pymwp: A Static Analyzer Determining Polynomial Growth Bounds

Tool User Guide

Clément Aubert Thomas Rubiano Neea Rusch Thomas Seiller

May 10, 2023

Contents

1 Introduction **2**
- 1.1 Property of Interest . . . 2
- 1.2 What mwp-Flow Analysis Computes . . . 2
- 1.3 Interpreting mwp-Bounds . . . 3

2 Installation **4**

3 Examples **5**
- 3.1 Binary Assignment . . . 6
- 3.2 Exponential Program . . . 7
- 3.3 While Analysis . . . 8
- 3.4 Infinite Program . . . 9
- 3.5 Challenge Example . . . 10

4 Learn More **11**

1 Introduction

pymwp ("pai m-w-p") is a tool for automatically performing static analysis on programs written in a subset of the C language. It analyzes resource usage and determines if program variables' growth rates are no more than polynomially related to their inputs sizes.

The theoretical foundations are described in paper the "*mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity*" (cf. "Learn More" for additional references and links). The technique is generic and applicable to any imperative language. pymwp is an implementation demonstrating this technique concretely on C programs. The technique is originally inspired by "*A Flow Calculus of mwp-Bounds for Complexity Analysis*".

This guide explains pymwp usage and behavior through several examples.

1.1 Property of Interest

For an imperative program, the goal is to discover a polynomially bounded data-flow relation, between the *initial values* of the variables (denoted `X1,…,Xn`) and its *final values* (denoted `X1',…,Xn'`).

For a program written in C language, this property can be presented as follows.

```c
void main(int X1, int X2, int X3){
   // initial values  ↑

   /*
    * various commands involving
    * variables X1, X2, X3
    */

   // X1', X2', X3' (final values)
}
```

Question: $\forall n$, is $\mathtt{X}_n \rightsquigarrow \mathtt{X}'_n$ polynomially bounded in inputs?

We answer this question using the mwp-flow analysis, implemented in the pymwp static analyzer.

1.2 What mwp-Flow Analysis Computes

The mwp-flow analysis works to establish a polynomial growth bound for input variables by applying inference rules to program's commands.

Internally, the analysis tracks *coefficients* representing *dependencies* between program's variables. These coefficients (or "flows") are $0, m, w, p$ and ∞. They characterize how data flows between variables.

- 0 — no dependency
- m — maximal (of linear)
- w — weak polynomial
- p — polynomial
- ∞ — infinite

with ordering: $0 < m < w < p < \infty$. The analysis name also comes from these coefficients.

After analysis, two outcomes are possible. (A) The program's variables values can be bounded by a polynomial in the input's values, or (B) the analysis determines it is impossible to establish such a bound. Due to non-determinism, many derivation paths need to be explored to determine this result.

The analysis succeeds if – for some derivation – no pair of variables is characterized by ∞-flow. That is, obtaining an ∞-free derivation implies existence of a polynomial growth bound; i.e., the program has the property of interest, or we can say that the program is *derivable*. The soundness theorem of the mwp-calculus guarantees that if such derivation exists, the program variables' value growth is polynomially bounded in inputs.

Program fails the analysis if every derivation contains an ∞ coefficient. Then it is not possible to establish polynomial growth bound. For these programs, pymwp reports an ∞-result.

1.3 Interpreting mwp-Bounds

If the analysis is successful, i.e., polynomial growth bound exists, it is represented using *mwp-bounds*.

An mwp-bound is a number theoretic expression of form: $\max(\vec{x}, poly_1(\vec{y})) + poly_2(\vec{z})$.

Disjoint variable lists $\vec{x}$, $\vec{y}$ and $\vec{z}$ capture dependencies of an input variable. Dependencies characterized by m-flow are in $\vec{x}$, w-flow in $\vec{y}$, and p-flow in $\vec{z}$. The polynomials $poly_1$ and $poly_2$ are built up from constants, variables, and operators $+$ and $\times$. Each variable list may be empty and $poly_1$ and $poly_2$ may not be present.

For multiple input variables, the result is a conjunction of mwp-bounds, one for each input variable.

Example 1. Assume program has one input variable named `X`, and we have obtained a bound: $\mathtt{X}' \leq \mathtt{X}$. The bound expression means the final value $\mathtt{X}'$ depends only on its own initial value `X`.

Example 2. Assume program has two inputs, `X` and `Y`, and we obtained a bound: $\mathtt{X}' \leq \mathtt{X} \wedge \mathtt{Y}' \leq \max(\mathtt{X}, 0) + \mathtt{Y}$.

- Final value $\mathtt{X}'$ depends on its own initial value `X`.
- Final value $\mathtt{Y}'$ depends on initial values of inputs `X` and `Y`.
- The bound expression can be simplified to $\mathtt{X}' \leq \mathtt{X} \wedge \mathtt{Y}' \leq \mathtt{X} + \mathtt{Y}$.

2 Installation

This section explains how to get started using pymwp. We recommend installing pymwp locally for an interactive tutorial experience.

