Flowchart Programs, Regular Expressions, and Decidability of Polynomial Growth-Rate

Amir M. Ben-Amram
School of Computer Science, Tel-Aviv Academic College, Israel
amirben@mta.ac.il

Aviad Pineles
paviad2@gmail.com

We present a new method for inferring complexity properties for a class of programs in the form of flowcharts annotated with loop information. Specifically, our method can (soundly and completely) decide if computed values are polynomially bounded as a function of the input; and similarly for the running time. Such complexity properties are undecidable for a Turing-complete programming language, and a common work-around in program analysis is to settle for sound but incomplete solutions. In contrast, we consider a class of programs that is Turing-incomplete, but strong enough to include several challenges for this kind of analysis. For a related language that has *well-structured* syntax, similar to Meyer and Ritchie's LOOP programs, the problem has been previously proved to be decidable. The analysis relied on the compositionality of programs, hence the challenge in obtaining similar results for flowchart programs with arbitrary control-flow graphs. Our answer to the challenge is twofold: first, we propose a class of loop-annotated flowcharts, which is more general than the class of flowcharts that directly represent structured programs; secondly, we present a technique to reuse the ideas from the work on structured programs and apply them to such flowcharts. The technique is inspired by the classic translation of non-deterministic automata to regular expressions, but we obviate the exponential cost of constructing such an expression, obtaining a polynomial-time analysis. These ideas may well be applicable to other analysis problems.

Keywords: polynomial time complexity, cost analysis, program transformation

1 Introduction

Devising algorithms that deduce complexity properties of a given program is a classic problem of program analysis [33, 30, 23], and has received considerable attention in recent years. Ideally, a static-analysis tool will warn us in compilation-time whenever our program fails a complexity specification. For instance, we could be warned of algorithms whose running time is not polynomially bounded, or algorithms that compute super-polynomially large values.

While practical tools typically attempt to infer explicit bounds, as a theoretical problem it is sufficiently challenging to look at the classification problem—polynomial or not. Since deciding such a property for all programs in a Turing-complete language is impossible, there are two ways to approach the problem. There is much work (as the above-cited) which targets a Turing-complete language and settles for an incomplete solution. Other works investigate the decidability of the problem in restricted languages. In [6], the problem of *polynomial boundedness* is shown decidable (moreover, in PTIME) for a "core" imperative language L_{BJK} with a bounded loop command[1]. The language of [6] only has numeric variables, with the basic operations of addition and multiplication. Figure 1 shows the syntax of the language; regarding its semantics, it is important that the language has a non-deterministic choice instead of conditionals, and bounded loops ("do at most n times") which are also non-deterministic (for further details see [6]).

[1] Since we consider explicitly-bounded loops, the problem of classifying the time complexity of the program is equivalent to the problem of classifying growth rates of variables.

G.W. Hamilton, A. Lisitsa, A.P. Nemytykh (Eds): VPT 2016
EPTCS 216, 2016, pp. 24–49, doi:10.4204/EPTCS.216.2

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

Figure 1: Syntax of the structured core language L_{BJK} of [6].

The design of this core language is influenced by the approach “abstract and conquer.” This means that while we are solving a subproblem, this could be used also as a component in a partial solution to the “real” problem (handling a realistic language) by using it as a *back end*, assuming that some *front end* analyser translates source programs, in a conservative fashion, into core programs. Abstraction (e.g., making branches non-deterministic by simply hiding the conditionals) is clearly doable, while sophisticated *static analysis* technology may allow the construction of more precise front ends. Importantly, current static analysis techniques allow for computing *loop bounds* [1, 2, 17] and thus transforming an arbitrary loop into our bounded-loop form. These intentions motivate the choice of including non-determinism in our core language.

Another reason for making the core language non-deterministic is the desire to find a decidable problem. In fact, including precise test as conditionals (say, testing for equality of variables) is easily seen to lead to undecidability, and [7] shows that having a deterministic loop (with precise iteration count, rather than just a bound) breaks decidability, too.

To carry this research forward, we ask: in what ways can the core language be extended, while maintaining decidability? In this paper we consider an extension that involves the control structure of the program: we extend from nested loop programs to *unstructured* (“flowchart”) programs. The details of this extension are motivated, as we will explain, by looking at the way certain complexity analysers for realistic programs work, trying to break the process into a front-end and a back-end, and make the back-end decision problem solvable. Theoretically, the result is a polynomial-time decidability result for a language *FC* that strictly extends L_{BJK}. Our analysis algorithm for the *FC* language is obtained by a technique that is more general than the particular application. The technique arose from trying to adapt the analysis of [6]. The challenge was that the analysis was compositional and essentially based on the well-nested structure of the core language. How can such an analysis be performed on arbitrary control-flow graphs? Our solution is guided by the standard transformation of a finite automaton (NFA) to a regular expression. A regular expression is a sort of well-structured program. In order to make this as explicit as possible, we define a programming language which is written in a syntax similar to regexp’s. However, semantically it is an extension of L_{BJK} (thus, as a by-product, we find a structured language with precisely the expressivity of *FC*). We show that by composing the NFA-to-regexp transformation with an analysis of the structured language (a natural extension of the analysis of [6]) we obtain an analyzer for *FC* that runs in polynomial time (somewhat surprising, as the general construction of regular expressions is exponential-time). We believe that many static analysis algorithms that process general control-flow graphs may implicitly use the same approach, but that we are first to make it explicit.

The rest of this paper is structured as follows: first we motivate, then formally define, our concept of loop-annotated flowcharts. Then, we define the regexp-like language, called Loop-Annotated Regular Expressions (LARE, for short), and explain how we translate flowcharts to LARE. Finally we give the algorithm that analyzes LARE. Some of the more technical definitions and proofs, and in particular, the correctness proofs of the analysis algorithm, which are long and complex, are deferred to appendices.

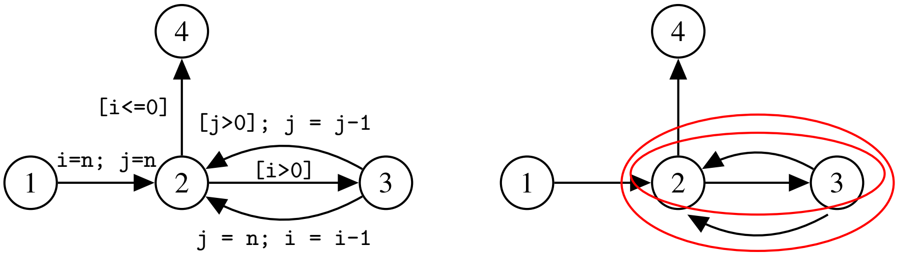
2 Loop-Annotated Flowchart Programs: Motivation

The algorithm presented in this paper analyzes programs in a language which we call *Loop-Annotated Flowchart Programs*. This language has the important features that (i) program form is an arbitrary control-flow graph with instructions on arcs; (ii) variables hold non-negative integers; (iii) the instruction set is highly limited; (iv) information about loop bounds is supplied as "annotations," presented in the next section. The design results from two goals, on one hand we are looking for a decidable case; on the other hand we are motivated by looking at tools that analyze real-world programs. We next explain this motivation informally, focusing on the tool [2]. The tool generates from the input program a control-flow graph, where arcs carry both guards and updates; for example, the C program shown in Figure 2 (a) might be represented as in Figure 2 (b).

The tool uses a linear-programming based algorithm to find *ranking functions* for parts (subgraphs) of the control-flow graph. A ranking function is a combination of the program variables that is non-increasing throughout the subgraph while on certain arcs it is non-negative and strictly decreasing. This implies a bound on the number transitions (program steps) that correspond to the latter kind of arcs. If there remains a strongly-connected subgraph for which an iteration bound is not implied, a ranking function for the subgraph is necessary. For example, in Figure 2, the algorithm of [2] reports the function $\mathtt{i}+\mathtt{n}$, holding throughout the strongly-connected component of the graph, and strictly decreasing on the bottom arc from (3) to (2); this implies an upper bound of $2\mathtt{n}$ (the initial value of this function) on the number of times one can take this arc while staying within the component. The algorithm next excludes this arc and finds the ranking function $\mathtt{i}+\mathtt{j}$ for the remaining cycle (note that $\mathtt{j}$ would have sufficed; ranking functions are not unique). The second function implies the bound $\mathtt{i}+\mathtt{n}$ on the number of iterations through that cycle (in terms of the values of variables on entrance to this loop). Thus, the analysis decomposes the program into two nested loops (Figure 2 (c)), also establishing, for each loop, an iteration bound in terms of variables that do not change in the loop. Note that this decomposition is not evident from the program text (where there is just one loop construct), and is not unique, in general. Moreover, it depends on semantic analysis and cannot be determined just from the graph structure. *Our focus in this work is on the analysis of a program that is already decomposed and annotated with bounds.* Thus the above discussion should be understood purely as motivation; we do not deal with the art of abstracting source programs in realistic languages. Our decidability results only concern questions about a loop-annotated flowcharts. We note that this does not translate to a decidability result regarding the source programs, since the process of identifying and annotating loops in not well defined (there may be many correct annotations for given problem); moreover, undecidable problems may be encoded into this phase. For simplicity, in our input language, loop bounds will always be specified as a single variable. This is not a restriction, since if we are considering a given program where a loop bound is an expression in program variables (e.g., a sum, as above) we can generate an auxiliary variable to store the loop bound.

Another motivation to handle flowcharts is the usage of abstraction refinement techniques, where the control-flow graph representing a program is expanded to represent additional properties of the current or past states [29]. Even when starting from a structured program, the structure of the resulting control-flow graph might not correspond to that of the source program. As an example, Pineles [28] extends the language treated here with reset assignments `X := 0`. In [4], such assignments were added to L_{BJK}, which required a non-trivial addition to the analysis algorithm. In contradiction, [28] handles the extension by program transformation: we refine the control-flow graph with respect to the history of resets. This allows for eliminating the resets, so the growth-rate analysis does not have to handle them. However, it also changes the graph structure so it no longer corresponds to the original, structured program.

```
assume (n>=0);
i = n;
j = n;
while (i > 0) {
  if (j>0) {
    j = j-1;
  } else {
    j = n;
    i = i-1;
  }
}
```



(a) (b) (c)

Figure 2: Illustration of the decomposition of a flowchart (coming from a C program) into nested loops. Note that this is not a program in our language but in a language of guarded assignments, which is, hopefully, self-explanatory. We only present it to illustrate the considerations in Section 2.

3 Loop-Annotated Flowchart Programs

3.1 Program form and informal semantics

Data Our programs operate on a finite (per program) set of variables, each holding a number. The variables are typically denoted by $\mathtt{X}_1,\dots,\mathtt{X}_n$, and a state of the program's storage is thus a vector $(x_1,\dots,x_n)$. The initial contents of the variables are regarded as input to the program, and their final value as output. We assume that the only type of data is nonnegative integers. More generality is possible, e.g., considering signed integers. This will be left out of the present paper.

