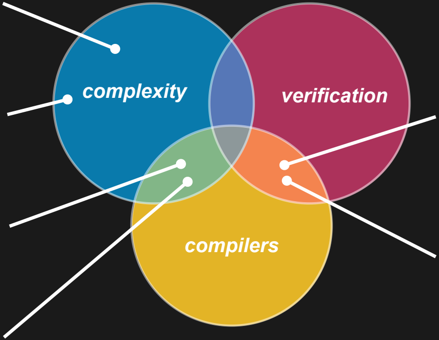
On Implicit Computational Complexity with Applications to Real-World Programs

Neea Rusch
Augusta University

17 May 2022

Topics: static analysis +



A Flow Calculus of *mwp*-Bounds for Complexity Analysis

Tight Polynomial Worst-Case Bounds for Loop Programs

Implicit Complexity in Theory and Practice

Loop Quasi-Invariant Chunk Detection

A Formally-Verified C Static Analyzer

Formal Verification of an SSA-Based Middle-End for CompCert

```
> is my program
 behavior correct?
```

```
git commit -m

"Works on my machine"
```

```
> run tests

Name          Stmts   Miss   Cover
----------------------------------
matrix.py       155      0    91 %
analysis.py     222     12    86 %
```

coverage 89%

static analysis

💯

Static analysis offers much stronger guarantees

- Evaluates program behavior for all inputs
- Analyzes programs statically, without execution
- Typically performed using an automated tool
- Study various properties: data flow, errors, resources, . . .
- More use cases: optimize programs, improve compilers

. . . but static analysis is difficult

- Limited information: only what is known statically or at compile time
- Analyser itself is software — can we trust its result?
- Rice's theorem: all non-trivial semantic properties are undecidable

Why complexity analysis?

According to Jean-Yves Moyen[1], there are many good reasons.

Different programs can compute the same function, and knowing their resource usage is useful:

- Predict the amount of resources needed to ensure it can run on a given computer
- Determine which parts of the program are (or are not) efficient

---

[1] Moyen, Jean-Yves. 2017. "*Implicit Complexity in Theory and Practice.*" Habilitation à Diriger des Recherches (HDR). University of Copenhagen.

Why complexity analysis?

"Certifying program resource usages is possibly as crucial as the specification of program correctness, since a guaranteed correct program whose memory usage exceeds available resources is, in fact, unreliable."[2]

---

[2] Aubert, Clément, et al. 2022. "*mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity.*"

Traditional Computational Complexity theory

Traditional approach uses computational models.

- Models lack expressivity – not used to program anything
- Real programs are not suitable for analysis on these models

Implicit Computational Complexity (ICC) theory

Definition by Romain Péchoux:[3]

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.

---

[3] Péchoux, Romain. 2020. "*Complexité implicite: bilan et perspectives.*" Habilitation à Diriger des Recherches (HDR). Université de Lorraine.

“A Flow Calculus of mwp-Bounds for Complexity Analysis”

Neil D. Jones and Lars Kristiansen (2009)

mwp-Analysis: Introduction

Is growth of variable values polynomially bounded?

- Will use the *mwp*-Calculus to determine this
- Program variables are collected in a matrix
- Flows in matrix characterize variable dependencies:

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

  weaker
  ↓
  stronger

- If derivation exists, then polynomially bounded

mwp-Analysis: Program Syntax

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

mwp-Analysis: Derivation Example

Let's analyze this program: `loop X3 {X2 = X1 + X2}`

mwp-Analysis: Derivation Example

```
loop X3 { X2 = X1 + X2 }
```

$$\frac{}{\vdash \mathsf{X}_i : \left\{\substack{m \\ i}\right\}}\ \text{E1}$$

mwp-Analysis: Derivation Example

`loop X3 { X2 = X1 + X2 }`

$$\texttt{X1} : \begin{bmatrix} m \\ 0 \\ 0 \end{bmatrix} \qquad \texttt{X2} : \begin{bmatrix} 0 \\ m \\ 0 \end{bmatrix} \qquad \frac{}{\vdash \mathsf{X}_i : \{{}^{m}_{i}\}}\ \text{E1}$$

mwp-Analysis: Derivation Example

```
loop X3 { X2 = X1 + X2 }
```

$\texttt{X1} : \begin{bmatrix} m \\ 0 \\ 0 \end{bmatrix}$

$\texttt{X2} : \begin{bmatrix} 0 \\ m \\ 0 \end{bmatrix}$

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

$$\frac{\vdash \mathsf{e1} : V_1 \quad \vdash \mathsf{e2} : V_2}{\vdash \mathsf{e1} + \mathsf{e2} : pV_1 \oplus V_2} \; \text{E3}$$

