|\!)^\tau \bullet \mathbf{p} = \langle x_1 + x_3 + \tau x_3, \; x_2 + x_3 + x_4 + \tau x_3, \; x_3, \; x_3 \rangle$$

and this is as high as we get. So we finish by substituting x_3 (being the loop bound) for τ, obtaining the result

$$\langle x_1 + x_3 + {x_3}^2, \; x_2 + x_3 + x_4 + {x_3}^2, \; x_3, \; x_3 \rangle.$$

We conclude this section with some comments about the improvement of the above algorithm on [3]. First, in [3], iterative kernels were not used. Generalization was applied to *idempotent* elements in the closure set (satisfying $\mathbf{p} \bullet \mathbf{p} = \mathbf{p}$); an idempotent AMP is not necessarily iterative, and the definition of the generalization operator was slightly more complex than the definition for iterative AMPs. The choice to focus on idempotent elements was guided by an algebraic theorem that we used to prove the algorithm correct (Simon's Forest Factorization Theorem). Another difference is that in the original algorithm, the computation of the closure and subsequent generalization had to be performed in several rounds (until the set stabilized); in contrast, when generalization is applied to iterative kernels as done here, it suffices to compute closure, generalize and close once more, as expressed by (6.2).

7. The Polynomial-Space Algorithm

In this section we provide the polynomial-space algorithm for the univariate bound problem. We will try to present the algorithm in a way which motivates it and clarifies why it works. First, let us remind the reader that we are trying to compute attainable functions: such a function describes a bound that can be reached (for almost all inputs, on at least one possible computation for every such input), and therefore provides a lower bound on the worst-case result; moreover, we know by Theorem 4.1 that a complete set of attainable bounds exists, in the sense that it also implies a tight upper bound (specifically, the **max** of this finite set of functions is a worst-case upper bound). Intuitively, since an attainable function is "something that can happen," to find such functions we just have to evaluate the program symbolically and see what it computes: expressed as a function of the input, this will be an attainable function. Our evaluator will be non-deterministic and so generate a set of attainable functions. It will be then necessary to prove that this is a complete set.

7.1. **The Simple Case: Without Additions in Loops.** As long as we do not need generalization—which is only necessary when additions are performed within a loop— our analysis (Section 6) is nothing more than a symbolic evaluation, where instead of maintaining a concrete state as a vector of numbers and applying the operations numerically, we maintain AMPs that represent the computation so far, and apply symbolic arithmetics to them. While $[\![C]\!]^S$ represents all the possible effects of C, as a set, we develop in this section a non-deterministic evaluator, which only follows one possible path. Thus a `choose` command will be implemented verbatim—by non-deterministically choosing one branch or the other. Similarly, in a loop, the non-deterministic evaluator literally iterates its body a non-deterministic number of times. The information that it has to maintain, besides the current command, is just the current symbolic state. Importantly, since we are provided with an initial univariate state, all our computation is with such states, which are much more compact than AMPs. Given a degree bound d, a univariate state can be represented by a vector in $[d]^n$. This representation requires polynomial space.

Next, we give a set of definitions for a simple non-deterministic interpreter Ev_S, which does not handle generalization (we give this simple version first for the sake of presentation). Throughout this presentation, $\mathbf{s}$ refers to a univariate state.

The interpretation of `skip`, assignment and multiplication commands is the obvious

$$\mathrm{Ev}_S[\![\, \mathtt{skip} \,]\!]\mathbf{s} \Rightarrow \mathbf{s}\,; \quad \mathrm{Ev}_S[\![\, \mathtt{X}_i \!:=\! \mathtt{X}_j \,]\!]\mathbf{s} \Rightarrow \mathbf{set}_{ij} \bullet \mathbf{s}\,; \quad \mathrm{Ev}_S[\![\, \mathtt{X}_i \!:=\! \mathtt{X}_j * \mathtt{X}_k \,]\!]\mathbf{s} \Rightarrow \mathbf{mul}_{ijk} \bullet \mathbf{s}$$

where **set**, **mul** are as in the last section, and $\Rightarrow$ is the evaluation relation. Since the addition of abstract univariate monomials x^a and x^b is $x^{\max(a,b)}$, we could simulate addition $\mathtt{X}_i \!:=\! \mathtt{X}_j + \mathtt{X}_k$ by replacing $\mathbf{s}[i]$ with $\max(\mathbf{s}[j], \mathbf{s}[k])$; however we prefer to use non-determinism and define two alternatives for evaluating this command:

$$\mathrm{Ev}_S[\![\, \mathtt{X}_i \!:=\! \mathtt{X}_j + \mathtt{X}_k \,]\!]\mathbf{s} \quad \Rightarrow \quad \mathbf{set}_{ij} \bullet \mathbf{s}$$
$$\mathrm{Ev}_S[\![\, \mathtt{X}_i \!:=\! \mathtt{X}_j + \mathtt{X}_k \,]\!]\mathbf{s} \quad \Rightarrow \quad \mathbf{set}_{ik} \bullet \mathbf{s}$$

This non-determinism means that we can get the precise result as well as another one which is lower and seems redundant. However, this redundancy turns out to be advantageous later, when we get to multivariate bounds.

The interpretation of the choice and sequencing commands is natural, using non-determinism to implement `choose`.

$$\mathrm{Ev}_S[\![\texttt{choose C}_1 \texttt{ or C}_2]\!]\mathbf{s} \quad \Rightarrow \quad \mathrm{Ev}_S[\![\texttt{C}_1]\!]\mathbf{s}$$

$$\mathrm{Ev}_S[\![\texttt{choose C}_1 \texttt{ or C}_2]\!]\mathbf{s} \quad \Rightarrow \quad \mathrm{Ev}_S[\![\texttt{C}_2]\!]\mathbf{s}$$

$$\mathrm{Ev}_S[\![\texttt{C}_1 \texttt{; C}_2]\!]\mathbf{s} \qquad\qquad \Rightarrow \quad \mathrm{Ev}_S[\![\texttt{C}_2]\!]\,(\mathrm{Ev}_S[\![\texttt{C}_1]\!]\mathbf{s})$$

For the loop, since we are assuming that there is no addition—hence generalization is not necessary—we just compute composition-closure, i.e., iterate the loop body any finite number of times, using the auxiliary function $\mathrm{Ev}_S{}^*[\![\texttt{C}]\!]$:

$$\mathrm{Ev}_S{}^*[\![\texttt{C}]\!]\mathbf{s} \qquad\qquad \Rightarrow \quad \mathbf{s}$$

$$\mathrm{Ev}_S{}^*[\![\texttt{C}]\!]\mathbf{s} \qquad\qquad \Rightarrow \quad \mathrm{Ev}_S{}^*[\![\texttt{C}]\!]\,(\mathrm{Ev}_S[\![\texttt{C}]\!]\mathbf{s})$$

$$\mathrm{Ev}_S[\![\texttt{loop X \{C\}}]\!]\mathbf{s} \quad = \quad \mathrm{Ev}_S{}^*[\![\texttt{C}]\!]\mathbf{s}$$

For any $\mathbf{x}$, the set $[\![\texttt{C}]\!]^S \bullet \mathbf{x}$ and the set of possible results of evaluating $\mathrm{Ev}_S[\![\texttt{C}]\!]\mathbf{x}$ have the same maximal elements; thus they define the same worst-case bound.

The difference between computing with full AMPs and evaluating with univariate states as above can be explained as follows: the effect of any computation path of the program is given by an expression

$$\mathbf{p}_1\mathbf{p}_2\ldots\mathbf{p}_j$$

where $\mathbf{p}_i$ represents the ith atomic command (i.e., an assignment) in the path, and juxtaposition $\mathbf{pq}$ means applying $\mathbf{q}$ after $\mathbf{p}$ (left-to-right composition). In order to evaluate the result given an initial state $\mathbf{i}$, we compute

$$\mathbf{i}\mathbf{p}_1\mathbf{p}_2\ldots\mathbf{p}_j\,.$$

The point is that thanks to associativity, we can parenthesize such an expression in different ways. The expression

$$\mathbf{i}(\mathbf{p}_1\mathbf{p}_2\ldots\mathbf{p}_j)$$

represents transforming the path to an AMP (our symbolic semantics from the last section) and then applying the AMP to the given initial state. On the other hand, Ev_S implements the computation

$$((\mathbf{i}\mathbf{p}_1)\mathbf{p}_2)\ldots\mathbf{p}_j$$

(which is our default reading of a product—left-associative), which gains efficiency, since we only maintain a univariate state.

7.2. **Data-flow matrices.** When we consider loops that include addition, we have to introduce generalization, in order to account for repeated increments. From the definition of the function LC (Eq. 6.2), we can represent each of the values of $LC([\![\texttt{C}]\!]^S)$ by an expression of this form:

$$\mathbf{p}_{11}\mathbf{p}_{12}\ldots\mathbf{p}_{1j_1}\,(\!|\mathbf{p}_{21}\mathbf{p}_{22}\ldots\mathbf{p}_{2j_2}|\!)^\tau\,\mathbf{p}_{31}\mathbf{p}_{32}\ldots\mathbf{p}_{3j_3}\,(\!|\mathbf{p}_{41}\mathbf{p}_{42}\ldots\mathbf{p}_{4j_4}|\!)^\tau\cdots \qquad (7.1)$$

where each $\mathbf{p}_{ij}$ is an AMP from $[\![\texttt{C}]\!]^S$, representing a single iteration.

Our interpreter will start with a given univariate state and repeatedly apply the body of the loop; due to non-determinism, each application is equivalent to the application of one possible AMP from $[\![\texttt{C}]\!]^S$, so we are effectively applying $\mathbf{p}_{11}\mathbf{p}_{12}\ldots$. However, the parenthesized expressions need a different handling. In order to compute $(\!|\mathbf{p}|\!)^\tau$, even if we only want to evaluate it on a univariate state—i.e., $\mathbf{i}(\!|\mathbf{p}|\!)^\tau$—we need to know more about $\mathbf{p}$:

namely, we have to identify self-dependent variables, and to ensure that generalization is sound (i.e., only applied to iterative fragments). In order to track variable dependences and verify self-dependence, we now introduce *data-flow matrices* and add their maintenance to our interpreter.

Definition 7.1 (Data-Flow Matrix)**.** Given $\mathbf{p} \in \mathtt{AMPol}$, its data-flow matrix $\mathtt{DFM}(\mathbf{p})$ is a Boolean matrix $M \in M_n(\mathbb{B})$ such that M_{ij} is 1 if $\mathbf{p}[j]$ depends on x_i, and 0 otherwise.

Example 7.2. A data-flow matrix can be associated with any sequence of assignments. Consider the command $\mathtt{X_1{:}{=}X_1{*}X_2;\ X_3{:}{=}X_2{+}X_3}$, in a program with $n = 3$ variables, it is represented by the AMP $\mathbf{p} = \langle x_1 x_2, x_2, x_2 + x_3 \rangle$. The data-flow matrix is

$$\mathtt{DFM}(\mathbf{p}) = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

In previous works [23, 16, 4], matrices were not Boolean, and the values of their components used to differentiate types of value dependence (the precise meaning of "type" varies). For our algorithm, we find that we only need to track one type of dependence, namely linear dependence. Non-linear dependence will not be tracked at all (it could be, but this would complicate the algorithm; we go by the rule of maintaining the minimal information necessary for a purpose).

Definition 7.3 (Linear Data-Flow Matrix)**.** Given $\mathbf{p} \in \mathtt{AMPol}$, its *linear* data-flow matrix $\mathtt{LDFM}(\mathbf{p})$ is a Boolean matrix $M \in M_n(\mathbb{B})$ such that M_{ij} is 1 if $\mathbf{p}[j]$ depends on x_i *linearly*, and 0 otherwise.

Example 7.4. For $\mathbf{p}$ as in Example 7.2,

$$\mathtt{LDFM}(\mathbf{p}) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

In order to compute data-flow matrices for composite commands we use ordinary Boolean matrix product. The reader is invited to verify that, if A represents a command $\mathtt{C}$ and B represents command $\mathtt{D}$, then $A{\cdot}B$ represents the data-flow in $\mathtt{C;D}$. This works both for $\mathtt{DFM}$ and for $\mathtt{LDFM}$. In terms of AMPs, we have

Lemma 7.5. *The matrix abstraction commutes with left-to-right composition:* $\mathtt{DFM}(\mathbf{pq}) = \mathtt{DFM}(\mathbf{p})\mathtt{DFM}(\mathbf{q})$ *and* $\mathtt{LDFM}(\mathbf{pq}) = \mathtt{LDFM}(\mathbf{p})\mathtt{LDFM}(\mathbf{q})$.

Let us define some notations. I is the identity matrix (we always assume that the dimension is fixed by context; it is the number of variables in the program under analysis). The "assignment" matrix $A(j, i)$ represents the data-flow in the command $\mathtt{X_i{:}{=}X_j}$. It differs from I in a single column, column i, which is the jth unit vector:

$$A(j, i) = \begin{array}{c} \\ \\ \\ \\ j \end{array} \begin{bmatrix} 1 & & 0 & & \\ & \ddots & 0 & & \\ & & 0 & & \\ & & 1 & \ddots & \\ & & 0 & & 1 \end{bmatrix}$$

$$
\begin{aligned}
\mathrm{Ev}_M[\![\,\texttt{skip}\,]\!]\mathbf{s} &\Rightarrow (\mathbf{s}, I) \\
\mathrm{Ev}_M[\![\,\texttt{X}_i\texttt{:=}\texttt{X}_j\,]\!]\mathbf{s} &\Rightarrow (\mathbf{set}_{ij} \bullet \mathbf{s}, A(j,i)) \\
\mathrm{Ev}_M[\![\,\texttt{X}_i\texttt{:=}\texttt{X}_j*\texttt{X}_k\,]\!]\mathbf{s} &\Rightarrow (\mathbf{mul}_{ijk} \bullet \mathbf{s}, E(i)) \\
\mathrm{Ev}_M[\![\,\texttt{X}_i\texttt{:=}\texttt{X}_j+\texttt{X}_k\,]\!]\mathbf{s} &\Rightarrow (\mathbf{set}_{ij} \bullet \mathbf{s}, A(j,i)+A(k,i)) \\
\mathrm{Ev}_M[\![\,\texttt{X}_i\texttt{:=}\texttt{X}_j+\texttt{X}_k\,]\!]\mathbf{s} &\Rightarrow (\mathbf{set}_{ik} \bullet \mathbf{s}, A(j,i)+A(k,i)) \\
\mathrm{Ev}_M[\![\texttt{choose C}_1 \texttt{ or C}_2]\!]\mathbf{s} &\Rightarrow \mathrm{Ev}_M[\![\texttt{C}_1]\!]\mathbf{s} \\
\mathrm{Ev}_M[\![\texttt{choose C}_1 \texttt{ or C}_2]\!]\mathbf{s} &\Rightarrow \mathrm{Ev}_M[\![\texttt{C}_2]\!]\mathbf{s} \\
\mathrm{Ev}_M[\![\texttt{C}_1\texttt{; C}_2]\!]\mathbf{s} &\Rightarrow \texttt{let } (\mathbf{s}_1, M_1) = \mathrm{Ev}_M[\![\texttt{C}_1]\!]\mathbf{s}; \quad \texttt{let } (\mathbf{s}_2, M_2) = \mathrm{Ev}_M[\![\texttt{C}_2]\!]\mathbf{s}_1 \\
& \qquad \texttt{in} \quad (\mathbf{s}_2, M_1 {\cdot} M_2)
\end{aligned}
$$

Figure 2: The main function of the analyzer, except for loop handling.

We similarly define $E(i)$ ("erase") to differ from I in having the ith column zero. This represents the LDFM of $\texttt{X}_i\texttt{:=}\texttt{X}_j*\texttt{X}_k$. The following observations will be useful.

- Let M be an LDFM representing some transformation $\mathbf{p}$. Then $I \wedge M$ (the $\wedge$ is applied component-wise) gives the self-dependent variables of $\mathbf{p}$ (as the non-zero diagonal elements).
- $(I \wedge M) \cdot M$ is precisely $\texttt{LDFM}(\!(\mathbf{p})\!)$.

7.3. **Maintaining Data-Flow Matrices in Non-Looping Commands.** Next, we give to each non-loop command a (non-deterministic) semantics in terms of both a univariate state and a data-flow matrix. The evaluator Ev_M computes this semantics. It is invoked as $\mathrm{Ev}_M[\![\texttt{C}]\!]\mathbf{s}$, to evaluate command $\texttt{C}$ over initial univariate state $\mathbf{s}$. This returns a pair: the final state and a data-flow matrix, abstracting the computation that took the given state to the final one. See Figure 2. Note that in the case of addition, even if we use the fact that the sum of two abstract monomials equals one of them, we still record the data-flow from both summands.

7.4. **Handling loops.** When we handle a loop, we will be (in essence) evaluating expressions of the type in Section (7.1). We now outline the ideas used for simulating a parenthesized sub-expression $(\!(\mathbf{p}_1\mathbf{p}_2\ldots\mathbf{p}_j)\!)^\tau$.

We introduce the following notation for changing a set of components of $\mathbf{x}$:

Definition 7.6. For $Z \subset [n]$, and an n-tuple $\mathbf{x}$, $\mathbf{x}[Z \leftarrow 0]$ is obtained by changing the components indexed by Z to 0.