Instructions Atomic commands, or *instructions*, modify the values of variables. A *core instruction set* for our work, corresponding to the instructions of L_{BJK}, consists of the instructions

    X := Y,   X := Y + Z,   X := Y * Z,   skip,   X := **

where X, Y and Z are variable names. The skip instruction is a no-op and could be replaced with X := X. The last form means "set X to an unknown value," which of course will have no upper bound in terms of the input; it is included because it is useful in abstracting realistic programs. We remark that, with an eye to abstraction of real programs, we may also use the *weak assignment* forms

$\mathtt{X} :\leq \mathtt{Y}, \quad \mathtt{X} :\leq \mathtt{Y} + \mathtt{Z}, \quad \mathtt{X} :\leq \mathtt{Y} * \mathtt{Z},$

which set X to a non-determined value between 0 and the right-hand side. A detailed semantics for the instructions may be found in [8].

Definition 3.1. A *loop-annotated flowchart-program* (abbreviated to "a program," when context permits) p consists of:

- A finite set of variables $\mathtt{X}_1,\dots,\mathtt{X}_n$.
- A *control-flow graph* (CFG) which is a directed graph $G_{\mathtt{p}} = (\mathtt{Loc}_{\mathtt{p}}, \mathtt{Arc}_{\mathtt{p}})$. The nodes $\mathtt{Loc}_{\mathtt{p}}$ are called *locations*.
- A map Inst() from CFG arcs to instructions (we sometimes abuse the term and refer to arcs as "instructions;" the meaning should be clear from context).

- A non-empty set of entry nodes (nodes with no predecessors), P_{entry}, and a non-empty set of terminal nodes (nodes with no successors), P_{term} (while program units in most languages have a single entry point, the generality of allowing a set of entry nodes is useful in our algorithm).
- A *loop tree*, described next.

Definition 3.2. A loop tree for program $\mathtt{p}$ is a set $\mathscr{L}_{\mathtt{p}}$ of subsets of $\mathtt{Arc}_{\mathtt{p}}$, called loops, which form a rooted tree under a relation which we call "nesting". Loops nested in $L \in \mathscr{L}_{\mathtt{p}}$ are required to be disjoint, strict subsets of L. The root is the whole CFG. With each non-root $L \in \mathscr{L}_{\mathtt{p}}$ is associated a variable, called its *bound*; technically, we denote by $\mathtt{Bound}(L)$ this variable's index, so that the bound is $\mathtt{X}_{\mathtt{Bound}(L)}$. In addition, a "cut set" $\mathtt{Cutset}(L)$ is provided, which consists of arcs that belong to L but not to its descendants. In a valid program, if $a \in L$, instruction $\mathtt{Inst}(a)$ must not modify $\mathtt{X}_{\mathtt{Bound}(L)}$. In addition, every cycle C in the CFG must include an arc from the cutset of the lowest loop L containing C.

For intuition, we might think of cut-set instructions as maintaining a counter which ensures that flow passes through such arcs at most $\mathtt{X}_{\mathtt{Bound}(L)}$ times. We make the assumption—without loss of generality—that cut-set arcs represent transitions that do not alter the values of variables; this condition can always be achieved by adding arcs to the graph. We shall use the symbol ¢ to mark these arcs in a diagram. Note that the notion of "a loop" is very flexible. It is a set of arcs, in particular is not required to be strongly connected (though this would be the natural situation). One benefit of this flexibility is that this loop information can persist through program transformations.

Semantics of programs is mostly straight-forward. The loop information is interpreted as follows: once a loop L is entered, at most B cutset arcs of L may be traversed before the loop is exited, where B is the value of the loop bound. For full details of the definition, see Appendix A.

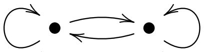
3.2 Growth-rate analysis

The *polynomial-bound analysis problem* is to find, for a given program, which output variables are bounded by a polynomial in the initial values of all variables. This is the problem we focus on; the following variants can be easily reduced to it: (1) *feasibility*—find whether all the values generated throughout any computation (rather than outputs only) are polynomially bounded in the initial values. (2) *Polynomial running time*—find whether the worst-case time complexity of the program (i.e., number of steps) is polynomial.

3.3 Flowcharts versus structured programs

Flowcharts are more expressive than structured programs[2] over the same instruction set, since they can have complex "non-structured" control flow; we now propose a formal argument to support this claim. It is natural to say that a flowchart F is *strongly equivalent* to a program P if they have the same set of traces, a trace being the sequence of atomic commands performed in a computation. More precisely, let $\mathscr{T}_F(\vec{x})$ (respectively $\mathscr{T}_P(\vec{x})$) be the set of traces of the flowchart (resp. structured program) when the initial state is $\vec{x}$; say that F and P are *equivalent* if $\mathscr{T}_F(\vec{x}) = \mathscr{T}_P(\vec{x})$ for all $\vec{x}$. This clearly implies $\bigcup_{\vec{x}} \mathscr{T}_F(\vec{x}) = \bigcup_{\vec{x}} \mathscr{T}_P(\vec{x})$, an equality that we call *weak equivalence*. This last set is a regular language over the alphabet of instructions. Note that by ignoring the semantics of instructions and considering them as abstract symbols, a flowchart is reduced to a finite automaton (NFA). A structured program

[2] This means programs sharing the structure of L_{BJK}: they include loop commands, commands that branch into two sub-commands, and atomic commands, and have the single-entry-single-exit property.

can be directly represented by a regular expression over the same alphabet. By [15], a flowchart that includes, in a single loop, the 2-node digraph  (labeled with distinct instructions) has the property that any weakly-equivalent structured program has star-height of 2 at least (i.e., it includes the pattern $(...(...)^*...)^*$). Now, the flowchart program, as it has a single loop, has linear running time, whereas the structured program will have a worst-case running time quadratic at least. Hence, they cannot be strongly equivalent. In other words, such an annotated flowchart has no equivalent structured program (though both have polynomial running time; indeed, a FC-to-L_{BJK} transformation that preserves polynomiality follows from our results, but it has another drawback—an exponential blow-up in size).

4 Loop-Annotated Regular Expression Programs

In this section we introduce *Loop-Annotated Regular Expression Programs*, LARE, which is a program form designed to exploit the analogy of structured programs to regular expressions and flowcharts to automata. This language is based on regular expressions but it represents programs in a superset of L_{BJK}, and is endowed with computational semantics. However, occasionally we do refer to the standard semantics of a regular expression as describing a set of strings. To emphasize the former interpretation, we may use the term "program" instead of "expression".

Definition 4.1. The class of Loop-Annotated Regular-Expression Programs LARE is constructed as follows. First, *atomic* expressions are:

1. A set Σ of *basic instructions*, which serve as *symbols* of the expressions.
2. An additional, special symbol, ¢, called the cut-arc symbol.
3. The empty-string constant ε.

Secondly, expressions (atomic or not) can be combined in the following ways

1. Concatenation: EF where E, F are LARE expressions.
2. Alternation (or non-deterministic choice), $E|F$ where E, F are expressions.
3. Iteration: E^* where E is an expression.
4. Loop annotation: if E is an expression and $\mathtt{X}_\ell$ a variable, we can form the expression $[_\ell E]$, provided that E does not include any assignment to $\mathtt{X}_\ell$. The pair of syntactic elements $[_\ell$ and $]$ is called *loop brackets*.

If we ignore the loop brackets, an LARE expression is just a regular expression which generates a set of strings over Σ. We use $[\![E]\!]_{str}$ to denote this set. The loop brackets have a role in defining the computational semantics of LARE. First, we state a validity requirement: In a *well formed* expression, every iteration construct E^* must appear inside some pair of loop brackets. Moreover, every string that E generates must include a ¢. E.g., $(a(¢b)^*d)^*$ is invalid, because the expression $(a(¢b)^*d)$ generates the ¢-free string ad.

Parentheses will be used to indicate expression structure, as usual (also using standard precedence and associativity for operators).

Semantically, an LARE program executes a (non-deterministically chosen) sequence of instructions that, viewed as a string over Σ, belongs to the language $[\![E]\!]_{str}$. However, the sequence also has to satisfy the loop bounds, in the sense that the number of ¢'s encountered in a subsequence generated by expression $[_\ell E]$, discounting those in the scope of any inner pair of brackets, is bounded by $\mathtt{X}_\ell$. For full details of the definition, see Appendix A.

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

$[_4(\text{¢}(\boxed{\mathtt{X}_3 := \mathtt{X}_1 + \mathtt{X}_2}\boxed{\mathtt{X}_2 := \mathtt{X}_3}))^*]$

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

$[_4(\text{¢}(\boxed{\mathtt{X}_2 := \mathtt{X}_1 + \mathtt{X}_4} \mid \boxed{\mathtt{X}_2 := \mathtt{X}_2 + \mathtt{X}_4}))^*]$

Figure 3: programs in L_{BJK} and their expression as LARE, to illustrate the latter.

The loop brackets make possible the correspondence between LARE and flowcharts. We take the following result to be intuitive, and we state it without details:

THEOREM 4.2. *An LARE program can be represented as a semantically-equivalent loop-annotated flowchart over the same set of instructions.*

The semantic equivalence between the two program forms is one of trace semantics—as long as program locations (nodes in the flowchart graph) are disregarded in the traces (this is also formalized in Appendix A).

It should also be pretty obvious that the *structured core language* L_{BJK} can be embedded in LARE—this is really just a change of syntax, where, importantly, a loop `loop` $\mathtt{X}_\ell$ `{C}` becomes $[_\ell(\text{¢}E)^*]$, where E represents `C`. Hence, in this translation, the iteration operator (star), cut-arc symbol and loop brackets always work together. Generally, this is not required in LARE, allowing more flexibility, which is required for their equivalence to flowcharts.

5 Translating Flowchart Programs into LARE

We now arrive at the translation of a flowchart program into LARE, which was the point of introducing it. This translation is not difficult, but since our analysis rests on it, we give it in some detail. Our algorithm is based on the classical *Rip* algorithm to transform an NFA into a regular expression ([32, Sec. 1.3]), which we extend in order to handle loops correctly. We present the algorithm in some stages, bottom-up, starting with a part which is just as in the NFA algorithm.

Notation: the algorithm manipulates a graph which has LARE expressions on each arc (this generalizes an ordinary flowchart, where each arc carries a single instruction, in the same way the *GNFA* generalizes an NFA in [32]; due to lack of space we do not elaborate). Denote by $\mathtt{Rex}_e$ the expression on the arc e (we may write uv for e when there is a single arc from u to v).

5.1 Ripping a node

By "ripping" a node we remove it from the graph, obtaining a smaller graph with equivalent semantics

Algorithm RIPONE(g, v):
accepts a control-flow graph g annotated with LARE expressions, and an internal node v (i.e., v has both in-going and out-going arcs).

1. Merge parallel arcs in g, if any, by combining the expressions with the alternation operator.
2. For every path of length two, uvw, add an arc e' from u to w and let $\mathtt{Rex}_{e'}$ be $\mathtt{Rex}_{uv}(\mathtt{Rex}_{vv})^*\mathtt{Rex}_{vw}$ (if v has no self-loop, we get $\mathtt{Rex}_{uv}\mathtt{Rex}_{vw}$).
3. Remove v from the graph.

5.2 Contracting a loop

Next we define a procedure to contract a subgraph g, representing a leaf loop (that has no child loops), by ripping its internal nodes. We assume that the graph has some entry nodes and some exit nodes and that these are the only nodes that connect to the rest of the program. The effect of the contraction procedure below is that only entry and exit nodes remain; and arcs connect entries to exits.

Algorithm $\textsc{ContractSimple}(g,l)$, where g is a control-flow graph and l, an index of a loop variable:

1. For every node v which is not an entry or exit node in g, perform $\textsc{RipOne}(g,v)$.
2. For every arc e in this bipartite graph, replace its label $\mathtt{Rex}_e$ by $[_l\, \mathtt{Rex}_e]$.

