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)

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

The Universe of
mathematical functions

(Images: NASA)

Computational Complexity (CC)

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

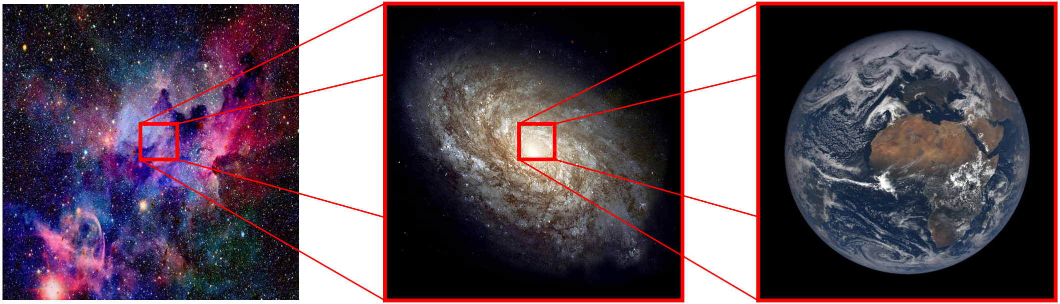
The Universe of mathematical functions

The Galaxy of **computable** functions

(Images: NASA)

Computational Complexity (CC)

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



The Universe of mathematical functions

The Galaxy of **computable** functions

The Planet of **tractable** functions

Assume Cobham-Edmonds thesis: **tractable/feasible** = **polynomial time**.

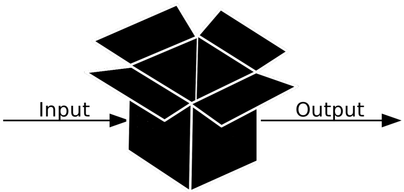
(Images: NASA)

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 $\mathcal{C}$:

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

Examples of complexity class $\mathcal{C}$

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

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 reccurence [Leivant-Marion94],
  - tiering [Marion11],
  - read-only/write-only: [Jones01], [De Carvalho-Simonsen14].
- **matrix-based** type systems:
  - μ-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**:
  - $!A \multimap \S A$ in LAL,
  - $1 \rightarrow 0$ in tier-based approaches,
  - Read-Only $\rightarrow$ Write-Only in Jones/Simonsen
  - $\begin{bmatrix} \ddots & & \\ & P & \\ & & \ddots \end{bmatrix}$ in mwp (whereas $\begin{bmatrix} \ddots & & \\ & M & \\ & & \ddots \end{bmatrix}$ 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:

$$\begin{array}{ccccccccc} & t_1 & \to & t_2 & \to & \dots & \to & t_n \\ P \geq & [t_1] & \geq & [t_2] & \geq & \dots & \geq & [t_n] \end{array}$$

  - (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 Σ_0^2-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 $\overbrace{(\mathbb{N} \to \mathbb{N})}^{\phi} \to \mathbb{N}$
2nd-order polynomial time is taken to be the class of Basic Feasible Functionals ($\mathtt{BFF}$)

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 $[\![\mathcal{R}]\!] = \mathtt{BFF}$.

N.B.: The problem was solved for **type-1** polytime $\mathtt{FP}$ by Bellantoni and Cook in 1992.

A reminder on 2nd order polynomial time

BFF was introduced by Melhorn in 1976.

Theorem [Cook and Urquhart [1989]]

$\mathtt{BFF} = \lambda(\mathtt{FP} \cup \{\mathcal{R}\})_2$

$\mathcal{R}$ is a type-2 bounded iterator:

$$\mathcal{R}(\epsilon, a) = a$$
$$\mathcal{R}(ix, a) = \min(\phi(ix, \mathcal{R}(x, a)), \psi(ix))$$

Theorem [Cook and Kapron [1990]]

The set of functionals computable by an OTM in time $P(|\phi|, |\mathbf{a}|)$ 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 $n^{\phi,\mathbf{a}}$ be the biggest size of $\mathbf{a}$ and of an oracle's answer in the run of $M(\phi, \mathbf{a})$.
An OTM is in OPT if its runtime is bounded by $P(n^{\phi,\mathbf{a}})$, for some type-1 polynomial P.

BFF $\subsetneq$ OPT as it contains exponential functions.