We can use this notation with univariate states as well as with (abstract) MPs. In fact, we note that $\mathbf{x}[Z \leftarrow 0] = Id[Z \leftarrow 0] \bullet \mathbf{x}$.

Let us recall how, in the Closure Algorithm, we arrive at $(\!(\mathbf{p})\!)^\tau$ for some $\mathbf{p} \in Cl([\![\texttt{C}]\!]^S)$.

(1) We form $\mathbf{p} \in Cl([\![\texttt{C}]\!]^S)$ as $\mathbf{p} = \mathbf{p}_1\mathbf{p}_2\ldots\mathbf{p}_j$ where every $\mathbf{p}_i \in [\![\texttt{C}]\!]^S$ represents the effect of an iteration.
(2) We reduce $\mathbf{p}$ to $(\!(\mathbf{p})\!)$. This involves deleting monomials which are not self-dependent.
(3) We apply generalization to form $(\!(\mathbf{p})\!)^\tau$.

$$\mathrm{Ev}_M[\![\texttt{loop X}_\ell\texttt{\{C\}}]\!]\mathbf{s} \ \Rightarrow \ \mathrm{Ev}_M{}^\circledast[\![\texttt{C}]\!](\ell, \mathbf{s})$$

$$\mathrm{Ev}_M{}^*[\![\texttt{C}]\!]\mathbf{s} \qquad\qquad \Rightarrow \ (\mathbf{s}, I)$$

$$\mathrm{Ev}_M{}^*[\![\texttt{C}]\!]\mathbf{s} \qquad\qquad \Rightarrow \ \texttt{let } (\mathbf{s}_1, M_1) = \mathrm{Ev}_M[\![\texttt{C}]\!]\mathbf{s}; \ \ \texttt{let } (\mathbf{s}_2, M_2) = \mathrm{Ev}_M{}^*[\![\texttt{C}]\!]\mathbf{s}_1$$
$$\texttt{in } \ (\mathbf{s}_2, M_1{\cdot}M_2)$$

$$\mathrm{Ev}_M{}^\circledast[\![\texttt{C}]\!](\ell, \mathbf{s}) \qquad \Rightarrow \ (\mathbf{s}, I)$$

$$\mathrm{Ev}_M{}^\circledast[\![\texttt{C}]\!](\ell, \mathbf{s}) \qquad \Rightarrow \ \texttt{let } Z \subset [n]; \texttt{let } (\mathbf{s}_1, M_1) = GEN(Z, \ell,\, \mathrm{Ev}_M{}^*[\![\texttt{C}]\!](\mathbf{s}[Z \leftarrow 0]));$$
$$\texttt{let } (\mathbf{s}_2, M_2) = \mathrm{Ev}_M{}^\circledast[\![\texttt{C}]\!](\ell, \mathbf{s}_1)$$
$$\texttt{in } \ (\mathbf{s}_2, M_1{\cdot}M_2)$$

Figure 3: Loop handling (up to the *GEN* function). The construct let $Z \subset [n]$ non-deterministically selects a set.

Our PSPACE algorithm never forms $\mathbf{p}$. But it goes through $\mathbf{p}_1, \mathbf{p}_2 \ldots \mathbf{p}_j$ while simulating their action on an initial univariate state, say $\mathbf{x}$. If we always do this faithfully, we end up with a result that corresponds to an application of $\mathbf{p}$ to the univariate state, $\mathbf{p} \bullet \mathbf{x}$. Now we have two challenges:

(1) how to obtain $(\![\mathbf{p}]\!) \bullet \mathbf{x}$, which may differ from $\mathbf{p} \bullet \mathbf{x}$.

(2) how to simulate generalization.

The solution to both challenges involves two tools: first, the data-flow matrix; second, we modify the initial state for the simulation of this sequence of loop iterations by non-deterministically setting a set of components to *zero*. Let us explain how this handles the challenges.

(1) $(\![\mathbf{p}]\!)$ is obtained from $\mathbf{p}$ by deleting all non-iterative monomials. It is easy to see that

$$(\![\mathbf{p}]\!) \bullet \mathbf{x} = \mathbf{p} \bullet (\mathbf{x}[\overline{\mathrm{SD}(\mathbf{p})} \leftarrow 0]),$$

where by $\overline{S}$ we mean the complement of S. Our interpreter will guess a set Z and replace $\mathbf{x}$ with $\mathbf{x}[Z \leftarrow 0]$. After simulating a number of iterations of the loop—corresponding to the application of some $\mathbf{p} \in Cl([\![\texttt{C}]\!]^S)$—we verify that we made a valid guess, meaning that all variables not in Z are self-dependent in $\mathbf{p}$, using the associated data-flow matrix. Note that a set larger than $\overline{\mathrm{SD}(\mathbf{p})}$ is considered valid; this is important for the next point.

(2) Consider $\mathbf{p}[i]$ of the form $x_i + q$, where $q(x_1, \ldots, x_n)$ depends only on self-dependent variables: such an entry is subject to generalization, multiplying q by τ, which is later replaced by the loop bound x_ℓ. In algorithm *GEN* below, we shall multiply by x_ℓ directly without the intermediate use of the τ symbol. So the expected result is $x_i + x_\ell \cdot q$, which, when evaluated with abstract univariate states, is $\max(x_i, x_\ell \cdot q)$. We see that we can get a correct result in the following way: we non-deterministically either set x_i to 0, and at the end multiply the i'th component of the result by x_ℓ, or we modify neither x_i nor the final state. In the first case we compute $x_\ell \cdot q$, and in the second $\max(x_i, q)$. The latter result is always sound (it skips generalization), while the former is sound as long as generalization can be rightfully applied, a condition which can be checked using the data-flow matrix (namely that we have evaluated an iterative MP and that $i \in \mathrm{SD}(\mathbf{p})$).

The code for interpreting a loop is presented in Figure 3. The non-deterministic alternatives for $\mathrm{Ev}_M{}^*$ and $\mathrm{Ev}_M{}^\circledast$ correspond to zero or more loop iterations respectively.

Note how the definition of $\mathrm{Ev}_M{}^\circledast$ corresponds to the expression $Cl(Gen(Cl(\llbracket \mathsf{C} \rrbracket^S)))$ in Equation 6.2: an element of the inner closure, $Cl(\llbracket \mathsf{C} \rrbracket^S)$, corresponds to a simulation of a finite number of iterations, as done by function $\mathrm{Ev}_M{}^*$; an element of the outer closure is a composition of a finite number of elements, some of which are generalized (we actually generalize whenever possible).

Function GEN checks, using the data-flow matrix, whether we have indeed simulated an iterative MP, and if we did, it applies generalization by multiplying by x_ℓ the outcome of self-dependent variables that have been initialized to zero in this sub-computation. As explained above, the effect is that we multiply the increments (the term q where $\mathbf{p}[i] = x_i + q$).

$GEN(Z, \ell, (\mathbf{s}, M)) =$ if $(\forall i \notin Z : M_{ii} = 1)$ then $(\mathbf{s}^{gen}, M^{gen})$ else $(\mathbf{s}, M)$

where

$$\mathbf{s}^{gen} = \mathbf{s} \prod_{i \in Z : M_{ii} = 1} \mathbf{mul}_{ii\ell} \qquad M^{gen} = (I \wedge M) M (I \wedge \overline{M}) \vee (I \wedge M)$$

Note that the multiplication is performed by applying $\mathbf{mul}_{ii\ell}$ for each such i. There may be several such indices, hence the use of $\prod$ notation in $\mathbf{s}^{gen}$ (it expresses iteration of postfix composition). The formula for M^{gen} deserves some explanation. Note that it does not depend on Z at all. This is important for the correctness proof. Informally, assuming we have reached the current state by simulating a sequence of assignments equivalent to an AMP $\mathbf{p}$, the expression $(I \wedge M)M$ is $\mathtt{LDFM}(\langle\!\langle \mathbf{p} \rangle\!\rangle)$; multiplying by $(I \wedge \overline{M})$ results in *removing* from this matrix the columns that correspond to self-dependent entries; and then we put them back by adding $I \wedge M$. The point of this exercise can be seen in the next example.

Example 7.7. We illustrate an application of function GEN. We refer back to Example 6.1. Recall that the analysis of the loop body produced the following AMPs

$$\mathcal{S} = \{\langle x_1 + x_3, \ x_2 + x_4, \ x_3, \ x_3\rangle, \quad \langle x_1, \ x_2 + x_3 + x_4, \ x_3, \ x_3\rangle\}.$$

We focus on the path corresponding to the first AMP, and consider its evaluation with initial state $\mathbf{s} = \langle x^1, x^2, x^3, x^4\rangle$. Applying $\mathrm{Ev}_M{}^*$, we may obtain the final result $\mathbf{s}' = \langle x^3, x^4, x^3, x^3\rangle$ along with the matrix

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

What GEN does depends on the choice of Z. In the case $Z = \emptyset$, it returns the above result, which is equivalent to skipping generalization. Instead, $\mathrm{Ev}_M{}^\circledast$ may choose a non-empty set; for example, let $Z = \{1, 4\}$. Then $\mathrm{Ev}_M{}^\circledast$ modifies $\mathbf{s}$ by setting the first and fourth component to zero before invoking $\mathrm{Ev}_M{}^*$; so its result changes into $\mathbf{s}'' = \langle x^3, x^2, x^3, x^3\rangle$, while the matrix is unaffected. Next, function GEN checks the matrix: it confirms the condition for generalization, as $M_{22} = M_{33} = 1$. Only $i = 1$ satisfies $i \in Z \wedge M_{ii} = 1$, so just this component is multiplied by the loop bound $\mathbf{s}''[3] = x^3$ and we get $\mathbf{s}^{gen} = \langle x^6, x^2, x^3, x^3\rangle$. For clarity, we now break the computation of M^{gen} in parts: first,

$$M_1 = (I \wedge M) M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

removes the dependence on x_4, which is not self-dependent. In other words, M_1 represents the iterative part of a corresponding AMP. Then,

$$M_1(I \wedge \overline{M}) \vee (I \wedge M) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \vee \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

effectively replaces the first column with a unit column. Why? In M, we had two 1's in this column, corresponding to the expression $x_1 + x_3$. Generalization turns this into $x_1 + x_3 \cdot x_3$. But we are only interested in linear terms, so we replace this column with a unit column, erasing the dependence on x_3.

8. Complexity

We now consider the space complexity of the algorithm. The size of a univariate state, implemented as a degree vector, is $O(n(1 + \log d))$ bits where $d \geq 1$ is the highest degree reached. In the problem formulation where a degree is specified as input, the algorithm can safely truncate any higher degree down to d, as this will still give a correct answer to the user's question. It remains to consider the space occupied by the recursion stack. This will be proportional to the depth of the program's syntax tree once we avoid the calls of functions $\mathrm{Ev}_M{}^*$ and $\mathrm{Ev}_M{}^{\circledast}$ to themselves, or rather change them into tail calls. This can be done by routine rewriting of the functions, using an auxiliary "accumulator" parameter, so for instance the code

$$\mathrm{Ev}_M{}^*[\![C]\!]\mathbf{s} \quad \Rightarrow \quad (\mathbf{s}, I)$$
$$\mathrm{Ev}_M{}^*[\![C]\!]\mathbf{s} \quad \Rightarrow \quad \texttt{let } (\mathbf{s}_1, M_1) = \mathrm{Ev}_M[\![C]\!]\mathbf{s}; \quad \texttt{let } (\mathbf{s}_2, M_2) = \mathrm{Ev}_M{}^*[\![C]\!]\mathbf{s}_1$$
$$\texttt{in} \quad (\mathbf{s}_2, M_1 \cdot M_2)$$

changes into

$$\mathrm{TailEv}_M{}^*[\![C]\!](\mathbf{s}, A) \quad \Rightarrow \quad (\mathbf{s}, A)$$
$$\mathrm{TailEv}_M{}^*[\![C]\!](\mathbf{s}, A) \quad \Rightarrow \quad \texttt{let } (\mathbf{s}_1, M_1) = \mathrm{Ev}_M[\![C]\!]\mathbf{s}$$
$$\texttt{in} \quad \mathrm{TailEv}_M{}^*[\![C]\!](\mathbf{s}_1, A \cdot M_1)$$

and calls to $\mathrm{Ev}_M{}^*$ from $\mathrm{Ev}_M{}^{\circledast}$ pass I for the accumulator parameter. Thus we have

Theorem 8.1. *The univariate polynomial-bound problem (Problem 3) as well as the lower-bound generation problem (Problem 4) has polynomial space complexity.*

Note that the non-deterministic nature of the algorithm places the decision problem in NPSPACE, however by Savitch's theorem this implies PSPACE as well. The corresponding generation problem can also be determinized by essentially turning it into an exhaustive search, still in polynomial space. Finally we would like to recall that a program in our language can have exponentially-growing variables as well, and emphasize that handling them, as described in [3], does not increase the complexity of our problem (the output in such a case will simply indicate that the variable has an exponential lower bound; no tight bound will be given).

9. Proof of the PSPACE Algorithm

The purpose of this section is to show that our PSPACE algorithm, Ev_M, obtains correct results. How do we define correctness? The original goal of the algorithm is to obtain symbolic expressions that tightly bound the concrete *numeric* results obtained by a core-language program, applied to an initial integer-valued state. However, thanks to our previous work, we know that such a set of symbolic expressions—namely AMPs—is obtained by the abstract semantics of Section 6. So now we are able to define $[\![\mathtt{C}]\!]^S$ as our reference and define our goal as matching the results it provides, specialized to a univariate initial state. So the purpose of this section is to prove the following:

Theorem 9.1. *The interpreter Ev_M satisfies these two correctness claims:*

(1) *Soundness: Given any univariate state $\mathbf{x}$, if $\mathrm{Ev}_M[\![\mathtt{C}]\!]\mathbf{x} \Rightarrow^* (\mathbf{y}, M)$ (for any M) then $\exists \mathbf{p} \in [\![\mathtt{C}]\!]^S$ such that $\mathbf{y} \leq \mathbf{p} \bullet \mathbf{x}$.*

(2) *Completeness: Given any $\mathbf{p} \in [\![\mathtt{C}]\!]^S$, and univariate state $\mathbf{x}$, there is $\mathbf{y} \geq \mathbf{p} \bullet \mathbf{x}$ such that $\mathrm{Ev}_M[\![\mathtt{C}]\!]\mathbf{x} \Rightarrow^* (\mathbf{y}, M)$ for some M.*

Note that we can also state this as follows: for any $\mathbf{x}$, the set $[\![\mathtt{C}]\!]^S \bullet \mathbf{x}$ and the set of possible results of evaluating $\mathrm{Ev}_M[\![\mathtt{C}]\!]\mathbf{x}$ have the same maximal elements; thus they define the same worst-case bound.

Obviously, the proof is inductive and to carry it out completely we also have to assert something about the matrices. We define a notation for brevity:

Definition 9.2. Let f be a program function that, given a univariate state, evaluates to a pair—a univariate state and a matrix. Let $\mathcal{S}$ be a set of AMPs. We write:

- $f \hookrightarrow \mathcal{S}$ if $\forall \mathbf{x}$, if $f\mathbf{x} \Rightarrow^* (\mathbf{y}, M)$ then $\exists \mathbf{p} \in \mathcal{S}$ such that $\mathbf{y} \leq \mathbf{p} \bullet \mathbf{x}$ and $M = \mathtt{LDFM}(\mathbf{p})$.
- $f \hookleftarrow \mathcal{S}$ if $\forall \mathbf{p} \in \mathcal{S}$, $\forall \mathbf{x}$, $\exists (\mathbf{y}, M)$ such that $f\mathbf{x} \Rightarrow^* (\mathbf{y}, M)$, $\mathbf{y} \geq \mathbf{p} \bullet \mathbf{x}$ and $M = \mathtt{LDFM}(\mathbf{p})$.

The full correctness claim will be the following:

Theorem 9.3. *The interpreter Ev_M satisfies these two correctness claims:*

(1) *Soundness: Given any command $\mathtt{C}$, $\mathrm{Ev}_M[\![\mathtt{C}]\!] \hookrightarrow [\![\mathtt{C}]\!]^S$.*

(2) *Completeness: Given any command $\mathtt{C}$, $\mathrm{Ev}_M[\![\mathtt{C}]\!] \hookleftarrow [\![\mathtt{C}]\!]^S$.*

Correctness is straight-forward for straight-line code (this basically amounts to the associativity argument proposed in Section 7.1, with Lemma 7.5 justifying the matrices), and it trivially extends to commands with branching, and even to loops without additions (since they are analyzed by unrolling finitely many times). It is where *generalization* is used that correctness is subtle and so, this is the focus of this section. Since the case of straight-line (or more generally, loop-free) code is simple, we will state the properties without a detailed proof.

9.1. **Correctness without loops.**

Lemma 9.4. *Let $\mathtt{C}$ be a loop-free command. Then $\mathrm{Ev}_M[\![\mathtt{C}]\!] \hookrightarrow [\![\mathtt{C}]\!]^S$ and $\mathrm{Ev}_M[\![\mathtt{C}]\!] \hookleftarrow [\![\mathtt{C}]\!]^S$.*

$\mathrm{Ev}_M{}^*$ just iterates Ev_M an undetermined number of times, which matches the definition of composition-closure precisely so the following is also straight-forward.

