HAL open science

pymwp: A Static Analyzer Determining Polynomial Growth Bounds

Clément Aubert, Thomas Rubiano, Neea Rusch, Thomas Seiller

To cite this version:

Clément Aubert, Thomas Rubiano, Neea Rusch, Thomas Seiller. pymwp: A Static Analyzer Determining Polynomial Growth Bounds. Automated Technology for Verification and Analysis (ATVA 2023), Oct 2023, Singapour, France. 10.1007/978-3-031-45332-8_14 . hal-03269121v4

HAL Id: hal-03269121

https://hal.science/hal-03269121v4

Submitted on 16 May 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

pymwp: A Static Analyzer Determining Polynomial Growth Bounds*

Clément Aubert[1 [0000−0001−6346−3043]], Thomas Rubiano[2], Neea Rusch[1 [0000−0002−7354−5330]], and Thomas Seiller[2,3 [0000−0001−6313−0898]]

[1] School of Computer and Cyber Sciences, Augusta University
[2] LIPN – UMR 7030 Université Sorbonne Paris Nord
[3] CNRS

Abstract. We present pymwp, a static analyzer that automatically computes, if they exist, polynomial bounds relating input and output sizes. In case of exponential growth, our tool detects precisely which dependencies between variables induced it. Based on the sound mwp-flow calculus, the analysis captures bounds on large classes of programs by being non-deterministic and not requiring termination. For this reason, implementing this calculus required solving several non-trivial implementation problems, to handle its complexity and non-determinism, but also to provide meaningful feedback to the programmer. The duality of the analysis result and compositionality of the calculus make our approach original in the landscape of complexity analyzers. We conclude by demonstrating experimentally how pymwp is a practical and performant static analyzer to automatically evaluate variable growth bounds of `C` programs.

Keywords: Static Program Analysis · Automatic Complexity Analysis · Program Verification · Bound Inference · Flow Analysis

1 Introduction – Making Use of Implicit Complexity

Certification of any program is incomplete if it ignores resource considerations, as runtime failure will occur if usage exceeds available capacity. To address this deficiency, automatic complexity analysis produced many different implementations [9,13,14,15] with varying features. This paper presents the development and specificities of our automatic static complexity analyzer, pymwp.

The first original dimension of our tool is its inspiration, coming from Implicit Computational Complexity (ICC) [10]. This field designs systems guaranteeing program's runtime resource usage that tend to possess practically useful properties. For this reason, it is conjectured that ICC systems could be used to achieve realistic complexity analysis [18, p. 16]. Our series of work [5,6] is testing this hypothesis, and resulted in the tool we present in this paper: pymwp is one of the first ICC-inspired applications, and the first mechanization of the specific technique it implements. Let us first exemplify what pymwp calculates.

* This research is supported by the Transatlantic Research Partnership. Rubiano and Seiller are supported by the Île-de-France through the DIM RFSI project "CoHOp".

Example 1. Consider an imperative program with a fixed number of parameters:

```
void increasing(int X1, int X2, int X3) {
 while (X2 < X1) { X2 = X1 + X1; }
 while (X3 < X2) { X3 = X2 + X2; }
}
```

Independently of the arguments passed (henceforth called initial values), once computation concludes, `X1` will hold the same value, but the values held by `X2` and `X3` may have changed. By manual analysis, we can deduce that the variable values "growth bound" between the *initial values* `X1`, `X2` and `X3` (overloading initial values and parameter names) and their *final values* (denoted `X1'`, `X2'` and `X3'`), omitting constants, is $\texttt{X1'} = \texttt{X1}$, $\texttt{X2'} \leq \max(\texttt{X1}, \texttt{X2})$ and $\texttt{X3'} \leq \max(\texttt{X3}, \texttt{X2} + \texttt{X1})$.[4] Therefore, for all initial values, the value growth of the variable's value is bounded by a polynomial w.r.t. its initial values. Our analysis is designed to either produce such bounds, or to pinpoint variables that grow exponentially.

Introducing more variables, or potentially non-terminating iteration, or complicating the logic would make manual analysis difficult. However, our static analyzer handles all those cases automatically. It determines if a program accepts at least one polynomial bounding the final value of its variables in terms of their initial values—what we call its *growth bound*. If a bound cannot be established, it provides feedback on sources of failure, identifying variable pairs that have "too strong" dependencies. The technique is sound [16, p. 11], meaning a positive result guarantees program has satisfactory value growth behavior at runtime.

The mwp-flow analysis [16], that powers this tool, is of interest for its flexibility, originality, and uncommon features [9] such as being compositional and not requiring termination. However, using it to implement an automatic analyzer required important theoretical adjustments, and to sidestep or solve computationally expensive steps in the derivation of the bounds. For example, approaches to determine bounds were motivated by the need to compute them rapidly and present them in a concise human-interpretable manner, which is problematic for potentially exponential number of outputs. The theoretical improvements were presented previously [5] and serve as basis for pymwp. In this paper, we focus on the tool and its recent advancements, with following contributions.