Check environment requirements

pymwp requires Python runtime 3.7 – 3.11 (current latest).

To check currently installed version, run:

```
python3 --version
```

On systems that default to Python 3 runtime, use `python` instead of `python3`.

Instructions for installing Python for different operating systems can be found at python.org ↗.

Install pymwp

Install pymwp from Python Package Index:

```
pip3 install pymwp==0.4.2
```

Download examples

Download a set of examples.

```
wget https://github.com/statycc/pymwp/releases/download/0.4.2/examples.zip
```

Alternatively, download the set of examples using a web browser:

https://github.com/statycc/pymwp/releases/download/0.4.2/examples.zip

Unzip the examples

Unzip `examples.zip` using your preferred approach.

This completes the setup.

3 Examples

We now demonstrate use of pymwp on several examples. To ease the presentation, we will use multiple command line arguments.

- `--fin` — always run analysis to completion (even on failure)
- `--info` — reduces amount of terminal output to info level
- `--no_time` — omits timestamps from output log

For a complete list of available command arguments, run:

```
pymwp --help
```

3.1 Binary Assignment

This example shows that assigning a compounded expression to a variable results in correct analysis.

Analyzed Program: assign_expression.c

```
int foo(int y1, int y2){
    y2 = y1 + y1;
}
```

It is straightforward to observe that this program has a polynomial growth bound. The precise value of that bound is $y1' = y1 \land y2' \leq 2 * y1$. Although the program is simple, it is interesting because binary operations introduce complexity in program analysis.

CLI Command

The current working directory should be the location of unzipped examples from Installation Step 4.

```
pymwp basics/assign_expression.c --fin --info --no_time
```

Output:

```
INFO (result): Bound: y1' ≤ y1 ∧ y2' ≤ y1
INFO (result): Bounds: 3
INFO (result): Total time: 0.0 s (0 ms)
INFO (file_io): saved result in output/assign_expression.json
```

Discussion

The analysis correctly assigns a polynomial bound to the program. The bound obtained by the analyzer is $y1' \leq y1 \land y2' \leq y1$. Comparing to the precise value determined earlier, this bound is correct, because we omit constants in the analysis results.

Due to non-determinism, the analyzer finds three different derivations that yield a bound. From the `.json` file, that captures the analysis result in more technical detail, it is possible to determine these three bounds are:

- $y1' \leq y1 \land y2' \leq \max(0,0) + y1$
- $y1' \leq y1 \land y2' \leq \max(0,0) + y1$
- $y1' \leq y1 \land y2' \leq \max(0,y1) + 0$

They all simplify to $y1' \leq y1 \land y2' \leq y1$. This concludes the obtained result matches the expected result.

3.2 Exponential Program

A program computing the exponentiation returns an infinite coefficient, no matter the derivation strategy chosen.

Analyzed Program: exponent_2.c

```
int foo(int base, int exp, int i, int result){
    while (i < exp){
        result = result * base;
        i = i + 1;
    }
}
```

This program's variable `result` grows exponentially. It is impossible to find a polynomial growth bound, and the analysis is expected to report ∞-result. This example demonstrates how pymwp arrives to that conclusion.

CLI Command

```
pymwp infinite/exponent_2.c --fin --info --no_time
```

Output:

```
INFO (result): foo is infinite
INFO (result): Possibly problematic flows:
INFO (result): base → result || exp → result || i → result || result → result
INFO (result): Total time: 0.0 s (2 ms)
INFO (file_io): saved result in output/exponent_2.json
```

Discussion

The output shows that the analyzer correctly detects that no bound can be established, and we obtain ∞-result. The output also gives a list of problematic flows. This list indicates all variable pairs, that along some derivation paths, cause ∞ coefficients to occur. The arrow direction means data flows from `source` → `target`. We can see the problem with this program is the data flowing to `result` variable. This clearly indicates the origin of the problem, and allows programmer to determine if the issue can be repaired, to improve program's complexity properties.

3.3 While Analysis

A program that shows infinite coefficients for some derivations.

Analyzed Program: notinfinite_3.c

```
int foo(int X0, int X1, int X2, int X3){
    if (X1 == 1){
        X1 = X2+X1;
        X2 = X3+X2;
    }
    while(X0<10){
        X0 = X1+X2;
    }
}
```

This program contains decision logic, iteration, and multiple variables. Determining if a polynomial growth bound exists is not immediate by inspection. It is therefore an interesting candidate for analysis with pymwp!

CLI Command

```
pymwp not_infinite/notinfinite_3.c --fin --info --no_time
```

Output:

```
INFO (result): Bound: X0' ≤ max(X0,X1)+X2*X3 ∧ X1' ≤ X1+X2 ∧ X2' ≤ X2+X3 ∧ X3' ≤ X3
INFO (result): Bounds: 9
INFO (result): Total time: 0.0 s (3 ms)
INFO (file_io): saved result in output/notinfinite_3.json
```

Discussion

Compared to previous examples, the analysis is now getting more complicated. We can observe this in the number of discovered bounds and the form of the bound expression. The number of times the loop iterates, or which branch of the `if` statement is taken, is not a barrier to determining the result.