Lemma 9.5. *Let $\mathtt{C}$ be a loop-free command. Then $\mathrm{Ev}_M{}^*[\![\mathtt{C}]\!] \hookrightarrow Cl([\![\mathtt{C}]\!]^S)$ and $\mathrm{Ev}_M{}^*[\![\mathtt{C}]\!] \hookleftarrow Cl([\![\mathtt{C}]\!]^S)$.*

9.2. **Soundness of the analysis of loops.** We now wish to extend the soundness claim to loops. The proof is by structural induction, where all loop-free commands are covered by the above lemmas, and similarly, given correctness for commands C and D, correctness for their sequential composition, or non-deterministic choice, follow straight from definitions. So the main task is to prove the correctness for loops, namely the correctness of $\mathrm{Ev}_M^{\circledast}$. The main task is to relate the possible results of $GEN(Z, \ell, \mathrm{Ev}_M^*[\![C]\!](\mathbf{x}[Z \leftarrow 0]))$, where Z ranges over subsets of $[n]$, with members of the AMP set $Gen(Cl([\![C]\!]^S))[x_\ell/\tau]$. Note that in the expression $Gen(Cl([\![C]\!]^S))[x_\ell/\tau]$, unlike in Section 6, we substitute the x_ℓ for τ immediately after generalization; this is done for the sake of comparing the results, since Ev_M does not use τ. This modification of the Closure Algorithm is harmless, as for AMPs $\mathbf{p}$, $\mathbf{q}$ from this set, we can use the rule $(\mathbf{p} \bullet \mathbf{q})[x_\ell/\tau] = (\mathbf{p}[x_\ell/\tau]) \bullet (\mathbf{q}[x_\ell/\tau])$, due to the assumption that X_ℓ is not modified inside the loop.

Lemma 9.6. *For any univariate state* $\mathbf{x}$, *and* $Z \subset [n]$, *if* $GEN(Z, \ell, \mathrm{Ev}_M^*[\![C]\!](\mathbf{x}[Z \leftarrow 0])) \Rightarrow^* (\mathbf{y}, M)$, *then there is* $\mathbf{r} \in Gen(Cl([\![C]\!]^S))$ *such that* $\mathbf{y} \leq \mathbf{r}[x_\ell/\tau] \bullet \mathbf{x}$, *and* $M = \mathrm{LDFM}(\mathbf{r}[x_\ell/\tau])$.

Note that this lemma constitutes an induction step; it is used under the inductive assumption that for the loop body C, we have soundness.

Proof. Let $\mathrm{Ev}_M^*[\![C]\!](\mathbf{x}[Z \leftarrow 0]) \Rightarrow^* (\mathbf{y}, M)$. When generalization is not applied, i.e., $\mathbf{y} = \mathbf{s}'$ and $M = M'$, we have, by the inductive assumption, $\mathbf{p} \in Cl([\![C]\!]^S)$ such that $\mathbf{s}' \leq \mathbf{p} \bullet (\mathbf{x}[Z \leftarrow 0]) \leq \mathbf{p} \bullet \mathbf{x}$ and $M' = \mathrm{LDFM}(\mathbf{p})$. Further, $\mathbf{p} = \mathbf{p}[x_\ell/\tau]$ (since there is no τ).

Next, suppose that matrix M' satisfies the condition for generalization ($\forall i \notin Z : M'_{ii} = 1$). By the inductive assumption, we can choose $\mathbf{p} \in Cl([\![C]\!]^S)$ such that $\mathbf{s}' \leq \mathbf{p} \bullet \mathbf{x}[Z \leftarrow 0]$ and $M' = \mathrm{LDFM}(\mathbf{p})$. We now wish to show that $\mathbf{y} = \mathbf{s}^{gen} \leq (\!|\mathbf{p}|\!)^\tau[x_\ell/\tau] \bullet \mathbf{x}$. We focus on the non-trivial case which is that of an index k to which generalization applies, i.e., $k \in Z$ and $M'_{kk} = 1$. Thus x_k is self-dependent in $\mathbf{p}$. Let $\mathbf{p}_Z = \mathbf{p} \bullet Id[Z \leftarrow 0]$, then $\mathbf{p}_Z$ is iterative. In fact, $\mathbf{p}_Z \sqsubseteq (\!|\mathbf{p}|\!)$. Write $(\!|\mathbf{p}|\!)[k] = x_k + q_k$, where q_k is a polynomial not dependent on x_k. Since $k \in Z$, $\mathbf{p}_Z[k] \sqsubseteq q_k$. Moreover, $\mathbf{s}'[k] \leq \mathbf{p}[k] \bullet \mathbf{x}[Z \leftarrow 0] = \mathbf{p}_Z[k] \bullet \mathbf{x}$. Now, in the Closure Algorithm, function Gen sets $(\!|\mathbf{p}|\!)^\tau[k] = x_k + q_k\tau$ (the τ subsequently substituted with x_ℓ), so multiplying $\mathbf{s}'[k]$ by $\mathbf{s}'[\ell]$, as GEN does, produces

$$\mathbf{s}^{gen}[k] \leq (\mathbf{p}_Z[k] \bullet \mathbf{x}) \cdot \mathbf{s}'[\ell] \leq ((\!|\mathbf{p}|\!)^\tau[k][x_\ell/\tau]) \bullet \mathbf{x}.$$

We conclude that the lemma is fulfilled by $\mathbf{r} = (\!|\mathbf{p}|\!)^\tau$.

Regarding the computation of the matrix M^{gen}, we refer to the text in Section 7.4. It explains why, given $M' = \mathrm{LDFM}(\mathbf{p})$, we get $M^{gen} = \mathrm{LDFM}((\!|\mathbf{p}|\!)^\tau[x_\ell/\tau])$. $\square$

$\mathrm{Ev}_M^{\circledast}$ combines an undetermined number of calls to $GEN(Z, \ell, \mathrm{Ev}_M^*[\![C]\!](\mathbf{x}[Z \leftarrow 0]))$, for arbitrary sets Z, which by straight-forward induction, and the definition of $[\![\texttt{loop } X_\ell \texttt{ \{C\}}]\!]^S$, gives

Lemma 9.7 (Soundness for loops)**.** *Let* C *be a command such that* $\mathrm{Ev}_M[\![C]\!] \hookleftarrow [\![C]\!]^S$. *Then*

$$\lambda \mathbf{s} \,.\, \mathrm{Ev}_M^{\circledast}[\![C]\!](\ell, \mathbf{s}) \hookrightarrow Cl(Gen(Cl([\![C]\!]^S)))[x_\ell/\tau].$$

Hence,

$$\mathrm{Ev}_M[\![\texttt{loop } X_\ell \texttt{ \{C\}}]\!] \hookrightarrow [\![\texttt{loop } X_\ell \texttt{ \{C\}}]\!]^S.$$

9.3. **Completeness for an unnested loop.** Again, the crux of this proof will be the application of generalization. Here we want to show that when the Closure Algorithm generalizes, $\mathrm{Ev}_M{}^\circledast$ can match its result. Based on the presentation in Section 7.4, we might expect that for $\mathbf{p} \in Cl(\llbracket \mathtt{C} \rrbracket^S)$, given that $\mathrm{Ev}_M{}^*$ can match $\mathbf{p} \bullet \mathbf{x}$, we will capture the results of generalization by making a correct guess of the self-dependent variables and the accumulators. Let us first formalize the latter term.

Definition 9.8. Let $\mathbf{p} \in \mathtt{AMPol}$ be iterative. Recall that for $i \in \mathrm{SD}(\mathbf{p})$ where $\mathbf{p}[i] = x_i + q_i$,

$$\mathbf{p}^\tau[i] = x_i + q_i \cdot \tau$$

So that, given a univariate state $\mathbf{x}$,

$$\mathbf{p}^\tau[i][x_\ell/\tau] \bullet \mathbf{x} = \max((q_i \bullet \mathbf{x}) \cdot \mathbf{x}[\ell], \ \mathbf{p}[i] \bullet \mathbf{x}).$$

Let $Acc(\ell, \mathbf{p}, \mathbf{x})$ be the set of indices i where the first term under the **max** is larger; intuitively, where generalization increases the result. We refer to $Acc(\ell, \mathbf{p}, \mathbf{x})$ as the *accumulators* in the computation under consideration (i.e., iterating $\mathbf{p}$ starting at state $\mathbf{x}$).

Recall that function GEN is applied to a state-matrix pair $(\mathbf{s}', M)$ obtained from $\mathrm{Ev}_M{}^*\llbracket \mathtt{C} \rrbracket \mathbf{s}$. By correctness of this analysis (i.e., the inductive assumption), we assume $\mathbf{s}' \geq \mathbf{p} \bullet \mathbf{s}$, where $\mathbf{p} \in Cl(\llbracket \mathtt{C} \rrbracket^S)$, and $M = \mathtt{LDFM}(\mathbf{p})$. We shall say that this application of $\mathrm{Ev}_M{}^*$ simulates $\mathbf{p}$ (in the completeness statement, we quantify over $\mathbf{p}$ first, so it may be considered fixed throughout the discussion). When $\mathrm{Ev}_M{}^*$ is applied to a state in which some entries are set to 0, namely, $\mathbf{s}[Z \leftarrow 0]$, this is equivalent to simulating $\mathbf{p} \bullet Id[Z \leftarrow 0]$ on $\mathbf{s}$. Then, GEN possibly multiplies some entries of $\mathbf{s}'$ by $\mathbf{s}'[\ell] = \mathbf{s}[\ell]$. This can be seen as modifying $\mathbf{p} \bullet Id[Z \leftarrow 0]$ into a new AMP $\mathbf{q}$ by multiplying some entries by x_ℓ, before applying the AMP to $\mathbf{s}$. We express this by writing $\mathbf{q} = \overline{GEN}(Z, \mathbf{p}, \ell)$, noting that the action turning $\mathbf{p}$ into $\mathbf{q}$ depends on $\mathbf{p}$, M (which is determined by $\mathbf{p}$) and Z, but not on $\mathbf{s}$. We also note that M^{gen} depends only on M.

Based on the presentation of the algorithm, the reader may infer that we intend that, for an iterative $\mathbf{p}$, letting $A = Acc(\ell, \mathbf{p}, \mathbf{x})$, we should have $\overline{GEN}(A, \mathbf{p}, \ell) \bullet \mathbf{x} = \mathbf{p}^\tau[x_\ell/\tau] \bullet \mathbf{x}$. However, this is not always the case, as shown by the following example.

Example 9.9. Let $\mathbf{p} = \langle x_1, \ x_2 + x_1, \ x_3 + x_2, \ x_4 \rangle$, and $\ell = 4$. Note that $\mathbf{p}$ is iterative. Let $\mathbf{x} = \langle x^2, x^2, x^1, x^1 \rangle$. Then $Acc(\ell, \mathbf{p}, \mathbf{x}) = \{2, 3\}$. Letting $A = \{2, 3\}$, it is easy to check that $\mathbf{q} = \overline{GEN}(A, \mathbf{p}, \ell) = \langle x_1, \ x_1 x_4, \ 0, \ x_4 \rangle$. Since we have begun our computation with $\mathbf{x}[A \leftarrow 0] = \langle x^2, 0, 0, x^1 \rangle$, the final state is $\langle x^2, x^3, 0, x^1 \rangle$. Note that the result falls short of $\mathbf{p}^\tau[x_\ell/\tau] \bullet \mathbf{x} = \langle x^2, x^3, x^3, x^1 \rangle$.

The solution to this mismatch is given by Lemma 9.12 below, which shows that by calling GEN *multiple times* in succession, in other words, by going through a number of recursive calls of $\mathrm{Ev}_M{}^\circledast$, completeness is recovered.

We shall now make some preparations for this lemma.

The Dependence Graph. The matrix $\mathtt{DFM}(\mathbf{p})$ may be seen as the adjacency matrix of a graph, which we call $G(\mathbf{p})$. Arcs in this graph represent data-flow in $\mathbf{p}$ (we use $\mathtt{DFM}(\mathbf{p})$ and not $\mathtt{LDFM}(\mathbf{p})$, so this includes non-linear dependence). Paths in the graph correspond to data-flow effected by iterating $\mathbf{p}$. 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 9.10. *If $\mathbf{p}$ is iterative, and causes no exponential growth when iterated on concrete data, then $G(\mathbf{p})$ has no cycles of length greater than 1.*

Note that the assumption refers to concrete computation $(\mathbf{p} \circ \mathbf{p} \circ \ldots)$, and that we are building on the assumption that the loop under analysis is polynomially bounded (this was presumably verified beforehand, see Section 3).

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. Since $\mathbf{p}$ is iterative, all these variables must be self-dependent (as some other entry depends on them). Thus we have $\mathbf{p}[i] = x_i + q(x_1, \ldots, x_n)$ where q involves x_k; but $\mathbf{p}[k]$ depends on x_{k-1}, and so on; hence $\mathbf{p}^{(r-1)}[k] \geq x_i$, and $\mathbf{p}^{(r)}[i] \geq 2x_i$. Thus iteration of $\mathbf{p}$ generates exponential growth, contradicting the assumption. □

When focusing on a particular AMP we may 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 this property as *neat*[3].

We also state a rather evident property of an iterative MP:

Lemma 9.11. *If $\mathbf{p}$ is iterative, then $\mathbf{p} \bullet \mathbf{p} \geq \mathbf{p}$. A fortiori, $\mathbf{p} \bullet \mathbf{p}^\tau \geq \mathbf{p}$.*

Proof. Consider the calculation of $(\mathbf{p} \bullet \mathbf{p})[k]$; it is obtained by substituting $\mathbf{p}[i]$ for each occurrence of x_i in $\mathbf{p}[k]$. By iterativity, we know that $\mathbf{p}[i]$ includes x_i, so we can clearly only enlarge the polynomial. □

Lemma 9.12. *Let $\mathbf{p} \in \mathtt{AMPol}$ be neat and iterative, and let $\mathbf{x}$ be a univariate state. There are sets $Z_1, \ldots, Z_{n-1}$ such that*

$$\mathbf{q}_{n-1} \bullet \mathbf{q}_{n-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x} \geq \mathbf{p}^\tau[x_\ell/\tau] \bullet \mathbf{x}$$

where $\mathbf{q}_i = \overline{GEN}(Z_i, \mathbf{p}, \ell)$.

Proof. Let $A = Acc(\ell, \mathbf{p}, \mathbf{x})$. We define sets Z_i for $i = 1, 2, \ldots, n-1$ as follows:

$$Z_i = A \cap \{i + 1\} \cup \overline{SD(\mathbf{p})}.$$

We now state an inductive claim and prove it. The claim is: for $1 \leq i \leq n$, for all $j \leq i$,

$$(\mathbf{q}_{i-1} \bullet \mathbf{q}_{i-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x})[j] \geq (\mathbf{p}^\tau[x_\ell/\tau] \bullet \mathbf{x})[j]. \tag{9.1}$$

Note that (9.1) specializes to the lemma's statement for $i = n$.

To start the induction, consider $i = 1$. Then we only have to prove that

$$\mathbf{x}[1] \geq (\mathbf{p}^\tau[x_\ell/\tau] \bullet \mathbf{x})[1].$$

By neatness, $\mathbf{p}[1]$ can only depend on x_1, in fact $\mathbf{p}[1] = x_1$. Also $\mathbf{p}^\tau[1] = x_1$. So equality holds.

Now, let $k > 1$, and assume that (9.1) holds when $i = k - 1$. To prove it for $i = k$, we consider some sub-cases.

(1) $j \leq k$ and $j \notin SD(\mathbf{p})$: in this case, $\mathbf{p}[j]$ only depends on x_t with $t < k$ where, by IH,

$$(\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x})[t] \geq (\mathbf{p}^\tau[x_\ell/\tau] \bullet \mathbf{x})[t].$$

---

[3]Readers who like linear algebra may draw some intuition about neat MPs from thinking about triangular matrices whose diagonal elements are in $\{0, 1\}$. Interestingly, this structure has also popped up in other works in the area [12].