1. We present the static analyzer pymwp in Sect. 3. It evaluates automatically if an input program has a polynomial growth bound and provides actionable feedback on failure. Our tool is easy to use and install; open-source, well-documented, and persistently available for future reuse.
2. Implementing the theoretical mwp-calculus required to solve several non-trivial implementation problems. Specifically, how to obtain fast and concise results was solved by recent tool developments, and discussed in Sect. 4.

[4] Observe that the bound for `X3'` involves `X1` *and* `X2`: the presence of `X1` in the bound of `X2'` transitively impacts the bound for `X3'`, because the analysis is compositional.

3. Sect. 5 demonstrates that pymwp is a practical and performant static analyzer by experimentally analyzing a set of canonical `C` programs. The evaluation also includes every example presented in this paper.

2 Calculating Bounds with mwp-Analysis

Given a deterministic imperative program over integers constructed using `while`, `if` and assignments, the mwp-analysis aims at discovering the polynomials bounding the variables final values `X1`', ..., `Xn`' in terms of their initial values `X1`, ..., `Xn` [16, p. 5]. This section gives insight on how to interpret the results of pymwp, exemplifies those bounds in more detail, and identifies its distinctive features in the landscape of automatic complexity analysis.

2.1 Interpreting Analysis Results: mwp-Bounds and ∞

The mwp-flow analysis internally captures dependencies between program's variables to determine existence of growth bounds and locates problematic data flow relations. A flow can be 0, meaning no dependency; *m*aximal of linear, *w*eak polynomial, *p*olynomial or ∞, in increasing order of dependency. When the value of *every variable* in a program is bounded by at most a polynomial in initial values, the flow calculus assigns each variable an *mwp-bound*. It is a number-theoretic expression of form $\max(\vec{x}, \text{poly}_1(\vec{y})) + \text{poly}_2(\vec{z})$, where variables characterized by m-flow are listed in $\vec{x}$; w-flows in $\vec{y}$, and p-flows in $\vec{z}$. Honest polynomials poly_1 and poly_2 are build up from constants and variables by applying $+$ and $\times$. Any of the three variable lists might be empty and poly_1 and poly_2 may not be present. A bound of a program is conjunction ($\wedge$) of mwp-bounds. Variables that depend "too strongly" are assigned ∞-flow, to indicate exponential growth.

Expression	reads as "The growth of `X`' is bounded by ..."
`X`' ≤ 0	"... a constant."
`X`' $\leq$ `X`	"... a polynomial in `X`." (its initial value)
`X`$' \leq \max($`X`$,$ `X1`$) +$ `X2` $\times$ `X3`	"... a polynomial in `X` or `X1`, `X2` and `X3`."

Determining program bounds is complicated because the flow calculus is non-deterministic. This enables to analyze a larger class of programs, but also means that one program may be assigned multiple bounds. If a program is assigned a bound, it is derivable in the calculus. An impossibility result occurs when all derivation "paths" yields an ∞-result.

2.2 Additional Foundational Examples

One important and original aspect is that the mwp-flow analysis ignores Boolean conditions, assuming that both `if`-branches evaluate, and that loops executes an arbitrary number of cycles. This lets pymwp analyze non-terminating programs without complications, and justifies why all conditions will be abstracted as `b`.

Letting `C1` ≡ `X2 = X1 + X1` and `C2` ≡ `X3 = X2 + X2`, Example 1 established that the iterative composition `while b C1; while b C2` has a polynomial growth bound, i.e., the property of interest.[5] We now elaborate on mwp-flow analysis behavior by inspecting two more expository programs, letting `C3` ≡ `X3 = X3 * X3`.

Example 2. Consider program `while b C3`. Even if `C3` in itself admits the bound $\texttt{X3'} \leq \texttt{X3}$, the value stored in variable `X3` will grow exponentially on each iteration. Therefore, the program cannot get a growth bound, due to the ∞ flow between `X3` and itself introduced by the `while` statement.

Example 3. Combining elements from the previous two examples, we construct `while b C1; while b C2; C3`. Variables `X1` and `X2` are unaffected by `C3`, but `X3` changes. We over-approximate the final value of `X3` to obtain the program's growth bound $\texttt{X1'} \leq \texttt{X1} \wedge \texttt{X2'} \leq \max(\texttt{X2}, \texttt{X1}) \wedge \texttt{X3'} \leq \texttt{X1} + \texttt{X2} + \texttt{X3}$. This example shows how partial results ($\texttt{X3'} \leq \max(\texttt{X3}, \texttt{X1}+\texttt{X2})$ and $\texttt{X3'} \leq \texttt{X3}$) can be combined to obtain new bounds ($\texttt{X3'} \leq \texttt{X1} + \texttt{X2} + \texttt{X3}$) by compositionality.

In the tool user guide, we present even more examples with in-depth discussion, to elaborate on the behavior and results of mwp-analysis.