Theorem [Kapron and Steinberg [2018]]

$\mathtt{BFF} = \lambda(\mathrm{OPT} \cap \mathrm{FLR})_2 = \lambda(\mathrm{OPT} \cap \mathrm{FLAR})_2$

- 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

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

not (FLR) if $\phi \searrow$

Example

```
while (x<n && y<8){
        y = φ(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 $\phi = \lambda z.4$

How to get rid of (Oracle Turing) machines?

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

Imperative PL on words with oracles

$$\mathit{Expressions} \ni \mathtt{e} ::= \mathtt{x} \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{op}(\mathtt{e}, \ldots, \mathtt{e}) \mid \phi(\mathtt{e} \upharpoonright \mathtt{e})$$

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

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

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

Tier-based type discipline

Tiers $\mathsf{k}, \mathsf{k}', \ldots$ 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 $\mathsf{k}^- < \mathsf{k}$,
- data of upper tier $\mathsf{k}^+ \geq \mathsf{k}$ cannot increase (in size).

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

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

$$\frac{\vdash \mathtt{x} : (\mathsf{k}_1, \mathsf{k}_{in}, \mathsf{k}_{out}) \qquad \vdash \mathsf{e} : (\mathsf{k}_2, \mathsf{k}_{in}, \mathsf{k}_{out}) \qquad \mathsf{k}_1 \leq \mathsf{k}_2}{\vdash \mathtt{x} := \mathsf{e} \;\; : (\mathsf{k}_1, \mathsf{k}_{in}, \mathsf{k}_{out})} \text{ (Asg)}$$

$$\frac{\vdash \mathsf{e} : (\mathsf{k}, \mathsf{k}_{in}, \mathsf{k}_{out}) \qquad \vdash \mathtt{st} : (\mathsf{k}, \mathsf{k}, \mathsf{k}_{out}) \qquad 1 \leq \mathsf{k} \leq \mathsf{k}_{out}}{\vdash \mathtt{while}(\mathsf{e})\{\mathtt{st}\} \;\; : (\mathsf{k}, \mathsf{k}_{in}, \mathsf{k}_{out})} \text{ (Wh)}$$

$$\frac{\vdash \mathsf{e} : (\mathsf{k}, \mathsf{k}_{in}, \mathsf{k}_{out}) \;\; \vdash \mathsf{e}' : (\mathsf{k}_{out}, \mathsf{k}_{in}, \mathsf{k}_{out}) \qquad \mathsf{k} < \mathsf{k}_{in} \leq \mathsf{k}_{out}}{\vdash \phi(\mathsf{e} \upharpoonright \mathsf{e}') : (\mathsf{k}, \mathsf{k}_{in}, \mathsf{k}_{out})} \text{ (Orc)}$$

$$\vdots$$

Illustrating example

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

$\mathtt{y} = \mathtt{x};$
$\mathtt{z} = \mathit{false};$
$\mathtt{while}(\mathtt{x}^1 >= 0)\{$
$\quad \mathtt{if}(\phi(\mathtt{y}^0 \restriction \mathtt{x}^1) == 0)\{$
$\quad\quad \mathtt{z}^0 = \mathit{true};$
$\quad \}\ \mathtt{else}\ \{;\}$
$\quad \mathtt{x}^1 = \mathtt{x}^1 - 1;$
$\}$
$\mathtt{return}\ \ \mathtt{z}$

- The program is typable and the while body has tier $(1, 1, 1)$.
- The computed function is in $\mathtt{OPT} \cap \mathtt{FLAR}$.

A tier-based characterization of BFF

- Let SAFE be the set of typable programs.
- Let SN be the set of strongly normalizing programs.
- Let $[\![\mathsf{X}]\!]$ be the set of functions computed by programs in X.

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

$\mathrm{BFF} = \lambda([\![\mathsf{SAFE} \cap \mathsf{SN}]\!])_2$

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.
$\rightarrow$ cannot be done straightforwardly as it breaks soundness.

Extended language (e_{i}: e is a type-i object)

$$
\begin{array}{lrcl}
(\texttt{Expressions}) & \texttt{e} & ::= & \texttt{x}_0 \mid \texttt{op}(\texttt{e},\ldots,\texttt{e}) \mid \texttt{x}_1(\texttt{e} \upharpoonright \texttt{e}) \\
