ICC@ICC: a taste of 2nd-order polytime complexity

Romain Péchoux,
Inria team Mocqua - CNRS, Inria, Université de Lorraine - LORIA

LoReL’s seminar

November 28th, 2022
Today’s talk

Today, we will focus on:

1. a brief overview of ICC (Implicit Computational Complexity)
2. a characterization of BFF (Basic Feasible Functionals)
   - $\approx$ 2nd order polynomial time
   - a work with Emmanuel Hainry, Bruce Kapron, and Jean-Yves Marion
Computational Complexity (CC) studies problems/functions wrt resource usage.

The Universe of mathematical functions

(Images: NASA)
Computational Complexity (CC) studies problems/functions wrt resource usage.

The Universe of mathematical functions

The Galaxy of computable functions

(Images: NASA)
Computational Complexity (CC) studies problems/functions wrt resource usage.

Assume Cobham-Edmonds thesis: tractable/feasible $\equiv$ polynomial time.

(Images: NASA)
Implicit computational complexity (ICC)

**ICC**: Subfield of CC aiming at providing characterizations of complexity classes:
- **machine-independent**
- with **no prior knowledge** on the complexity analyzed codes

If the characterization is **tractable** then ICC provides **automatic** static complexity analysis methods for **high level** PL.

State of the art:
- 30 years of intensive research,
- hundreds of publications,
- some academic tools
  - (Costa, SPEED, TcT, ...).
The ICC approach

**ICC criterion**

Take your favourite PL $\mathcal{L}$ and your favourite complexity class $\mathcal{C}$:

$$\mathcal{R} \subseteq \mathcal{L} \text{ is an ICC criterion if } \{[p] \mid p \in \mathcal{R}\} = \mathcal{C}.$$  

**Examples of complexity class $\mathcal{C}$**

- P, FP,
- PSPACE, FPSPACE,
- EXP, 2-EXP, \ldots, ELEMENTARY,
- NP,
- NC$^0$, NC$^1$, \ldots, NC
- PP, BPP, EQP, BQP, \ldots

**Examples of programming language $\mathcal{L}$**

- lambda-calculi,
- term rewrite systems,
- process calculi,
- reactive programs,
- imperative and OO programs,
- probabilistic and quantum programs.
A bunch of techniques (1/2)

Some ICC criteria

▶ **function algebra**: [Cobham65], [Bellantoni-Cook92], [Clote99] for a survey

▶ **linear logic** based approaches
  ▶ light logics: LLL [Girard87], ILAL [Asperti-Roversi02], DLAL [Baillot-Terui04],
  ▶ soft logics: SLL [Lafont04], STA [Gaboardi-Ronchi Della Rocca07],
  ▶ non size-increasing [Hofmann99].

▶ “**potential**” based methods
  ▶ interpretations: “quasi” [Bonfante-Marion-Moyen11], “sup” [Marion-Péchoux09],
    higher-order [Baillot-Dal Lago16],
  ▶ amortized resource analysis: [Jost et al.10], [Hoffmann-Hofmann10],
  ▶ sized-types: [Vasconcelos08], [Avanzini-Dal Lago17],
  ▶ cost semantics: [Danner et al.15].
A bunch of techniques (2/2)

Some ICC criteria

- **control flow (tiering-based)** techniques:
  - safe recursion [Bellantoni-Cook92],
  - ramified recurrence [Leivant-Marion94],
  - tiering [Marion11],
  - read-only/write-only: [Jones01], [De Carvalho-Simonsen14].

- **matrix-based** type systems:
  - $\mu$-measure [Niggl-Wunderlich06],
  - mwp bounds [Kristiansen-Jones09], resource control graphs [Moyen09].

- **empirical approaches** (some of them using abstract interpretations): COSTA [Albert et al.07], SPEED [Gulwani09], TcT [Avanzini-Moser-Schaper16].
Main techniques (1/2): typing

Tractable functions

Characterized by all techniques by preventing exponentiation, i.e. by preventing the iteration of methods duplicating the size of their inputs.

- Prevent iteration with a type discipline:
  - !A \rightarrow \aleph A in LAL,
  - 1 \rightarrow 0 in tier-based approaches,
  - Read-Only \rightarrow Write-Only in Jones/Simonsen
  - \begin{pmatrix} P \end{pmatrix} in mwp (whereas \begin{pmatrix} M \end{pmatrix} is required for iterability).
Main techniques (2/2): potentials

Tractable functions

Characterized by all techniques by preventing exponentiation, i.e. by preventing the iteration of methods duplicating the size of their inputs.