2.3 Originalities of mwp-flow Analysis

The mwp technique offers many properties that make it unique and practically useful. It is a syntactic analysis, not based on general purpose reasoners e.g., abstract interpreters or model checkers. It requires little structure, and no manual annotation from the analyzed program. This enables its implementation on any imperative programming language, and potentially at different stages of compilation. Compositionality is another significant feature. Non-compositional techniques require inlining programs and are common among automated complexity analyses [9]. With compositionality, analysis can be performed on parts of whole-programs, and after refactoring, repeated only on those parts that changed.

Several tools that evaluate resource bounds already exist [1,2,8,12,13,14,15,17]; including LOOPUS [25] and C4B [9], that specialize in `C` language inputs. Comprehensive evaluations of these tools have also been performed recently [9,11,25]. The main distinguishing factor between these tools and pymwp is the program's complexity property of interest: pymwp evaluates the existential of polynomial growth bounds w.r.t. initial values. We illustrate the difference in obtained bounds in Table 1. It is not an extensive comparison but suffices to show that pymwp differs in its aims from the other related techniques.

3 Technical Overview of pymwp

In this section we present the main contribution of the paper: the pymwp static analyzer. It is a command-line tool that analyzes programs written in subset

[5] pymwp actually outputs $\texttt{X1'} \leq \texttt{X1} \wedge \texttt{X2'} \leq \max(\texttt{X2}, \texttt{X1}) \wedge \texttt{X3'} \leq \max(\texttt{X3}, \texttt{X1} + \texttt{X2})$.

Input	LOOPUS	C4B	pymwp
`t19.c`	$\max(0, i - 10^2) + \max(0, k + i + 51)$	$50 + \lvert[-1, i]\rvert + \lvert[0, k]\rvert$	$\texttt{i'} \leq \texttt{i} + \texttt{k} \wedge \texttt{k'} \leq \texttt{k}$
`t20.c`	$2 \cdot \max(0, y - x) + \max(0, x - y)$	$\lvert[x, y]\rvert + \lvert[y, x]\rvert$	$\texttt{x'} \leq \texttt{x} \wedge \texttt{y'} \leq \texttt{y}$
`t47.c`	$1 + \max(n, 0)$	$1 + \lvert[0, n]\rvert$	$\texttt{n'} \leq \texttt{n} \wedge \texttt{flag'} \leq 0$

Table 1. Comparison of obtained resource bounds for various `C` language analyzers, on examples from Carbonneaux et al. [9, p. 26]. LOOPUS and C4B find asymptotically tight bounds based on amortization. LOOPUS calculates bounds on loop iterations, C4B derives global whole-program bounds, and pymwp analyzes variables growths. The inputs are part of Sect. 5 benchmark suite, and available in the pymwp repository [24].

of `C` programming language presented in Sect. 3.3. The name alludes to its implementation language, Python, which we selected for its flexibility and use in previous related works [4,19,20]. Our tool takes as input a path to a `C` program, and returns for each function it contains a growth bound—if at least one can be established—or a list of variable dependencies that may cause the exponential growth .[6] The pymwp development is open source [24] with releases published at Python Package Index (PyPI) [22], GitHub [24] and Zenodo [7]. A tool user guide is available at `https://statycc.github.io/.github/pymwp/`.

3.1 Program Analysis in Action

The default procedure for performing mwp-analysis is as follows:

1. Parse input file to obtain an abstract syntax tree (AST).
2. Initialize a `Result` object T.
3. For each function (or "program", interchangeably) in the AST:
   (a) Create an initial `Relation` R—briefly, this complex structure represents variables and their dependencies, at a program point (Sect. 4.1).
   (b) Sequentially for each statement in function body:
      i. Recursively apply inference rules to obtain R_i.
      ii. Compose R_i with previous relation: $R = R \circ R_i$.
      iii. If no bound exists, terminate analysis of function body.
   (c) If bounds exist, evaluate R to determine the bounds (Sect. 4.3)
   (d) Append function analysis result to T.
4. Return T.

3.2 Usage

There are multiple ways to use pymwp. It has a text-based application interface, and can be run from terminal, or it can be imported as a Python module into larger software engineering developments. The analysis is automatic and read-only, therefore it is possible to pair pymwp with other tools and integrate it into

[6] Obtaining this feedback requires to specify the `--fin` argument.

compilation or verification toolchains. The online demo provides one example use case. It is a web server application with pymwp as a package dependency. Other derived uses can be developed similarly. The easiest way to install pymwp is from PyPI, using command `pip install pymwp`. The default interaction command is

```
___pymwp path/to/file.c [args]
```

where the first positional argument is required. By default, pymwp displays the analysis result with logging information, and writes the result to a file. This behavior is customizable by specifying arguments. For a list of currently supported arguments run `pymwp -help`.

3.3 Scope of Analyzable Programs

The programs analyzable with pymwp are determined by its supported syntax. pymwp delegates the task of parsing `C` files to its dependency, pycparser [21], which aims to support the full `C99` specification. Programs that cannot be parsed will expectedly throw an error. Otherwise, analysis proceeds on the generated AST, and pymwp handles nodes that are syntactically supported by its calculus.[7] It skips unsupported nodes with a warning. We decided on this permissive approach, because it allows to obtain partial results and manually inspect unsupported operations. However, to establish a guaranteed of bound, the input program must fully conform to the supported syntax of the calculus. Currently the syntax has limitations, e.g., arrays and pointer operations are unsupported. Extending the analysis to richer syntax is a direction for future work.