The full $\textsc{Contract}$ procedure handles a loop in the context of a larger flowchart. Since loops may in general, share nodes, we first isolate the loop from the rest of the graph and ensure that it has well-defined entry and exit nodes, by creating new nodes for this purpose, which we call virtual entry/exit nodes. More precisely, a *boundary node* of a subgraph is a node which is incident to arcs outside the subgraph as well as within. If v is such a node, it is replaced (in the most general case) by four nodes: $v_{\overline{L}}$ for paths that stay outside the loop, v_{in} to enter the loop, v_{out} to exit it, and v_L for paths that go through v but stay inside the loop (this may be best understood with a drawing—see Figure 4). Note that we are not handling nested loops yet. The procedure is illustrated by Figure 5.

Algorithm $\textsc{Contract}(L)$

1. For every boundary node v of L, create new nodes v_L, $v_{\overline{L}}$, v_{in} and v_{out}. Any arc incident to v is replaced by two arcs as shown in Figure 4).
2. Let g' consist of the subgraph spanned by internal (non-boundary) nodes of L, plus, for a boundary node v, the nodes v_{in}, v_{out}, v_L.
3. Perform $\textsc{ContractSimple}(g', \mathtt{X}_{\mathtt{Bound}(L)})$. This removes all the internal nodes of L, leaving only the entry and exit nodes.

5.3 Converting a whole program

The algorithm to convert a whole flowchart program to LARE now follows easily. The input to the algorithm is, in general, a loop in the loop tree of the flowchart program (Definition 3.2). To convert the entire program, we apply $\textsc{ConvertFC}$ to the root of the tree.

Algorithm $\textsc{ConvertFC}(L)$

1. Perform recursive calls to $\textsc{ConvertFC}$ for all the children of L. In these recursive calls, any virtual nodes created will be shared (i.e., if v is a specific node of L and it has incident arcs in two subloops, there will still be only one node v_{in} and only one v_{out}. This subtlety arises because our loops are defined to be edge-disjoint but not vertex-disjoint).
2. Perform $\textsc{ContractSimple}(L, \mathtt{X}_{\mathtt{Bound}(L)})$.

As an exception, for the root loop we simplify $\textsc{Contract}$ by not creating virtual nodes. This works because it has entry nodes and exit nodes by definition, and we do not want to alter their identity.

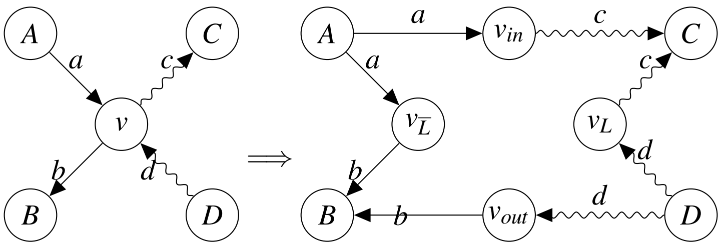


Figure 4: Transforming a boundary node v. The curly arcs belong to the loop under consideration. Arc labels $a, b, \ldots$ represent instructions.

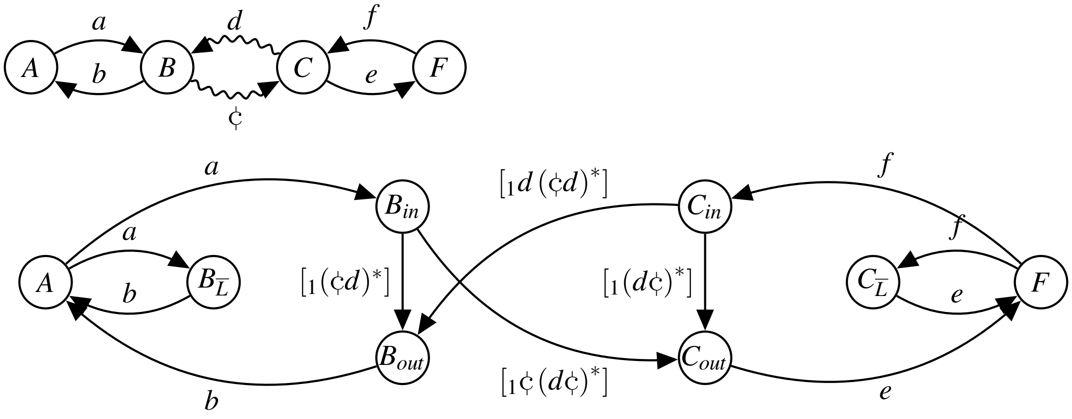


Figure 5: A flowchart with a loop $\{B, C\}$ (above) and the result of contracting this loop. We assume that the bounding variable for the loop is $\texttt{X}_1$. Note that nodes B_L, C_L are not present since contraction of the loop has removed them.

6 The Growth-Rate Analysis Algorithms

In this section we reach the main theoretical contribution: a polynomial-time decision procedure for polynomial growth rates. We implement the idea of extending results from well-structured programs to arbitrary control-flow graphs by exploiting the translation of graphs to regular expressions. First, we extend the analysis of [6] to LARE (the extension is easy, but for the sake of completeness we give the algorithm in a self-contained form. This version is based on [4] rather than [6]).

For the rest of this article, let $\mathbb{I} = \{1, \ldots, n\}$ be the set of variable indices.

6.1 Analysis of LARE commands

We define an interpretation of LARE commands over the domain of *dependency sets*. By applying this interpretation to a command, we find how the magnitude of the values of each at the end of the computation depends on the initial values.

6.2 Dependencies and dependency sets

Definition 6.1. The set of *dependency types* is $\mathbb{D} = \{1, 1^+, 2, 3\}$, with order $1 < 1^+ < 2 < 3$, and maximum operator $\sqcup$. We write $x \simeq 1$ for $x \in \{1, 1^+\}$.

The following verbal descriptions may give intuition to the meaning of dependency types:

1 =*identity dependency*,

1^+ =*additive dependency*,

2 =*multiplicative dependency*,

3 =*exponential dependency* (more precisely, super-polynomial).

Definition 6.2. The set of *dependencies* is $\mathbb{F}$, which is the union of two sets:
(1) The set of *unary dependencies*, isomorphic to $\mathbb{I}\times\mathbb{D}\times\mathbb{I}$. The notation for an element is $i \xrightarrow{\delta} j$.
(2) The set of *binary dependencies*, isomorphic to $\mathbb{I}\times\mathbb{I}\times\mathbb{I}\times\mathbb{I}$, notated ${}^{i}_{j} \rightrightarrows {}^{k}_{\ell}$.

Intuitively, binary dependencies represent conjunctions of unary dependencies. The fact that it is necessary and sufficient to handle conjunctions of pairs of unary dependencies (but not of larger sets) is a non-trivial property which is key to this algorithm's correctness.

Definition 6.3. A *dependency set* is a subset of $\mathbb{F}$, with the proviso that a binary dependency ${}^{i}_{j} \rightrightarrows {}^{k}_{\ell}$ may appear in the set only if it also includes $i \xrightarrow{\alpha} k$ and $j \xrightarrow{\beta} \ell$, for some $\alpha,\beta \simeq 1$, and $i \neq j \vee k \neq \ell$.

The function COMPLETE adds to a dependency set all binary dependencies that can be added according to the above rule. That is:
For a dependency set S, we let $\text{COMPLETE}(S) \stackrel{\text{def}}{=} S \cup \{ {}^{i}_{j} \rightrightarrows {}^{k}_{\ell} \mid i \xrightarrow{\alpha} k \in S \wedge j \xrightarrow{\beta} \ell \in S,\ \alpha,\beta \simeq 1\}$. We then define the *identity* dependency set is $I_{dep} \stackrel{\text{def}}{=} \text{COMPLETE}(\ \{i \xrightarrow{1} i \mid i \in \mathbb{I}\}\ )$.

6.3 Interpretation of LARE

To give a dependency-set semantics to an LARE program e, which we denote by $[\![\texttt{e}]\!]_{dep}$, we give a semantics to every symbol and every operation.

6.3.1 Symbols

The symbols of LARE are atomic instructions, which update the state, and are supposed to be associated with some dependency sets. In order to justify our claim to a complete solution for the core instruction set, we give the dependency sets corresponding to these assignment instructions:

$$\begin{aligned}
[\![\texttt{skip}]\!]_{dep} &= I_{dep} \\
[\![\texttt{X}_r \texttt{ := } \texttt{X}_s]\!]_{dep} &= \text{COMPLETE}(\ \{s \xrightarrow{1} r\} \cup \{i \xrightarrow{1} i \mid i \neq r\}\ ) \\
[\![\texttt{X}_r \texttt{ := } \texttt{X}_s + \texttt{X}_t]\!]_{dep} &= \text{COMPLETE}(\ \{s \xrightarrow{1^+} r, t \xrightarrow{1^+} r\} \cup \{i \xrightarrow{1} i \mid i \neq r\}\ ) && \text{when } s \neq t \\
[\![\texttt{X}_r \texttt{ := } \texttt{X}_s + \texttt{X}_s]\!]_{dep} &= \text{COMPLETE}(\ \{s \xrightarrow{2} r\} \cup \{i \xrightarrow{1} i \mid i \neq r\}) \\
[\![\texttt{X}_r \texttt{ := } \texttt{X}_s * \texttt{X}_t]\!]_{dep} &= \text{COMPLETE}(\ \{s \xrightarrow{2} r, t \xrightarrow{2} r\} \cup \{i \xrightarrow{1} i \mid i \neq r\}\ )
\end{aligned}$$

The atomic command `X := **` (set to unbounded value) is not directly representable. We handle it as `X := HUGE` where `HUGE` is a special variable. A weak assignment $\texttt{X} :\leq \texttt{e}$ is abstracted the same as the corresponding deterministic assignment `X := e`.

6.3.2 LARE operators

Alternation is interpreted by set union:

$$[\![E|F]\!]_{dep} = [\![E]\!]_{dep} \cup [\![F]\!]_{dep}$$

Concatenation is interpreted as a component-wise product of sets:

$$[\![EF]\!]_{dep} = [\![E]\!]_{dep} \cdot [\![F]\!]_{dep}$$

where the product of dependencies is given by

Definition 6.4 (dependency composition)**.**

$$\begin{aligned}
(i \xrightarrow{\alpha} j) \cdot (j \xrightarrow{\beta} k) &= i \xrightarrow{\alpha \sqcup \beta} k \\
(i \xrightarrow{\alpha} j) \cdot (\substack{j \\ j} \rightrightarrows \substack{k \\ k'}) &= (\substack{i \\ i} \rightrightarrows \substack{k \\ k'}), \quad \text{provided } \alpha \simeq 1 \\
(\substack{i \\ i'} \rightrightarrows \substack{j \\ j}) \cdot (j \xrightarrow{\alpha} k) &= (\substack{i \\ i'} \rightrightarrows \substack{k \\ k}), \quad \text{provided } \alpha \simeq 1 \\
(\substack{i \\ i'} \rightrightarrows \substack{j \\ j'}) \cdot (\substack{j \\ j'} \rightrightarrows \substack{k \\ k'}) &= \begin{cases} \substack{i \\ i'} \rightrightarrows \substack{k \\ k'}, & \text{if } i \neq i' \text{ or } k \neq k' \\ i \xrightarrow{2} k, & \text{if } i = i' \text{ and } k = k' \end{cases}
\end{aligned}$$

The last sub-case in the definition handles commands whose effect is to double a variable's value by making two copies of it and adding them together.