- By using a potential-based constraints implying a decrease along reduction:

  \[
  t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n
  \]

  \[
  P \geq [t_1] \geq [t_2] \geq \ldots \geq [t_n]
  \]

- (polynomial) interpretations-based methods,
- amortized resource analysis,
- ert-transformers method [Kaminski et al.06],
- sized-types.
## Intensional limits

### Definition [Intensional completeness]

A characterization is intensionally complete if any tractable algorithm computing this function is accepted.

### Theorem [Hajek79]

Providing an intensionally-complete characterization of tractable functions is a $\Sigma^2_0$-complete problem.

However, for automation purpose, the studied characterizations are decidable (even better tractable).

### Observation

Hence there are false negative.
Beyond ICC: extensions

**Intensional improvements**
- Soft Type Assignment
  [Gaboardi-Ronchi Della Rocca07]
- Dual Light Affine Logic
  [Baillot-Terui04]
- Sup-interpretations
  [Marion-Péchoux09]

**Adaptations of existing tools**
- Tiering on imperative programs
  [Marion11], [Marion-Leivant13]
- Tiering on OO programs
  [Hainry-Péchoux18]
- Interpretations of HO-TRS (STTRS)
  [Baillot-Dal Lago12]

**Extensions to new paradigms**
- Concurrent systems
  - Light logics and multi-threads
    [Amadio-Madet11]
  - Soft logics and processes
    [Martini-Dal Lago-Sangiorgi16]
- Probabilistic programs:
  [Avanzini-Dal Lago-Ghyselen19]
- Quantum programs
  [Dal Lago-Masini-Zorzi10]
- Real functions
  [Bournez-Gomaa-Hainry11]
- Coinductive data
  [Gaboardi-Péchoux15]
Summary on ICC

Strong links with other research domains:

- Termination techniques (often coming from and/or combined with)
- Computability theory (Primrec, undecidable classes, . . .)
- Finite model theory (common goals)
- Static analysis (type systems, abstract interpretations, empirical approaches)

A survey on ICC in my HDR, available at https://members.loria.fr/RPechoux/
What about 2nd order complexity classes?

2nd-order objects are functions in \( (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \)

2nd-order polynomial time is taken to be the class of Basic Feasible Functionals (BFF)

Goal (Open problem for more than 20 years)

Find a **tractable** static analysis technique for certifying **2nd order polynomial time** complexity.

Rephrasing: Find a tractable restriction \( \mathcal{R} \) such that \( \llbracket \mathcal{R} \rrbracket = \text{BFF} \).

N.B.: The problem was solved for **type-1** polytime FP by Bellantoni and Cook in 1992.
A reminder on 2nd order polynomial time

BFF was introduced by Melhorn in 1976.

\[
\text{Theorem [Cook and Urquhart [1989]]}
\]

\[
\text{BFF} = \lambda (\text{FP} \cup \{R\})_2
\]

\[R\text{ is a type-2 bounded iterator:}
\]
\[
R(\epsilon, a) = a
\]
\[
R(ix, a) = \min(\phi(ix, R(x, a)), \psi(ix))
\]

\[
\text{Theorem [Cook and Kapron [1990]]}
\]

The set of functionals computable by an OTM in time \(P(|\phi|, |a|)\) is exactly BFF.

2nd order polynomials and size function are defined by:

- \(P(X_1, X_0) ::= c \in \mathbb{N} \mid X_0 \mid X_1(P) \mid P + P \mid P \times P\)
- \(|\phi|(n) = \max_{|x| \leq n} |\phi(x)|\)
How to get rid of 2nd order polynomials?

**Definition [Oracle Polynomial Time (OPT) [Cook92]]**

Let \( n^{\phi,a} \) be the biggest size of \( a \) and of an oracle’s answer in the run of \( M(\phi, a) \).
An OTM is in OPT if its runtime is bounded by \( P(n^{\phi,a}) \), for some type-1 polynomial \( P \).

\[ \text{BFF} \subsetneq \text{OPT} \text{ as it contains exponential functions.} \]

**Theorem [Kapron and Steinberg [2018]]**

\[ \text{BFF} = \lambda(\text{OPT} \cap \text{FLR})_2 = \lambda(\text{OPT} \cap \text{FLAR})_2 \]

- FLR = Finite Length Revision
- FLAR = Finite LookAhead Revision
Finite Length Revision

Definition [Finite Length Revision - Kawamura and Steinberg [2017]]

An OTM is in FLR, if, for any input, the number of times the oracle answer is bigger than all of the previous oracle answers is bounded by a constant.