4 Implementation Advancements

Notable technical progress has occurred since the initial mention of pymwp in the literature [5].[8] We will discuss those solutions in this section.

4.1 Motivations for Refining Analysis Results

Understanding pymwp's advances requires to briefly reflect on its past. The mwp-flow, as originally designed [16], is an inference system that has an unbearable computational cost, as it manipulates non-deterministically an exponential number of sizable matrices [5, Sect. 2.3] to try to establish a bound. Our enhanced mwp-technique [5] resolved this challenge by internalizing the non-determinism. This way, all derivations—including the ones that will fail—are constructed at the same time, moving the problem from "Is there a derivation?" to "Among all the derivations you constructed, is there one without ∞ coefficient?"—an equivalent question that however complicates the production of the actual bound.

[7] List of supported features: `https://statycc.github.io/pymwp/features`.
[8] Full comparison: `https://github.com/statycc/pymwp/compare/FSCD22...0.4.2`.

While answering the first question is too computationally expensive, pymwp's FSCD 2022 version can answer the second, and it can further, if all derivations contain ∞ coefficients, terminate early for faster result [5, Sect. 4.4]. This was achieved thanks to a complex `Relation` data structure,[9] but extracting finer information from that data structure remained an outstanding problem. In particular, we wanted to provide the following feedback to the programmer: (i) If no bound exists, the location of the exponential growth. (ii) If bounds exist, the value of at least one of them. The current version of pymwp can now provide this feedback, thanks to a long maturation that we now detail.

4.2 Exposing Sources of Failure

Since pymwp identifies polynomial bounds, it reports failure on programs containing at least one variable whose value grows exponentially w.r.t. at least one of its initial value. Earlier tool versions would indicate that failure was detected without reporting the involved variables. Determining this information is complicated because of our treatment of non-determinism, but it is valuable, as addressing one of those points of failure would suffice to obtain a polynomial growth bound. Even if the program cannot be refactored satisfactorily, then analyzing the exponential growth allows to assess potential impact on the parent software application.

Our solution is to record additional information about ∞-coefficient in the `Relation` data structure, and to list all variable pairs on which failure may occur. Since detailed failure information may not be relevant in some use-cases, and is costly to compute, it was added as an optional `-fin` argument.

Example 4. From our tool user guide (output abridged for clarity):

```
int foo(int X1, int X2, int X3){
 if (X1 == 1){
 X1 = X2+X1;
 X2 = X3+X2;
 }
 while(X1 < 10){
 X1 = X2+X1;
 }
}
```

```
$ pymwp infinite/infinite_3.c
↪ --fin
foo is infinite
Possibly problematic flows:
X1 → X1 ‖ X2 → X1 ‖ X3 →
↪ X1
```

Reads as "`X1` depends too strongly on all variables."

4.3 Efficiently Determining Bounds

In the alternative case, where bounds are determined to exist, the next step is to evaluate the bounds—step 3(c) in the pymwp workflow (Sect. 3.1). This is

[9] A complex data structure sounds daunting, but it is in fact one of the highlights of the system, and enables to solve a difficult derivation problem efficiently. For details, see the documentation at `https://statycc.github.io/pymwp/relation`.

problematic because the calculus can yield an exponential number of bounds w.r.t. the program size, as illustrated in Table 3 with e.g., benchmark 32. long. As a result, the evaluation phase—e.g., extracting from this conglomerate of derivations the bounds—is increasingly costly. Handling this task efficiently required to discover a computational solution, and finding a compact format to represent the results in interpretable and memory-efficient manner. For simplicity we describe this process only at high-level, but refer to the implementation for complete details.

Determining mwp-bounds requires two separate steps, starting with the `Relation` data structure generated during analysis phase. The first challenge is to determine which paths in our conglomerate of derivations produce bounds (i.e., does not contain ∞). A naïve brute force solution would iterate over all paths, but this is too slow for practical use. Instead, we developed a set-theoretic approach, that determines first all derivation paths that lead to ∞ and then negates those paths. We capture this process in a structure called `Choice`, and the result of this computation is called a choice vector. A choice vector contains all derivation paths yielding a bound in a compact, regular expression-like representation. Once those paths are known, it is possible to extract from a `Relation` an mwp-bound (represented as a `Bound` object), by applying a selected path. Currently, we take the first choice from the choice vector, and display it as a result. Leveraging this set of bounds and its utilities is discussed in conclusion and left for future work.

5 Experimental Evaluation

To establish that pymwp is a practical and performant static analyzer, we evaluated it on a benchmark suite of canonical `C` programs. We ran the analyzer on the benchmarks and measured the results, thus conducting an evaluation of performance and behavioral correctness. We did not perform tool comparison or use a standard suite for two reasons: absence of a representative comparison target (cf. Sect. 2.3) and syntactic restrictions that limit the scope of analyzable programs (cf. Sect. 3.3). However, the choice methodology judiciously evaluates pymwp, and facilitates transparency and reproduction of experiments. We actively put heavy emphasis to ensure—with software engineering best practices e.g., tests, documentation [23], and long-term archival deposits [7]—that pymwp, and the evaluation presented here, are available and reusable for future comparisons.