(\texttt{Statements}) & \texttt{st} & ::= & [\texttt{x}_0 := \texttt{e}]; \mid \texttt{st st} \mid \texttt{if}(\texttt{e})\{\texttt{st}\}\{\texttt{st}\} \mid \texttt{while}(\texttt{e})\{\texttt{st}\} \\
(\texttt{Procedures}) & \texttt{P} & ::= & \texttt{P}(\overline{\texttt{x}_1},\overline{\texttt{x}_0})\{\texttt{st}\ \ \texttt{return}\ \ \texttt{x}_0\} \\
(\texttt{Terms}) & \texttt{t} & ::= & \texttt{x} \mid \lambda\texttt{x}.\texttt{t} \mid \texttt{t@t} \mid \texttt{call P}(\overline{\{\texttt{x}_0 \rightarrow \texttt{t}_0\}},\overline{\texttt{t}_0}) \\
(\texttt{Programs}) & \texttt{prog} & ::= & \texttt{t}_0 \mid \texttt{declare P in prog}
\end{array}
$$

Solution: type-1 arguments in a procedure call are restricted to closures $\{\texttt{x}_0 \rightarrow \texttt{t}_0\}$.

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 λ-abstractions are of order $\leq i$.

- SAFE_i is the set of type-i typable programs.
  - Remark: SAFE_0 is the set of typable programs without lambda-abstraction.
- SN is still the set of strongly normalizing programs.

Example

$$
\begin{array}{l}
\texttt{prog}(\phi,\texttt{w}) \triangleq \texttt{declare KS(Y, v) \{} \\
\qquad \texttt{u} := \texttt{10;} \\
\qquad \texttt{z} := \varepsilon\texttt{;} \\
\qquad \texttt{while (}\texttt{v}^{1} \neq \texttt{0) \{ //}\ k_{in} = k_{out} = 1 \\
\qquad\quad \texttt{v}^{1} := \texttt{v-1;} \\
\qquad\quad \texttt{z}^{0} := \texttt{Y(}\texttt{z}^{0} \upharpoonright \texttt{u}^{1}\texttt{)} \\
\qquad \texttt{\}} \\
\qquad \texttt{return z} \\
\quad \texttt{\}} \\
\quad \texttt{in call KS(\{x} \rightarrow \phi\ \texttt{@}\ (\phi\ \texttt{@ x)\}, w)}
\end{array}
$$

- $[\![\texttt{prog}]\!] \in (\mathbb{W} \to \mathbb{W}) \to \mathbb{W} \to \mathbb{W}$
- $[\![\texttt{prog}]\!](\phi^{\mathbb{W}\to\mathbb{W}}, \texttt{w}^{\mathbb{W}}) = F_{|\texttt{w}|}(\phi)$ with $\begin{cases} F_0(\phi) = \epsilon \\ F_{n+1}(\phi) = (\phi \circ \phi)(F_n(\phi)^{\leq |10|}) \end{cases}$
- $\texttt{prog} \in \mathsf{SAFE}_0 \cap \mathsf{SN}$ whereas $[\![\texttt{prog}]\!] \notin \texttt{OPT} \cap \texttt{FLAR}$.

First implicit and complete characterizations of $\mathtt{BFF}$

Characterizations without lambda-closure:

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

$\forall i \geq 0,\ [\![\mathsf{SAFE}_i \cap \mathsf{SN}]\!] = \mathtt{BFF}$

Lambda-abstraction is not required for completeness:

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

$[\![\mathsf{SAFE}_0 \cap \mathsf{SN}]\!] = \mathtt{BFF}$

In particular $[\![\mathtt{prog}]\!] \in [\![\mathsf{SAFE}_0 \cap \mathsf{SN}]\!]$.

$\rightarrow$ Can we weaken the SN requirement?

How to get rid of Strong Normalization?

We consider Size Change Termination (SCT).

General idea

Program:

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

$\Longrightarrow$

Size change graph abstraction:

$$\begin{pmatrix} x \xrightarrow{-1} x \\ y \qquad\quad y \end{pmatrix}^{\omega}$$

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."

$\rightarrow$ 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]]

$\forall i \geq 0,\ [\![\mathsf{SAFE}_i \cap \mathsf{SCP}_S]\!] = \mathtt{BFF}$

SCP_S can be decided in time quadratic in the program size.

Theorem [Type inference]

- $\mathtt{prog} \in \cup_i \mathsf{SAFE}_i \cap \mathsf{SCP}_S$ is Ptime-complete (using Mairson[2004]).
- $\mathtt{prog} \in \mathsf{SAFE}_0 \cap \mathsf{SCP}_S$ is in time cubic in $|\mathtt{prog}|$ (using HKMP[2022]).

Conclusion

Conclusion

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.