# Fresh Perspectives on Implicit Computational Complexity

Neea Rusch · Augusta University

3 June 2025 @ Uppsala University

# Static program analysis is fascinating

**Functional equivalance**
Two programs are functionally equivalent if they compute the same output for every input.

However, such programs can differ in non-functional properties.

# Static program analysis is fascinating

Exercise: compare the following two programs for differences.

```
if (bit) { Z = X; }
    else { Z = Y; }
```

vs.

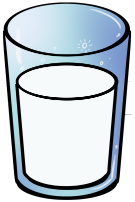
```
Z = X * bit + Y * (1 - bit);
```

Assume standard semantics, variables `X,Y,Z` $\in \mathbb{Z}$, and `bit` $\in \{0, 1\}$.

**Bad news!** There exists no general method for analyzing interesting semantic properties; i.e., every non-trivial semantic property is undecidable.[1]

**Good news!** We can always build increasingly better approximative techniques.

This is colloquially called the "full employment theorem for static program analysis designers"[2]



[1] Rice, "Classes of recursively enumerable sets and their decision problems".

[2] Møller and Schwartzbach, *Static Program Analysis*, p. 4.

# Techniques of static program analysis

There are many syntactical techniques for reasoning about programs: data-flow analysis, type systems, abstract interpretation, etc.

The “toolbox” of this talk comes from **implicit computational complexity**.

# Agenda

By the end of this talk, you should have learned three things about implicit computational complexity:

1. What is it?
2. How does it work?
3. What is it good for?

# Classic Complexity Theory

Characterizes complexity classes in terms of machine models.

Programs are classified into classes based on resource usage.

Resources of interest are typically time, space, etc.

# Very Brief History

| Year | Description |
|---|---|
| 1966 | Cobham–Edmonds's Thesis: relates P time with feasible functions[3][4] |
| 1992 | First implicit characterizations of complexity |
| | – Stephen Bellantoni and Stephen Cook: safe recursion[5] |
| | – Daniel Leivant: stratified recurrence[6] |

[3] Cobham, "The Intrinsic Computational Difficulty of Functions".

[4] Edmonds, "Paths, trees, and flowers".

[5] Bellantoni and Cook, "A new recursion-theoretic characterization of the polytime functions".

[6] Leivant, "Stratified functional programs and computational 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.[7]

[7] Péchoux, "Complexité implicite : bilan et perspectives".

# Implicit Computational Complexity (ICC)

| | |
|---|---|
| **Programming language** | C, Java, $\lambda$-calculus… |
| + **restriction** | type system, syntax structure, data flow… |
| $\Rightarrow$ **complexity class** | PTIME, EXP, L, PP… |

# Advantages of Implicit Computational Complexity

- Natural characterizations of central complexity results.
- Better understanding of complexity classes.
  For example: complexity classes are intrinsic mathematical entities that do not depend on a particular machine model.
- Quantifies the computational power available in programming languages by construction.
- Potential to convert complexity-theoretic insights to practical program analyses.

# Restricting languages is a bit controversial

## What are the practical limitations of a non-turing complete language?

Asked 14 years ago Modified 4 months ago Viewed 12k times

74

As there are non-Turing complete languages out there, and given I didn't study Comp Sci at university, could someone explain something that a Turing-incomplete language (like Coq) cannot do?

## Practical non-Turing-complete languages?

Asked 15 years, 8 months ago Modified 10 years, 2 months ago Viewed 22k times

55

Nearly all programming languages used are Turing Complete, and while this affords the language to represent any computable algorithm, it also comes with its own set of problems. Seeing as all the algorithms I write are intended to halt, I would like to be able to represent them in a language that guarantees they will halt.

# Programming languages with ~~restrictions~~ guarantees

**(safe) Rust**
no memory errors, no data races, controlled aliasing

**Total functional programming**
programs are provably terminating

**Theorem-proving languages**
require termination, but enable constructing formal proofs

**Synchronous languages**
for real-time reactive systems with response-time and memory usage restrictions

# The challenge with Implicit Computational Complexity

Despite the many compelling features, ICC has remained largely a theoretical novelty.

The practical power, limitations, and utilities of ICC are not well-understood.

This influences the continued development of ICC theories and limits exposure of ICC ideas and techniques in broader research communities.

# Hypothesis

Implicit computational complexity offers applied utilities when lifted outside the theoretical domain.

# Questions

- Can we develop practical resource analyses based on these theories?
- Is the theory correct: can we prove formally its soundness?
- If theories can be automated, what are their use cases?
- Can the theories be used to track other semantic properties?