Since $j \notin \mathrm{SD}(\mathbf{p})$, the construction of $\overline{GEN}(Z_{k-1}, \mathbf{p}, \ell)$ does not modify $\mathbf{p}[j]$, i.e.,

$$\mathbf{q}_{k-1}[j] \bullet (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x}) = \mathbf{p}[j] \bullet (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x})$$
$$\geq \mathbf{p}[j] \bullet (\mathbf{p}^\tau [x_\ell / \tau] \bullet \mathbf{x})$$
$$\geq \mathbf{p}[j] \bullet \mathbf{x} \qquad \qquad \text{by Lemma 9.11}$$
$$= \mathbf{p}[j]^\tau [x_\ell / \tau] \bullet \mathbf{x} \qquad \qquad \text{since } j \notin \mathrm{SD}(\mathbf{p}).$$

(2) $j < k$ where $j \in \mathrm{SD}(\mathbf{p})$: here $\mathbf{p}[j]$ depends on x_j. Yet $j \notin Z_{k-1}$, so the construction of $\overline{GEN}(Z_{k-1}, \mathbf{p}, \ell)$ does not modify $\mathbf{p}[j]$, i.e.,

$$\mathbf{q}_{k-1}[j] \bullet (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x}) = \mathbf{p}[j] \bullet (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x})$$
$$\geq (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x})[j] \qquad \qquad \text{(since } \mathbf{p}[j] \geq x_j)$$
$$\geq (\mathbf{p}^\tau [x_\ell / \tau] \bullet \mathbf{x})[j] \qquad \qquad \text{by IH.}$$

(3) $j = k \in \mathrm{SD}(\mathbf{p})$ but $k \notin A_{k-1}$: here $\mathbf{p}[j]$ depends on x_j. Yet $j \notin Z_{k-1}$, so the construction of $\overline{GEN}(Z_{k-1}, \mathbf{p}, \ell)$ does not modify $\mathbf{p}[j]$, i.e.,

$$\mathbf{q}_{k-1}[j] \bullet (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x}) = \mathbf{p}[j] \bullet (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x})$$
$$\geq \mathbf{p}[j] \bullet \mathbf{x} \qquad \qquad \text{(since } \mathbf{q}_i[j] \geq x_j \text{ for all } i)$$
$$\geq (\mathbf{p}^\tau [x_\ell / \tau] \bullet \mathbf{x})[j] \qquad \qquad \text{since } j \notin A_{k-1}.$$

(4) $j = k \in A_{k-1}$: here, $\mathbf{p}[j] = \mathbf{p}[k]$ depends on x_k. We may write $\mathbf{p}[k] = x_k + z$ for a polynomial z in $x_1, \ldots, x_{k-1}$. We have, by IH,

$$z \bullet (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x}) \geq z \bullet (\mathbf{p}^\tau [x_\ell / \tau] \bullet \mathbf{x})$$

Since $k \in A_{k-1}$, we know that

$$\mathbf{p}^\tau [k][x_\ell / \tau] \bullet \mathbf{x} = (z \bullet \mathbf{x}) \cdot \mathbf{x}[\ell] = (z \cdot x_\ell) \bullet \mathbf{x}$$

Also, since $k \in Z_{k-1}$,

$$\mathbf{q}_{k-1}[k] = \overline{GEN}(Z_{k-1}, \mathbf{p}, \ell)[k] = (\mathbf{p}[k] \bullet Id[Z_{k-1} \leftarrow 0]) \cdot x_\ell = z \cdot x_\ell$$

so

$$\mathbf{q}_{k-1}[k] \bullet (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x}) = (z \cdot x_\ell) \bullet (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x})$$
$$= \mathbf{p}^\tau [k][x_\ell / \tau] \bullet (\mathbf{q}_{k-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x})$$
$$\geq \mathbf{p}^\tau [k][x_\ell / \tau] \bullet (\mathbf{p}^\tau [x_\ell / \tau] \bullet \mathbf{x}) \qquad \qquad \text{by IH}$$
$$\geq (\mathbf{p}^\tau [x_\ell / \tau] \bullet \mathbf{x}) \qquad \qquad \text{by Lemma 9.11.}$$

$\square$

We are now ready to conclude

Lemma 9.13. *For any univariate state $\mathbf{x}$, for any $\mathbf{r} \in (Gen(Cl(\llbracket \mathtt{C} \rrbracket^S)))$, there are $\mathbf{y}, M$ such that $Ev_M{}^\circledast \llbracket \mathtt{C} \rrbracket (\ell, \mathbf{x}) \Rightarrow^* (\mathbf{y}, M)$, $\mathbf{y} \geq \mathbf{r}[x_\ell / \tau] \bullet \mathbf{x}$, and $M = \mathtt{LDFM}(\mathbf{r}[x_\ell / \tau])$.*

Proof. We focus on the interesting case, which is when generalization is applied to $(\!|\mathbf{p}|\!)$, for $\mathbf{p} \in Cl(\llbracket \mathtt{C} \rrbracket^S)$, to produce $\mathbf{r}$. Note that $(\!|\mathbf{p}|\!) = \mathbf{p} \bullet Id[\overline{\mathrm{SD}(\mathbf{p})} \leftarrow 0]$. For any set $Z_i \supseteq \overline{\mathrm{SD}(\mathbf{p})}$, it is easy to see that

$$\mathbf{q}_i \stackrel{def}{=} \overline{GEN}(Z_i, \mathbf{p}, \ell) = \overline{GEN}(Z_i, (\!|\mathbf{p}|\!), \ell).$$

Using Lemma 9.12 we have sets $Z_1, \ldots, Z_{n-1}$, containing $\overline{\mathrm{SD}(\mathbf{p})}$, such that

$$\mathbf{q}_{n-1} \bullet \mathbf{q}_{n-2} \bullet \ldots \bullet \mathbf{q}_1 \bullet \mathbf{x} \geq (\!|\mathbf{p}|\!)^\tau[x_\ell/\tau] \bullet \mathbf{x} = \mathbf{r}[x_\ell/\tau] \bullet \mathbf{x}\,.$$

We conclude, using Lemma 9.5, that starting with $\mathbf{x}$, and making at most n recursive calls, $\mathrm{Ev}_M{}^\circledast$ can reach $\mathbf{y} \geq \mathbf{r}[x_\ell/\tau] \bullet \mathbf{x}$ as desired. The correctness of the matrix is again the equation $M^{gen} = \mathtt{LDFM}((\!|\mathbf{p}|\!)^\tau[x_\ell/\tau])$, which has been argued in Section 7.4. $\square$

Straight-forward induction, and the definition of $[\![\mathtt{loop}\ \mathtt{X}_\ell\ \{\mathtt{C}\}]\!]^S$, give

Lemma 9.14 (Completeness for loops). *Let $\mathcal{C}$ be a command such that $\mathrm{Ev}_M[\![\mathtt{C}]\!] \hookleftarrow [\![\mathtt{C}]\!]^S$. Then*

$$\lambda\mathbf{s}\,.\,\mathrm{Ev}_M{}^\circledast[\![\mathtt{C}]\!](\ell, \mathbf{s}) \hookleftarrow Cl(Gen(Cl([\![\mathtt{C}]\!]^S)))[x_\ell/\tau]\,.$$

Hence,

$$\mathrm{Ev}_M[\![\mathtt{loop}\ \mathtt{X}_\ell\ \{\mathtt{C}\}]\!] \hookleftarrow [\![\mathtt{loop}\ \mathtt{X}_\ell\ \{\mathtt{C}\}]\!]^S\,.$$

10. PSPACE-Completeness of bound analysis

In this section we complement our PSPACE upper bound with a hardness result, to show that our classification of the problem's complexity as PSPACE is tight. The hardness proof is a reduction from *termination of Boolean Programs*, a known PSPACE-complete problem. First, we state the definition of the decision problem to which we reduce. This is a special case of the univariate-bound decision problem, with a fixed initial state.

Definition 10.1. The decision problem DEG (for "degree") is defined as the set of triples (P, j, d) such that P is a core-language program, where the maximal value of $\mathtt{X}_j$ at the completion of the program, in terms of the univariate input $\mathbf{x} = \langle x, x, \ldots, x \rangle$, has a polynomial lower bound of $\Omega(x^d)$.

The complexity of the problem is classified in relation to the "input size" defined as: $|P| + d$, where $|P|$ is the size of the syntax tree representing P. This means that we allow d to be represented in unary notation (and we will find that this does not affect the complexity class).

Definition 10.2. A *Boolean program* is an instruction sequence $b = \mathtt{1{:}I}_1\ \mathtt{2{:}I}_2\ldots\mathtt{m{:}I}_m$ specifying a computation on Boolean variables $\mathtt{B}_1, \ldots, \mathtt{B}_k$, with program locations $1, 2, \ldots, m$.

Instructions $\mathtt{I}_\ell$ have two forms: $\mathtt{X}_i\ \mathtt{:=}\ \mathtt{not}\ \mathtt{X}_i$, and $\mathtt{if}\ \mathtt{X}_i\ \mathtt{then}\ \mathtt{goto}\ \ell'\ \mathtt{else}\ \ell''$. Here $1 \leq i \leq k$ and $\ell, \ell', \ell'' \in \{0, 1, 2, \ldots, m\}$, where ℓ, ℓ', ℓ'' are always three different locations.

Semantics: the *computation* by b is a finite or infinite state sequence $b \vdash (\ell_1, \sigma_1) \to (\ell_2, \sigma_2) \to \ldots$, where each store σ assigns a truth value in $\{\mathit{true}, \mathit{false}\}$ to each of b's variables, and ℓ_t is the program location at time t.

We are considering *input free* programs. These programs have a fixed initial state: $\ell_1 = 1$, and σ_1 assigns *false* to every variable. Given state (ℓ_t, σ_t), if $\ell_t = 0$ then the computation has terminated, else the following rules apply.

If instruction $\mathtt{I}_{\ell_t}$ is $\mathtt{X}_i\ \mathtt{:=}\ \mathtt{not}\ \mathtt{X}_i$, then σ_{t+1} is identical to σ_t except that $\sigma_{t+1}(\mathtt{B}_i) = \neg\sigma_t(\mathtt{B}_i)$. Further, $\ell_{t+1} = (\ell_t + 1) \bmod (m+1)$.

If instruction $\mathtt{I}_{\ell_t}$ is $\mathtt{if}\ \mathtt{X}_i\ \mathtt{then}\ \mathtt{goto}\ \ell'\ \mathtt{else}\ \ell''$, then σ_{t+1} is identical to σ_t. Further, $\ell_{t+1} = \ell'$ if $\sigma_t(\mathtt{B}_i) = \mathit{true}$, and $\ell_{t+1} = \ell''$ if $\sigma_t(\mathtt{B}_i) = \mathit{false}$.

Finally, program b *terminates* if for some t: $b \vdash (\ell_1, \sigma_1) \to \ldots \to (\ell_t, \sigma_t) = (0, \sigma_t)$.

The following lemma is proved in [15, Chapter 28] (with a trivial difference, concerning acceptance instead of termination):

Lemma 10.3. *The following set is complete for PSPACE:*

$$\mathcal{B} = \{b \mid b \text{ is an input-free Boolean program and } b \text{ terminates}\}.$$

Theorem 10.4. *The problem DEG is PSPACE-hard, even for programs with a single loop.*

Proof. Reduction from problem $\mathcal{B}$ above.

Suppose program $b = \texttt{1:I}_1 \ \texttt{2:I}_2 \ldots \texttt{m:I}_m$ has k variables $\texttt{B}_1, \ldots, \texttt{B}_k$. Without loss of generality, each variable has value *false* after execution, if b terminates (just add at the end of b one test and one assignment for each variable.)

Program $\texttt{p}$ will have $2 + 2k(m+1)$ variables named $\texttt{X}_1, \texttt{X}_2$, and then $\texttt{X}_{\ell,i,v}$ for all $0 \leq \ell \leq m$, $1 \leq i \leq k$ and $v \in \{0,1\}$. Informally, the program simulates the Boolean program, such that the pair $\texttt{X}_{\ell,i,0}$ and $\texttt{X}_{\ell,i,1}$ represent the value b of variable $\texttt{B}_i$ when program location ℓ was last visited. This is not a deterministic simulation, due to the absence of deterministic branching; instead, the program may take many paths, most of which are wrong simulations; and the contents of variables $\texttt{X}_{\ell,i,v}$ will reflect the correctness of the simulation. In fact, the value of each such variable will be either x or x^2, where the value x denotes error. That is, if the program reaches location ℓ with $\texttt{B}_i = v$, only the path that simulates it correctly will have $\texttt{X}_{\ell,i,v} = x^2$.

We first define the initialization command $\texttt{INIT}$: for all i, it sets $\texttt{X}_{1,i,0}$ to $\texttt{X}_1^2$ (the initial value of $\texttt{B}_i$ is *false*). All other simulation variables are set to $\texttt{X}_1$.

For every program location ℓ we define a command $\texttt{C}_\ell$ "simulating" the corresponding instruction, as follows:

- For instruction $\ell : \texttt{X}_i := \texttt{not } \texttt{X}_i$,
  $\texttt{C}_\ell$ modifies only variables $\texttt{X}_{\ell',i,v}$ where $\ell' = (\ell + 1) \bmod (m+1)$. For each $v = 0, 1$, it sets:
    $\texttt{X}_{\ell',i,v} \quad \texttt{:= } \texttt{X}_{\ell,i,\neg v}$
    while for all $j \neq i$, and $v = 0, 1$, it sets
    $\texttt{X}_{\ell',i,v} \quad \texttt{:= } \texttt{X}_{\ell,i,v}$
- For instruction $\ell : \texttt{if } \texttt{X}_i \texttt{ then goto } \ell' \texttt{ else } \ell''$,
  $\texttt{C}_\ell$ modifies the variables $\texttt{X}_{\ell',i,v}$ and $\texttt{X}_{\ell'',i,v}$ as follows:
    $\texttt{X}_{\ell',i,0} \quad \texttt{:= } \texttt{X}_1; \qquad \triangleright \textit{0 is definitely an error here}$
    $\texttt{X}_{\ell',i,1} \quad \texttt{:= } \texttt{X}_{\ell,i,1}; \quad \triangleright \textit{1 is an error if it was so before}$
    $\texttt{X}_{\ell'',i,1} \quad \texttt{:= } \texttt{X}_1; \qquad \triangleright \textit{1 is definitely an error here}$
    $\texttt{X}_{\ell'',i,0} \quad \texttt{:= } \texttt{X}_{\ell,i,0} \quad \triangleright \textit{0 is an error if it was so before}$
    For all $j \neq i$, we simply have, for both values of v,
    $\texttt{X}_{\ell',i,v} \quad \texttt{:= } \texttt{X}_{\ell,i,v};$
    $\texttt{X}_{\ell'',i,v} \quad \texttt{:= } \texttt{X}_{\ell,i,v}$

Finally, these commands are put together to make the program $\texttt{p}$:

```
INIT;
loop X₁ { choose  C₁ or C₂ or ... Cₘ };
X₂ := X₀,₁,₀ * X₀,₂,₀ * ... * X₀,ₖ,₀
```

The outcome of the reduction is $(\texttt{p}, 2, 2k)$. Thus, we are asking whether the final value of $\texttt{X}_2$ depends on x with a degree of $2k$ or more, which will only happen if the loop is completed with x^2 in each of the variables $\texttt{X}_{0,i,0}$. This means that program b could be successfully simulated up to a point where the location was 0 and all variables *false* (that is, the program really terminated).

A comment is due on the subject of the loop bound (which we chose, somewhat arbitrarily, to be x). Obviously we do not know in advance how long b runs. But since it is input-free, if b terminates, its running time is a constant and the desired output (x^{2k}) will be achieved for all values of x large enough to permit the complete simulation of b. If b does not terminate, the output will always be bounded by x^{2k-1}.   □

What is surprising in the above proof is the simplicity of the programs; curiously, they do not contain implicit data-flow in loops, which was the main challenge in the problem's solution.

11. CHECKING MULTIVARIATE BOUNDS

In this section we are moving to multivariate bounds. We show that checking a multivariate bound can be reduced to the univariate problem. We focus on the following decision problem (first defined in Section 3).

Problem 1. The *multivariate bound decision problem* is: Given a core-language program P and a monomial $m(x_1, \ldots, x_n) = x_1^{d_1} \ldots x_n^{d_n}$, report whether m constitutes an attainable bound on the final value of X_n; namely whether constants $\mathbf{x}_0$, $c > 0$ exist such that

$$\forall \mathbf{x} \geq \mathbf{x}_0 \ \exists \mathbf{x}' \ . \ \mathbf{x}[\![P]\!]\mathbf{x}' \ \wedge \ x'_n \geq cm(\mathbf{x}) \, .$$

We later discuss how to solve the related search problem, asking to find a complete set of attainable bounds. If there is no polynomial upper bound on x'_n, then all monomials are attainable. This case can be detected in polynomial time using the algorithm of [4], so we assume henceforth that x'_n is polynomially bounded. In fact, as already discussed, we may assume that all super-polynomially growing variables have been excluded.

Given P, we consider the set of attainable monomials (forming positive instances of the problem). We represent a monomial by a column vector of degrees: $\mathbf{d} = \langle d_1, \ldots, d_n \rangle^T$. The main idea in this section is to make use of the *geometry* of this set of vectors by viewing the problem as a *linear programming* problem. Before proceeding, we recall some background knowledge.

11.1. **Polyhedra.** We recall some useful definitions and properties, all can be found in [25].

Point $\mathbf{x} \in \mathbb{Q}^n$ is *a convex combination* of points $\mathbf{x}_1, \ldots, \mathbf{x}_m$ if $\mathbf{x} = \sum_{i=1}^m a_i \cdot \mathbf{x}_i$, where all $a_i \geq 0$ and $\sum_i a_i = 1$. The *convex hull* of a set of points is the set of all their convex combinations.

A *rational convex polyhedron* $\mathcal{P} \subseteq \mathbb{Q}^n$ (*polyhedron* for short) can be defined in two equivalent ways:

(1) As the set of solutions of a set of inequalities $A\mathbf{x} \leq \mathbf{b}$, namely $\mathcal{P} = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} \leq \mathbf{b}\}$, where $A \in \mathbb{Q}^{m \times n}$ is a rational matrix of n columns and m rows, $\mathbf{x} \in \mathbb{Q}^n$ and $\mathbf{b} \in \mathbb{Q}^m$ are column vectors of n and m rational values respectively. Each linear inequality (specified by a row of A and the corresponding element of $\mathbf{b}$) is known as a *constraint*.

