Certifying Implicit Computational Complexity

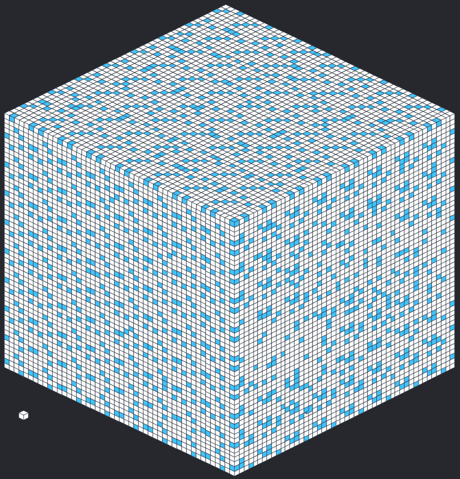
Neea Rusch
School of Computer and Cyber Sciences
Augusta University

PhD Research Proposal Presentation
16 December 2022

Resource bounds impact program correctness.

Excessive resource usage makes programs fail.

100^3 (1 million)



200000^3 (8 quadrillion)

Wish List

- Automatic analysis of complexity properties
- Ability to optimize resource usage
- Strong guarantees of correctness

Hypothesis

Implicit Computational Complexity (ICC) provides new approaches to automatic complexity analysis and can resolve certain limitations.

Presentation Outline

- ☐ Background

  Research directions:

- ☐ 1. mwp-Analysis Improvement and Implementation
- ☐ 2. Distributing and Parallelizing Non-canonical Loops
- ☐ 3. Formally Verified Complexity

Implicit Computational Complexity (ICC)

Let L be a programming language, C a complexity class, and $[\![p]\!]$ the function computed by program p.

Find a restriction $R \subseteq L$, such that the following equality holds:

$$\{[\![p]\!] \mid p \in R\} = C$$

The variables L, C, and R are the parameters that vary greatly between different ICC systems[1].

[1] Romain Péchoux. *Complexité implicite : bilan et perspectives*. Habilitation à Diriger des Recherches (HDR). 2020. URL: `https://hal.univ-lorraine.fr/tel-02978986`.

Analyzing Variable Value Growth

For a deterministic imperative program,
is the growth of input variable values polynomially bounded?

Example

```
C′ ≡ X1 := X2 + X3;
     X1 := X1 + X1
```

$[\![C']\!](x_1, x_2, x_3 \rightsquigarrow x'_1, x'_2, x'_3)$
implies $x'_1 \leq 2x_2 + 2x_3$
and $x'_2 \leq x_2$ and $x'_3 \leq x_3$.

```
C″ ≡ X1 := 1;
     loop X2 { X1 := X1 + X1 }
```

$[\![C'']\!](x_1, x_2 \rightsquigarrow x'_1, x'_2)$
implies $x'_1 \leq 2^{x_2}$ and $x'_2 \leq x_2$.

mwp-Flow Analysis[2]

- Tracks how each variable depends on other variables.
- Flows characterize dependencies:

  0 – no dependency
  m – maximal
  w – weak polynomial
  p – polynomial

  weaker ↕ *stronger*

- Apply inference rules to program statements.
- Collect analysis result in a matrix.

[2] Neil D. Jones and Lars Kristiansen. "A flow calculus of *mwp*-bounds for complexity analysis". In: *ACM Trans. Comput. Log.* 10.4 (Aug. 2009), 28:1–28:41. DOI: 10.1145/1555746.1555752.

mwp-Analysis Example

```
void main(int X1, int X2, int X3){
    if (X1 < X2) {
        X3 = X1 + X1;
    }
    else {
        X3 = X3 + X2;
    }
    while (X1 < 0){
        X1 = X2 + X3;
    }
}
```