# Questions

- Can we develop practical resource analyses based on these theories?
- Is the theory correct: can we prove formally its soundness?
- If theories can be automated, what are their use cases?
- Can the theories be used to track other semantic properties?

# The flow calculus of mwp-bounds[8]

Data flow analysis for certifying that **final values** computed by a deterministic imperative program will be bounded by **polynomials** in the program's **inputs**.

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

The goal is to discover a polynomially bounded data-flow relation for command `C`, between its variables' initial values $x_i$ and final values $x_i'$: $[\![\texttt{C}]\!](x_i \rightsquigarrow x_i')$.

$\texttt{C}' \equiv$
```
X1 := X2 + X3;
X1 := X1 + X1
```

$x_1' \leq 2x_2 + 2x_3$

$x_2' \leq x_2$

$x_3' \leq x_3$

PASS

$\texttt{C}'' \equiv$
```
X1 := 1;
loop X2 {X1 := X1 + X1}
```

$x_1' \leq 2^{x_2}$

$x_2' \leq x_2$

FAIL

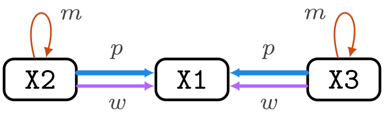
The goal is to discover a polynomially bounded data-flow relation for command `C`, between its variables' initial values $x_i$ and final values $x_i'$: $[\![\texttt{C}]\!](x_i \rightsquigarrow x_i')$.

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

$x_1' \leq 2x_2 + 2x_3$

$x_2' \leq x_2$

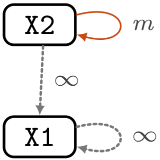
$x_3' \leq x_3$




PASS

The goal is to discover a polynomially bounded data-flow relation for command `C`, between its variables' initial values $x_i$ and final values $x_i'$: $[\![\texttt{C}]\!](x_i \rightsquigarrow x_i')$.

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

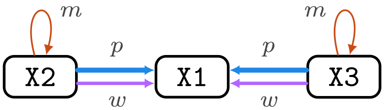


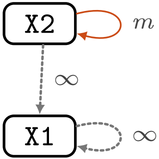

$x_1' \leq 2^{x_2}$

$x_2' \leq x_2$

FAIL

The goal is to discover a polynomially bounded data-flow relation for command `C`, between its variables' initial values $x_i$ and final values $x_i'$: $[\![\texttt{C}]\!](x_i \rightsquigarrow x_i')$.



# Mechanics of the flow calculus

## Imperative Language

(var) $\mathtt{X}_1 \mid \mathtt{X}_2 \mid \mathtt{X}_3 \mid \ldots$ (aexp) `e + e` | `e * e` (bexp) `e = e` | `e < e` | …

(com) `skip` | `X := e` | `C;C` | `if b then C else C` | `loop X {C}` | `while b do {C}`

# Mechanics of the flow calculus

## Flow coefficients (dependencies)

$0:$ no dependency $m:$ maximal $w:$ weak polynomial $p:$ polynomial

$\xrightarrow{\text{weaker...stronger}}$

# Mechanics of the flow calculus

## Inference rules

$$\frac{}{\vdash \texttt{Xi} : \{{}^{m}_{\texttt{i}}\}}\ \text{E1} \qquad \frac{\vdash \texttt{Xi} : V_1 \quad \vdash \texttt{Xj} : V_2}{\vdash \texttt{Xi}\star\ \texttt{Xj} : pV_1 \oplus V_2}\ \text{E3} \qquad \frac{\vdash \texttt{e} : V}{\vdash \texttt{Xj = e} : 1 \overset{\texttt{j}}{\leftarrow} V}\ \text{A} \qquad \ldots$$

# Mechanics of the flow calculus

**mwp-bound**

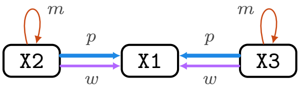
$$\max(\vec{x}, poly_1(\vec{y})) + poly_2(\vec{z})$$