(2) As the convex hull of a finite set of points $\mathbf{x}_i$ and rays $\mathbf{y}_j$:

$$\mathcal{P} = \text{convhull}\{\mathbf{x}_1, \ldots, \mathbf{x}_m\} + \text{cone}\{\mathbf{y}_1, \ldots, \mathbf{y}_t\}, \qquad\qquad (11.1)$$

or more explicitly: $\mathbf{x} \in \mathcal{P}$ if and only if $\mathbf{x} = \sum_{i=1}^m a_i \cdot \mathbf{x}_i + \sum_{j=1}^t b_j \cdot \mathbf{y}_j$ for some rationals $a_i, b_j \geq 0$, where $\sum_{i=1}^m a_i = 1$. The vectors $\mathbf{y}_1, \ldots, \mathbf{y}_t$ are recession directions

of $\mathcal{P}$, i.e., directions in which the polyhedron is unbounded; in terms of the constraint representation, they satisfy $A\mathbf{y}_i \leq \mathbf{0}$. The points $\mathbf{x}_i$ of a minimal set of generators are the *vertices* of $\mathcal{P}$.

For any set $\mathcal{S} \subseteq \mathbb{Q}^n$ we let $I(\mathcal{S})$ be $\mathcal{S} \cap \mathbb{Z}^n$, i.e., the set of integer points of $\mathcal{S}$. The *integer hull* of $\mathcal{S}$, commonly denoted by $\mathcal{S}_I$, is defined as the convex hull of $I(\mathcal{S})$. A polyhedron $\mathcal{P}$ is *integral* if $\mathcal{P} = \mathcal{P}_I$. This is equivalent to stating that it is generated (Eq. 11.1) by a set of integer points and rays. In particular, its vertices are integral.

We will be using some results from [25], regarding the *computational complexity* of algorithms on polyhedra. Thus we adhere to definitions used by the author, which we now recite for completeness. Schrijver denotes the bit-size of an integer x y $\|x\| = 1 + \lceil \log(|x|+1) \rceil$; the bit-size of an n-dimensional vector $\mathbf{a}$ is defined as $\|\mathbf{a}\| = n + \sum_{i=1}^{n} \|a_i\|$; and the bit-size of an inequality $\mathbf{a} \cdot \mathbf{x} \leq c$ as $1 + \|c\| + \|\mathbf{a}\|$. For a polyhedron $\mathcal{P} \subseteq \mathbb{Q}^n$ defined by $A\mathbf{x} \leq \mathbf{b}$, the *facet size*, denoted by $\|\mathcal{P}\|_\phi$, is the smallest number $\phi \geq n$ such that $\mathcal{P}$ may be described by some $A\mathbf{x} \leq \mathbf{b}$ where each inequality in $A\mathbf{x} \leq \mathbf{b}$ fits in ϕ bits. The *vertex size*, denoted by $\|\mathcal{P}\|_\psi$, is the smallest number $\psi \geq n$ such that $\mathcal{P}$ has a generator representation in which each of $\mathbf{x}_i$ and $\mathbf{y}_j$ fits in ψ bits (the size of a vector is calculated as above). The following theorem [25, part of Theorem 10.2] relates the two measures:

Theorem 11.1. *Let $\mathcal{P}$ be a rational polyhedron in $\mathbb{Q}^n$; then $\|\mathcal{P}\|_\phi \leq 4n^2 \|\mathcal{P}\|_\psi$.*

11.2. **Monomial bounds and linear programming.** From this point on, we fix $\mathcal{S}$ to mean the set of attainable monomials of a given program (as vectors in $\mathbb{N}^n$).

Let $\mathbf{u} = \langle u_1, \ldots, u_n \rangle \in \mathbb{N}^n$ be a (row) vector representing a univariate input, as in Section 7. Let $m(x_1, \ldots, x_n) = x_1^{d_1} \ldots x_n^{d_n}$. Instead of $x_1^{d_1} \ldots x_n^{d_n}$ we write, concisely, $\mathbf{x}^\mathbf{d}$. Then note that

$$m(x^{u_1}, \ldots, x^{u_n}) = x^{d_1 u_1 + \cdots + d_n u_n} = x^{\mathbf{u} \cdot \mathbf{d}}.$$

We now state

Lemma 11.2. *Finding a tight upper bound on the final value of X_n given an initial univariate state $\mathbf{u}$ is equivalent to the following optimization problem:*

$$\text{Given } \mathbf{u}, \text{ maximize } \mathbf{u} \cdot \mathbf{d} \quad \text{subject to } \mathbf{d} \in \mathcal{S}. \tag{11.2}$$

Note that the multivariate problem is simply to decide $\mathcal{S}$. The statement is by no means trivial. Luckily it comes easily out of the results of [3].

Proof. According to [3], given that the values computed by the program are polynomially bounded, the set $\mathcal{S}$ of all the attainable *multivariate* monomials provides tight upper bounds as well (more precisely, taking the maximum of these monomials gives a well-defined function— since the set is finite—and this function is an asymptotic upper bound). If we plug $\mathbf{u}$ as initial state, the set of bounds becomes $x^{\mathbf{u} \cdot \mathbf{d}}$ where $\mathbf{d}$ ranges over $\mathcal{S}$. These are univariate monomials, so they are fully ordered. The highest bound, namely the maximum value of $\mathbf{u} \cdot \mathbf{d}$, is the tight worst-case upper bound for the program. $\square$

So far, we have only considered bounds which are polynomials, where the exponents are integer. However, a function $m(x_1, \ldots, x_n) = x_1^{d_1} \ldots x_n^{d_n}$ where the d_i are non-negative *rational numbers* is a perfectly valid candidate for comparison with the results of a computation. Let $\mathcal{T}$ be the set of rational-valued vectors $\mathbf{d}$ such that the corresponding function m is attainable. Clearly $\mathcal{S} \subseteq \mathcal{T}$; in fact, $\mathcal{S} = I(\mathcal{T})$. The inclusion is clearly strict. However, when we consider the optimization problem (11.2), we have
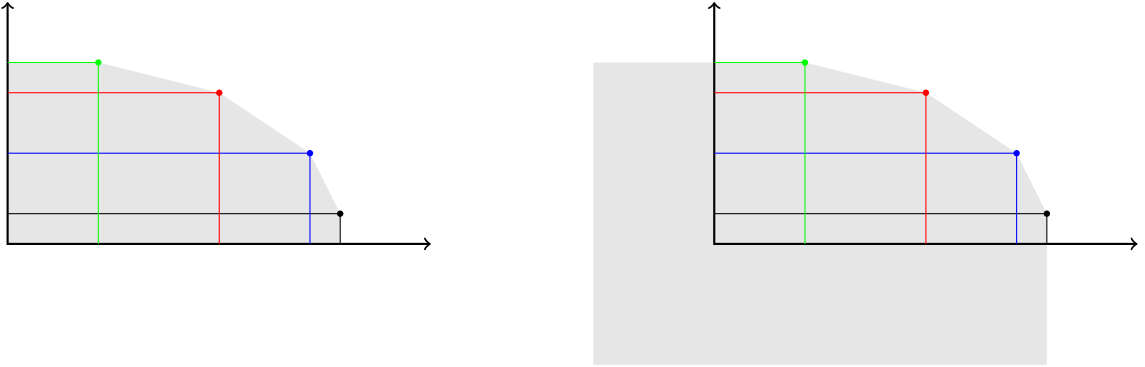
Figure 4: Assuming that the black, blue, red and green dots represent extremal elements of $\mathcal{S}$, the set $\mathcal{T}$ is represented by the convex hull of the union of four boxes (shaded area in the left-hand drawing). The right-hand drawing illustrates $\mathcal{T}'$.

Lemma 11.3. *For any vector* $\mathbf{u} \in \mathbb{N}^n$,

$$\max_{\mathbf{d} \in \mathcal{T}} \mathbf{u} \cdot \mathbf{d} = \max_{\mathbf{d} \in \mathcal{S}} \mathbf{u} \cdot \mathbf{d}.$$

Proof. Since $\mathcal{S} \subseteq \mathcal{T}$, clearly $\max_{\mathbf{d} \in \mathcal{T}} \mathbf{u} \cdot \mathbf{d} \geq \max_{\mathbf{d} \in \mathcal{S}} \mathbf{u} \cdot \mathbf{d}$. Suppose that the inequality is strict; then there is an attainable function $m = \mathbf{x}^{\mathbf{d}}$ (with $\mathbf{d} \in \mathcal{T}$) such that $m(\mathbf{u})$ has a greater exponent than $m'(\mathbf{u})$, for all $m' = \mathbf{x}^{\mathbf{d}'}$ where $\mathbf{d}' \in \mathcal{S}$. This contradicts Corollary 4.2. □

We now make a couple of observations on the shape of $\mathcal{T}$.

Lemma 11.4. *For any* $\mathbf{d} \in \mathcal{T}$, *all integer points of the box* $B_{\mathbf{0},\mathbf{d}} = \{\mathbf{x} \mid \mathbf{0} \leq \mathbf{x} \leq \mathbf{d}\}$ *are also in* $\mathcal{T}$.

Proof. Immediate since if $\langle a_1, \ldots, a_n \rangle \leq \langle b_1, \ldots, b_n \rangle$ then $x^{a_1} \cdots x^{a_n} \leq x^{b_1} \cdots x^{b_n}$. □

We conclude that $\mathcal{T}$ is a union of boxes. We can say more.

Lemma 11.5. *If functions* $f, g : \mathbb{N}^n \to \mathbb{R}$ *are attainable, so is* $\max(f, g)$.

This can be easily checked against the definition (Definition 3.1).

Lemma 11.6. $\mathcal{T}$ *is convex.*

Proof. Consider a monomial $m = \mathbf{x}^{\mathbf{d}}$ where $\mathbf{d}$ is the convex combination $\sum c_j \cdot \mathbf{d}_j$ of some $\mathbf{d}_j \in \mathcal{T}$, where $\sum_{j=1}^n c_j = 1$. We refer to the ith component of $\mathbf{d}_j$ as d_{ji}. Thus

$$m(\mathbf{x}) = \mathbf{x}^{\sum_j c_j \mathbf{d}_j} = \prod_i \prod_j x_i^{c_j d_{ji}} = \prod_j \prod_i x_i^{c_j d_{ji}} = \prod_j (\mathbf{x}^{\mathbf{d}_j})^{c_j}.$$

By the rational-weight form of the classic inequality of means [27, Eq. (2.7)], a weighted geometric mean is bounded by the corresponding weighted arithmetic mean:

$$\prod_j (\mathbf{x}^{\mathbf{d}_j})^{c_j} \leq \sum_j c_j (\mathbf{x}^{\mathbf{d}_j}) \leq \max_j (\mathbf{x}^{\mathbf{d}_j});$$

The latter is an attainable function, therefore so is m. Hence $\mathbf{d} \in \mathcal{T}$. □

So $\mathcal{T}$ is convex, includes $\mathcal{S}$, and the maximum of $\mathbf{u} \cdot \mathbf{d}$ for any $\mathbf{u} > \mathbf{0}$ is obtained at a point of $\mathcal{S}$. We conclude that $\mathcal{T}$ is an integral polyhedron [25, §16.3]. It equals the convex hull of the boxes whose upper right corners are points of $\mathcal{S}$ (Figure 11.2, left). It will be convenient in the next proof to extend $\mathcal{T}$ by allowing negative numbers as well. This gives a polyhedron $\mathcal{T}'$ which is unbounded in the negative direction (Figure 11.2, right); technically, $\mathcal{T}' = \mathcal{T} \cup \mathrm{cone}(\langle -1, 0, \ldots, 0 \rangle, \langle 0, -1, \ldots, 0 \rangle, \ldots, \langle 0, \ldots, 0, -1 \rangle)$.

Now we can use known results on the complexity of linear programming to obtain our result.

Lemma 11.7. *The facet size of $\mathcal{T}'$ is bounded by $4n^2(n + \|d_{max}\|)$, where d_{max} is the maximal attainable degree.*

This follows directly from Theorem 11.1, using the fact that our polyhedron is integral, i.e., its vertices are members of $\mathcal{S}$.

Theorem 11.8. *The multivariate-bound decision problem can be solved in space polynomial in the size of the given program and the maximal attainable degree d_{max}.*

Proof. As explained above, our approach is to show that we can decide whether $\mathbf{d} \in \mathcal{T}$. As PSPACE is closed under complement, we can consider the converse question: whether $\mathbf{d} \notin \mathcal{T}'$. This is equivalent to asking: is there a constraint $\mathbf{a} \cdot \mathbf{x} \le b$, satisfied by $\mathcal{T}'$, such that $\mathbf{a} \cdot \mathbf{d} > b$. Our decision procedure is non-deterministic and *guesses* $\mathbf{a}$. Then we need to find b, which amounts to maximizing $\mathbf{a} \cdot \mathbf{x}$ over $\mathbf{x} \in \mathcal{T}'$. Note that $\mathbf{a}$ will contain no negative numbers, since $\mathcal{T}'$ does not satisfy any constraint with negative coefficients. In case $\mathbf{a}$ has non-integer rational numbers, we can scale by their common denominator. Thus, w.l.o.g. we may assume $\mathbf{a}$ to be integer-valued, and by Lemmas 11.2 and 11.3, finding whether b exists is equivalent to solving the univariate decision problem (Problem 3 on p. 8) with initial state $\mathbf{a}$, querying the degree $\mathbf{a} \cdot \mathbf{d}$.

So, we have reduced the multivariate-bound decision problem to the univariate problem. What is the complexity of the resulting algorithm? Assuming that we know d_{max}, we can impose the bound given by Lemma 11.7. Since $\mathbf{a}$ consists of polynomially-big numbers (in terms of their bit size), our univariate algorithm (Section 7) solves the problem in polynomial space as stated.

What if we do not know d_{max}? In this case we can search for it; we give the procedure below. $\qquad\square$

In the case that d_{max} is not known, our complexity result is less satisfying when the monomial presented to the decision problem is of a degree much smaller than d_{max}. However, if what we really seek is not the decision problem for its own sake, but the discovery of tight upper bounds, then we do want to reach d_{max}, and we can then describe our algorithm's complexity as polynomial space in the *output-sensitive sense* (that is, depending on a quantity which is not the size of the input but of the output). First, let us state the complexity of the search procedure.

Theorem 11.9. *Given a core-language program P, the index of a chosen variable X_j, we can maximize $d_{max} = \sum_i d_i$ over all monomials $\mathbf{x^d}$ that constitute an attainable lower bound on the final value of X_n. The algorithm's space complexity is polynomial in the size of P plus $\|d_{max}\|$.*

Proof. Recall that a polynomial-space non-deterministic algorithm can be determinized in polynomial space (Savitch's Theorem). We search for d_{max} using a deterministic implementation of our non-deterministic algorithm (Section 7). This deterministic algorithm can be

used to give a definite answer to the query: is a given degree d attainable for $\mathtt{X}_n$. We start with $d = 1$ and repeatedly increase d (reusing the memory) until we hit a negative answer (as usual we assume that the possibility of super-polynomial growth has been eliminated first using [4]). We can actually be quicker and only loop through powers of two, since this would give an approximation to d_{max} which is close enough to be used in Theorem 11.8. $\quad\square$

12. Generating Multivariate Bounds Directly

Our analysis algorithm from Section 7 can actually be modified to yield multivariate bounds directly, as we show next. Given this statement, the reader may wonder why we bothered with the version for univariate bounds, and the reduction of multi-to-uni-variate. Let us reveal right away, then, that despite the simplicity of the following algorithm, we do not have a direct *proof* for its correctness and we approach it based on the previous algorithms.

The new algorithm is obtained thanks to a rather simple observation about the analyzer Ev_M, namely, that the only operator applied to our (univariate) monomials is product. Since the product of multivariate monomials is again a (multivariate) monomial, we can replace the data type of our symbolic state $\mathbf{s}$ from an n-tuple of *univariate* monomials to an n-tuple of *multivariate* ones. Such an n-tuple is a special case of AMP, and we also refer to it as *a monomial state* because it is used in the evaluator to abstract a state, just as the univariate state in the previous algorithm. Other than this change of datatype, no change to the code is necessary. In terms of implementation, a multivariate monomial is implemented as either an n-tuple of degrees, or a special constant representing 0. The space complexity of this representation remains polynomial in the number of variables and the maximum degree d_{max}. We call the evaluator, thus modified, Ev_{MM}. To illustrate the idea, consider a 3-variable program. Then its initial state is represented by $Id = \langle x_1, x_2, x_3 \rangle$. Here are two examples of simple straight-line computations:

$$\mathrm{Ev}_{MM} [\![\, \mathtt{X}_3 \mathtt{:=} \mathtt{X}_2 \mathtt{*} \mathtt{X}_1 \,]\!] Id \qquad\qquad \Rightarrow \quad (\langle x_1,\, x_2,\, x_1 x_2 \rangle, E(3))$$

$$\mathrm{Ev}_{MM} [\![\, \mathtt{X}_3 \mathtt{:=} \mathtt{X}_2 \mathtt{*} \mathtt{X}_1 \, ; \quad \mathtt{X}_2 \mathtt{:=} \mathtt{X}_3 \,]\!] Id \quad \Rightarrow^* \quad (\langle x_1,\, x_1 x_2,\, x_1 x_2 \rangle, E(3) A(3, 2))$$

where the data-flow matrices $A(3, 2)$ and $E(3)$ are as in Section 7.2.