Iteration To define the interpretation of the star operator, we first define $LFP(S)$ to be the least fixpoint (under set containment) of the function $f(X) = I_{dep} \cup X \cup (X \cdot S)$.

Finally, we define the so-called *loop correction* operator, which represents the *implicit dependence* of variables updated in a loop on the loop bound. First, we define it for single dependencies:

$$\begin{aligned}
LC_\ell(i \xrightarrow{1^+} i) &= \{\ell \xrightarrow{2} i\} \\
LC_\ell(i \xrightarrow{2} i) &= \{\ell \xrightarrow{3} i\} \qquad (1) \\
LC_\ell(\Delta) &= \{\} \qquad \text{for all other } \Delta \in \mathbb{F},
\end{aligned}$$

and then extend it to sets by $LC_\ell(S) = S \cup \bigcup_{D \in S} LC_\ell(D)$ (note that we define it so that $LC_\ell(S) \supseteq S$). Using these definitions, let $F = LFP([\![E]\!]_{dep})$, then

$$[\![E^*]\!]_{dep} = LC_\ell(F) \cdot F$$

where ℓ is the index of the bounding variable for the closest enclosing bracket construct.

In this analysis, the brackets themselves do not imply any computation in the abstract semantics. Their role is just to determine ℓ in the above rule for loop correction.

EXAMPLE 6.5. Consider the loop in Figure 3 (left). It can be expressed as an LARE command of the form $[_4(\cent E)^*]$, where E represents the loop body. The set of *unary* dependencies for E is

$$1 \xrightarrow{1} 1, \quad 1 \xrightarrow{1^+} 3, \quad 2 \xrightarrow{1^+} 3, \quad 1 \xrightarrow{1^+} 2, \quad 2 \xrightarrow{1^+} 2, \quad 4 \xrightarrow{1} 4$$

Binary dependencies include all pairs of the above. Now consider the product $[\![E]\!]_{dep} \cdot [\![E]\!]_{dep}$; by the last case in Definition 6.4, it includes $1 \xrightarrow{2} 2$. This demonstrates the role of this definition in expressing the fact that when the loop is iterated, a multiple of the initial value of $\mathtt{X}_1$ is accumulated in $\mathtt{X}_2$. Now, consider the dependency $2 \xrightarrow{1^+} 2$, which may be interpreted as stating that $\mathtt{X}_2$ is an accumulator. Since this dependency exists in the closure $F = LFP([\![E]\!]_{dep})$, this will produce an additional dependency when we apply the *LC* operator, specifically, it generates $4 \xrightarrow{2} 2$. Indeed, $\mathtt{X}_2$ accumulates a multiple of $\mathtt{X}_4$ (in fact, $\mathtt{X}_4 \cdot \mathtt{X}_1$; but we do not record the precise expression). Finally, when computing $LC_4(F) \cdot F$, we compose $4 \xrightarrow{2} 2$ with $2 \xrightarrow{1^+} 3$ obtaining $4 \xrightarrow{2} 3$, which reflects the flow of the product $\mathtt{X}_4 \cdot \mathtt{X}_1$ to $\mathtt{X}_3$; similarly, we obtain $1 \xrightarrow{2} 3$.

EXAMPLE 6.6. To illustrate the role of binary dependencies, let us compare the loop just analyzed with Figure 3 (right). Here, in the loop's body, the unary dependencies $1 \xrightarrow{1^+} 2$ and $2 \xrightarrow{1^+} 2$ exist, but not their conjunction (because they happen in alternative execution paths), and the result $1 \xrightarrow{2} 2$ does not arise.

6.4 Correctness and complexity

Our correctness claims state that the results of this analysis provide, for every variable, either a polynomial upper bound or an exponential lower bound. This assumes the core instruction set; if additional instructions are added, one of these two soundness claim may be compromised, depending on how closely the instruction is modeled by dependency sets (note that, interestingly, our result means that the core instruction set cannot express a computation with super-polynomial-sub-exponential growth rate).

THEOREM 6.7. *Let E be a well-formed LARE command, using the core instruction set. If $i \xrightarrow{3} j \in [\![E]\!]_{dep}$ for some i, then the values that $\mathtt{X}_j$ can take at the end of an execution of E grow, in the worst case, at least exponentially in terms of the initial variable values. If there is no such dependency, a polynomial bound on the final value of $\mathtt{X}_j$ exists.*

The proof of this theorem follows by showing that the dependency sets computed by our algorithm are all sound with respect to two interpretations: a lower-bound interpretation and an upper-bound interpretation. The interpretations are defined in terms of a *trace semantics* of the program. Due to the complexity of the proofs, they are deferred to appendices.

As to *complexity*, we claim that a straight-forward implementation of the algorithm is polynomial-time in the size of the program. The reason is that the size of a dependency set is polynomially bounded in the number of variables; most operations on such sets (union, composition) are clearly polynomial-time, the only non-trivial issue being the fixed-point computation for analyzing E^*. However, this is polynomial time because the height of the semi-lattice of dependency sets is polynomial.

6.5 Analysis of flowchart programs

The principle of the algorithm is to convert the source program to LARE program and perform the above analysis on the result. However, doing it in two steps, as just described, is inefficient, because converting an NFA to a regular expression has exponential cost (in fact, it is known that the *size* of the regular expression cannot be polynomially bounded).

To obtain an efficient algorithm we use the principle of function fusion [12], which basically means to eliminate intermediate structures when composing functions. We fuse together the functions of conversion from *FC* to LARE and analysis of the LARE program. The fused algorithm does not generate an expression, but directly computes its abstract semantics. Hence, it works on a graph with dependency sets as arc labels, rather than LARE expressions, and applies operations in the dependency-set domain instead of syntactic operations on expressions.

The fused algorithm runs in polynomial time. This follows by bounding the costs of the steps—applications of abstract operators—and the number of such operations that occur (as they dominate the running time). The complexity of abstract-domain operations is polynomial, see Section 6.4. For the number of operations, we consider the size of the graph manipulated by the algorithm, which is polynomial in the size of the original flowchart plus the number of additional nodes generated throughout the algorithm, due to splitting of boundary nodes. Since such splitting only occurs once per boundary node per loop, we have a polynomial bound in terms of the original graph.

7 Related Work

Growth-rate analysis is clearly related to the large body of work on static program analysis for discovering resource consumption (in particular, running time). But since we focus on decidable cases, based on weak languages, it is also related to work in Computability and Implicit Computational Complexity. Thus, there is a lot of work that could be mentioned, and this section will only give a brief overview and a few representative citations. However, we will dwell a bit longer on recent work that appears interestingly related.

Meyer and Ritchie [25] introduced the class of *loop programs*, which only has definite, bounded loops, so that *some* upper bound on their complexity can always be computed. Subsequent work [21, 22, 27, 19] attempted to analyze such programs more precisely; most of them proposed syntactic criteria, or analysis algorithms, that are sufficient for ensuring that the program lies in a desired class (say, polynomial-time programs), but are not both necessary and sufficient: thus, they do not address the decidability question (the exception is [22] which has a decidability result for a "core" language). As already mentioned, [6] introduced weak bounded loops (such that can exit early) into the loop language, plus other simplifications, and obtained decidability of polynomial growth-rate. Regarding the necessity of these simplifications, [7] showed undecidability for a language that can only do addition and definite loops (that cannot exit early).

Results that characterize programs in a way sufficient (but not necessary) to have a certain complexity resonate with the area called Implicit Computational Complexity (ICC), where one designs languages or program classes for capturing a complexity class; this was the goal in [21]. Later the approach seems to have focused on functional languages [3, 16, 18] and term rewriting systems [10].

Among related works in static program analysis, mostly pertinent are works directed at obtaining *symbolic, possibly asymptotic, complexity bounds* for programs (in a high-level language or an intermediate language) under generic cost models (either unit cost or a more flexible, parametrized cost model). A symbolic bound could be an expression like $2\mathrm{x}+10$, which may be more or less accurate, but to answer a simple binary question like "is there a polynomial bound" is not considered sufficient in this area (though one could argue that weeding out the super-polynomial programs should be of interest). Characteristic to this area is the fact that decidable subproblems have rarely been studied.

Wegbreit [33] presented the first, and very influential, system for automatically analysing a program's complexity. His system analyzes first-order LISP programs; broadly speaking, the system transforms the program into a set of recurrence equations for the complexity which have to be solved. Subsequent works along similar lines included [23, 30] and more recently [9] for functional programs, [14] for logic programs, and [1] for JBC.

Other techniques for resource/complexity analysis of realistic programming languages include abstract interpretation (for functional programs: [26]), counter instrumentation [17] and type systems [20, 31].

For programs where loops are not explicitly bounded, there is an obvious connection of finding bounds to proving termination. So techniques of termination proofs have migrated into bound computation. One example is size-change termination [24, 5] used in [34], and another is linear and lexicographic ranking functions [2]. This latter work, in particular, has inspired our notion of annotated flowcharts, as described in Section 2. By examining [2] one can see that they implicitly construct the loop tree. However, once this is done, they can compute a global bound only from loop bounds which are linear in the program's input. Thus the method precludes programs where a loop bound is a non-linear function of the input, possibly computed at a previous (or enclosing) loop. Recently, Brockschmidt et al. [11] presented an analysis algorithm, called `KoAT`, that deals with this challenge by alternating bound analysis (based on ranking functions) and size analysis (basically, analyzing the growth rate of variables); this solution seems to be very interestingly related to our work. We note the following points:

- `KoAT` handles unannotated programs in a realistic language, and it searches for ranking functions as part of the algorithm. It does not abstract programs into a weak programming language. No decidability result is claimed, nor does this seem to be a goal.
- `KoAT` produces explicit bounds, taking constants into account, while we only considered the decision problem of polynomial growth rate, and the goal was to answer it precisely.
- Interestingly, [11] use a "data-flow graph" in their algorithm, while we used such a graph in the proof (in fact, a similar graph was already used in [19]).

We can make a theoretically-meaningful comparison by focusing on the intersection of the two problems solved: namely, we can look only at our restricted programming language (which can be easily compiled into the input language of `KoAT`). For such programs we recognize all polynomial programs, while `KoAT` does not (but it provides explicit bounds when it does). This suggests an idea for further research, namely to make a closer comparison, and possibly merge the techniques.

8 Conclusion and Open Problems

This works addressed the decidability of a growth-rate property of programs, namely polynomial growth, in a weak programming language. Extending previous work that addressed compositionally-structured programs we have presented an analysis that works for flowchart programs with possibly complex control-flow graphs, provided with hierarchical loop information. We propose this program form as a way to extend program analysis algorithms from structured programs to less-structured ones. We prove that the polynomial growth problem is PTIME-decidable for our class of flowchart programs, with a restricted instruction set.

Unlike typical work in static analysis of programs (going by names such as *resource analysis* or *cost analysis*), our algorithm does not output full expressions for the complexity bounds. In principle, one could extend our algorithm to produce such bounds, since their calculation is implicit in the proof. We opted not to do it, since we focus on the problem that we can solve completely. We acknowledge that if we produce explicit polynomials, they will not be tight in general. It is an interesting problem, theoretically, to research the problem of computing precise bounds. Another theoretical challenge is to extend the language, e.g., precisely analyzing a larger set of instructions, or adding recursive procedures. In practice, one works with full languages and settles for sound-but-incomplete solutions, but we hope that our line of research can also contribute ideas to the more practical side of program analysis.