# Mechanics of the flow calculus

1. Imperative programming language [input]
2. Coefficients (dependencies)
3. Inference rules
4. mwp-bounds [output]

# Derivation success

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

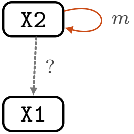



$$\dfrac{\dfrac{\dfrac{\overline{\vdash \texttt{X2} : \begin{pmatrix} 0 \\ m \\ 0 \end{pmatrix}}\ \text{E1} \qquad \overline{\vdash \texttt{X3} : \begin{pmatrix} 0 \\ 0 \\ m \end{pmatrix}}\ \text{E1}}{\vdash \texttt{X2+X3} : \begin{pmatrix} 0 \\ p \\ m \end{pmatrix}}\ \text{E3}}{\vdash \texttt{X1:=X2+X3} : \begin{pmatrix} 0 & 0 & 0 \\ p & m & 0 \\ m & 0 & m \end{pmatrix}}\ \text{A} \qquad \vdots \qquad \dfrac{}{\vdash \texttt{X1:=X1+X1} : \begin{pmatrix} p & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{pmatrix}}\ \text{A} \qquad \vdots}{\vdash \texttt{X1:=X2+X3;X1:=X1+X1} : \begin{pmatrix} 0 & 0 & 0 \\ p & m & 0 \\ p & 0 & m \end{pmatrix}}\ \text{C}$$

$$x_1' \leq x_2 + x_3 \quad \wedge \quad x_2' \leq x_2 \quad \wedge \quad x_3' \leq x_3$$

# Derivation failure

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




$$\dfrac{}{\vdash \texttt{X1:=1} : \begin{pmatrix} m \\ 0 \end{pmatrix}}\ \text{E1}$$

$$\vdots$$

$$\dfrac{}{\vdash \texttt{X1:=X1+X1} : \begin{pmatrix} p & 0 \\ 0 & m \end{pmatrix}}\ \text{A}$$

$$\vdots$$

✗

$$\forall i, M^*_{ii} = m \quad \dfrac{\vdash \texttt{C} : M}{\vdash \texttt{loop X}_\ell\texttt{\{C\}} : M^* \oplus \{\overset{p}{\ell} \rightarrow j \mid \exists i, M^*_{ij} = p\}}\ \text{L}$$

# Application challenges

All works nicely on paper.

But we have implementation problems:

- How to handle nondeterminism?
- How to handle derivation failure?

# Handling non-determinism

**Idea:** track derivation choices as functions from choices to coefficients.

- ▷ If a coefficient depends on a choice represent it as 3 elements.
- ▷ If independent, represented as a single element.

We track not only dependencies, but a history of derivation choices.

# FIX: Internalize non-determinism

```
X1 = X2 + X3
```

$$\downarrow$$

$$\cancel{\begin{pmatrix} m & 0 & 0 \\ m & m & 0 \\ p & 0 & m \end{pmatrix}} \quad \cancel{\begin{pmatrix} m & 0 & 0 \\ p & m & 0 \\ m & 0 & m \end{pmatrix}} \quad \cancel{\begin{pmatrix} m & 0 & 0 \\ w & m & 0 \\ w & 0 & m \end{pmatrix}}$$

$$\downarrow$$

$$\begin{pmatrix} m & 0 & 0 \\ m.\delta(0,0)+p.\delta(1,0)+w.\delta(2,0) & m & 0 \\ p.\delta(0,0)+m.\delta(1,0)+w.\delta(2,0) & 0 & m \end{pmatrix}$$

# The failure problem

`C ≡ while(b){X1:=X2+X2}`

Derivation of `X1:=X2+X2` yields two matrices: $\begin{pmatrix} 0 & 0 \\ p & m \end{pmatrix}$ and $\begin{pmatrix} 0 & 0 \\ w & m \end{pmatrix}$

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

$\Rightarrow$ derivation $\begin{pmatrix} 0 & 0 \\ p & m \end{pmatrix}$ fails but derivation $\begin{pmatrix} 0 & 0 \\ w & m \end{pmatrix}$ succeeds.

# FIX: New way to represent failure

**Idea:** We introduce $\infty$ flow to represent non-polynomial dependencies.

$$\{0, m, w, p, \infty\}$$

Every derivation can be completed without restarts.

Captures localized information about where failure occurs.

Once failure is introduced, it cannot be erased: $\infty \times^{\infty} 0 = \infty$.

```
C ≡ while(b){X1:=X2+X2}
```

$$\begin{pmatrix} m+\infty\delta(0,0)+\infty\delta(1,0) & 0 \\ \infty\delta(0,0)+\infty\delta(1,0)+w\delta(2,0) & m \end{pmatrix}$$

# Implementation: pymwp static analyzer[9]

A prototype analyzer for a subset of C99.

Source code and demo: statycc.github.io/pymwp/demo

Install: **pip install pymwp**

Usage

```
pymwp path/to/file.c  [ARGS]
```

[9] Aubert et al., "pymwp: A Static Analyzer Determining Polynomial Growth Bounds".

# The enhanced flow calculus

Q: Can we develop practical resource analyses based on these theories?
A: Yes, *eventually*.