	X1	X2	X3
X1	m	0	0
X2	0	m	0
X3	0	0	m

mwp-Analysis Example

```
void main(int X1, int X2, int X3){
    if (X1 < X2) {
        X3 = X1 + X1;
    }
    else {
        X3 = X3 + X2;
    }
    while (X1 < 0){
        X1 = X2 + X3;
    }
}
```

	X1	X2	X3
X1	m	0	$\boxed{p}$
X2	0	m	0
X3	0	0	m

mwp-Analysis Example

```
void main(int X1, int X2, int X3){
    if (X1 < X2) {
        X3 = X1 + X1;
    }
    else {
        X3 = X3 + X2;
    }
    while (X1 < 0){
        X1 = X2 + X3;
    }
}
```

	X1	X2	X3
X1	m	0	0
X2	0	m	p
X3	0	0	m

mwp-Analysis Example

```
void main(int X1, int X2, int X3){
    if (X1 < X2) {
        X3 = X1 + X1;
    }
    else {
        X3 = X3 + X2;
    }
    while (X1 < 0){
        X1 = X2 + X3;
    }
}
```

	X1	X2	X3
X1	m	0	p
X2	0	m	p
X3	0	0	m

mwp-Analysis Example

```
void main(int X1, int X2, int X3){
    if (X1 < X2) {
        X3 = X1 + X1;
    }
    else {
        X3 = X3 + X2;
    }
    while (X1 < 0){
        X1 = X2 + X3;
    }
}
```

	X1	X2	X3
X1	m	0	0
X2	$\boxed{w}$	m	0
X3	$\boxed{w}$	0	m

mwp-Analysis Example

```
void main(int X1, int X2, int X3){
    if (X1 < X2) {
        X3 = X1 + X1;
    }
    else {
        X3 = X3 + X2;
    }
    while (X1 < 0){
        X1 = X2 + X3;
    }
}
```

$$\begin{array}{c|ccc} & \mathtt{X1} & \mathtt{X2} & \mathtt{X3} \\ \hline \mathtt{X1} & m & 0 & 0 \\ \mathtt{X2} & w & m & 0 \\ \mathtt{X3} & w & 0 & m \end{array}$$

$= M^*$

Side condition: $\forall i, M^*_{ii} = m$ and $\forall i, j, M^*_{ij} \neq p$

mwp-Analysis Example

```
void main(int X1, int X2, int X3){
    if (X1 < X2) {
        X3 = X1 + X1;
    }
    else {
        X3 = X3 + X2;
    }
    while (X1 < 0){
        X1 = X2 + X3;
    }
}
```

$$
\begin{array}{c|ccc}
 & \texttt{X1} & \texttt{X2} & \texttt{X3} \\
\hline
\texttt{X1} & p & 0 & p \\
\texttt{X2} & p & m & p \\
\texttt{X3} & w & 0 & m
\end{array}
$$

$= \texttt{C;C}$

mwp-Analysis Example - Final Result

```
void main(int X1, int X2, int X3){
    if (X1 < X2) {
        X3 = X1 + X1;
    }
    else {
        X3 = X3 + X2;
    }
    while (X1 < 0){
        X1 = X2 + X3;
    }
}
```

	X1	X2	X3
X1	p	0	p
X2	p	m	p
X3	w	0	m

mwp-Analysis Soundness

For program $\texttt{C}$ and mwp-matrix M,

- $\vdash \texttt{C} : M$ means calculus *assigns* matrix M to command $\texttt{C}$.
- $\texttt{C}$ is *derivable* if the calculus assigns at least one matrix to it.
- Relation $\vdash \texttt{C} : M$ holds iff there exists a derivation in the calculus.

Theorem (Soundness[3])

$\vdash \texttt{C} : M$ *implies* $\vDash \texttt{C} : M$.

[3] Jones and Kristiansen, "A flow calculus of *mwp*-bounds for complexity analysis", p. 11.