Proving the correctness of the algorithm falls, as usual, into soundness and completeness.

12.1. **Soundness.** The soundness for straight-line code is straight-forward to prove, so we content ourself with stating it:

Lemma 12.1. *Let $\mathcal{C}$ be a loop-free command. Given any monomial state $\mathbf{s}$, if $\mathrm{Ev}_{MM} [\![\mathcal{C}]\!] \mathbf{s} \Rightarrow^* (\mathbf{g}, M)$ (for any M) then $\exists \mathbf{p} \in [\![\mathcal{C}]\!]^S$ such that $\mathbf{g} \sqsubseteq \mathbf{p} \bullet \mathbf{s}$, and $M = \mathtt{LDFM}(\mathbf{p})$.*

This means that if we set the initial state to Id, the monomials we generate are attainable in the sense of Corollary 4.2. We now wish to show the same property for loops. As in Section 9, the extension to $\mathrm{Ev}_{MM}{}^*$ is immediate:

Lemma 12.2. *Let $\mathcal{C}$ be a loop-free command. Given any monomial state $\mathbf{s}$, if $\mathrm{Ev}_{MM}{}^* [\![\mathcal{C}]\!] \mathbf{s} \Rightarrow^* (\mathbf{g}, M)$ then $\exists \mathbf{p} \in Cl([\![\mathcal{C}]\!]^S)$ such that $\mathbf{g} \sqsubseteq \mathbf{p} \bullet \mathbf{s}$, and $M = \mathtt{LDFM}(\mathbf{p})$.*

As in Section 9, we proceed to cover any command by structural induction, where all loop-free commands are covered by the above lemmas, and similarly, given correctness for commands $\mathtt{C}$ and $\mathtt{D}$, correctness for their sequential composition, or non-deterministic choice,

follow straight from definitions. So the main task is to prove the correctness for loops, more precisely the correctness of $\mathrm{Ev}_{MM}{}^{\circledR}$; to this end we compare the possible results of $GEN(Z, \ell, \mathrm{Ev}_{MM}{}^*[\![\mathtt{C}]\!](\mathbf{s}[Z \leftarrow 0]))$ with our reference result $Gen(Cl([\![\mathtt{C}]\!]^S))[x_\ell/\tau]$.

The soundness part of the proof repeats the arguments used with Ev_M almost verbatim.

Lemma 12.3 (Soundness of GEN)**.** *For any monomial state* $\mathbf{s}$*, and* $Z \subset [n]$*, if* $(\mathbf{g}, M) \in GEN(Z, \ell, \mathrm{Ev}_{MM}{}^*[\![\mathtt{C}]\!](\mathbf{s}[Z \leftarrow 0]))$*, then there is* $\mathbf{r} \in Gen(Cl([\![\mathtt{C}]\!]^S))$ *such that* $\mathbf{g} \sqsubseteq (\mathbf{r}[x_\ell/\tau]) \bullet \mathbf{s}$*, and* $\mathtt{LDFM}(\mathbf{r})[x_\ell/\tau] = M$*.*

Proof. Let $(\mathbf{s}', M') = \mathrm{Ev}_{MM}{}^*[\![\mathtt{C}]\!](\mathbf{s}[Z \leftarrow 0])$. When generalization is not applied, i.e., $\mathbf{g} = \mathbf{s}'$ and $M = M'$, we have, by inductive hypothesis, $\mathbf{p} \in Cl([\![\mathtt{C}]\!]^S)$ such that $\mathbf{s}' = \mathbf{p} \bullet (\mathbf{s}[Z \leftarrow 0]) \sqsubseteq \mathbf{p} \bullet \mathbf{s}$ and $M' = \mathtt{LDFM}(\mathbf{p})$, which is the same as $\mathtt{LDFM}(\mathbf{r})[x_\ell/\tau]$ (since there is no τ).

Next, suppose that matrix M' satisfies the condition for generalization ($\forall i \notin Z : M'_{ii} = 1$), and let $\mathbf{p}$ be an element of $Cl([\![\mathtt{C}]\!]^S)$ such that $\mathbf{s}' \sqsubseteq \mathbf{p} \bullet \mathbf{s}[Z \leftarrow 0]$ and $M' = \mathtt{LDFM}(\mathbf{p})$. Consider an index k to which generalization applies, i.e., $k \in Z$ and $M'_{kk} = 1$. Thus x_k is self-dependent in $\mathbf{p}$. Let $\mathbf{p}_Z = \mathbf{p} \bullet Id[Z \leftarrow 0]$. Then $\mathbf{p}_Z$ is iterative; in fact, $\mathbf{p}_Z \sqsubseteq (\!|\mathbf{p}|\!)$. Moreover, since $k \in Z$, $\mathbf{p}_Z[k] \sqsubseteq q$ such that $(\!|\mathbf{p}|\!)[k] = x_k + q$. Now, in the Closure Algorithm, function Gen multiplies the polynomial q by τ (which is then substituted with x_ℓ), so multiplying $\mathbf{p}_Z[k] \bullet \mathbf{s}$ by $\mathbf{s}'[\ell]$, as GEN does, produces

$$\mathbf{s}^{gen}[k] \sqsubseteq (\mathbf{p}_Z[k] \bullet \mathbf{x}) \cdot \mathbf{s}'[\ell] \sqsubseteq ((\!|\mathbf{p}|\!)^\tau[k][x_\ell/\tau]) \bullet \mathbf{x}.$$

We take $\mathbf{r}$ to be $(\!|\mathbf{p}|\!)^\tau$. Correctness of M is as in the previous proofs. $\qquad\square$

Once we can do the induction step for loops, we easily get

Lemma 12.4 (Soundness of Ev_{MM})**.** *Let* $\mathtt{P}$ *be a core-language program. If* $\mathrm{Ev}_{MM}[\![\mathtt{P}]\!]Id \Rightarrow^* (\mathbf{g}, M)$ *(for any* M*) then* $\exists \mathbf{p} \in [\![\mathtt{P}]\!]^S$ *such that* $\mathbf{g} \sqsubseteq \mathbf{p}$*.*

12.2. **Completeness.** We formulate our completeness claim based on the geometric view taken in the previous section. We maintain the convention that $\mathcal{S}$ denotes the set of attainable monomials (asymptotic lower bounds on the final value of $\mathtt{X}_n$) for a program under discussion (arbitrary but fixed through the discussion), and $\mathcal{T}$, $\mathcal{T}'$ its extensions to rational-number polyhedra.

Now, in the previous subsection we have argued for soundness of Ev_{MM} in the same way that we did for Ev_M, that is by comparison to the Closure Algorithm, and established the relation $\mathbf{g} \sqsubseteq \mathbf{p} \bullet \mathbf{s}$ between Ev_{MM}'s result and some $\mathbf{p} \in [\![\mathtt{C}]\!]^S$. A corresponding completeness claim might be that we can reach any monomial state $\mathbf{g} \sqsubseteq \mathbf{p} \bullet \mathbf{s}$. This is, however, not the case. Consider the two assignments:

```
X₂ := X₁ + X₂;
X₂ := X₂ * X₂
```

In Ev_{MM} (starting with Id), after the first assignment we have two monomial states, $\langle x_1, x_1 \rangle$ and $\langle x_1, x_2 \rangle$; and after the second, we have $\langle x_1, x_1^2 \rangle$ and $\langle x_1, x_2^2 \rangle$. But the Closure Algorithm generates $\langle x_1, \ x_1^2 + x_1 x_2 + x_2^2 \rangle$. Note that there is a monomial, $x_1 x_2$, which is not obtained by Ev_{MM}. But this is OK, because the degree vector $(1, 1)$ lies on the line between $(2, 0)$ and $(0, 2)$; i.e., it is not a vertex of the attainable-monomial polyhedron. In order to guarantee that we find the vertices, it suffices to compare the results on univariate states—this is the main insight of Section 11.

Lemma 12.5. *Let C be any command. Given any monomial state* $\mathbf{s}$*, any* $\mathbf{p} \in [\![C]\!]^S$*, and any univariate state* $\mathbf{x}$*, there are a monomial state* $\mathbf{g}$ *and matrix* M *such that*

$$Ev_{MM}[\![C]\!]\mathbf{s} \Rightarrow^* (\mathbf{g}, M),$$

$$\mathbf{g} \bullet \mathbf{x} \geq \mathbf{p} \bullet \mathbf{s} \bullet \mathbf{x} \text{ and}$$

$$M = \texttt{LDFM}(\mathbf{p}).$$

Proof. Fix $\mathbf{s}$, $\mathbf{p}$ and $\mathbf{x}$. By Theorem 9.1 there is $\mathbf{y} \geq \mathbf{p} \bullet \mathbf{s} \bullet \mathbf{x}$ such that

$$\mathrm{Ev}_M[\![C]\!](\mathbf{s} \bullet \mathbf{x}) \Rightarrow^* (\mathbf{y}, \texttt{LDFM}(\mathbf{p})).$$

Now, since Ev_{MM} simulates the execution of Ev_M step by step, with the sole difference that it operates on the elements of $\mathbf{s}$ instead of those of $\mathbf{s} \bullet \mathbf{x}$, and that the only operations applied to monomials are copying and product, it should be clear that

$$(\mathrm{Ev}_{MM}[\![C]\!]\mathbf{s}) \bullet \mathbf{x}$$

evaluates to the same set of values as

$$\mathrm{Ev}_M[\![C]\!](\mathbf{s} \bullet \mathbf{x})$$

and this justifies the lemma. $\qquad\square$

Corollary 12.6 (Completeness of Ev_{MM})**.** *Let C be a core-language program. Let* $m = x_1^{d_1} \cdots x_n^{d_n}$ *be a monomial representing an attainable bound on the final value of* X_n *after executing C, and such that its degree vector* $\mathbf{d}$ *is a vertex of* $\mathcal{T}'$*. Then* $\mathrm{Ev}_{MM}[\![C]\!]Id \Rightarrow^* (\mathbf{g}, M)$ *(for some* M*) such that* $\mathbf{g}[n] = m$*.*

Proof. Since $\mathbf{d}$ is a vertex, there is a univariate state $\mathbf{x}$ for which $\mathbf{d}$ is the unique solution to the optimization problem (11.2), meaning that $m \bullet \mathbf{x}$ maximizes the nth component in $\mathbf{p} \bullet \mathbf{x}$ when $\mathbf{p}$ ranges over $[\![C]\!]^S$. By the last lemma, this can be matched by Ev_{MM}, and since Ev_{MM} is also sound (i.e., it only produces attainable monomials), it must produce m. $\quad\square$

Since we can find any vertex of the polyhedron, we have a complete solution in the sense that the **max** of all these monomials is a tight multivariate upper bound on the queried variable, as in Corollary 4.2.

12.3. **Discussion.** We have given a polynomial-space solution to the problem of *generating* a bound. Since the algorithm can generate all the vertices of the attainable-monomial polyhedron, it could be used to implement an alternative *decision* procedure as well, replacing the proof of Theorem 11.8. Note that, ultimately, both proofs rely on the insight expressed by Lemma 11.2, and even if this is a redundancy, we thought that presenting both approaches makes the picture more complete.

Looking back at Ev_{MM}, as we have already noticed that the initial state does not really influence any decision by the algorithm, it is easy to see that we could rewrite the algorithm to remove this parameter. The algorithm then abstracts a command to a set in $\texttt{AMPol} \times M_n(\mathbb{B})$, rather than a function that maps initial states to such sets. Next, we give this version in equational form, in the style of the Closure Algorithm. The result set may be of exponential size, so the non-deterministic interpreter form of the algorithm is useful to prove the PSPACE complexity class, but we feel that the following presentation is of a certain elegance. The abstraction of command C is denoted $[\![C]\!]^M$ (M for Monomials).

We define $[\![\mathtt{C}]\!]^M$, first for non-looping commands:

$$
\begin{array}{rcl}
[\![\,\mathtt{skip}\,]\!]^M & = & \{(Id, I)\} \\
[\![\,\mathtt{X}_i\,\mathtt{:=}\,\mathtt{X}_j\,]\!]^M & = & \{(\mathbf{set}_{ij}, A(j,i))\} \\
[\![\,\mathtt{X}_i\,\mathtt{:=}\,\mathtt{X}_j\,\mathtt{+}\,\mathtt{X}_k\,]\!]^M & = & \{(\mathbf{set}_{ij}, A(j,i)+A(k,i)),\ (\mathbf{set}_{ik}, A(j,i)+A(k,i))\} \\
[\![\,\mathtt{X}_i\,\mathtt{:=}\,\mathtt{X}_j\,\mathtt{*}\,\mathtt{X}_k\,]\!]^M & = & \{(\mathbf{mul}_{ijk}, E(i))\} \\
[\![\,\mathtt{choose\ C}_1\ \mathtt{or\ C}_2\,]\!]^S & = & [\![\mathtt{C}_1]\!]^S \cup [\![\mathtt{C}_2]\!]^S \\
[\![\mathtt{C}_1;\mathtt{C}_2]\!]^S & = & [\![\mathtt{C}_2]\!]^S \odot [\![\mathtt{C}_1]\!]^S\,,
\end{array}
$$

where $(\mathbf{p}, A) \odot (\mathbf{q}, B) \stackrel{def}{=} (\mathbf{p} \bullet \mathbf{q},\ A \cdot B)$.

Finally,

$$[\![\mathtt{loop\ X}_\ell\ \mathtt{\{C\}}]\!]^M = Cl(Gen(\ell, Cl([\![\mathtt{C}]\!]^M)))$$

where $Cl()$ denotes closure under $\odot$, and

$$Gen(\ell, \mathcal{S}) = \bigcup_{\substack{(\mathbf{p},M)\in\mathcal{S} \\ Z\subset[n]}} GEN(Z, \ell,\ (\mathbf{p} \bullet (Id[Z \leftarrow 0]), M))\,,$$

with function GEN defined just as in Ev_{MM} (namely just as in Ev_M, but calculating with multivariate monomials), as illustrated in the following example.

Example 12.7. Consider the following loop:

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

The abstraction of the *loop body* clearly yields the following pairs:

$$\left(\langle x_1,\ x_2,\ x_1,\ x_2,\ x_5\rangle,\ \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}\right),$$

$$\left(\langle x_3,\ x_2,\ x_3,\ x_4,\ x_5\rangle,\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}\right),$$

$$\left(\langle x_4,\ x_2,\ x_3,\ x_4,\ x_5\rangle,\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}\right)$$

In the closure of the loop body's abstraction we will find compositions of these results; e.g., composing the first with the second (respectively the third) corresponds to simulating the instructions:

$X_3 := X_1; X_4 := X_2;$
$X_1 := X_3 + X_4;$

and yields the results:

$$(\langle x_1,\ x_2,\ x_1,\ x_2,\ x_5\rangle, \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}),$$

$$(\langle x_2,\ x_2,\ x_1,\ x_2,\ x_5\rangle, \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}).$$

To illustrate the effect of GEN, let $Z = \{1,3,4\}$. Inspecting the above matrix, we note that $3, 4$ must be in the set to permit generalization, since they denote non-self-dependent entries; the presence of 1 in Z implies that we treat X_1 as an accumulator. The above two matrices produce, respectively, the following results

$$(\langle 0,\ x_2,\ 0,\ x_2,\ x_5\rangle\quad, \begin{bmatrix} 1 & 0 & 1 & 0 & o \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}),$$

$$(\langle x_2 x_5,\ x_2,\ 0,\ x_2,\ x_5\rangle, \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}).$$

We state the correctness of $[\![\cdot]\!]^M$, which is based on the results established for Ev_{MM}, by the simple observation that

$$\mathbf{p} \in [\![\mathtt{C}]\!]^M \iff (\mathrm{Ev}_{MM}[\![\mathtt{C}]\!]Id \Rightarrow^* (\mathbf{p}, \mathtt{LDFM}(\mathbf{p}))).$$

Theorem 12.8. *Let* P *be a core-language program. The set* $[\![\mathtt{P}]\!]^M$ *satisfies the following properties:*

(1) *(soundness) If* $\mathbf{m} \in [\![\mathtt{P}]\!]^M$ *then* $\exists \mathbf{p} \in [\![\mathtt{P}]\!]^S$ *such that* $\mathbf{m} \sqsubseteq \mathbf{p}$.

(2) *(completeness) Let* $m = x_1^{d_1} \cdots x_n^{d_n}$ *be a monomial representing an attainable bound on the final value of variable* X_n *after executing* P, *and such that its degree vector* $\mathbf{d}$ *is a vertex of* $\mathcal{T}'$. *Then there is* $\mathbf{m} \in [\![\mathtt{P}]\!]^M$ *such that* $\mathbf{m}[n] = m$.