A Formal Semantics of FC and LARE

This appendix gives additional details on the formal semantics of both FC programs and LARE programs. This is required for justifying the correspondence of the two formalisms (which we just state: we skip the correctness proof of the conversion algorithm, which is quite trite once the definitions are in place), and for correctness proof of the growth-rate analysis for LARE (described in the following two appendices).

In both cases we state the semantics in terms of *traces*, sometimes called transition sequences. Those for FC are “more concrete” in that they involve the nodes and arcs of the CFG.

A.1 Semantics of FC

Consider an FC program $\texttt{p}$ with variables $\texttt{X}_1,\ldots,\texttt{X}_n$, and control-flow graph $G_{\texttt{p}} = (\texttt{Loc}_{\texttt{p}}, \texttt{Arc}_{\texttt{p}})$. For $a \in \texttt{Arc}_{\texttt{p}}$, write $a : P \to Q$ if P is its source location and Q its target.

Definition A.1 (states)**.** The set of states of $\texttt{p}$ is $\texttt{St}_{\texttt{p}} = \texttt{Loc}_{\texttt{p}} \times \mathbb{N}^n$, such that $s = (P,\vec{x})$ indicates that the program is at location P and $\vec{x}$ specifies the values of the variables.

Definition A.2 (transitions)**.** A transition is a pair of states, a *source state* s and a *target state* s', related by an instruction a of $\texttt{p}$. More precisely: $a : P \to Q$, $s = (P,\vec{x})$, $s' = (Q,\vec{x}')$, and the relation of $\vec{x}'$ to $\vec{x}$ is determined by the semantics of $\texttt{Inst}(a)$. When this holds, we write $s \overset{a}{\to} s'$. The set of transitions is $\mathscr{T}_{\texttt{p}}$.

Definition A.3 (transition sequence)**.** A *transition sequence* (or trace) of $\texttt{p}$ is a finite sequence of consecutive state transitions $\tilde{s} = s_0 \overset{a_1}{\longrightarrow} s_1 \ldots s_{t-1} \overset{a_t}{\longrightarrow} s_t$, where the instruction sequence $a_1 a_2 \ldots a_t$ corresponds to a CFG path. We refer to the arcs of the path as *the arcs of* $\tilde{s}$. The set of traces is denoted $\mathscr{T}_{\texttt{p}}^{+}$.

The definition of a transition sequence does not take the loop bounds into account. Thus it allows for sequences which do not respect the bounds. To enforce the bounds, we introduce the next definition.

Definition A.4 (properly bounded)**.** For a *transition sequence* $\tilde{s}$, let $L \in \mathscr{L}_{\texttt{p}}$ be the smallest loop that includes all arcs of $\tilde{s}$ (the smallest enclosing loop). Let L° be L minus any nested loop. Then, $\tilde{s}$ is *properly bounded* if the following conditions hold:

1. If L is not the root, let $\ell = \texttt{Bound}(L)$; then the number of occurrences in $\tilde{s}$ of any ¢ arc from L° is at most the value of x_{ℓ} (which does not change throughout $\tilde{s}$).
2. If L is the root, any $a \in L^{\circ}$ occurs at most once.
3. Every contiguous subsequence of $\tilde{s}$ is properly bounded.

We say the properly-bounded transition sequence is *a run of loop L* when L is the smallest enclosing loop, as in the definition. We say that a transition sequence is *complete* if it starts with an entry node of the flowchart and ends with an exit node.

We let $\mathscr{T}_{\texttt{p}}^{\oplus}$ denote the set of properly-bounded transition sequences for $\texttt{p}$.

A.2 Semantics of LARE

In the semantics of LARE programs, states only describe the values of variables, but not a program location (we may call them *pure states* when distinction is important). The set of pure states $\mathtt{St}$ is related to $\mathtt{St}_{\mathtt{p}}$ by an obvious abstraction relation. The evolution of such states is described by (pure) transitions:

Definition A.5 (pure transitions)**.** A pure transition is a pair of states, a *source state* s and a *target state* s', related by an instruction a out of the vocabulary of atomic instructions. We write this as $s \xrightarrow{a} s'$. We presume that a transition correctly reflects the semantics of the atomic instruction.

Definition A.6 (traces)**.** A *trace* is a finite sequence of consecutive transitions $s_0 \xrightarrow{a_0} s_1 \ldots s_{t-1} \xrightarrow{a_{t-1}} s_t$. The function $\textsc{Erase}(\sigma)$ removes the ¢ transitions from the trace σ, which is valid because this instruction does not change the state. We define $\|\sigma\|$ to be the number of ¢'s in σ. The set of all traces is denoted by $\mathscr{T}$ (the number of variables is tacitly assumed to be fixed). We write $s \overset{\sigma}{\rightsquigarrow} s'$ to indicate that $s[0] = s$ and $s[t] = s'$.

Concatenation of finite traces λ, ρ is written as $\lambda \fatsemi \rho$ and requires the final state of λ to be the initial state of ρ. As a special case, we denote an *empty* sequence by ε and define $\varepsilon \fatsemi \rho = \rho \fatsemi \varepsilon = \rho$ for any ρ.

We define the *trace* semantics $\llbracket E \rrbracket_{ts}$ of an expression E by structural induction. First, recall that each *symbol* corresponds to a single instruction. For our core instruction set, semantics should be obvious, so we skip the details. As for *composite programs*, we have

$$\llbracket E|F \rrbracket_{ts} = \llbracket E \rrbracket_{ts} \cup \llbracket F \rrbracket_{ts}$$
$$\llbracket EF \rrbracket_{ts} = \llbracket E \rrbracket_{ts} \fatsemi \llbracket F \rrbracket_{ts}$$

where the last operation is, naturally, the component-wise concatenation of E and F:

$$L \fatsemi R \overset{\text{def}}{=} \{\lambda \fatsemi \rho \mid \lambda \in L, \rho \in R, \text{and } \lambda \fatsemi \rho \text{ is defined}\}.$$

Finally, for the looping constructs, we define $\llbracket E^* \rrbracket_{ts}$ to be the reflexive-transitive closure of $\llbracket E \rrbracket_{ts}$ under the concatenation operation $\fatsemi$, and

$$\llbracket\, [_\ell E]\, \rrbracket_{ts} = \{\textsc{Erase}(\sigma) \mid \sigma \in \llbracket E \rrbracket_{ts}, \|\sigma\| \le (\sigma[0])_\ell\}\,.$$

A.3 Correspondence of the semantics

Let $\mathtt{p}$ be a flowchart program. Let $\mathscr{T}_{\mathtt{p}}^{+}$ be the set of properly-bounded traces for $\mathtt{p}$. As traces of LARE do not involve locations, we define the function $\textsc{AddLoc}(S, P, Q)$ that inserts locations in traces $\tilde{s} \in S$ so that the initial location is P, the final location is Q, and intermediate locations are the anonymous location $\bullet$. We also define a converse function $\textsc{AnonLoc}(S, P, Q)$, where S is a set of flowchart traces, which replaces the program locations in every $\tilde{s} \in S$ (except the first location and the last) by $\bullet$, provided the trace starts at P and ends at Q (otherwise, it is ignored). Then the correspondence of a LARE E to a flowchart program $\mathtt{p}$ with entry point P and exit Q is expressed by the equation:

$$\textsc{AnonLoc}(\mathscr{T}_{\mathtt{p}}^{\oplus}, P, Q) = \textsc{AddLoc}(\llbracket E \rrbracket_{ts}, P, Q)\,.$$

B Preliminaries for the Proofs

This section includes some preliminaries for the correctness proof of the polynomial-bound analysis (or rather proofs: in the next section, we prove that the analysis provides sound lower bounds on the worst-case growth, and in the next, that it provides sound upper bounds). Thus we have a sound and complete decision procedure for the problem of polynomial growth rate. The proofs in this paper are short presentations, while full details can be found in the technical report [8].

Our analysis of LARE programs (Section 6) involves an operation (loop correction) that refers to the bounding variable of the enclosing loop. In order to make the analysis fully compositional (which is easier to reason about), we avoid the need to "peek" at the enclosing context by introducing a dummy variable for "iteration count," denoted by x_{n+1} (we may call it also "the iteration variable", note however that it is just a place-holder, later to be replaced). We thus write $\mathbb{I}_E = \{1,\dots,n+1\}$ for the extended set of variable indices and extend the notation for dependencies to $\mathbb{I}_E$. To the interpretation of all atomic programs we add $n+1 \xrightarrow{1} n+1$, and the loop corrector uses x_{n+1} rather than x_ℓ (we thus have LC_{n+1} and not LC_ℓ). The bracket construct now has a non-trivial interpretation, namely let $\text{SUBST}(\ell, n+1, S)$ be the result of substituting any dependency $n+1 \xrightarrow{\delta} j$ by $\ell \xrightarrow{\delta} j$ in the dependency set S. Then

$$[\![\, [_\ell E]\,]\!]_{dep} = \text{SUBST}(\ell, n+1, [\![E]\!]_{dep})\,.$$

We further add to dependencies a property called *color*: black is the default color and red is special. A dependency is given red if and only if it is of the form $n+1 \xrightarrow{\delta} i$ with $i \neq n+1$.

Some useful observations are: (i) In any derivation of dependencies for a LARE program, $LC_{n+1}(D) = \emptyset$ whenever D is red. (ii) In any composition $D_1 \cdot D_2$ in a derivation, at most one of D_1 and D_2 is red. (iii) The type of a red dependence is always 2 or 3. (iv) A black dependence having $n+1$ as source index must be the identity dependence $n+1 \xrightarrow{1} n+1$.

Conventions and notations For $\vec{x} = (x_1,\dots,x_n)$, x_{min} is $\min\{x_i\}$. The relation $\vec{x} \sqsupseteq t$ means: $x_{min} \geq t$.

C Lower-Bound Soundness

This section proves soundness of the lower-bound aspect of polynomial-bound analysis, leading to the conclusion that $x \xrightarrow{3} y$ indicates certain exponential growth. This is the more intuitive interpretation of our abstract domain: we interpret every dependency that we derive as an indication of something that *can* happen. In a sense, the heart of the proof is the proper definition of the concrete meaning of the various dependency types, i.e., when a dependency type is considered to hold in a certain set of execution traces. Afterwards, it only rests to verify that they are computed correctly for every type of commands. This is mostly technical and is not given in this paper in full detail.

We give the definition for unary dependencies (black and red) first and then the binary ones.

Definition C.1 (unary, black dependencies)**.** Let $\rho \subset \mathscr{T}$. We say that ρ satisfies the dependency $D = i \xrightarrow{\delta} j$ written $\rho \models D$, if there are integer constants d, t, b, where $d \geq 0$ and $t \geq b \geq 0$, as well as a real constant $c > 0$, such that for all $\vec{x} \sqsupseteq t$ there is a sequence $\sigma = (\vec{x}, \dots, \vec{y}) \in \rho$ with $\|\sigma\| = b$ satisfying:

(M) $\vec{y} \sqsupseteq x_{min}$,
(SU1) $\delta \geq 1 \quad \Rightarrow \quad y_j \geq x_i$,
(SU1+) $\delta = 1^+ \quad \Rightarrow \quad y_j \geq x_i + x_{min}$,
(SU2) $\delta = 2 \quad \Rightarrow \quad y_j \geq 2(x_i - d)$,
(SU3) $\delta = 3 \quad \Rightarrow \quad y_j \geq 2^{c(x_i - d)}$.