$$\frac{\vdash \mathsf{e1} : V_1 \quad \vdash \mathsf{e2} : V_2}{\vdash \mathsf{e1} + \mathsf{e2} : V_1 \oplus pV_2} \; \text{E4}$$

mwp-Analysis: Derivation Example

`loop X3 { X2 = X1 + X2 }`

$$\texttt{X1} : \begin{bmatrix} m \\ 0 \\ 0 \end{bmatrix} \qquad \texttt{X2} : \begin{bmatrix} 0 \\ m \\ 0 \end{bmatrix}$$

$$\texttt{X1 + X2} : \begin{bmatrix} p \\ m \\ 0 \end{bmatrix}$$

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

$$\frac{\vdash \mathsf{e1} : V_1 \quad \vdash \mathsf{e2} : V_2}{\vdash \mathsf{e1} + \mathsf{e2} : pV_1 \oplus V_2} \text{ E3}$$

$$\frac{\vdash \mathsf{e1} : V_1 \quad \vdash \mathsf{e2} : V_2}{\vdash \mathsf{e1} + \mathsf{e2} : V_1 \oplus pV_2} \text{ E4}$$

mwp-Analysis: Derivation Example

`loop X3 { X2 = X1 + X2 }`

$$\texttt{X1 + X2} : \begin{bmatrix} p \\ m \\ 0 \end{bmatrix}$$

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

mwp-Analysis: Derivation Example

`loop X3 { X2 = X1 + X2 }`

$$\texttt{X1 + X2} : \begin{bmatrix} p \\ m \\ 0 \end{bmatrix}$$

$$\texttt{X2 = X1 + X2} : \begin{bmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix}$$

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

mwp-Analysis: Derivation Example

`loop X3` $\{$ `X2 = X1 + X2` $\}$

`X2 = X1 + X2` $: \begin{bmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & 0 & m \end{bmatrix}$

$$\forall i, [M^*_{ii} = m] \ \frac{\vdash \mathsf{C} : M}{\vdash \mathsf{loop}\ \mathsf{X}_l\{\mathsf{C}\} : M^* \oplus \{ {}^p_l\!\!\rightarrow j \mid \exists i[M^*_{ij} = p]\}} \ \mathsf{L}$$

mwp-Analysis: Derivation Example

$$\texttt{loop X3 } \{\texttt{X2 = X1 + X2}\} : \begin{bmatrix} m & p & 0 \\ 0 & m & 0 \\ 0 & p & m \end{bmatrix}$$

mwp-Analysis: Debrief

- …it works!
- When $\models C : M$ holds, the bound property is guaranteed: invalid programs are not accepted.
- It is a theoretical approach: does it work on real programs?
- How big is the class of programs that can be analyzed?
- The bound is coarse
- The syntax is restrictive

About tight bounds

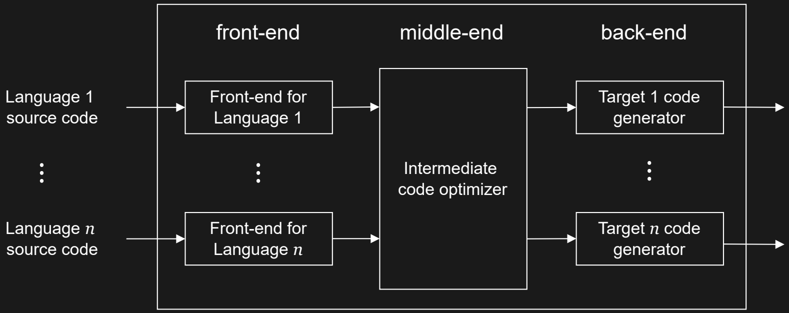
In "*Tight Polynomial Worst-Case Bounds for Loop Programs*", Amir Ben-Amram and Geoff Hamilton (2020) show that for a simple imperative core language

- It is possible to obtain asymptotically-tight Θ-bound, up to multiplicative constant factor
- The bound is multivariate and disjunctive
  e.g., $\langle x_1^2, x_2, x_2 + x_3 \rangle$ is tight bound of x_1, x_2 and x_3
- Complete solution: if polynomial bound exists it will be found

…but what to do about restrictive syntax?

Compilers

Classic architecture has 3 parts



front-end

middle-end

back-end

Language 1 source code

⋮

Language n source code

Front-end for Language 1

⋮

Front-end for Language n

Intermediate code optimizer

Target 1 code generator

⋮

Target n code generator

Compilers

Compilers are the natural place to introduce ICC systems[4].