From the bound expression, we can determine the following.

- `X0` has the most complicated dependency relation. Its mwp-bound combines the impact of the `if` statement, the `while` loop, and the chance that the loop may not execute.
- `X1` and `X2` have fairly simple growth dependencies, originating from the `if` statement.
- `X3` is the simplest case – it never changes. Therefore, it only depends on itself.

Overall, the analysis concludes the program has a polynomial growth bound.

3.4 Infinite Program

A program that shows infinite coefficients for all choices.

Analyzed Program: infinite_3.c

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

If you studied the previous example carefully, you might notice that this example is *very similar*. There is a subtle differences: variable `X0` has been removed and its usages changed to `X1`. This example demonstrates how this seemingly small change impacts the analysis result.

CLI Command

```
pymwp infinite/infinite_3.c --fin --info --no_time
```

Output:

```
INFO (result): foo is infinite
INFO (result): Possibly problematic flows:
INFO (result): X1 → X1 ‖ X2 → X1 ‖ X3 → X1
INFO (result): Total time: 0.0 s (2 ms)
INFO (file_io): saved result in output/infinite_3.json
```

Discussion

We can observe the result is ∞. Thus, even a small change can change the analysis result entirely.

The output reveals the problem arises from how data flows from source variables `X1`, `X2`, and `X3`, to target variable `X1`. Observe that even though there is no direct assignment from `X3` to `X1`, the analysis correctly identifies this dependency relation, that occurs through `X2`.

From the output, we have identified the point and source of failure. Conversely, we know other variable pairs are not problematic. By focusing on how to avoid "too strong" dependencies targeting variable `X1`, programmer may be able to refactor and improve the program's complexity properties.

3.5 Challenge Example

Try to guess the analysis outcome before determining the result with pymwp.

Analyzed Program

```
int foo(int X0, int X1, int X2){
    if (X0) {
      X2 = X0 + X1;
    }
    else{
      X2 = X2 + X1;
    }
    X0 = X2 + X1;
    X1 = X0 + X2;
    while(X2){
      X2 = X1 + X0;
    }
}
```

After seeing the various preceding examples – with and without polynomial bounds – we present the following challenge. By inspection, try to determine if this program is polynomially bounded w.r.t. its input values.

It is unknown which `if` branch will be taken, and whether the `while` loop will terminate, but this is not a problem for determining the result.

CLI Command

```
pymwp other/dense_loop.c --fin --no_time --info
```

Output:

```
INFO (result): Bound: X0' ≤ max(X0,X2)+X1 ∧ X1' ≤ X0*X1*X2 ∧ X2' ≤ max(X0,X2)+X1
INFO (result): Bounds: 81
INFO (result): Total time: 0.0 s (29 ms)
INFO (file_io): saved result in output/dense_loop.json
```

Discussion

Even with just 3 variables we can see—in the obtained bound expression and the number of bounds—that this is a complicated derivation problem. The analyzer determines the program has a polynomial growth bound. Let us reason informally and intuitively why this obtained result is correct.

We can observe in the bound expression, that all three variables have complicated dependencies on one another; this corresponds to what is also observable in the input program.

Regarding variables `X0` and `X1`, observe there is no command, with either as a target variable, that would give rise to exponential value growth (need iteration). Therefore, they must have polynomial growth bounds.

Variable `X2` is more complicated. The program has a `while` loop performing assignments to `X2` (potentially problematic), and the `while` loop may or may not execute.

- Case 1: loop condition is initially false. Then, final value of `X2` depends on the `if` statement, and in either branch, it will have polynomially bounded growth.

- Case 2: loop condition true – at least one iteration will occur. The program iteratively assigns values to `X2` inside the `while` loop. However, notice the command is loop invariant. No matter how many times the loop iterates, the final value of `X2` is `X1` + `X0`. We already know those two variables have polynomial growth bounds. Therefore, `X2` also grows polynomially w.r.t. its input values.

This reasoning concurs with the result determined by pymwp.

---

4 Learn More

This guide has offered only a *very short* introduction to mwp-analysis and pymwp.
If you wish to explore more, have a look at:

- "*mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity*"[1]
  Describes theoretical foundations behind pymwp.
- "*A Flow Calculus of mwp-Bounds for Complexity Analysis*"[2]
  Original mwp-flow analysis technique.
- Documentation[3] – Includes information about supported C language features, etc.
- Online demo[4] – Run pymwp online on more than 40 examples.
- Source code[5] – pymwp is open source.
- License[6] – pymwp is licensed under GPLv3.

[1] https://doi.org/10.4230/LIPIcs.FSCD.2022.26
[2] https://doi.org/10.1145/1555746.1555752
[3] https://statycc.github.io/pymwp
[4] https://statycc.github.io/pymwp/demo/
[5] https://github.com/statycc/pymwp
[6] https://github.com/statycc/pymwp/blob/main/LICENSE