Note that the soundness clause indicates that $\mathbf{m}$ is an attainable AMP for P; and that in the completeness clause, the choice of X_n is arbitrary. The same property holds for any chosen variable (and the corresponding set of attainable monomial bounds).

13. A Few Simple Extensions

A natural follow-up to the above results is to extend them to richer programming languages. In this section we briefly discuss four such extensions. First, two trivial ones:

- We can allow any polynomial (with positive coefficients) as the right-hand side of an assignment—this is just syntactic sugar.
- We can include the weak (non-deterministic) assignment form which says "let X be a natural number *bounded* by the following expression." This construct may be useful if we want to use our language to approximately simulate the behavior of programs that have arithmetic expressions which we do not support, as long as a preprocessor can statically determine polynomial bounds on such expressions. No change to the algorithm is required.

Next, two extensions which are not trivial, but very easy.

Resets. [2] shows that it is possible to enrich the core language by a reset instruction, X := 0, and still solve the problem of distinguishing polynomially-bounded variables from potentially super-polynomial ones. Conceptually this is perhaps not a big change, but technically it caused a jump in the complexity of the solution, from PTIME to PSPACE-complete. Our tight-bound problem is already PSPACE-complete, and we can extend our solution to handle resets without a further increase in complexity. In fact, with the abstract-interpreter algorithm, adding the resets is very smooth. We add a special constant $\underline{0}$, which means *definitely zero* — the symbolic value of a variable that has been reset. This value is treated using the natural laws, i.e., $\underline{0} + x = x$, $\underline{0} * x = \underline{0}$. Resetting a variable cuts data-flow, so the abstraction of X := 0 reflects this in the LDFM:

$$\mathrm{Ev}_M[\![\, \mathtt{X}_i \mathtt{:=} \mathtt{0} \,]\!]\mathbf{s} \;\;\Rightarrow\;\; (\mathbf{s}[i \leftarrow \underline{0}], E(i))$$

Additional changes are necessary in the analysis of loops. If the loop bound is definitely zero, we know for certain that the loop body is not executed (unlike the general case, where skipping the loop is just a *possibility*). Thus we rewrite the corresponding section of our interpreter as follows:

$$\mathrm{Ev}_M[\![\mathtt{loop}\ \mathtt{X}_\ell \mathtt{\{C\}}]\!]\mathbf{s} = \begin{cases} (\mathbf{s}, I) & \text{if } \mathbf{s}[\ell] = \underline{0}, \\ \mathrm{Ev}_M{}^{\circledast}[\![\mathtt{C}]\!](\ell, \mathbf{s}) & \text{otherwise.} \end{cases}$$

and similarly for Ev_{MM}. Inside $\mathrm{Ev}_M{}^{\circledast}$ (respectively $\mathrm{Ev}_{MM}{}^{\circledast}$), we have to deal with the case that a definitely-zero variable is selected for the set Z of variables to be set to zero. Let us recall that this set serves two purposes:

(1) To mask out non-self-dependent variables. If the variable is initially definitely zero, there is of course no need to mask it out, and moreover, replacing $\underline{0}$ with an ordinary zero may cause imprecision, in case that an internal loop depends on that variable.

(2) To isolate the increment q in computations that set $x'_i = x_i + q$, by putting 0 in x_i. This has to work, even in the case that x_i is initially $\underline{0}$; while, as above, replacing $\underline{0}$ with an ordinary zero would be wrong.

From these considerations, the correct code for $\mathrm{Ev}_M{}^{\circledast}$ becomes the following.

$$\mathrm{Ev}_M{}^{\circledast}[\![\mathtt{C}]\!](\ell, \mathbf{s}) \;\; \Rightarrow \;\; (\mathbf{s}, I)$$

$$\mathrm{Ev}_M{}^{\circledast}[\![\mathtt{C}]\!](\ell, \mathbf{s}) \;\; \Rightarrow \;\; \begin{aligned}&\mathtt{let}\ \ Z \subseteq \{i \mid \mathbf{s}[i] \neq \underline{0}\};\\ &\mathtt{let}\ \ (\mathbf{s}_1, M_1) = GEN(Z, \ell, \mathrm{Ev}_M{}^{*}[\![\mathtt{C}]\!](\mathbf{s}[Z \leftarrow 0]));\\ &\mathtt{let}\ \ (\mathbf{s}_2, M_2) = \mathrm{Ev}_M{}^{\circledast}[\![\mathtt{C}]\!](\ell, \mathbf{s}_1)\\ &\mathtt{in}\ \ \ (\mathbf{s}_2, M_1 {\cdot} M_2)\end{aligned}$$

while in function GEN:

$$GEN(Z, \ell, (\mathbf{s}, M)) = \text{if } (\forall i \notin Z : (\mathbf{s}[i] = \underline{0} \vee M_{ii} = 1))$$
$$\text{then } (\mathbf{s}^{gen}, M^{gen}) \text{ else } (\mathbf{s}, M)$$

where

$$\mathbf{s}^{gen} = \mathbf{s} \prod_{i:M_{ii}=1 \wedge (i \in Z \vee \mathbf{s}[i]=\underline{0})} \mathbf{mul}_{ii\ell}$$
$$M^{gen} = (I \wedge M)M(I \wedge \overline{M}) \vee (I \wedge M)$$

Unbounded "value". In [6] we proposed to extend our language with an instruction $\mathtt{X} \;{:}{=}\; {*}$, sometimes called "havoc $\mathtt{X}$." The intention is to indicate that $\mathtt{X}$ is set to a value which is not expressible in the core language. This may be useful if we want to use our core-language programs as an abstraction of real programs, that possibly perform computations that we cannot express or bound by a polynomial. It may also be used to model a **while**-loop (as done, for example, in [16]): we do not attempt to analyze the loop for termination or even for bounds (this is outside the scope of our algorithm; such loops depend on concrete conditionals which we do not model at all). So all we suggest is to analyze what happens in the loop under the assumption that we cannot bound the number of iterations. This can be simulated by setting the loop bound to $*$.

The implementation is very similar to that of reset, and moreover, they can be combined. The rules for calculating with $*$ are: $* \cdot 0 = 0$, and $* \cdot m = *$ for any other monomial m. And, of course, if an output variable comes out as $*$ it is reported to have no polynomial upper bound.

14. 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 [29, 24, 19]. It is also an area of active research, with tools being currently (or recently) developed including COSTA [1], APROVE [11], CIAOPP [20] , C^4B [9], LOOPUS [26] and COFLOCO [10]—and this is just for imperative programs. There is also work on functional and logic programs, term rewriting systems, recurrence relations, etc. which we cannot attempt to survey here.

Our programming language, which is based on bounded loops, is similar to Meyer and Ritchie's language of *Loop Programs* [21], which differed by being deterministic and including constants; it is capable of computing all the primitive recursive functions and too strong to obtain decidability results. Similar languages have been used as objects for analysis of growth-rate (in particular, identifying polynomial complexity) in [17, 18, 23, 16]. These works, too, considered deterministic languages for which precise analysis is impossible, but

on reading them, one can clearly see that there are some clear limits to the aspects of the language that they analyze (such as using a loop counter as a bound, not relying on the assumption that the iteration count is always completed). Such considerations led to the definition of the weak, non-deterministic language in [4]. Recent work in static analysis of fully-capable programs [8] 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, based on loop bounds provided by the loop analysis. This may be seen, loosely speaking, as a process that reduces general programs to bounded-loop programs which are then analyzed. Our previous paper [3] was, however, the first that gave a solution for computing tight polynomial bounds which is complete for the class of programs we consider.

The problem of deciding termination of a weak programming language has been considered in several works; one well-studied example considers single-path while loops with integer variables, affine updates, and affine guard conditions [28, 7, 13]. Its decidability was first proved in [13], but it is not shown to what complexity class this problem belongs. It would be interesting to see whether this can also be shown to be PSPACE-complete.

Dealing with a somewhat different problem, [22, 14] both check, or find, *invariants* in the form of polynomial equations, so they can identify cases where a final result is *precisely* polynomial in the input values. 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. However, their language has no mechanism to represent bounded loops, whereas in our work, it is all about these loops.

15. Conclusion

We have complemented previous results on the computability of polynomial bounds in the weak imperative language defined in [4]. While [4] established that the existence of polynomial bounds is decidable, [3] showed that tight bounds are computable. Here we have shown that they can be computed in polynomial space, and that this is optimal in the sense of PSPACE-hardness. We have thus settled the question of the complexity class of this program-analysis problem. Interestingly, this improvement required some new ideas on top of [3], including the connection of multivariate bounds to univariate bounds based on the geometry of the set of monomial bounds, and the computation of bounds using an abstraction which is space-economical, including only degree vectors and data-flow matrices.

Some challenging open problems regarding bound computation remain:

(1) Whether the problem is solvable at all for various extensions to the language (see [5, 3] for further discussion).
(2) How to compute more precise polynomial bounds (with explicit constant factors); can we make them tight?
(3) Computation of tight super-polynomial bounds (currently, if a variable cannot be bounded by a polynomial, we give no upper bound).

Another problem we are interested in is: if one of our programs computes a polynomially-bounded result, how high can the degree of the bounding polynomial be, in terms of the size of the program? We conjecture that the degree is at most exponential in the program size. However, a proof which we attempted (in an earlier technical report) turned out to be wrong, so the problem stands open.

Acknowledgement. We thank the anonymous referees, whose diligent reviews have been invaluable to our work on this article. On the occasion of his 80th birthday, we would also like to acknowledge the enormous contributions made by Neil D. Jones in this area, and his influence on our work.

Appendix A. Justification of The Closure Algorithm

The purpose of this section is to clarify how the Closure Algorithm (Section 6) relates to the algorithm we published in [3], and justify the correctness of the new version. The justification amounts to a proof that the algorithm computes a set of lower bounds which is equivalent to the set computed by [3]; we are thus relying on the correctness proof of [3].

A.1. **Background.** We begin with a stock of definitions and auxiliary results from [3]. This subsection contains no novel material and is included for completeness.

Definition A.1 (Subsumption order)**.** For τ-multivariate polynomials p, q we define $p \lhd q$ to hold if for every monomial $\mathfrak{m}$ of p some multiple of $\mathfrak{m}$ appears in q. We then say that p is subsumed by q.

For example, we have $x_1 x_2 \lhd \tau x_1 x_2 + x_3$. Moreover, $x_1 + \tau x_1 \lhd \tau x_1$. It should be clear that this order agrees with the asymptotic growth-rate order commonly denoted by $p = O(q)$. The definition is extended to MPs component-wise. Note that $\lhd$ is a coarser relation than the fragment relation $\sqsubseteq$, i.e., $\mathbf{p} \sqsubseteq \mathbf{q} \Rightarrow \mathbf{p} \lhd \mathbf{q}$. For example, $x_1 x_2$ is not a fragment of $\tau x_1 x_2$, but is subsumed by it.

Definition A.2. $\mathbf{p} \in \tau\mathtt{AMPol}$ 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.

Let $\mathbf{p} \in \tau\mathbf{MPol}$. Recall (Section 9) that its *dependence graph* $G(\mathbf{p})$ is a directed graph with node set $[n]$, which includes an arc $i \to j$ if and only if $\mathbf{p}[j]$ depends on x_i.

The following lemma (similar to Lemma 9.10) is from [3].

Lemma A.3. *Suppose that the SDL $\mathcal{S}$ is polynomially bounded, $\mathbf{p} \in \tau\mathbf{MPol}$ is attainable over $\mathcal{S}$, and $\alpha(\mathbf{p})$ is idempotent. Then $G(\mathbf{p})$ does not have any simple cycle longer than one arc. In other words, $G(\mathbf{p})$ is quasi-acyclic.*

We assume that the loop under analysis is polynomially bounded; therefore we can rely on the above property of $G(\mathbf{p})$, being quasi-acyclic. 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 refer to an (abstract) MP satisfying this as *neat*.

A.2. **The Old Closure Algorithm.** We now recall the algorithm from [3], to facilitate its comparison with the current version. We are only considering the SDL analysis component—the rest is identical. The following function from [3] is equivalent in role to function LC (Equation 6.2):

Solve($\mathcal{S}$)

(1) Set $T = \mathcal{S}$.

(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.

The reader may have noticed that in this algorithm, generalization is applied to idempotent elements, instead of iterative elements. As an idempotent element is not necessarily iterative, the definition of generalization we used (Definition 6.15) does not suffice.

Definition A.4 (extended generalization). Let $\mathbf{p} \in \tau\texttt{AMPol}$; define $\mathbf{p}^\tau$ by:

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

where $\mathbf{p}[i]''$ is defined as the sum of all iterative monomials of $\mathbf{p}[i]$, except for x_i (a monomial in $\mathbf{p}$ is iterative if it consists of variables self-dependent in $\mathbf{p}$. We employ the unmotivated notation $\mathbf{p}[i]''$ just for consistency with [3]).

Note that the addition in this definition is abstract (see example below). It is easy to verify that Definition 6.15 agrees with this definition (up to subsumption) if $\mathbf{p}$ is iterative (and, in particular, τ-free; monomials that include τ are not considered iterative).

Example A.5. We illustrate Definition A.4 with an example. Let $\mathbf{p} = \langle x_1,\ x_2,\ x_2,\ x_4 + \tau x_2 + \tau x_1 + x_1 x_2 \rangle$. Then $\mathrm{SD}(\mathbf{p}) = \{1, 2, 4\}$. This τ-AMP is not iterative, because of the τ-monomials in $\mathbf{p}[4]$. However, it is idempotent, and applying Definition A.4, $\mathbf{p}^\tau = \langle x_1,\ x_2,\ x_2,\ x_4 + \tau x_2 + \tau x_1 + \tau x_1 x_2 \rangle$. Note also that $\mathbf{p}$ is neat: $G(\mathbf{p})$ is shown in Figure 5.
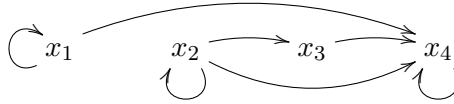

Figure 5: Dependence graph for the example MP. Nodes are labeled x_j rather than just j for readability.

Example A.6. Next, we give an example for a complete computation of SOLVE. We consider the loop:

$$\texttt{loop } \mathrm{X}_3 \ \{ \ \mathrm{X}_1 := \ \mathrm{X}_1 \ + \ \mathrm{X}_2; \ \ \mathrm{X}_2 := \ \mathrm{X}_2 \ + \ \mathrm{X}_3; \ \ \mathrm{X}_4 := \ \mathrm{X}_3 \ \}$$

The body of the loop is evaluated symbolically and yields the abstract multi-polynomial:

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

This singleton is the initial set T. To obtain $Cl(T)$, we include the identity MP, and also compute the compositions

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

$$(\mathbf{p})^{\bullet(3)} = (\mathbf{p})^{\bullet(2)}.$$

Here the closure computation stops. Next, in the generalization phase, since $\mathbf{p}^{\bullet(2)}$ is idempotent, we compute:

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

So, the following set results:

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

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

applying closure again, we obtain some additional results:

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

The last element is idempotent but applying generalization does not generate anything new. Thus the final set T has been reached. The reader may reconsider the source code to verify that we have indeed obtained tight bounds for the loop. Note that $\mathbf{p}$ itself is not idempotent, but it is iterative. Our improved version of the Closure Algorithm (Section 6) would generalize $\mathbf{p}$, adding $\mathbf{p}^\tau = \langle x_1 + \tau x_2,\ x_2 + \tau x_3,\ x_3,\ x_3\rangle$. This result is subsumed by $\mathbf{q}$ above, showing that the final sets of bounds resulting from the two algorithms are equivalent.

Our new algorithm is simpler than the old one as it performs only one round of generalization—it does not "iterate until fixed-point" like SOLVE. This is a simplification, since we do not have to consider the issue of correctly generalizing MPs that already include τ. It can be proved that every idempotent AMP has an iterative kernel; but many AMPs which are not idempotent also have an iterative kernel. One could say that the new algorithm is more eager to generalize. The motivation for this change was to elimnate the idempotence condition: in order to judge that an AMP is idempotent we need full information about it. Our PSPACE algorithm operates with very partial information and we derive its soundness by establishing that what it generalizes is an iterative fragement of *some* AMP in the closure, which we cannot guarantee to be idempotent. Thus, we made this change in order to justify the PSPACE algorithm. Nonetheless, we believe that our current variant may be more efficient in practice than the old one, mostly because of the elimination of the outer "iterate until fixed point" loop. It would be our choice, should we prefer to implement a closure-based algorithm (perhaps because determinization of the PSPACE algorithm, which is non-deterministic, would result in a deterministic algorithm worse in practice).

A.3. **Equivalence of the algorithms.** We now prove that our revised algorithm gives results equivalent to the former version. Basically, we prove subsumption of the two sets of results in each other.