- All derivations captured deterministically in one (complex) matrix.
- The mwp-bounds are “lost” (this has been resolved since).
- The enhanced system gives more information and captures a larger class of programs than the original system.
- Many open problems remain: formalization, increasing precision, analyzing lower bounds, …

# From Theory to Applications

Implicit computational complexity provides orthogonal techniques for automatic resource analysis.

Attempts to implement and apply the theories expose their limitations.

Those investigations lead to improvements in the theories.

# Implicit Complexity in Program Analysis

We can implement program analyses from two directions:

**Top-down**: reducing a rich language to a restricted subset.

**Bottom-up**: reasoning about programs before any programs exist

# Extended Utilities

The techniques developed in implicit computational complexity can be adjusted to tracking other semantic properties.

We have investigated applications in parallelizing transformations[10], and security (ongoing).

These investigations highlight the flexibility of ICC techniques.

[10] Aubert et al., "Distributing and Parallelizing Non-canonical Loops".

# Take-home Messages

Implicit computational complexity…

- Gives us a rich toolbox of techniques for resource analysis.
- Can be used *beyond* resources, to track other non-functional properties.
- Should be viewed in broader sense than the name implies.

# Nondeterministic Inference Rules

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

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

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

$$\frac{\vdash \texttt{Xi} : V_1 \quad \vdash \texttt{Xj} : V_2}{\vdash \texttt{Xi+Xj} : pV_1 \oplus V_2}\ \text{E3}$$

$$\frac{\vdash \texttt{Xi} : V_1 \quad \vdash \texttt{Xj} : V_2}{\vdash \texttt{Xi+Xj} : V_1 \oplus pV_2}\ \text{E4}$$

$$\frac{\vdash \texttt{e} : V}{\vdash \texttt{Xj = e} : 1 \overset{\texttt{j}}{\leftarrow} V}\ \text{A}$$

$$\forall i, M^*_{ii} = m \quad \frac{\vdash \texttt{C} : M}{\vdash \textbf{loop}\ \texttt{X}_\ell\ \{\texttt{C}\} : M^* \oplus \{^{p}_{\ell} \rightarrow j \mid \exists i, M^*_{ij} = p\}}\ \text{L}$$

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

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

# Deterministic Inference Rules

$$\frac{\star \in \{+,-\}}{\vdash \mathtt{Xi}\star\mathtt{Xj} : (0 \mapsto \{{}^{m}_{\mathtt{i}}, {}^{p}_{\mathtt{j}}\}) \oplus (1 \mapsto \{{}^{p}_{\mathtt{i}}, {}^{m}_{\mathtt{j}}\}) \oplus (2 \mapsto \{{}^{w}_{\mathtt{i}}, {}^{w}_{\mathtt{j}}\})}\ \mathsf{E}^{\mathsf{A}}$$

$$\frac{}{\vdash \mathtt{Xi}*\mathtt{Xj} : \{{}^{w}_{\mathtt{i}}, {}^{w}_{\mathtt{j}}\}}\ \mathsf{E}^{\mathsf{M}}$$

$$\frac{}{\vdash \mathtt{Xi} : \{{}^{m}_{\mathtt{i}}\}}\ \mathsf{E}^{\mathsf{S}}$$

$$\frac{\vdash \mathtt{e} : V}{\vdash \mathtt{Xj}\ \mathtt{=}\ \mathtt{e} : 1 \overset{\mathtt{j}}{\leftarrow} V}\ \mathsf{A}$$

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

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

$$\frac{\vdash \mathtt{C} : M}{\vdash \mathtt{loop}\ \mathtt{Xl}\ \{\mathtt{C}\} : M^* \oplus \{{}^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} \oplus \{{}^{p}_{\mathtt{l}} \to j \mid \exists i, M^*_{ij} = p\}}\ \mathsf{L}^{\infty}$$

$$\frac{\vdash \mathtt{C} : M}{\vdash \mathtt{while}\ \mathtt{b}\ \mathtt{do}\ \{\mathtt{C}\} : M^* \oplus \{{}^{\infty}_{j} \to j \mid M^*_{jj} \neq m\} \oplus \{{}^{\infty}_{i} \to j \mid M^*_{ij} = p\}}\ \mathsf{W}^{\infty}$$

$$\frac{}{\vdash \mathtt{Xi}\ \mathtt{=}\ \mathtt{F(X1,\cdots,\ Xn)} : 1 \overset{\mathtt{i}}{\leftarrow} ((M_f^1)\delta(0,c) \oplus \cdots \oplus (M_f^k)\delta(0,c)\delta(k,c))}\ \mathsf{F}$$