As an exception, dependencies $n+1 \xrightarrow{1} n+1$ are considered to be satisfied by any (non-empty) ρ.

Note that property (M), which also figures in the following definitions, means that no variable has been reset to zero, for example. Indeed, handling such extensions requires a more difficult analysis [4].

Red dependencies express a condition similar to that of black dependencies of the same type, but they express a dependence on the iteration count $\|\sigma\|$ and they should be satisfied by infinite sets of traces whose iteration counts form an arithmetic sequence.

Definition C.2 (unary, red dependencies)**.** Let $\rho \subset \mathscr{T}$. We say that ρ satisfies the red dependency $D = n+1 \xrightarrow{\delta} j$ written $\rho \models D$, if there are integer constants d, t, b, p, where $p > 0$, $d \geq 0$, $t \geq b \geq 0$, as well as a real constant $c > 0$, such that for all $i \geq 0$, for all $\vec{x} \sqsupseteq t + ip$, there is a sequence $\sigma = (\vec{x}, \ldots, \vec{y}) \in \rho$, with $\|\sigma\| = b + ip$, satisfying:

(M) $\vec{y} \sqsupseteq x_{min}$,
(SU2) $\delta = 2 \quad \Rightarrow \quad y_j \geq 2(\|\sigma\| - d)$,
(SU3) $\delta = 3 \quad \Rightarrow \quad y_j \geq 2^{c(\|\sigma\| - d)}$.

Definition C.3 (binary dependencies)**.** Let $\rho \subset \mathscr{T}$. We say that ρ satisfies the dependency $D = {}^{i}_{j} \rightrightarrows {}^{k}_{\ell}$ written $\rho \models D$, if there are constants $t \geq b \geq 0$ such that for all $\vec{x} \sqsupseteq t$ there is $\sigma = (\vec{x}, \ldots, \vec{y}) \in \rho$ where $\|\sigma\| = b$, satisfying:

(M) $\vec{y} \sqsupseteq x_{min}$,
(SB1) $y_k \geq x_i \wedge y_\ell \geq x_j$,
(SB2) $k = \ell \Rightarrow y_k \geq x_i + x_j$.

Using these definitions we can state the soundness theorem:

THEOREM C.4 (lower-bound soundness)**.** *If $D \in [\![E]\!]_{dep}$ then $[\![E]\!]_{ts} \models D$.*

The next subsections justify the soundness for each LARE constructor in turn; this yields Theorem C.4 by simple induction.

C.1 Atomic programs

This is a trivial part. All atomic programs in our core instruction set induce obvious dependencies. Note that for a multiplication instruction, `X`$_i$ `:=` `X`$_j$`*X`$_k$, we need a threshold $t \geq 2$ to justify the lower bounds $y_i \geq 2x_j$ and $y_i \geq 2x_k$.

C.2 Alternation and composition

Alternation is interpreted as set union in the concrete semantics, and since the property $\rho \models D$ is existential, we immediately have

LEMMA C.5. *If $[\![E_1]\!]_{ts} \models D$, or $[\![E_2]\!]_{ts} \models D$, then $[\![E_1 | E_2]\!]_{ts} \models D$.*

For composition, we claim:

LEMMA C.6. *If $[\![E_1]\!]_{ts} \models D_1$ and $[\![E_2]\!]_{ts} \models D_2$ then $[\![E_1 E_2]\!]_{ts} \models D_1 \cdot D_2$.*

This follows by considering $\rho_1, \rho_2 \subset \mathscr{T}$, such that $\rho_1 \models D_1$ and $\rho_2 \models D_2$, and proving that $(\rho_1 ⨟ \rho_2) \models D_1 \cdot D_2$. The proof is a tedious case analysis, according to the types of D_1 and D_2, and follows the corresponding case in the definition of the product (Definition 6.4). We will skip it mostly, giving one case for example, involving binary dependencies: Suppose that $D_1 = \substack{i\\i'} \rightrightarrows \substack{j\\j'}$ and $D_2 = \substack{j\\j'} \rightrightarrows \substack{k\\k}$, and $i \neq i'$. Then $D_1 \cdot D_2 = \substack{i\\i'} \rightrightarrows \substack{k\\k}$. Let t_1, b_1 (respectively, t_2, b_2) be the constants involved in the application of definition C.3 to these dependencies. Thus there is, for all $\vec{x} \sqsupseteq t_1$ a sequence $\sigma_1 \in \rho_1$ satisfying requirements (M) and (SB1), starting with $\vec{x}$ and ending with $\vec{y} \sqsupseteq x_{min}$. We can restrict attention to $\vec{x} \sqsupseteq \max(t_1, t_2)$ which implies $\vec{y} \sqsupseteq t_2$. For such $\vec{y}$ there will be a $\sigma_2 \in \rho_2$, satisfying $\vec{z} \sqsupseteq y_{min}$, plus (M)–(SB2). Then $y_j \geq x_i$, $y_{j'} \geq x_{i'}$, $z_k \geq y_j + y_{j'}$. It follows that $z_k \geq x_i + x_{i'}$, so the conclusion $(\rho_1 ⨟ \rho_2) \models D_1 \cdot D_2$ holds.

C.3 Analyzing loops

In this analysis we assume for simplicity (and omitting the justification) that our experssions are rewritten so that for every starred expression there is just one ¢, in its beginning, thus: $(\text{¢}E)^*$.

Now, recall that $[\![E^*]\!]_{dep} = LC_\ell(F) \cdot F$, with $F = LFP([\![E]\!]_{dep})$. We first note that if for some finite m, $[\![E]\!]_{ts} \models D_i$ for $i = 1, \ldots, m$, then $[\![(\text{¢}E)^*]\!]_{ts} \models D_1 \cdot D_2 \ldots \cdot D_m$. Consequently $[\![(\text{¢}E)^*]\!]_{ts} \models D$ for all $D \in LFP([\![E]\!]_{dep})$. It rests to justify the use of the loop correction operator.

LEMMA C.7. *Let $F = LFP([\![E]\!]_{dep})$, $D \in F$ and $R \in LC_{n+1}(D)$. Then $[\![(\text{¢}E)^*]\!]_{ts} \models R$.*

Proof. Note that if $LC_{n+1}(D) = \emptyset$, there is nothing to prove. Otherwise, $LC_{n+1}(D) = \{n+1 \xrightarrow{\lambda} i\}$ for some $\lambda \geq 2$. From $F = LFP([\![E]\!]_{dep})$, it clearly follows that, for some $m > 0$, D is in $([\![E]\!]_{dep})^m$; hence $([\![\text{¢}E]\!]_{ts})^m \models D$. Denote, for brevity, $[\![\text{¢}E]\!]_{ts}$ by ρ. Checking the cases in the definition of LC (Page 34) we see that D must be $i \xrightarrow{\delta} i$ with $i \neq n+1$ and $\delta \in \{1^+, 2\}$. In particular, D is black. Hence, there are t, b, d such that

> For all $\vec{x} \sqsupseteq t$ there is a $\sigma = (\vec{x}, \ldots, \vec{y}) \in \rho^m$ where $\|\sigma\| = b$, $\vec{y} \sqsupseteq x_{min}$ and $y_i \geq x_i + x_{min}$ (for $\delta = 1^+$) or $y_i \geq 2(x_i - d)$ (for $\delta = 2$).

Note that $b > 0$, since an empty trace only satisfies dependencies of type 1 (for the same reason, $m > 0$).

By easy induction we can now derive for any $s > 0$, and any $\vec{x}$ as above, a trace $\pi_s \in \rho^{ms}$ starting at $\vec{x}$ as above, and ending with a state $\vec{z}$ satisfying $z_{min} \geq x_{min}$ and $z_i \geq x_i + s x_{min}$ (for $\delta = 1^+$) or $z_i \geq 2^s(x_i - d)$ (for $\delta = 2$). Moreover, $\|\pi_s\| = sb$. Thus,

$$\begin{aligned} z_i &\geq x_i + s x_{min} \geq x_i + (\|\pi_s\|/b) x_{min} && \text{(for } \delta = 1^+\text{), or} \\ z_i &\geq 2^s(x_i - d) \geq 2^{\|\pi_s\|}(x_i - d) && \text{(for } \delta = 2\text{)}. \end{aligned}$$

Choosing $c' = 1/b$ we may conclude that if $\|\pi_s\|$ is large enough, and $x_{min} > \max(d, 2b, t)$, we have

$$\begin{aligned} z_i &\geq 2 \cdot \|\pi_s\| && \text{(for } \delta = 1^+\text{), or} \\ z_i &\geq 2^{c' \cdot \|\pi_s\|} && \text{(for } \delta = 2\text{)}. \end{aligned}$$

By the definition of the semantics of the iteration construct, $\pi_s \in [\![(\text{¢}E)^*]\!]_{ts}$. The sequences π_s thus satisfy the requirements per Definition C.2, with appropriate constants (which are not hard to derive from the above discussion, in particular we note that the period of the arithmetic sequence is b). □

We conclude with the bracket construct. Recall that

$$[\![\, [_\ell E]\,]\!]_{ts} = \{\textsc{Erase}(\sigma) \mid \sigma \in [\![E]\!]_{ts}, \|\sigma\| \le (\sigma[0])_\ell\}\,.$$

Clearly, $\textsc{Erase}(\sigma)$ has the same initial and final states as σ. Thus a lower bound on the final state of σ is valid for $\textsc{Erase}(\sigma)$. Suppose that $[\![E]\!]_{ts} \models D$; in most cases this implies $[\![\, [_\ell E]\,]\!]_{ts} \models D$ trivially, with the exception being $D = n+1 \xrightarrow{\alpha} k$, a red dependency. In this case the lower bound involves $\|\sigma\|$, which ranges over a set $B = \{b, b+p, b+2p, b+3p, \dots\}$. Choosing the longest among these sequences that satisfy $\|\sigma\| \le (\sigma[0])_\ell$, we obtain $\|\sigma\| \ge (\sigma[0])_\ell - p$. Hence, denoting the initial and final states of σ by $\vec{x}$ and $\vec{y}$, respectively, we obtain a result of the form

$$\begin{aligned} y_i &\ge 2(x_\ell - p - d) && \text{(for } \alpha = 2\text{), or} \\ y_i &\ge 2^{c(x_\ell - p - d)} && \text{(for } \alpha = 3\text{)} \end{aligned}$$

which satisfies (SU2) or (SU3), respectively. We also note that (SU1) will be satisfied once the threshold t large enough.

We have now proved the soundness of all analysis rules, and Theorem C.4 immediately follows.

D Upper-Bound Soundness

The upper-bound soundness result will show that the absence of a type-3 dependency for a result variable implies that it is polynomially bounded in all executions. Thus, here we need an interpretation of the abstract value (the dependency set) which is *universal* in terms of applying to all traces, and moreover, it has to take into account *all variables* simultaneously. For intuition, consider an assignment $\texttt{X}_i$ `:=` $\texttt{X}_j$`+`$\texttt{X}_k$. To prove that the result is exponential, it suffices to know that one of $\texttt{X}_j$, $\texttt{X}_k$ is exponential. But to prove that the result is polynomial, we need to know that both of them are.

