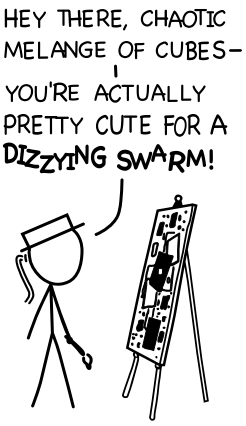
Applied Implicit Computational Complexity

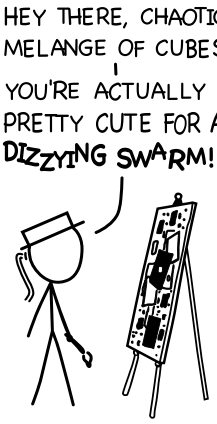
Neea Rusch · Augusta University

Doctoral Symposium @ ECOOP 2025

3 July 2025



Inspired by xkcd, *Abstract Pickup* and xkcd, *Academia vs. Business*



Inspired by xkcd, *Abstract Pickup* and xkcd, *Academia vs. Business*

Example

Compare the two programs

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

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

vs.

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

Static program analysis

Many standard techniques: data flow analysis, type systems, constraint-based analysis, abstract interpretation,…[1]

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

[1] See, e.g., Møller and Schwartzbach, *Static Program Analysis* and Nielson, Nielson, and Hankin, *Principles of program analysis*.

Agenda

By the end of this talk, you should have learned three things:

1. What is implicit computational complexity?
2. How does it work? (by example)
3. What are some of the applications?

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.[2]

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

Implicit Computational Complexity (ICC)

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

ICC by Example: Safe Recursion

Programming language: general recursive functions.

Programs: built by composition, primitive recursion, and minimization.

Addition

$$\begin{aligned} \mathsf{add}(0, x) &= x \\ \mathsf{add}(n+1, x) &= \mathsf{add}(n, x) + 1 \end{aligned}$$

Exponentiation

$$\begin{aligned} \exp(0) &= 1 \\ \exp(n+1) &= \mathsf{add}(\exp(n), \exp(n)) \end{aligned}$$

Dal Lago, "Implicit computation complexity in higher-order programming languages: A Survey in Memory of Martin Hofmann".

ICC by Example: Safe Recursion

Exponentiation

$$\mathsf{exp}(0) = 1$$
$$\mathsf{exp}(n+1) = \mathsf{add}(\mathsf{exp}(n), \mathsf{exp}(n))$$

Problem: Composing exp allows forming exponentials of arbitrary size.
Primitive recursion is well beyond any reasonable complexity class.
But, for expressivity, we also do not want to remove primitive recursion.

ICC by Example: Safe Recursion

Solution: distinguishing arguments that matter for complexity.

Arguments of any function f are either normal or safe.

- *normal* – if they can have a big impact on complexity of f
- *safe* – when they influence behavior of f very mildly.

Primitive recursion can now be restricted.

ICC by Example: Safe Recursion

New recursion scheme: $\vec{x}$ are normal $\vec{y}$ are safe parameters.

$$f(0, \vec{x}; \vec{y}) = h(\vec{x}; \vec{y})$$
$$f(n+1, \vec{x}; \vec{y}) = g(n, \vec{x}; \vec{y}, f(n, \vec{x}; \vec{y}))$$

- first parameter of f must be normal
- recursive call must be forwarded to a safe argument of g

Theorem. The function algebra defined by safe recursion equals the class FP of polynomial time computable functions.[3]

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

The (Applied) ICC Challenge

ICC offers many compelling features, but has remained largely a theoretical novelty. The practical power, limitations, and capabilities of ICC are not well-understood.

Hypothesis

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

I have been exploring this hypothesis in two directions.

ICC in Automatic Resource Analysis

Apply ICC systems in the originally designed ways, toward resource 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 program exists.

ICC in Automatic Resource Analysis

Some research questions:

- Can we develop practical resource analyses based on ICC theories?
- Is the theory correct: can we prove soundness formally?
- If theories can be automated, what are their use cases?

Analyzing Extended Program Properties

ICC focuses on programming languages, i.e., aim to show that a program P satisfies some desirable property ϕ (typically resource usage), $P \vdash \phi$.

Idea: Change the *analyzed property*.

Analyzing Extended Program Properties

We have applied ICC in solving two other program analysis problems.

- Program transformation to increase parallelization potential
- Analysis of security properties, i.e., non-interference

Analyzing Extended Program Properties

Select findings

- ICC systems can be *flexible*: allow tracking other program properties.
- The original complexity result may be lost, but something else is gained.
- This kind of application develops deeper understanding of the underlying theory.
- Pushes to introduce ICC to other research communities.

Takeaway Messages

1. ICC offers complementary techniques for analyzing resource usage.
2. It is possible to adjust ICC analyses to track other program properties.
3. ICC has potential to guarantee properties by construction.

Implicit computational complexity is broader in utility than the name implies!

My defense date is 15 August 2025 · see **neea.pl**