Presentation Outline

✓ Background

Research directions:

☐ 1. mwp-Analysis Improvement and Implementation

☐ 2. Distributing and Parallelizing Non-canonical Loops

☐ 3. Formally Verified Complexity

mwp-Analysis Improvement and Implementation

The mwp-analysis has many useful properties:

- Compositional method
- Termination, loop condition are abstracted
- Language agnostic syntax
- Multivariate result, etc.

But is it *really* automatable?

Several Open Questions

- Powerfulness – what is the size of the class programs that can be analyzed?
- Richness – can it be extended to analyze more commands?
- Utility – what else can be done with this analysis?

Several Practical Limitations

- How to handle analysis failure?
- How to manage nondeterminism of rules?
- How to efficiently determine if program `C` is derivable?

mwp-Analysis Improvement and Implementation – Approach

- Adjusted the mathematical framework to have deterministic rules.
- Extended the supported syntax with function calls, incl. recursion.
- Created a static analyzer implementation[4] and measured its performance.
- Split computation into two phases: existence of bound vs. calculating it.
- Developed an efficient evaluation strategy.

[4] Clément Aubert et al. *pymwp: A Tool for Guaranteeing Complexity Bounds for C Programs*. Version 1.0. Oct. 2022. DOI: 10.5281/zenodo.7159134. URL: https://github.com/statycc/pymwp.

mwp-Analysis Improvement and Implementation – Example

```
int foo(int x, int y){
 while(0){x=y+y;}
}
```

$$
\begin{array}{c c}
 & \begin{array}{cc} \mathrm{x} & \mathrm{y} \end{array} \\
\begin{array}{c} \mathrm{x} \\ \mathrm{y} \end{array} &
\begin{bmatrix}
m + \infty\delta(0,0) + \infty\delta(1,0) & 0 \\
\infty\delta(0,0) + \infty\delta(1,0) + w\delta(2,0) & m
\end{bmatrix}
\end{array}
$$

https://statycc.github.io/pymwp/demo/#basics_while_2.c

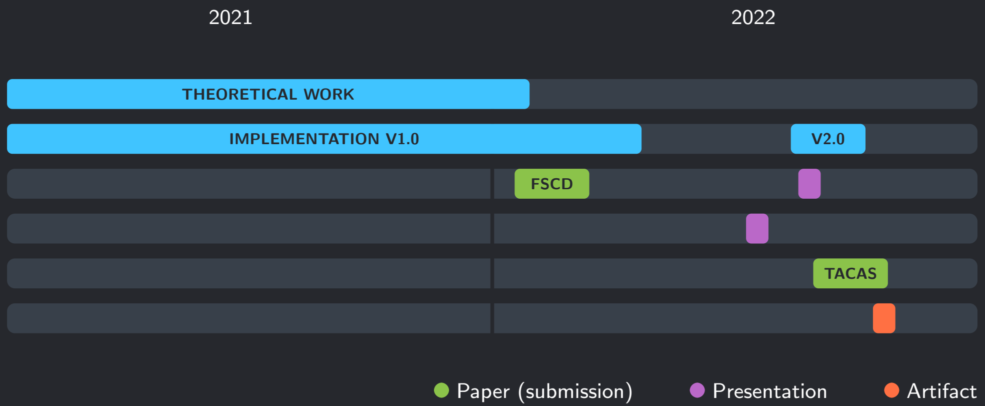
Publications

Aubert, Clément, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity". In: *FSCD 2022*. Vol. 228. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 26:1–26:23. DOI: `10.4230/LIPIcs.FSCD.2022.26`.

— . "pymwp: A Tool for Guaranteeing Complexity Bounds for C Programs". Submitted to 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2023.

Aubert, Clément, Thomas Rubiano, Neea Rusch, and Thomas Seiller. *pymwp: A Tool for Guaranteeing Complexity Bounds for C Programs*. Version 1.0. Oct. 2022. DOI: `10.5281/zenodo.7159134`. URL: `https://github.com/statycc/pymwp`.