Multi-polynomials. We introduce the notation $\vec{p}$ for a collection of polynomials p_j. The range of the indices is implicit, and should be assumed by the reader to be $\{1, \dots, n\}$ unless the context dictates otherwise. We may refer to such a collection as a *multi-polynomial*. Its purpose it to express simultaneous polynomial bounds on several variables.

We say that variable x_i *participates* in polynomial $p(\vec{x})$ (or that the polynomial depends on x_i) if x_i appears in a monomial of p that has a non-zero coefficient. We use the notation $p[x_k|\pi(k)]$ to indicate that p depends only on variables x_k where k satisfies the predicate π. For example, $p[x_k|k=1]$ indicates a polynomial dependent only on x_1.

Multi-polynomials can be composed, written $\vec{p} \circ \vec{q}$, provided that for every x_i that participates in $\vec{p}$, the polynomial q_i is defined.

In the proof we use polynomials of $n+1$ variables, where $x_1, \dots, x_n$ represent the initial state of the computation under consideration while the last variable, x_{n+1}, is used to represent the number of ¢ symbols in a trace. Once polynomial bounds of this form (to be called *extended polynomials*) are established for a program E, we obtain bounds for $[_\ell E]$ by substituting the value of the loop control variable $\texttt{x}_\ell$ for x_{n+1}.

Dependency matrices. Just as we aggregate upper bounds in a multi-polynomial, we have to aggregate dependencies as well. We use an $(n+1) \times (n+1)$ matrix A to denote a collection of unary dependencies, i.e., A_{ij} shows the dependence of x_j on x_i. Thus, $A \in \mathbb{D}_0{}^{(n+1)\times(n+1)}$, where $\mathbb{D}_0 = \{0,1,1^+,2,3\}$ is the set of dependency types together with 0, a bottom element representing no dependence. Our matrices have to satisfy a certain validity condition in order to make sense as a set of dependencies. We call A *admissible* if the following conditions are satisfied:

$$A_{(n+1)(n+1)} = 1 \tag{2}$$

$$(\forall i,j) A_{ij} = 1 \Rightarrow A_{kj} = 0 \text{ for all } k \neq i\,, \tag{3}$$

where the first condition arises from the special role of x_{n+1}, and the second one from the purpose of a dependency $i \xrightarrow{1} j$, which is supposed to mean that x_j obtains its value from x_i, without any additions (note the difference of 1 to 1^+).

We denote by $A \leq B$ the natural component-wise comparison.

When S and T are sets of matrices, $S \leq T$ means $\forall A \in S\ \exists B \in T\ A \leq B$.

We introduce a "sum" operation on $\mathbb{D}_0$, denoted by $+$, as follows: $1^+ + 1^+ = 2$; and $\alpha + \beta = \alpha \sqcup \beta$ for all other α, β. We then define matrix product in $\mathbb{D}_0{}^{(n+1)\times(n+1)}$ in the usual way, using the operations $\cdot$ and $+$.

The set-of-matrices abstraction. The core idea of this proof is to replace our abstract domain. Instead of sets of unary and binary dependencies, we generate matrices. The unary dependencies are represented as entries in the matrix, while binary dependencies correspond to the co-existence of two $1/1^+$ entries in the same matrix. However, we do not define a new abstract interpreter, but obtain the matrices from the sets of dependencies, as follows.

Definition D.1. Let S be a set of dependence facts. Then $\widehat{\mathfrak{M}}(S)$ is the set of all admissible A satisfying

$$\begin{array}{ll} \text{(M1)} & \forall i,j\ .\ A_{ij} \neq 0 \Rightarrow i \xrightarrow{A_{ij}} j \in S \\ \text{(M2)} & \forall i,j,k,l\ .\ A_{ik}, A_{jl} \simeq 1 \wedge (i \neq j \vee k \neq l) \ \Rightarrow\ \genfrac{}{}{0pt}{}{i}{j} \rightrightarrows \genfrac{}{}{0pt}{}{k}{l} \in S\,. \end{array}$$

Let $\mathfrak{M}(S)$ be the set of maxima of $\widehat{\mathfrak{M}}(S)$.

The *matrix abstraction* of the analysis results for E is the set $\mathfrak{M}[\![E]\!]_{dep}$ (it seems neat to omit the parentheses in $\mathfrak{M}([\![E]\!]_{dep})$).

EXAMPLE D.2. Consider an LARE program E representing the command

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

The dependency set for this command is

$$1 \xrightarrow{1} 1,\quad 1 \xrightarrow{2} 3,\quad 2 \xrightarrow{1} 2,\quad 3 \xrightarrow{1} 2,\quad 3 \xrightarrow{1} 3,\quad 4 \xrightarrow{1} 4,\quad \genfrac{}{}{0pt}{}{1}{3} \rightrightarrows \genfrac{}{}{0pt}{}{1}{2},\quad \genfrac{}{}{0pt}{}{4}{3} \rightrightarrows \genfrac{}{}{0pt}{}{4}{2},$$

along with all binary dependencies of the form $\genfrac{}{}{0pt}{}{i}{j} \rightrightarrows \genfrac{}{}{0pt}{}{i}{j}$ (since they all occur in the `skip` branch). Here are two elements of $\mathfrak{M}[\![E]\!]_{dep}$

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 0 & 2 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 \\ 4 & 0 & 0 & 0 & 1 \end{array} \qquad \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 1 \end{array}$$

Clearly, the first represents the dependencies arising from choosing the first branch, while the second represents the second branch. Note that a matrix including both $3 \xrightarrow{1} 2$ and $3 \xrightarrow{1} 3$ is excluded by the constraint (M2).

The following lemma (which we cite here without proof) shows that the matrix sets obtained in this way satisfy some constraints of the kind we would expect a semantic abstraction to fulfill.

LEMMA D.3. *The following facts are true for the matrix representation $\mathfrak{M}[\![E]\!]_{dep}$.*

1. *For an atomic program E, $\mathfrak{M}[\![E]\!]_{dep}$ consists of a single matrix.*
2. $\mathfrak{M}[\![E_1|E_2]\!]_{dep} \geq \mathfrak{M}[\![E_1]\!]_{dep} \cup \mathfrak{M}[\![E_2]\!]_{dep}$.
3. $\mathfrak{M}[\![E_1E_2]\!]_{dep} \geq \mathfrak{M}[\![E_1]\!]_{dep} \cdot \mathfrak{M}[\![E_2]\!]_{dep}$.
4. *(a)* $\mathfrak{M}[\![(\cent E)^*]\!]_{dep} \geq \{I_{n+1}\}$.
   (b) $\mathfrak{M}[\![(\cent E)^*]\!]_{dep} \geq \mathfrak{M}[\![(\cent E)^*]\!]_{dep} \cdot \big(\mathfrak{M}[\![(\cent E)^*]\!]_{dep} \cup \{I_{n+1}\}\big)$.

We proceed to give these matrices a meaning in terms of polynomial upper bounds. The crux of the proof will be to prove that the matrices computed for a program are sound when interpreted in this fashion. For simplicity, throughout the rest of the section we assume that our analysis concludes that all variables are polynomially bounded and prove the soundness of that. Thus, we assume that no 3-entries occur in our matrices.

Definition D.4 (concretization of dependence vectors)**.** Let v be a vector in $\{0,1,1^+,2\}^{(n+1)}$. We define a set of functions, $\Gamma(v)$, to include all polynomials of form

$$\left(\sum_{i:\ v_i \simeq 1} a_i x_i\right) + P[x_i \mid v_i = 2]$$

with $a_i \in \{0,1\}$. Note that if v is the zero vector, we get the constant function 0.

Definition D.5 (polynomial upper bounds)**.** Let $\rho \subset \mathscr{T}$. We say that ρ admits a polynomial p as an upper bound on variable j, or that p bounds variable j in ρ, if the following holds:

$$\sigma \in \rho, \quad \vec{x} \overset{\sigma}{\rightsquigarrow} \vec{y}, \quad x_{n+1} \geq \|\sigma\| \quad \Longrightarrow \quad y_j \leq p(\vec{x}). \tag{4}$$

We also apply this expression to $j = n+1$ (the dummy variable). Here the requirement is $p(\vec{x}) = x_{n+1}$.

Definition D.6 (description by a matrix)**.** Let $\rho \subset \mathscr{T}$ and A an admissible matrix. Suppose that there is an upper bound p_{Aj} for variable j in ρ, and $p_{Aj} \in \Gamma(A_{\bullet j})$. We then say that A describes variable j in ρ.

We say that an admissible matrix A describes $\rho \subseteq \mathscr{T}$ (via a multi-polynomial $\vec{p}_A$) if $\vec{p}_{Aj} \in \Gamma(A_{\bullet j})$ bounds variable j in ρ for all j. Concisely, we write: $\rho \models A : \vec{p}_A$ or, elliptically, $\rho \models A$.

We say that a set $\mathscr{A}$ of matrices describes ρ if ρ has a cover $\rho \subseteq \bigcup_{A \in \mathscr{A}} \rho_A$ such that every matrix A describes its corresponding subset ρ_A, i.e., $\rho_A \models A$. We concisely write this statement as $\rho \models \mathscr{A}$.

Now we have a concise statement of the main theorem of this section:

THEOREM D.7 (soundness: upper bounds)**.** *For any LARE program E, $[\![E]\!]_{ts} \models \mathfrak{M}[\![E]\!]_{dep}$.*

D.1 The proof

This theorem is proved by structural induction on E. The base cases are the atomic programs, for which the statement is straight-forward to verify. The case of $E_1|E_2$ is also straight-forward, but the case of E_1E_2 is slightly harder. The argument for this case should be clear from the statement of the following two lemmata, whose proofs we omit.

LEMMA D.8. *Let A, B be matrices; p a polynomial, $\vec{q}$ a multi-polynomial, and assume that:*

(1) For a certain j, $p \in \Gamma(B_{\bullet j})$;
(2) For all k such that x_k participates in p, q_k is defined and $q_k \in \Gamma(A_{\bullet k})$;
Then $p \circ \vec{q} \in \Gamma((AB)_{\bullet j})$.

LEMMA D.9. *Let $\rho_1, \rho_2 \subseteq \mathscr{T}$, so that $\rho_1 \models A : \vec{q}$ and $\rho_2 \models B : \vec{p}$. Then $\rho_1 \fatsemi \rho_2 \models AB : (\vec{p} \circ \vec{q})$.*

Finally we come to the hardest part, which is to prove the following

LEMMA D.10. *If $[\![E]\!]_{ts} \models \mathfrak{M}[\![E]\!]_{dep}$, then $[\![(\cent E)^*]\!]_{ts} \models \mathfrak{M}[\![(\cent E)^*]\!]_{dep}$.*

For this proof, we let $\mathscr{M} = \mathfrak{M}[\![E]\!]_{dep} \cup \{I_{n+1}\}$, and $\mathscr{M}^* = \mathfrak{M}[\![E^*]\!]_{dep}$.

As the proof is difficult, we only bring some main ideas. The first is the so-called *size-relation graph*, SRG. This is a graph which shows all the dependencies at once: its nodes are $\{1, \ldots, n+1\}$ and there is an arc $i \rightarrow j$, labeled δ, if δ is the highest value of entry A_{ij} in $\mathscr{M}^*$, and is non-zero.