5.1 Methodology

Benchmarks Description The suite contains 50 `C` programs, written in the subset of `C99` syntax supported by pymwp. The benchmarks are designed purposely to exercise various data flows that pose challenges to the analyzer, e.g., increasing parameters, binary operations, loops and decisions, and various combinations of those operations. The benchmarks are organized into seven categories based on their expected result (∞ vs. non-∞); origin in related publications [5,16], and in this tool paper; and interest (basic examples, others). We

omit these categories here, but they are apparent in the benchmarks distribution. The suite is available from pymwp repository [24], and as a release asset on GitHub, and Zenodo [7].

Metrics For each benchmark, we record 1. benchmark name, corresponding to `C` file name, followed by "*: program name*" if a file contains multiple programs. 2. The lines of code (loc) in the benchmark. Observe this number ranges between 4 and 45: this is reasonable and representative, because the analysis is compositional. Analysis of even a large `C` file reduces to analysis of its functions, that would be expectedly similar in size to these benchmarks. 3. The time (ms) required to complete program analysis. We use milliseconds for precision since all analyses conclude within seconds. For ∞-programs, the time is for performing full evaluation with feedback, although a result of existential failure could be obtained faster. 4. Number of program initial values (vars), which internally impacts complexity of the analysis. 5. Number of polynomial bounds discovered by the analyzer. The number of bounds is 0 if the result is ∞. 6. If a program is derivable, we capture one of its bounds.

Experimental Setup The measurements were performed on a Linux x86_64, kernel v.5.4.0-1096-gcp, Ubuntu 18.04, with 8 cores and 32 GB virtual memory. The machine impacts only observed execution time; other metrics are deterministic. The software environment was Python runtime 3.8.0, gcc 7.5.0, GNU Make 4.1, and the dev dependencies of pymwp. Because the measurement utilities of pymwp are not distributed with its release, the experiments must be run from source. We used source code version 🏷 0.4.2. The command to repeat experiments is `make bench`. It runs analysis on benchmarks and generates two tables of results.

5.2 Results

The evaluation results are presented in Table 2. The analyzer correctly finds a polynomial bound for noninfinite benchmarks, and rejects exponential and infinite benchmarks. The analyzer is also able to derive bounds for potentially non-terminating `while` benchmarks. Observe that the analysis concludes rapidly even for a long example with 45 loc, and for explosion, that has initial values count 18. The number of bounds for long is high, because it is a complicated derivation with high degree of internalized non-determinism.

For programs that have polynomial growth bounds, we give a simplified example bound in Table 3. We omit in this representation variables whose only dependency is on self, e.g., `X'` $\leq$ `X`.[10] The table serves to demonstrate that pymwp can derive complex multivariate bounds automatically, and to present the result of a non-deterministic computation in a digestible form. It also clarifies what the analyzer computes and that those results are original in form.

[10] Bound of example5_1 does not appear in Table 3 because of this simplification.

#	Benchmark	loc	ms	vars	bounds	#	Benchmark	loc	ms	vars	bounds
1.	assign_expression	8	0	2	3	26.	infinite_4	9	2189	5	0
2.	assign_variable	9	0	2	3	27.	infinite_5	11	518	5	0
3.	dense	16	15	3	81	28.	infinite_6	14	1031	4	0
4.	dense_loop	17	66	3	81	29.	infinite_7	15	298	5	0
5.	example14: *f*	4	2	2	1	30.	infinite_8	23	722	6	0
6.	example14: *foo*	11	0	2	3	31.	inline_variable	9	0	2	3
7.	example16	15	7	4	27	32.	long	45	2875	5	177147
8.	example3_1_a	10	1	3	9	33.	notinfinite_2	4	1	2	9
9.	example3_1_b	10	2	3	9	34.	notinfinite_3	9	7	4	9
10.	example3_1_c	11	3	3	1	35.	notinfinite_4	11	30	5	3
11.	example3_1_d	12	1	2	0	36.	notinfinite_5	11	29	4	9
12.	example3_2	12	2	3	0	37.	notinfinite_6	16	34	4	81
13.	example3_4	22	14	5	0	38.	notinfinite_7	15	283	5	9
14.	example5_1	10	0	2	1	39.	notinfinite_8	22	856	6	27
15.	example7_10	10	1	3	9	40.	simplified_dense	9	1	2	9
16.	example7_11	11	9	4	27	41.	t19_c4b	9	2	2	81
17.	example8	8	1	3	9	42.	t20_c4b	7	1	2	9
18.	explosion	23	405	18	729	43.	t47_c4b	12	1	2	3
19.	exponent_1	16	7	4	0	44.	tool_ex_1	7	5	3	1
20.	exponent_2	13	4	4	0	45.	tool_ex_2	7	0	2	0
21.	gcd	12	10	2	0	46.	tool_ex_3	9	8	3	3
22.	if	7	0	2	3	47.	while_1	7	1	2	3
23.	if_else	7	0	2	9	48.	while_2	7	1	2	1
24.	infinite_2	6	16	2	0	49.	while_if	9	3	3	9
25.	infinite_3	9	7	3	0	50.	xnu	26	17	5	6561

Table 2. Benchmark results for a canonical test suite of `C` programs. Benchmark that have 0 bounds represents case where analyzer reports an ∞-result.