Timeline: mwp-Analysis Improvement and Implementation



Presentation Outline

✓ Background

Research directions:

✓ 1. mwp-Analysis Improvement and Implementation

☐ 2. Distributing and Parallelizing Non-canonical Loops

☐ 3. Formally Verified Complexity

New and legacy code needs to be transformed to utilize multicore architectures.

$$\Downarrow$$

Idea: find an automatic way to reduce program execution time by parallelization.

Distributing and Parallelizing Non-canonical Loops

We present a program transformation technique to distribute loops.
Enables discovery of parallelization potential in previously uncovered cases.

High-level idea

```
while(t[i] != j){
  s1[i] = j*j;
  s2[i] = 1/j;
  i++;
}
```

$\Rightarrow$

```
# parallel
while(t[i1] != j){
  s1[i1] = j*j; i1++;
}

# parallel
while(t[i2] != j){
  s2[i2] = 1/j; i2++;
}
```

Loop Fission Algorithm

Our algorithm performs loop fission transformation.

- Uses ICC-inspired data-flow analysis to analyze dependencies.
- Establishes cliques between statements.
- Split independent cliques into multiple loops.

Loop Fission Algorithm Features

- Applicable even if iteration space is unknown.
- Loop-agnostic: `for`, `while`, `do...while`; complex conditions, etc.
- Can be mapped to any imperative language (high level ...IR)

Other Contributions

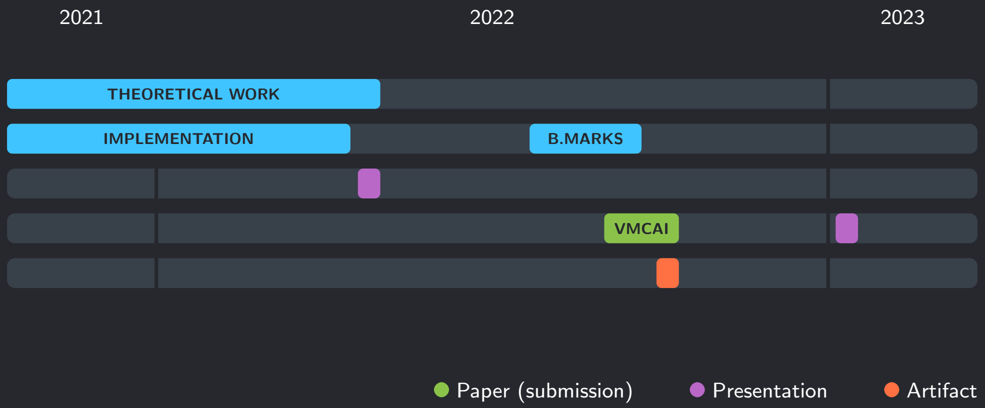
- Transformation correctness proof.
- Experimental evaluation that measures expected gain.

Publications

Aubert, Clément, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "Distributing and Parallelizing Non-canonical Loops". To appear in Verification, Model Checking, and Abstract Interpretation (VMCAI). 2023.

Aubert, Clément, Thomas Rubiano, Neea Rusch, and Thomas Seiller. *Distributing and Parallelizing Non-canonical Loops – Artifact*. Version 1.0. Sept. 2022. DOI: 10.5281/zenodo.7080145. URL: https://github.com/statycc/loop-fission.

Timeline: Distributing and Parallelizing Non-canonical Loops



Presentation Outline

- ✓ Background

  Research directions:

- ✓ 1. mwp-Analysis Improvement and Implementation
- ✓ 2. Distributing and Parallelizing Non-canonical Loops
- ☐ 3. Formally Verified Complexity

Recall mwp-analysis soundness theorem:

$$\vdash \texttt{C} : M \text{ implies } \vDash \texttt{C} : M.$$

Proving Programs

- Prove that some property holds with the strongest possible guarantee.
- Done using an interactive theorem prover.
- Construct rigorous logical arguments.
- Machine-checkable for correctness.

