Implicit characterization of the class of Basic Feasible Functionals

Romain Péchoux

CNRS, Inria, Université de Lorraine - LORIA,
* joint work with Emmanuel Hainry, Bruce Kapron and Jean-Yves Marion

Complexity Days 2023

Motivations

We aim at providing characterizations of complexity classes:

- **machine-independent**,
- with **no prior knowledge** on the complexity of analyzed codes.

If the characterization is **tractable** then we obtain an **automated complexity analysis** for a high-level programming language.

State of the art:

- 30 years of intensive research,
- hundreds of publications,
- some tools: Costa, SPEED, TcT, ...

The Implicit Computational Complexity approach

Methodology

Consider your favorite programming language $\mathcal{L}$ and your favorite complexity class $\mathcal{C}$.

$$\text{Find a tractable restriction } \mathcal{R} \subseteq \mathcal{L} \text{ such that } [\![\mathcal{R}]\!] = \mathcal{C},$$

where $[\![X]\!]$ is the set of functions computed by programs in X.

Examples of (type-1) complexity class $\mathcal{C}$

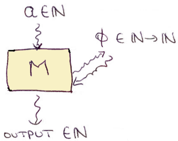
- P, FP,
- PSPACE, FPSPACE,
- NP,
- PP, BPP, EQP, BQP,
- ...

Examples of restriction $\mathcal{R}$

- type systems:
  - linear logic, sized types, ...
- interpretation methods,
- amortized resource analysis,
- ...

What about type-2 complexity classes?

Type-2 objects are functions in $\overbrace{(\mathbb{N} \to \mathbb{N})}^{\phi} \times \overbrace{\mathbb{N}}^{a} \to \mathbb{N}$



Type-2 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** technique for certifying **type-2 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 type-2 polynomial time

BFF was introduced by Melhorn in 1976.

Theorem [Cook and Urquhart [1989]]

$\mathrm{BFF} = \lambda(\mathrm{FP} \cup \{\mathcal{I}\})_2$

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

$$\mathcal{I}^{f,g}(\epsilon, a) = a$$
$$\mathcal{I}^{f,g}(ix, a) = \min(f(ix, \mathcal{I}^{f,g}(x, a)), g(ix))$$

$\lambda(X)_2$: type-2 restriction of the simply-typed lambda-closure using constants in X.

Theorem [Kapron and Cook [FOCS1991]]

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

Type-2 polynomials and size function are defined by:

- $P(X_1, X_0) ::= c \in \mathbb{N} \mid X_0 \mid P + P \mid P \times P \mid X_1(P)$
- $|\phi|(n) = \max_{|x| \leq n} |\phi(x)|$

How to get rid of type-2 polynomials?

$\rightarrow$ Type-2 polynomials are not tractable.

Definition [Oracle Polynomial Time ($\mathtt{OPT}$) – Cook [1992]]

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

$\mathtt{BFF} \subsetneq \mathtt{OPT}$, as $\mathtt{OPT}$ contains exponential functions.

Theorem [Kapron and Steinberg [LICS2018]]

$\mathtt{BFF} = \lambda(\llbracket \mathtt{OPT} \cap \mathtt{FLAR} \rrbracket)_2$

$\rightarrow$ $\mathtt{FLAR}$ = Finite LookAhead Revision

Finite LookAhead Revision

Definition [Finite LookAhead Revision - Kapron and Steinberg [LICS2018]]

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;
}
```

in FLAR.

The constant bound is 0.

Example

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

not in FLAR for $\phi = \lambda x.4$

but is in FLR (I will briefly mention this class in the conclusion)

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

$$\mathit{Expressions} \ni E ::= \mathtt{x} \mid \mathtt{true} \mid \mathtt{false} \mid op(E,\dots,E) \mid \boldsymbol{\phi(\mathbf{E} \restriction \mathbf{E})}$$

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

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

- 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**: $w \restriction v = w_1 \dots w_{|v|}$.

Tier-based type discipline

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

The type system ensures some non-interference properties.

In a tier $\mathbf{k}$ command:

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

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

1. The tier $\mathbf{k}$ implements the non-interference policy.
2. The *innermost* tier $\mathbf{k}_{in}$ is used for declassification.
3. The *outermost* tier $\mathbf{k}_{out}$ is used to ensure `FLAR` on oracle calls.

Tier-based type system: an overview

Typing rules

$$\frac{\vdash \mathtt{x} : (\mathbf{k}_1, \mathbf{k}_{in}, \mathbf{k}_{out}) \quad \vdash E : (\mathbf{k}_2, \mathbf{k}_{in}, \mathbf{k}_{out}) \quad \mathbf{k}_1 \leq \mathbf{k}_2}{\vdash \mathtt{x} := E \ : (\mathbf{k}_1, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{(Asg)}$$

$$\frac{\vdash E : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \quad \vdash C : (\mathbf{k}, \mathbf{k}, \mathbf{k}_{out}) \quad \mathbf{1} \leq \mathbf{k} \leq \mathbf{k}_{out}}{\vdash \mathtt{while}(E)\{C\} \ : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{(Wh)}$$

$$\frac{\vdash E : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out}) \ \vdash E' : (\mathbf{k}_{out}, \mathbf{k}_{in}, \mathbf{k}_{out}) \quad \mathbf{k} < \mathbf{k}_{in} \leq \mathbf{k}_{out}}{\vdash \phi(E \upharpoonright E') : (\mathbf{k}, \mathbf{k}_{in}, \mathbf{k}_{out})} \text{(Orc)}$$

$$\vdots$$

Illustrating example

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

```
y = x;
z = false;
while(x^1 >= 0){
   if(φ(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 $(\mathbf{1}, \mathbf{1}, \mathbf{1})$.
- 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 $[\![X]\!]$ 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)

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

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

Type system

The extended type system just consists of two layers:

- SAFE procedures (using the HKMP[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

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

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

First characterizations of BFF

Characterizations without external lambda-closure:

Theorem [Hainry-Kapron-Marion-Péchoux [FoSSaCS22]]

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

Surprisingly, the internal lambda-abstraction is not required for completeness.

$\rightarrow$ 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:

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

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

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

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** (decidable type inference in polytime) $\Rightarrow$ it can be automated.

Open issues

- **expressive power** (capture more false negatives),
- **extension to Finite Length Revision** (harder, WIP using declassification).

Thank you for your attention !