- Most work is done in Intermediate Representation (IR)
- IR is generic, typed, assembly-like
- Analyses and optimizations already occur in these intermediate passes
- Any language supported by front-end can be analyzed

Maybe this will work for ICC analysis on real programs?

---

[4] Moyen, Jean-Yves. 2017. "*Implicit Complexity in Theory and Practice.*" Habilitation à Diriger des Recherches (HDR). University of Copenhagen.

ICC meets compilers

In "*Loop Quasi-Invariant Chunk Detection*" by Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller (2017):

- Introduce an automatable loop optimization technique
- Analyzes loop *quasi-invariants*, that become fixed after finite iterations; the number of iterations is *invariance degree*
- Method can handle blocks of statements and arbitrary depth of invariance degree
- If a chunk is an inner loop, hoisting it reduces complexity

ICC meets compilers

In "*Loop Quasi-Invariant Chunk Detection*" by Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller (2017):

- Paper comes with two artifacts: standalone tool and LLVM compiler prototype pass
- Implementation assumes programs in static single assignment (SSA) form
- SSA-form is property of some IRs where variables are assigned once

This is the first known application of ICC techniques in a mainstream compiler.

. . . but recall this initial challenge

Analyser itself is software

can we trust its result?

Formally verified software

- Correctly implemented program may not behave correctly as an executable
- Result of static analysis may not hold in the executable program
- . . . *unless* compiler guarantees preservation of semantics
- We can use mechanical proofs to establish rigorous guarantees of correctness using proof assistants

How realistic is this approach?

We already have the CompCert compiler

The CompCert C verified compiler[5]

- A realistic, high-assurance compiler for almost all of `C`
- Comes with a mathematical, machine-checked proofs
- Generated executable code behaves exactly as prescribed by the semantics of the source program

---

[5] https://compcert.org

Formally verified static analysis is doable

In "*A Formally-Verified C Static Analyzer*" by Jacques-Henri Jourdan et al. (2015):

- Verasco – the first formally verified static analyzer
- Based on abstract interpretation and detects runtime errors
- Integrates with CompCert such that guarantees established by Verasco carry over to the compiled code

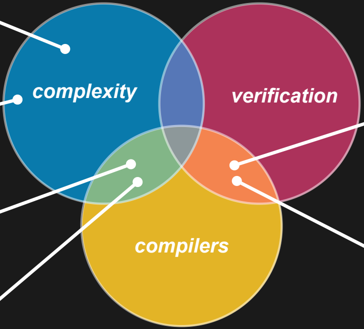
But what about SSA-form?

Formally verified SSA-form middle-end also exists

In "*Formal Verification of an SSA-based Middle-end for CompCert*" by Gilles Barthe, Delphine Demange, and David Pichardie (2014):

- SSA form is useful for many optimizations, but not used in CompCert
- The result is a formally verified middle-end implementation
- Middle-end translates in and out of SSA form and performs sample optimization

All the necessary pieces are now in place



A Flow Calculus of *mwp*-Bounds for Complexity Analysis

Tight Polynomial Worst-Case Bounds for Loop Programs

Implicit Complexity in Theory and Practice

Loop Quasi-Invariant Chunk Detection

A Formally-Verified C Static Analyzer

Formal Verification of an SSA-Based Middle-End for CompCert

Future directions

Extensions of Implicit Computational Complexity

- So far these techniques exist almost only on paper
- Powerfulness — what can be said about the classes of programs they can analyse?
- Applied applications and study of extended properties
  - power usage, error growth, etc.
  - optimizations based on these analyses

Future directions

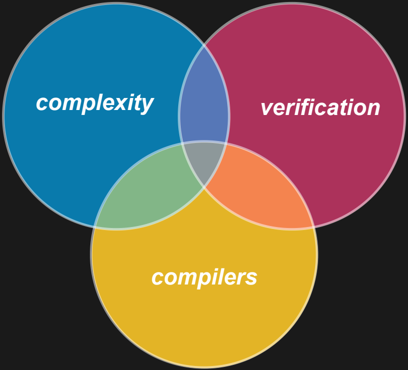
Integrating ICC-based analyses in compilers

- Do these systems work on real languages, with memory accesses, classes, recursion, etc.?
- This is a realistic target for applying these methods

Future directions

Verified complexity analysis

- Gives strongest possible assurance of result correctness
- Implementations using other techniques and for other properties exist — but not verified complexity analysis
- The *mwp*-analysis is a potentially good candidate


complexity
verification
compilers