Trade-off

Mechanical proofs require specifying every detail (slow, tedious).

$\Updownarrow$

Get the strongest possible guarantee of correctness.

My Goal

Prove the mwp-analysis technique.

- As defined in the original paper.
- Using the Coq proof assistant.

Steps - 1 of 4

Define the programming language under analysis.

- Simple, memory-less imperative language.
- Syntax: variables, arithmetic and boolean exp., commands.

Steps - 2 of 4

Define the mathematical machinery.

- Need e.g., matrices, semi-ring.
- Other related mathematical concepts e.g., honest polynomial.

Steps - 3 of 4

Implementing the typing system.

- Define the flow calculus rules.
- Define a typing system.

Steps - 4 of 4

Prove the paper lemmas and theorems.

- There are 8 lemmas and 7 theorems.
- The soundness theorem, $\vdash \texttt{C} : M$ implies $\vDash \texttt{C} : M$, is essential.
- "These proofs are long, technical and occasionally highly nontrivial."[5]

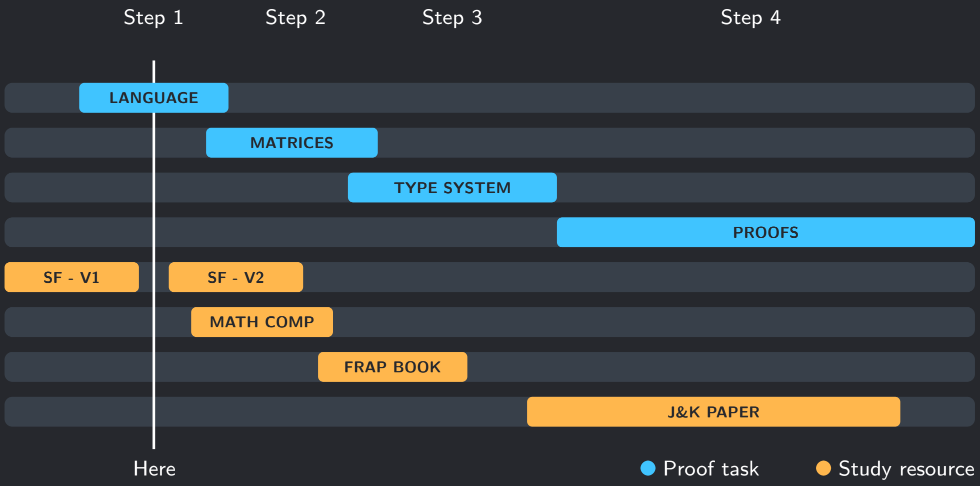
[5] Jones and Kristiansen, "A flow calculus of *mwp*-bounds for complexity analysis", p. 2.

Expected Main Result

A *certified* complexity analysis technique.

- Proves a positive result obtained by analysis is correct.
- Establishes certified "growth bound" on input variable values.

Timeline and Progress



Possible Future Directions

Many directions can follow from the correctness proof
e.g., a formally verified static analyzer.

- Adjusting analysis makes it practical and fast.
- Proof would show the original technique is correct, but not fast.
- It should be possible to combine those two results.

Preliminary Papers

Aubert, Clément, Thomas Rubiano, Neea Rusch, and Thomas Seiller. "Certifying Complexity Analysis". At the Ninth International Workshop on Coq for Programming Languages (CoqPL). 2023.

Rusch, Neea. "Formally Verified Resource Bounds Through Implicit Computational Complexity". In: *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2022. Association for Computing Machinery, 2022. DOI: 10.1145/3563768.3565545.

Presentation Outline

- ✓ Background

Research directions:

- ✓ 1. mwp-Analysis Improvement and Implementation
- ✓ 2. Distributing and Parallelizing Non-canonical Loops
- ✓ 3. Formally Verified Complexity

Summary

Implicit Computational Complexity (ICC) provides new approaches to automatic complexity analysis and can resolve certain limitations.