#	Benchmark bound	#	Benchmark bound
1.	$\mathtt{y2}' \leq \mathtt{y1}$	34.	$\mathtt{X0}' \leq \max(\mathtt{X0}, \mathtt{X1}) + \mathtt{X2} \times \mathtt{X3}$ $\wedge\ \mathtt{X1}' \leq \mathtt{X1} + \mathtt{X2} \wedge \mathtt{X2}' \leq \mathtt{X2} + \mathtt{X3}$
2.	$\mathtt{x}' \leq \mathtt{y}$	35.	$\mathtt{X1}' \leq \max(\mathtt{X1}, \mathtt{X2} + \mathtt{X3}) \wedge \mathtt{X2}' \leq \max(\mathtt{X2}, \mathtt{X3})$ $\wedge\ \mathtt{X4}' \leq \max(\mathtt{X4}, \mathtt{X5})$
3.	$\mathtt{X0}' \leq \max(\mathtt{X0}, \mathtt{X2}) + \mathtt{X1} \wedge \mathtt{X1}' \leq \mathtt{X0} \times \mathtt{X1} \times \mathtt{X2}$ $\wedge\ \mathtt{X2}' \leq \max(\mathtt{X0}, \mathtt{X2}) + \mathtt{X1}$	36.	$\mathtt{X1}' \leq \max(\mathtt{X1}, \mathtt{X4}) + \mathtt{X2} \times \mathtt{X3}$ $\wedge\ \mathtt{X2}' \leq \max(\mathtt{X2}, \mathtt{X4}) + \mathtt{X3}$ $\wedge\ \mathtt{X3}' \leq \max(\mathtt{X3}, \mathtt{X4})$
4.	$\mathtt{X0}' \leq \max(\mathtt{X0}, \mathtt{X2}) + \mathtt{X1} \wedge \mathtt{X1}' \leq \mathtt{X0} \times \mathtt{X1} \times \mathtt{X2}$ $\wedge\ \mathtt{X2}' \leq \max(\mathtt{X0}, \mathtt{X2}) + \mathtt{X1}$	37.	$\mathtt{X1}' \leq \max(\mathtt{X1}, \mathtt{X4}) + \mathtt{X2} \times \mathtt{X3}$ $\wedge\ \mathtt{X2}' \leq \max(\mathtt{X2}, \mathtt{X4}) + \mathtt{X3}$
5.	$\mathtt{X2}' \leq \max(\mathtt{X2}, \mathtt{X1})$	38.	$\mathtt{X1}' \leq \max(\mathtt{X1}, \mathtt{X2} + \mathtt{X3} + \mathtt{X4} + \mathtt{X5})$ $\wedge\ \mathtt{X2}' \leq \max(\mathtt{X2}, \mathtt{X3} + \mathtt{X4} + \mathtt{X5})$ $\wedge\ \mathtt{X3}' \leq \max(\mathtt{X3}, \mathtt{X4} + \mathtt{X5})$ $\wedge\ \mathtt{X4}' \leq \max(\mathtt{X4}, \mathtt{X5})$
6.	$\mathtt{X2}' \leq \mathtt{X1}$	39.	$\mathtt{X1}' \leq \mathtt{X1} + \mathtt{X2} \times \mathtt{X3} \times \mathtt{X4} \times \mathtt{X5}$ $\wedge\ \mathtt{X2}' \leq \max(\mathtt{X2}, \mathtt{X1} + \mathtt{X3} + \mathtt{X4} + \mathtt{X5})$ $\wedge\ \mathtt{X3}' \leq \max(\mathtt{X3}, \mathtt{X1} + \mathtt{X4} + \mathtt{X5}) + \mathtt{X2}$ $\wedge\ \mathtt{X4}' \leq \max(\mathtt{X4}, \mathtt{X1} + \mathtt{X5}) + \mathtt{X2}$ $\wedge\ \mathtt{X6}' \leq \max(\mathtt{X6}, \mathtt{X1} + \mathtt{X3} + \mathtt{X4} + \mathtt{X5}) + \mathtt{X2}$