The decomposition of this graph into strongly connected components (SCCs), and their topological numebring, plays a crucial role. We show that intra-components arcs must be labeled with 1 or 1^+; intuitively in there had been a 2 there, a variable would have grown exponentially in the loop. The topological ordering of the components induces a "tiering" of variables so that non-linear data-flow only flows into higher-numbered SCCs.

The second important idea turns up when we want to compute bounds for variables that belong to a non-singleton SCC. It turns out to be necessary to compute bounds not only on the values of those variables but also on the sums of sets of variables. We can illustrate this point using the program

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

To prove that there is no exponential growth in this loop, it is crucial to use the fact that each of the first two assignments command makes the sum $\mathtt{X}_3 + \mathtt{X}_4$ equal to $\mathtt{X}_1 + \mathtt{X}_2$. This implies that subsequently executing the third assignment amounts to setting $\mathtt{X}_1$ to $\mathtt{X}_1 + \mathtt{X}_2$; repeatedly doing so yields polynomial growth. If we only state bounds on each variable separately we cannot complete the proof.

Basically, the sets of interest here are sets of variables that exchange their values, but do not duplicate them. We introduce another definition:

Definition D.11. Let C be a strongly connected component of the SRG. We define $\mathscr{F}(C)$ to be the family of sets $X \subseteq C$, such that for any matrix $A \in \mathscr{M}^*$, and $i \in C$,

$$A_{ij} > 0 \,\wedge\, A_{ik} > 0 \,\wedge\, j,k \in X \,\Rightarrow\, j = k\,.$$

Such a set is called *duplication-free*.

EXAMPLE D.12. Consider the program shown above and its SRG, shown in Figure 6. Let C be the SCC consisting of the nodes x_1, x_3 and x_4. The set $\{1,3,4\}$, representing all three nodes of the component, is *not* duplication-free. In fact, analysis of the assignment $\mathtt{X}_1 := \mathtt{X}_3 + \mathtt{X}_4$ yields dependences $3 \xrightarrow{1^+} 1$ and $3 \xrightarrow{1} 3$ (among others) as well as binary dependence $\substack{3\\3} \rightrightarrows \substack{1\\3}$. Consequently, $\mathscr{M}$ includes a matrix A with $A_{31} = 1^+$ and $A_{33} = 1$. Hence any set containing both 1 and 3 is not duplication-free. However, the set $\{3,4\}$ is, and this is used in proving that there is no exponential growth.

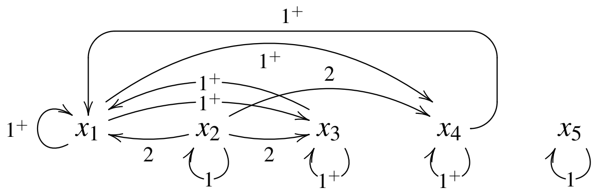


Figure 6: SRG for the example program. Nodes are labeled x_j rather than just j for readability.

We now summarize in a nutshell the proof of Lemma D.10. The proof constructs, for every $A \in \mathscr{M}^*$, and every duplication-free set X, a multi-polynomial $\vec{h}_{A,X}$, and proves that every trace in $[\![(\not{c}E)^*]\!]_{ts}$ is upper-bounded by one of these multi-polynomials (in the sense of Definition D.5, extended to comprise bounds on sets of variables). We remark that this provides bounds for individual variables since a singleton set is duplication-free. These polynomials are constructed by induction on the topological ordering of the SCCs, i.e., at every step of this induction we construct the polynomials, simultaneously, for all of $\mathscr{F}(C)$ for a component C. There is no space here to give the construction, not to show its soundness; the interested reader is referred to our technical report.

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla & Damiano Zanardini (2012): *Cost analysis of object-oriented bytecode programs*. Theoretical Computer Science 413(1), pp. 142–159, doi:10.1016/j.tcs.2011.07.009.

[2] Christophe Alias, Alain Darte, Paul Feautrier & Laure Gonnord (2010): *Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs*. In: Static Analysis, Proceedings of the 17th International Symposium, LNCS 6337, Springer, pp. 117–133, doi:10.1007/978-3-642-15769-1_8.

[3] Stephen Bellantoni & Stephen A. Cook (1992): *A New Recursion-Theoretic Characterization of the Polytime Functions*. Computational Complexity 2, pp. 97–110, doi:10.1007/BF01201998.

[4] Amir M. Ben-Amram (2010): *On Decidable Growth-Rate Properties of Imperative Programs*. In Patrick Baillot, editor: International Workshop on Developments in Implicit Computational complExity (DICE 2010), EPTCS 23, pp. 1–14, doi:10.4204/EPTCS.23.1.

[5] Amir M. Ben-Amram (2011): *Monotonicity Constraints for Termination in the Integer Domain*. Logical Methods in Computer Science 7, pp. 1–43, doi:10.2168/LMCS-7(3:4)2011.

[6] Amir M. Ben-Amram, Neil D. Jones & Lars Kristiansen (2008): *Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time*. In: Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008, LNCS 5028, Springer, pp. 67–76, doi:10.1007/978-3-540-69407-6_7.

[7] Amir M. Ben-Amram & Lars Kristiansen (2012): *On the Edge of Decidability in Complexity Analysis of Loop Programs*. International Journal on the Foundations of Computer Science 23(7), pp. 1451–1464, doi:10.1142/S0129054112400588.

[8] Amir M. Ben-Amram & Aviad Pineles (2016): *Flowchart Programs, Regular Expressions, and Decidability of Polynomial Growth-Rate*. CoRR Technical Report 1410.4011v4. Available at `http://arxiv.org/abs/1410.4011v4`.

[9] Ralph Benzinger (2001): *Automated complexity analysis of Nuprl extracted programs*. Journal of Functional Programming 11(1), pp. 3–31. doi:10.1017/S0956796800003865.

[10] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion & Hélène Touzet (2001): *Algorithms with polynomial interpretation termination proof.* Journal of Functional Programming 11, pp. 33–53, doi:10.1017/S0956796800003877.

[11] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs & Jürgen Giesl (2014): *Alternating Runtime and Size Complexity Analysis of Integer Programs*. In: 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 140–155, doi:10.1007/978-3-642-54862-8_10.

[12] Wei-Ngan Chin (1992): *Safe Fusion of Functional Expressions*. In: Proceedings of the 1992 ACM Conference on LISP and Functional Programming, LFP '92, ACM, pp. 11–20, doi:10.1145/141471.141494.

[13] Patrick Cousot (2005): *Proving program invariance and termination by parametric abstraction, Lagrangian relaxation, and semidefinite programming*. In: 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI05), LNCS 3385, Springer, pp. 1–24, doi:10.1007/978-3-540-30579-8_1.

[14] Saumya Debray & Nai wei Lin (1993): *Cost Analysis of Logic Programs*. ACM Transactions on Programming Languages and Systems 15, pp. 48–62, doi:10.1145/161468.161472.

[15] Lawrence C. Eggan (1963): *Transition graphs and the star-height of regular events.* The Michigan mathematical journal 10(4), pp. 385–397. doi:10.1307/mmj/1028998975.

[16] Andreas Goerdt (1992): *Characterizing complexity classes by general recursive definitions in higher types*. Information and Computation 101(2), pp. 202 – 218, doi:10.1016/0890-5401(92)90062-K.

[17] Sumit Gulwani, Krishna K. Mehra & Trishul M. Chilimbi (2009): *SPEED: precise and efficient static estimation of program computational complexity*. In Zhong Shao & Benjamin C. Pierce, editors: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, ACM, pp. 127–139. doi:10.1145/1480881.1480898.

[18] Martin Hofmann (2003): *Linear types and non-size-increasing polynomial time computation*. Information and Computation 183(1), pp. 57–85, doi:10.1016/S0890-5401(03)00009-9.

[19] Neil D. Jones & Lars Kristiansen (2009): *A Flow Calculus of mwp-Bounds for Complexity Analysis*. ACM Trans. Computational Logic 10(4), pp. 1–41. doi:10.1145/1555746.1555752.

[20] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl & Martin Hofmann (2010): *Static determination of quantitative resource usage for higher-order programs*. In: The 37th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, (POPL), ACM, pp. 223–236. doi:10.1145/1706299.1706327.

[21] Takumi Kasai & Akeo Adachi (1980): *A Characterization of Time Complexity by Simple Loop Programs*. Journal of Computer and System Sciences 20(1), pp. 1–17. doi:10.1016/0022-0000(80)90001-X.

[22] Lars Kristiansen & Karl-Heinz Niggl (2004): *On the computational complexity of imperative programming languages*. Theoretical Computer Science 318(1-2), pp. 139–161. doi:10.1016/j.tcs.2003.10.016.

[23] Daniel Le Métayer (1988): *ACE: an automatic complexity evaluator*. ACM Trans. Program. Lang. Syst. 10(2), pp. 248–266. doi:10.1145/42190.42347.

[24] Chin Soon Lee, Neil D. Jones & Amir M. Ben-Amram (2001): *The Size-Change Principle for Program Termination*. In: Proceedings of the Twenty-Eigth ACM Symposium on Principles of Programming Languages (POPL), January 2001, ACM press, pp. 81–92. doi:10.1145/360204.360210.

[25] Albert R. Meyer & Dennis M. Ritchie (1967): *The complexity of loop programs*. In: Proc. 22nd ACM National Conference, Washington, DC, pp. 465–469.

[26] Manuel Montenegro, Ricardo Peña & Clara Segura (2010): *A Space Consumption Analysis by Abstract Interpretation*. In: Foundational and Practical Aspects of Resource Analysis, LNCS 6324, Springer, pp. 34–50. doi:10.1007/978-3-642-15331-0_3.

[27] Karl-Heinz Niggl & Henning Wunderlich (2006): *Certifying Polynomial Time and Linear/Polynomial Space for Imperative Programs*. SIAM J. Comput 35(5), pp. 1122–1147. doi:10.1137/S0097539704445597.

[28] Aviad Pineles (2014): *Growth-Rate Analysis of Flowchart Programs*. Available at `http://www2.mta.ac.il/~amirben/projects/aviad_final.pdf`. MSc project, The Academic College of Tel-Aviv Yaffo.

[29] Xavier Rival & Laurent Mauborgne (2007): *The trace Partitioning Abstract Domain*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29(5), p. 26. doi:10.1145/1275497.1275501.

[30] M. Rosendahl (1989): *Automatic Complexity Analysis*. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture FPCA'89*, ACM, pp. 144–156. doi:10.1145/99370.99381.

[31] Alejandro Serrano, Pedro López-García & Manuel V. Hermenegildo (2014): *Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types*. *TPLP* 14(4-5), pp. 739–754, doi:10.1017/S147106841400057X.

[32] Michael Sipser (2012): *Introduction to the theory of computation*, 3rd edition. PWS Publishing Company.

[33] Ben Wegbreit (1975): *Mechanical Program Analysis*. *Communications of the ACM* 18(9), pp. 528–539. doi:10.1145/361002.361016.

[34] Florian Zuleger, Sumit Gulwani, Moritz Sinn & Helmut Veith (2011): *Bound Analysis of Imperative Programs with the Size-Change Abstraction*. In: *Proceedings of the 8th International Symposium on Static Analysis, SAS 2011*, *LNCS* 6887, pp. 280–297. doi:10.1007/978-3-642-23702-7_22.