Lemma A.7. *Every $\mathbf{p}\in$ SOLVE$(\mathcal{S})$ is subsumed by a member of $LC(\mathcal{S})$.*

Proof. We prove inductively, based on the structure of the algorithm, that the set T, computed by SOLVE, is subsumed by $LC(\mathcal{S})$.
Base case: $T = \mathcal{S}$ is, of course, contained in $\mathcal{S}$.
Induction Step 1: Setting T to $Cl(T)$ preserves the relation because $LC(\mathcal{S})$ is closed under composition, as one can clearly see from its definition.
Induction Step 2: This step adds $\mathbf{p}^\tau$ to T for idempotent AMPs $\mathbf{p}$. We shall now prove that $\mathbf{p}^\tau \lhd LC(\mathcal{S})$. We consider two sub-cases. If $\mathbf{p}$ is τ-free, we infer that $\mathbf{p}\in Cl(\mathcal{S})$. Then it is clearly also in $LC(\mathcal{S})$. If $\mathbf{p}$ has τ's, we observe that $\mathbf{p}[1/\tau]$ is what one gets by performing the same *composition* operations that yielded $\mathbf{p}$, but skipping the generalizations. So $\mathbf{p}[1/\tau]\in Cl(\mathcal{S})\subseteq LC(\mathcal{S})$. It is easy to see that $\mathbf{p}[1/\tau]$ is idempotent as well; thus

$(\!|\mathbf{p}[1/\tau]|\!)^\tau$ is formed by function *Gen*. Also, by IH, there is $\mathbf{q} \in LC(\mathcal{S})$ such that $\mathbf{p} \lhd \mathbf{q}$. Using a few lemmas proved below, we conclude that $\mathbf{q} \bullet (\!|\mathbf{p}[1/\tau]|\!)^\tau \bullet \mathbf{p}[1/\tau] \rhd \mathbf{p}^\tau$; specifically,

$$\mathbf{q} \bullet (\!|\mathbf{p}[1/\tau]|\!)^\tau \bullet \mathbf{p}[1/\tau] \rhd \mathbf{p} \bullet (\!|\mathbf{p}[1/\tau]|\!)^\tau \bullet \mathbf{p}[1/\tau] \qquad \text{(Lemma A.8)}$$
$$\rhd \mathbf{p} \bullet (\mathbf{p}[1/\tau])^\tau \qquad \text{(Lemma A.11)}$$
$$\rhd \mathbf{p}^\tau \qquad \text{(Lemma A.9)}$$

The above product is in $LC(\mathcal{S})$, as we have established that each of its factors is present in the set $Gen(Cl(\mathcal{S}))$; we conclude $\mathbf{p}^\tau \lhd LC(\mathcal{S})$. □

Lemma A.8. *For $p, q \in \tau\mathtt{APol}$ and $\mathbf{r} \in \tau\mathtt{AMPol}$, if $\mathbf{q} \rhd \mathbf{p}$ then $\mathbf{q} \bullet \mathbf{r} \rhd \mathbf{p} \bullet \mathbf{r}$.*

Proof. Let $\mathfrak{m}$ be a monomial of $p \bullet \mathbf{r}$. It is obtained by substituting components $\mathbf{r}[i]$ into variables x_i of a monomial $\mathfrak{p}$ of p. Now, q is known to contain some multiple $\mathfrak{q}$ of $\mathfrak{p}$, and clearly substituting $\mathbf{r}$ in $\mathfrak{q}$ yields a multiple of $\mathfrak{m}$. □

Lemma A.9. *Let $\mathbf{p} \in \tau\mathtt{AMPol}$ be idempotent and neat. Then $\mathbf{p} \bullet (\mathbf{p}[1/\tau])^\tau \rhd \mathbf{p}^\tau$.*

Proof. We prove, by induction on i, that the inequality holds for the first i components of the respective AMPs; we write this using "array slice notation" as $(\mathbf{p} \bullet (\mathbf{p}[1/\tau])^\tau)[1..i] \rhd \mathbf{p}^\tau[1..i]$.

Base case: Since $\mathbf{p}$ is neat we know that $\mathbf{p}[1] = x_1$. It is also clear that $(\mathbf{p}[1/\tau])^\tau[1] = x_1$. Consequently, $\mathbf{p} \bullet (\mathbf{p}[1/\tau])^\tau[1] = x_1 = \mathbf{p}^\tau[1]$.

Induction step: let $i > 1$. If $i \notin \mathrm{SD}(\mathbf{p})$, then $\mathbf{p}[i]$ only depends on variables x_j with $j < i$. This allows us to use the induction hypothesis:

$$\mathbf{p} \bullet (\mathbf{p}[1/\tau])^\tau[i] = \mathbf{p} \bullet \mathbf{p} \bullet (\mathbf{p}[1/\tau])^\tau[i] \qquad \text{by idempotence}$$
$$= \mathbf{p}[i] \bullet \mathbf{p} \bullet (\mathbf{p}[1/\tau])^\tau$$
$$\rhd \mathbf{p}[i] \bullet \mathbf{p}^\tau \qquad \text{by IH}$$
$$\rhd \mathbf{p}[i] \bullet \mathbf{p} = \mathbf{p}[i] = \mathbf{p}^\tau[i] \qquad \text{by Def. A.4}.$$

If $i \in \mathrm{SD}(\mathbf{p})$, then $\mathbf{p}[i] = x_i + p_i(\mathbf{x}, \tau)$ where p_i only depends on variables x_j with $j < i$. For conciseness, define $\mathbf{q}$ to be $\mathbf{p} \bullet (\mathbf{p}[1/\tau])^\tau$. Now by idempotence of $\mathbf{p}$:

$$(\mathbf{p} \bullet (\mathbf{p}[1/\tau])^\tau)[i] = (\mathbf{p} \bullet \mathbf{p} \bullet (\mathbf{p}[1/\tau])^\tau)[i]$$
$$= \mathbf{p}[i] \bullet \mathbf{q}$$
$$= \mathbf{q}[i] + p_i \bullet \mathbf{q}$$
$$\rhd (\mathbf{p}[1/\tau])^\tau[i] + p_i \bullet \mathbf{q} \qquad \text{since } \mathbf{p}[i] \rhd x_i$$
$$\rhd x_i + \tau(\mathbf{p}[1/\tau])[i]'' + p_i \bullet \mathbf{q} \qquad \text{by Def. A.4}$$
$$\rhd x_i + \tau\mathbf{p}[i]'' + p_i \bullet \mathbf{q} \qquad \text{since } \mathbf{p}[i]'' \text{ is } \tau\text{-free}$$
$$= x_i + p_i \bullet \mathbf{q} + \tau\mathbf{p}[i]''$$
$$\rhd x_i + p_i \bullet \mathbf{p}^\tau + \tau\mathbf{p}[i]'' \qquad \text{by IH}$$
$$\rhd x_i + p_i \bullet \mathbf{p} + \tau\mathbf{p}[i]''$$
$$= (\mathbf{p} \bullet \mathbf{p})[i] + \tau\mathbf{p}[i]''$$
$$\rhd \mathbf{p}[i] + \tau\mathbf{p}[i]'' \quad = \quad \mathbf{p}^\tau[i].$$

□

Lemma A.10. *Assume that $\mathbf{p} \in \mathtt{AMPol}$ is taken from the closure set of a polynomially-bounded SDL. If $\mathbf{p}$ is idempotent, then $(\!|\mathbf{p}|\!) \bullet \mathbf{p} = \mathbf{p}$.*

Proof. $G(\mathbf{p})$ is quasi-acyclic by Lemma A.3 above. Thus, w.l.o.g., it is neat (that is, the variables are indexed to match the DAG order). Clearly, $(\!|\mathbf{p}|\!) \sqsubseteq \mathbf{p}$ and consequently $(\!|\mathbf{p}|\!) \bullet \mathbf{p} \sqsubseteq \mathbf{p} \bullet \mathbf{p} = \mathbf{p}$. Now, assume that equality does not hold. Let $\mathfrak{m}$ be a lexicographically biggest monomial[4] that appears in $\mathbf{p}$ and is missing in $(\!|\mathbf{p}|\!) \bullet \mathbf{p}$. Consider the calculation of $\mathbf{p} \bullet \mathbf{p}$. It generates $\mathfrak{m}$ by substituting monomials from the right-hand instance of $\mathbf{p}$ for the variables of a certain monomial $\mathfrak{n}$ taken from the left-hand instance. Because $\mathbf{p}$ is neat, substitution replaces a variable x_i either by x_i itself or by a lexicographically-smaller monomial. If all the variables of $\mathfrak{n}$ could be replaced by themselves, then $\mathfrak{n} = \mathfrak{m}$ would be a iterative monomial and therefore present in $(\!|\mathbf{p}|\!)$ and in $(\!|\mathbf{p}|\!) \bullet \mathbf{p}$. We conclude that some variables in $\mathfrak{n}$ are not self-dependent. It is further easy to see that $\mathfrak{m}$ is lexicographically smaller than $\mathfrak{n}$. By the choice of $\mathfrak{m}$, it follows that $\mathfrak{n}$ appears in $(\!|\mathbf{p}|\!) \bullet \mathbf{p}$. But then $\mathfrak{m}$ is present in $((\!|\mathbf{p}|\!) \bullet \mathbf{p}) \bullet \mathbf{p}$ which equals $(\!|\mathbf{p}|\!) \bullet \mathbf{p}$ (by associativity of composition and idempotence of $\mathbf{p}$). A contradiction has been shown. $\qquad\square$

Lemma A.11. *Assume that* $\mathbf{p} \in \mathtt{AMPol}$ *is taken from the closure set of a polynomially-bounded SDL. If* $\mathbf{p}$ *is idempotent, then* $(\!|\mathbf{p}|\!)^\tau \bullet \mathbf{p} \rhd \mathbf{p}^\tau$.

Proof. We already know that every monomial $\mathfrak{m}$ of $\mathbf{p}$ appears in $(\!|\mathbf{p}|\!) \bullet \mathbf{p}$ and therefore either $\mathfrak{m}$ or $\tau\mathfrak{m}$ appears in $(\!|\mathbf{p}|\!)^\tau \bullet \mathbf{p}$. So we have covered all the monomials of $\mathbf{p}^\tau$ except those affected by generalization, namely those included in the polynomials $\mathbf{p}[i]''$. By definition of $\mathbf{p}[i]''$, they are self-dependent and also appear in $(\!|\mathbf{p}|\!)$. In fact, $(\!|\mathbf{p}|\!)[i] = x_i + \mathbf{p}[i]''$ and $(\!|\mathbf{p}|\!)^\tau[i] = x_i + \tau\mathbf{p}[i]''$. Substituting, we find that $((\!|\mathbf{p}|\!)^\tau \bullet \mathbf{p})[i] \sqsupseteq x_i + \tau\mathbf{p}[i]''$. The statement of the lemma follows. $\qquad\square$

The proof of Lemma A.7 is now complete, and we move on to the converse subsumption. We will begin this time with an auxiliary result.

Lemma A.12. *Let* $\mathcal{S}$ *be a polynomially-bounded SDL, and let* $\mathbf{p} \in Cl(\mathcal{S})$. *Then there is a* $k > 0$ *such that* $\mathbf{p}^{(k)}$, *the AMP representation of a k-fold application of* $\mathbf{p}$, *is idempotent.*

Proof. This follows from the fact that $Cl(\mathcal{S})$ is finite (an easy consequence of polynomial boundedness). Finiteness implies that the sequence $\mathbf{p}^{(i)}$ must repeat elements. If $\mathbf{p}^{(i)} = \mathbf{p}^{(i+k)}$, $k > 0$, then for any $j \geq i$, $\mathbf{p}^{(j)} = \mathbf{p}^{(j+k)}$. Assuming (w.l.o.g.) $k \geq i$, we have $\mathbf{p}^{(k)} = \mathbf{p}^{(2k)}$, so $\mathbf{p}^{(k)}$ is idempotent. $\qquad\square$

Lemma A.13. *Let* $\mathcal{S}$ *be a polynomially-bounded SDL, and let* $\mathbf{q}$ *be an* iterative *fragment of any* $\mathbf{p} \in Cl(\mathcal{S})$. *Then*
(1) $\mathbf{q}^{(i)}$ *is also iterative;*
(2) $\mathbf{q}^{(i)} \sqsubseteq \mathbf{p}^{(i)}$;
(3) $\mathbf{q}^{(i)} \sqsubseteq \mathbf{q}^{(i+1)}$.

Proof. (1) It easy to see that if $i \in \mathrm{SD}(\mathbf{q})$, then $i \in \mathrm{SD}(\mathbf{q}^{(i)})$ for any i. On the other hand, as each component of $\mathbf{q}$ only depends on the self-dependent variables, the same is true for $\mathbf{q}^{(i)}$; thus $\mathbf{q}^{(i)}$ is iterative. In fact, it has the same set of self-dependent variables.
(2) This is true because $\mathbf{q} \sqsubseteq \mathbf{p}$ implies $\mathbf{q} \bullet \mathbf{r} \sqsubseteq \mathbf{p} \bullet \mathbf{r}$ for any $\mathbf{r}$.

---

[4]Monomials are ordered lexicographically by viewing $x_1^{d_1} \ldots x_n^{d_n}$ as $\langle d_n, \ldots, d_1 \rangle$. The reversal of the list has the effect that dropping the highest-numbered variable makes a monomial smaller, even if other degrees go up.

(3) Let $\mathfrak{m}$ be a monomial of $\mathbf{q}^{(i)}$. When we compose $\mathbf{q}^{(i)}$ over $\mathbf{q}$ (to obtain $\mathbf{q}^{(i+1)}$), we substitute $\mathbf{q}[j]$ for each variable x_j in $\mathfrak{m}$. Since $j \in \text{SD}(\mathbf{q})$, we have $x_j \sqsubseteq \mathbf{q}[j]$. Thus, among the monomials resulting from this substitution, there is a monomial identical to $\mathfrak{m}$.

$\square$

Lemma A.14. *Every $\mathbf{p} \in LC(\mathcal{S})$ is subsumed by a member of* SOLVE$(\mathcal{S})$.

Proof. We proceed by three steps, following the definition of LC. First, if $\mathbf{p} \in Cl(\mathcal{S})$ then it is clearly also in SOLVE$(\mathcal{S})$. Next, consider $(\!|\mathbf{p}|\!)^\tau$ for $\mathbf{p} \in Cl(\mathcal{S})$. By Lemma A.12, choose k such that $\mathbf{p}^{(k)}$ is idempotent. Hence $(\mathbf{p}^{(k)})^\tau \in$ SOLVE$(\mathcal{S})$. Applying Lemma A.13 to $(\!|\mathbf{p}|\!)$, we obtain:

$$(\!|\mathbf{p}|\!) \sqsubseteq (\!|\mathbf{p}|\!)^{(k)} \sqsubseteq \mathbf{p}^{(k)}$$

Moreover, it is not hard to extend these inequalities to

$$(\!|\mathbf{p}|\!)^\tau \sqsubseteq ((\!|\mathbf{p}|\!)^{(k)})^\tau \sqsubseteq (\mathbf{p}^{(k)})^\tau$$

Thus $(\!|\mathbf{p}|\!)^\tau$ is subsumed by a member of SOLVE$(\mathcal{S})$.

Finally, generalized AMPs are subjected to composition-closure again, but its results will certainly be subsumed by corresponding compositions in SOLVE$(\mathcal{S})$, as the latter is closed under composition, and composition respects the subsumption order. $\square$

REFERENCES

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

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

[3] A. M. Ben-Amram and G. W. Hamilton. Tight polynomial worst-case bounds for loop programs. *Logical Methods in Computer Science*, 16(2), 2020.

[4] A. M. Ben-Amram, N. D. Jones, and L. Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *LNCS*, pages 67–76. Springer, 2008.

[5] A. M. Ben-Amram and L. 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.

[6] A. M. Ben-Amram and A. Pineles. Flowchart programs, regular expressions, and decidability of polynomial growth-rate. In G. Hamilton, A. Lisitsa, and A. 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.

[7] M. Braverman. Termination of integer linear programs. In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA*, volume 4144 of *Lecture Notes in Computer Science*, pages 372–385. Springer, 2006.

[8] M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4):13:1–13:50, Aug. 2016.

[9] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. In *Proceedings of the ACM SIGPLAN 2015 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2015.

[10] A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In J. Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.

[11] 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.

[12] M. Hark, F. Frohn, and J. Giesl. Polynomial loops: Beyond termination. In E. Albert and L. Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 279–297. EasyChair, 2020.

[13] M. Hosseini, J. Ouaknine, and J. Worrell. Termination of linear loops over the integers. In *46th International Colloquium on Automata, Languages, and Programming, ICALP*, 2019.

[14] E. Hrushovski, J. Ouaknine, A. Pouly, , and J. 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. ACM, 2018.

[15] N. D. Jones. *Computability and Complexity From a Programming Perspective*. Foundations of Computing Series. MIT Press, 1997.

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

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

[18] L. Kristiansen and K.-H. Niggl. On the computational complexity of imperative programming languages. *Theor. Comp. Sci.*, 318(1-2):139–161, 2004.

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

[20] P. López-García, L. Darmawan, M. Klemen, U. Liqat, F. Bueno, and M. V. Hermenegildo. Interval-based resource usage verification by translation into horn clauses and an application to energy consumption. *TPLP*, 18(2):167–223, 2018.

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

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

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

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

[25] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, New York, 1986.

[26] M. Sinn, F. Zuleger, and H. Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 59(1):3–45, 2017.

[27] J. M. Steele. *The Cauchy-Schwarz Master Class: An Introduction to the Art of Mathematical Inequalities*. Cambridge University Press, 2004.

[28] A. Tiwari. Termination of linear programs. In *Computer Aided Verification*, pages 70–82, 2004.

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