- ✓ mwp-Analysis Improvement and Implementation
- ✓ Distributing and Parallelizing Non-canonical Loops
- $\rightarrow$ Formally Verified Complexity (current)

Many other directions can follow e.g., certified static analyzer for complexity.

mwp-Flow Analysis: Program Syntax

Variable	$\mathtt{X}_1 \mid \mathtt{X}_2 \mid \mathtt{X}_3 \mid \ldots$
Expression	$\mathtt{X} \mid \mathtt{e + e} \mid \mathtt{e * e}$
Boolean Exp.	`e = e`, `e < e`, etc.
Commands	$\mathtt{skip} \mid \mathtt{X := e} \mid \mathtt{C;C} \mid \mathtt{loop\ X}\ \{\mathtt{C}\} \mid$ $\mathtt{if\ b\ then\ C\ else\ C} \mid \mathtt{while\ b\ do}\ \{\mathtt{C}\}$

mwp-Flow Analysis: Inference Rules

$$\frac{}{\vdash \mathsf{Xi} : \{ {}^{m}_{\mathsf{i}} \}}\ \mathsf{E1}$$

$$\frac{\vdash \mathsf{C1} : M_1 \quad \vdash \mathsf{C2} : M_2}{\vdash \textbf{if}\ \ \mathsf{b}\ \textbf{then}\ \mathsf{C1}\ \textbf{else}\ \ \mathsf{C2} : M_1 \oplus M_2}\ \mathsf{I}$$

$$\frac{}{\vdash \mathsf{e} : \{ {}^{w}_{\mathsf{i}} \mid \mathsf{Xi} \in \mathrm{var}(\mathsf{e}) \}}\ \mathsf{E2}$$

$$\frac{\vdash \mathsf{Xi} : V_1 \quad \vdash \mathsf{Xj} : V_2}{\vdash \mathsf{Xi}{\star}\mathsf{Xj} : pV_1 \oplus V_2}\ \mathsf{E3}$$

$$\frac{\vdash \mathsf{Xi} : V_1 \quad \vdash \mathsf{Xj} : V_2}{\vdash \mathsf{Xi}{\star}\mathsf{Xj} : V_1 \oplus pV_2}\ \mathsf{E4}$$

$$\frac{\vdash \mathsf{e} : V}{\vdash \mathsf{Xj} = \mathsf{e} : 1 \xleftarrow{\mathsf{j}} V}\ \mathsf{A}$$

$$\forall i, M^*_{ii} = m \quad \frac{\vdash \mathsf{C} : M}{\vdash \textbf{loop}\ \mathsf{Xl}\ \{\mathsf{C}\} : M^* \oplus \{ {}^{p}_{\mathsf{l}} \to j \mid \exists i, M^*_{ij} = p \}}\ \mathsf{L}$$

$$\frac{\vdash \mathsf{C1} : M_1 \quad \vdash \mathsf{C2} : M_2}{\vdash \mathsf{C1};\ \mathsf{C2} : M_1 \otimes M_2}\ \mathsf{C}$$

$$\forall i, M^*_{ii} = m \text{ and } \forall i, j, M^*_{ij} \neq p \quad \frac{\vdash \mathsf{C} : M}{\vdash \textbf{while}\ \ \mathsf{b}\ \textbf{do}\ \{\mathsf{C}\} : M^*}\ \mathsf{W}$$

References

Jones, Neil D. and Lars Kristiansen. "A flow calculus of *mwp*-bounds for complexity analysis". In: *ACM Trans. Comput. Log.* 10.4 (Aug. 2009), 28:1–28:41. DOI: 10.1145/1555746.1555752.

Péchoux, Romain. *Complexité implicite : bilan et perspectives*. Habilitation à Diriger des Recherches (HDR). 2020. URL: https://hal.univ-lorraine.fr/tel-02978986.