Example

```c
while (x>0){
    y = \phi(x);
    x = x - 1;
}
not (FLR) if \phi ↘
```

Example

```c
while (x<n && y<8){
    y = \phi(x);
    x = x + 1;
}
(FLR) with constant 8
```
Finite LookAhead Revision

Definition [Finite LookAhead Revision - Kapron and Steinberg [2018]]

An OTM is in FLAR, if, for any input, the number of times a query is posed whose size exceeds the size of all previous queries is bounded by a constant.

Example

```
while (x>0){
    y = φ(x);
    x = x−1;
}
```

(FLAR) with constant 0

Example

```
while (x<n && y<8){
    y = φ(x);
    x = x+1;
}
```

not (FLAR) for \( φ = λz.4 \)
How to get rid of (Oracle Turing) machines?

→ Design a typed PL ensuring that computed functions are in $\text{OPT} \cap \text{FLAR}$.

**Imperative PL on words with oracles**

**Expressions** $\ni e ::= x | \text{true} | \text{false} | \text{op}(e, \ldots, e) | \phi(e \upharpoonright e)$

**Commands** $\ni \text{st} ::= x := e; \mid \text{st} \; \text{st} \mid \text{if}(e)\{\text{st}\} \text{else}\{\text{st}\} \mid \text{while}(e)\{\text{st}\}$

In an oracle call $\phi(w \upharpoonright v)$:

- $\phi$ computes a type-1 function on words, i.e. $\phi \in W \rightarrow W$.
- $w$ is the **oracle input**.
- $v$ is the **input bound**: $w \upharpoonright v = w_1 \ldots w_{|v|}$.
Tier-based type discipline

Tiers $k, k’, ...$ are security levels (in $\mathbb{N}$) assigned to Expressions and Commands.

The type system ensures some non-interference properties.

In a tier $k$ command:

- the program flow cannot be controlled by expressions of a lower tier $k^- < k$,
- data of upper tier $k^+ \geq k$ cannot increase (in size).

Judgments: $\Gamma, \Delta \vdash st : (k, k_{in}, k_{out})$ with $(k, k_{in}, k_{out}) \in \mathbb{N}^3$

1. The tier $k$ implements the non-interference policy.
2. The innermost tier $k_{in}$ is used for declassification.
3. The outermost tier $k_{out}$ is used to ensure FLAR on oracle calls.
Tier-based type system: an overview

Typing rules

\[ \frac{}{\vdash x : (k_1, k_{in}, k_{out})} \]
\[ \frac{}{\vdash e : (k_2, k_{in}, k_{out})} \quad k_1 \leq k_2 \] (Asg)
\[ \vdash x := e : (k_1, k_{in}, k_{out}) \]

\[ \frac{}{\vdash e : (k, k_{in}, k_{out})} \quad \vdash st : (k, k_{in}, k_{out}) \quad 1 \leq k \leq k_{out} \] (Wh)
\[ \vdash \text{while}(e\{st\}) : (k, k_{in}, k_{out}) \]

\[ \frac{}{\vdash e : (k, k_{in}, k_{out})} \quad \vdash e' : (k_{out}, k_{in}, k_{out}) \quad k < k_{in} \leq k_{out} \] (Orc)
\[ \vdash \phi(e \upharpoonright e') : (k, k_{in}, k_{out}) \]
\[ \vdots \]
Illustrating example

Program computing the decision problem $\exists n \leq x, \phi(n) = 0$.

```python
y = x;
z = false;
while(x^1 >= 0){
    if($\phi(y^0 | x^1) == 0$){
        z^0 = true;
    } else {
        x^1 = x^1 - 1;
    }
}
return z
```

- The program is typable and the while body has tier $(1, 1, 1)$.
- The computed function is in $OPT \cap FLAR$. 
A tier-based characterization of BFF

- Let SAFE be the set of typable programs.
- Let SN be the set of strongly normalizing programs.
- Let \([X]\) be the set of functions computed by programs in X.

**Theorem [Hainry-Kapron-Marion-Péchoux [LICS2020]]**

\[ \text{BFF} = \lambda([\text{SAFE} \cap \text{SN}])_2 \]

**Main drawbacks:**

- Lambda closure (for completeness)
- Termination assumption (for soundness)
How to get rid of the lambda-closure?

Naïve idea: internalize lambda-abstraction and application into the language. → cannot be done straightforwardly as it breaks soundness.