7.	$\mathtt{X1}' \leq \mathtt{R} + \mathtt{X1} \wedge \mathtt{X2}' \leq \mathtt{X1} \wedge \mathtt{X_1}' \leq \mathtt{X1} \wedge \mathtt{R}' \leq \mathtt{R} + \mathtt{X1}$	40.	$\mathtt{X0}' \leq \mathtt{X0} + \mathtt{X1} \wedge \mathtt{X1}' \leq \mathtt{X1} + \mathtt{X0}$
8.	$\mathtt{X1}' \leq \mathtt{X2} + \mathtt{X3}$	41.	$\mathtt{i}' \leq \mathtt{i} + \mathtt{k}$
9.	$\mathtt{X1}' \leq \mathtt{X2} \times \mathtt{X3}$	43.	$\mathtt{flag}' \leq 0$
10.	$\mathtt{X1}' \leq \max(\mathtt{X1}, \mathtt{X2} + \mathtt{X3})$	44.	$\mathtt{X2}' \leq \max(\mathtt{X2}, \mathtt{X1}) \wedge \mathtt{X3}' \leq \max(\mathtt{X3}, \mathtt{X1} + \mathtt{X2})$
15.	$\mathtt{X3}' \leq \mathtt{X3} + \mathtt{X1} \times \mathtt{X2}$	46.	$\mathtt{X2}' \leq \max(\mathtt{X2}, \mathtt{X1}) \wedge \mathtt{X3}' \leq \mathtt{X1} + \mathtt{X2} + \mathtt{X3}$
16.	$\mathtt{X1}' \leq \mathtt{X1} + \mathtt{X2} \times \mathtt{X3} \times \mathtt{X4} \wedge \mathtt{X2}' \leq \mathtt{X2} + \mathtt{X3} \times \mathtt{X4}$ $\wedge\ \mathtt{X3}' \leq \mathtt{X3} + \mathtt{X4}$	47.	$\mathtt{y}' \leq \max(\mathtt{x}, \mathtt{y})$
17.	$\mathtt{X1}' \leq \mathtt{X1} + \mathtt{X2} \times \mathtt{X3}$	48.	$\mathtt{x}' \leq \max(\mathtt{x}, \mathtt{y})$
18.	$\mathtt{x0}' \leq \mathtt{x1} + \mathtt{x2} \wedge \mathtt{x3}' \leq \mathtt{x4} + \mathtt{x5} \wedge \mathtt{x6}' \leq \mathtt{x7} + \mathtt{x8}$ $\wedge\ \mathtt{x9}' \leq \mathtt{x10} + \mathtt{x11} \wedge \mathtt{x12}' \leq \mathtt{x13} + \mathtt{x14}$ $\wedge\ \mathtt{x15}' \leq \mathtt{x16} + \mathtt{x17}$	49.	$\mathtt{y2}' \leq \max(\mathtt{y2}, \mathtt{y1}) \wedge \mathtt{r}' \leq \max(\mathtt{y2}, \mathtt{y1})$
22.	$\mathtt{y}' \leq \max(\mathtt{x}, \mathtt{y})$	50.	$\mathtt{beg}' \leq 0 \wedge \mathtt{end}' \leq 0 \wedge \mathtt{i}' \leq 0$
23.	$\mathtt{x}' \leq \max(\mathtt{x}, \mathtt{y}) \wedge \mathtt{y}' \leq \max(\mathtt{x}, \mathtt{y})$		
31.	$\mathtt{y2}' \leq \mathtt{y1}$		
32.	$\mathtt{X0}' \leq \mathtt{X2} + \mathtt{X1} \times \mathtt{X4} \wedge \mathtt{X1}' \leq \max(\mathtt{X2}, \mathtt{X3}, \mathtt{X4}) + \mathtt{X1}$ $\wedge\ \mathtt{X2}' \leq \max(\mathtt{X2}, \mathtt{X3}) + \mathtt{X1} \times \mathtt{X4}$ $\wedge\ \mathtt{X3}' \leq \max(\mathtt{X2}, \mathtt{X3}) + \mathtt{X1}$ $\wedge\ \mathtt{X4}' \leq \mathtt{X1} \times \mathtt{X2} \times \mathtt{X3} \times \mathtt{X4}$		
33.	$\mathtt{X0}' \leq \mathtt{X0} + \mathtt{X1} \wedge \mathtt{X1}' \leq \mathtt{X0} \times \mathtt{X1}$		