Extended language (e_i: e is a type-i object)

(Expressions)  e ::= x_0 | op(e,...,e) | x_1(e ↾ e)
(Statements)   st ::= [x_0 := e]; | st st | if(e){st}{st} | while(e){st}
(Procedures)   P ::= P(x_1,x_0){st return x_0}
(Terms)        t ::= x | λx.t | t@t | call P(x_0 → t_0,t_0)
(Programs)     prog ::= t_0 | declare P in prog

Solution: type-1 arguments in a procedure call are restricted to closures {x_0 → t_0}.
Type system

The extended type system just consists of two layers:

- SAFE procedures (using our [LICS2020] paper),
- Simply-typed terms on words $\mathbb{W}$.

Definitions

A program is a **type-i** program if all its $\lambda$-abstractions are of order $\leq i$.

- $\text{SAFE}_i$ is the set of type-i typable programs.
  - Remark: $\text{SAFE}_0$ is the set of typable programs without lambda-abstraction.
- SN is still the set of strongly normalizing programs.
Example

\[
\text{prog}(\phi, w) \triangleq \text{declare } \text{KS}(Y, v) \{ \\
\quad u := 10; \\
\quad z := \varepsilon; \\
\quad \text{while } (v^1 \neq 0) \{ // k_{in} = k_{out} = 1 \\
\qquad v^1 := v - 1; \\
\qquad z^0 := Y(z^0 \upharpoonright u^1) \\
\}\} \text{ return } z \\
\text{in call } \text{KS}({x \rightarrow \phi \circ (\phi \circ x)}, w)
\]

\[\boxed{\begin{align*}
\quad [\text{prog}] &\in (W \rightarrow W) \rightarrow W \rightarrow W \\
\quad [\text{prog}]_{(\phi_{W \rightarrow W}, w_{W})} &= F_{|w|}(\phi) \text{ with } \begin{cases} \\ F_0(\phi) = \varepsilon \\
F_{n+1}(\phi) = (\phi \circ \phi)(F_n(\phi) \leq |10|)
\end{cases} \\
\quad \text{prog} &\in \text{SAFE}_0 \cap \text{SN} \text{ whereas } [\text{prog}] \notin \text{OPT} \cap \text{FLAR}.
\end{align*}}\]
First implicit and complete characterizations of BFF

Characterizations without lambda-closure:

Theorem [Hainry-Kapron-Marion-Péchoux [FoSSaCS2022]]

\( \forall i \geq 0, \ \llbracket \text{SAFE}_i \cap \text{SN} \rrbracket = \text{BFF} \)

Lambda-abstraction is not required for completeness:

Theorem [Hainry-Kapron-Marion-Péchoux [FoSSaCS2022]]

\( \llbracket \text{SAFE}_0 \cap \text{SN} \rrbracket = \text{BFF} \)

In particular \( \llbracket \text{prog} \rrbracket \in \llbracket \text{SAFE}_0 \cap \text{SN} \rrbracket \).

→ Can we weaken the SN requirement?
How to get rid of Strong Normalization?

We consider Size Change Termination (SCT).

General idea

Program:

\[
\text{while } (x > 0) \{
  y = \phi(x); \\
  x = x - 1;
\}
\]

\[
\text{Size change graph abstraction:}
\begin{pmatrix}
  x & \rightarrow & x \\
  y & & y
\end{pmatrix}^\omega
\]

Theorem [Lee, Jones, and Ben Amram [POPL2001]]

“If every infinite computation would give rise to an infinitely decreasing value sequence in the size-change graph, then no infinite computation is possible.”

→ SCT is not “tractable”: PSPACE-complete.
Tractable characterizations of BFF

Completeness is preserved for SCT and for an instance SCP (Ben Amram-Lee [2007]).

Theorem [Hainry-Kapron-Marion-Péchoux [FoSSaCS2022]]

∀ \( i \geq 0 \), \( \text{SAFE}_i \cap \text{SCP}_S = \text{BFF} \)

\( \text{SCP}_S \) can be decided in time quadratic in the program size.

Theorem [Type inference]

- \( \text{prog} \in \bigcup_i \text{SAFE}_i \cap \text{SCP}_S \) is Ptime-complete (using Mairson[2004]).
- \( \text{prog} \in \text{SAFE}_0 \cap \text{SCP}_S \) is in time cubic in \( |\text{prog}| \) (using HKMP[2022]).
We have obtained **sound** and **complete** characterizations of type-2 polynomial time:

- **machine-independent**, 
  - a typed programming language with procedure calls
- **implicit**, 
  - no prior knowledge on the bound is required
- **tractable** and can thus be automated.
  - decidable type inference (in polynomial time)

**Open issues**

- What about Finite Length Revision (FLR)?
- Delineate a larger family of completeness preserving termination techniques.
- Adapt this method to a purely functional Programming Language.