Table 3. Examples of obtained bounds for corresponding benchmarks. For compactness, the bounds are simplified to exclude variables that have dependency only on self.

6 Conclusion

This paper presented pymwp, its recent technical advancements, and evaluated its performance. Our tool reasons efficiently about existence of the variables' growth bounds w.r.t. its initial value, and can be paired with other tools for extended verification and compound analyses. Possible enhancements of the tool involve extending it to support richer syntax, and exploring the space of discovered bounds. For example, we could investigate whether constraints such as "Is there a bound where this particular variable growth linearly?" can be satisfied. Another open question is to identify *distinct* bounds.

Beyond enhancements of pymwp, several future directions and extended applications can follow. Perhaps the most interesting of those is to formally verify the analysis technique, and work is already underway in that direction [3]. Since the analysis does not require much structure from an input program, it could be useful for analyzing intermediate representations during compilation. It could also find use cases in restricted domain-specific languages, and resource-restricted hardware, to establish guarantees of their runtime behavior. Long term, the fast compositional analysis could also be useful to construct IDE plug-ins to provide low-latency feedback to programmers.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of object-oriented bytecode programs. Theoretical Computer Science **413**(1), 142–159 (2012). https://doi.org/10.1016/j.tcs.2011.07.009
2. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Static Analysis: 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings 17. pp. 117–133. Springer International Publishing (2010). https://doi.org/10.1007/978-3-642-15769-1_8
3. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: Certifying complexity analysis (2023), https://hal.science/hal-04083105v1/file/main.pdf, presented at the Ninth International Workshop on Coq for Programming Languages (CoqPL)
4. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: LQICM On C Toy Parser (3 2021), https://github.com/statycc/LQICM_On_C_Toy_Parser
5. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: mwp-analysis improvement and implementation: Realizing implicit computational complexity. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Leibniz International Proceedings in Informatics, vol. 228, pp. 26:1–26:23. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.26
6. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: Realizing Implicit Computational Complexity (Mar 2022), https://hal.archives-ouvertes.fr/hal-03603510, presented at the 28th International Conference on Types for Proofs and Programs (TYPES 2022) (Recording)
7. Aubert, C., Rubiano, T., Rusch, N., Seiller, T.: pymwp: MWP analysis on C code in Python (May 2023). https://doi.org/10.5281/zenodo.7908484
8. Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing runtime and size complexity of integer programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **38**(4), 1–50 (2016). https://doi.org/10.1145/2866575
9. Carbonneaux, Q., Hoffmann, J., Shao, Z.: Compositional certified resource bounds. In: Grove, D., Blackburn, S.M. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 467–478. Association for Computing Machinery (2015). https://doi.org/10.1145/2737924.2737955
10. Dal Lago, U.: A short introduction to implicit computational complexity. In: Bezhanishvili, N., Goranko, V. (eds.) ESSLLI. Lecture Notes in Computer Science, vol. 7388, pp. 89–109. Springer (2011). https://doi.org/10.1007/978-3-642-31485-8_3
11. Flores Montoya, A.: Cost analysis of programs based on the refinement of cost relations. Ph.D. thesis, Technische Universität Darmstadt (2017), https://tuprints.ulb.tu-darmstadt.de/6746/1/Thesis_Antonio_Flores.pdf
12. Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Programming Languages and Systems (APLAS 2014). pp. 275–295. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-12736-1_15
13. Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, Carstenand Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S., Thiemann, R.: Analyzing program termination and complexity auto-

matically with aprove. Journal of Automated Reasoning **58**(1), 3–31 (2017). https://doi.org/10.1007/s10817-016-9388-y

14. Hainry, E., Jeandel, E., Péchoux, R., Zeyen, O.: Complexityparser: An automatic tool for certifying poly-time complexity of Java programs. In: Cerone, A., Ölveczky, P.C. (eds.) Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12819, pp. 357–365. Springer (2021). https://doi.org/10.1007/978-3-030-85315-0_20
15. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification - 24th International Conference, CAV 2012. LNCS, vol. 7358, pp. 781–786. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_64
16. Jones, N.D., Kristiansen, L.: A flow calculus of *mwp*-bounds for complexity analysis. ACM Transactions on Computational Logic **10**(4), 28:1–28:41 (2009). https://doi.org/10.1145/1555746.1555752
17. Moser, G., Schneckenreither, M.: Automated amortised resource analysis for term rewrite systems. In: Functional and Logic Programming (FLOPS 2018). pp. 214–229. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-90686-7_14
18. Moyen, J.: Implicit Complexity in Theory and Practice. Habilitation thesis, University of Copenhagen (2017), https://lipn.univ-paris13.fr/~moyen/papiers/Habilitation_JY_Moyen.pdf
19. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk detection. In: D'Souza, D., Kumar, K.N. (eds.) Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017, Pune, India, October 3-6, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10482. Springer (2017). https://doi.org/10.1007/978-3-319-68167-2_7
20. Moyen, J., Rubiano, T., Seiller, T.: Loop quasi-invariant chunk motion by peeling with statement composition. In: Bonfante, G., Moser, G. (eds.) Proceedings 8th Workshop on Developments in Implicit Computational Complexity and 5th Workshop on Foundational and Practical Aspects of Resource Analysis, DICE-FOPARA@ETAPS 2017, Uppsala, Sweden, April 22-23, 2017. Electronic Proceedings in Theoretical Computer Science, vol. 248, pp. 47–59 (2017). https://doi.org/10.4204/EPTCS.248.9, http://arxiv.org/abs/1704.05169
21. pycparser - Complete C99 parser in pure Python, https://github.com/eliben/pycparser
22. pymwp at Python Package Index (2023), https://pypi.org/project/pymwp/
23. pymwp documentation (2023), https://statycc.github.io/pymwp/
24. pymwp source code repository (2023), https://github.com/statycc/pymwp
25. Sinn, M., Zuleger, F., Veith, H.: Complexity and resource bound analysis of imperative programs using difference constraints. Journal of automated reasoning **59**, 3–45 (2017). https://doi.org/10.1007/s10817-016-9402-4