Implicit Complexity and Formal Proofs

H. Férée & S. Hym & M. Mayero & **J.-Y. Moyen** & D. Nowak
Jean-Yves.Moyen@lipn.univ-paris13.fr

School of Computing, University of Kent, United Kingdom
CRIStAL, CNRS & University of Lille, France
LIPN, University of Paris 13, France

October 10-11 2018

What is this talk about?

- Formal proof in Coq of the P-criterion: "PPO + QI $\equiv$ PTIME".
- Tool, integrated with Coq, to help show that a program satisfies the P-criterion.
- Some tricks and learning that could be useful for further similar projects.

Motivation: Cryptography

The need for formal proofs

- Cryptography needs to prove security of its protocols.
- “Paper” proofs are often discovered wrong a few year later.
  Formal proofs are needed.

The need for formal proofs

- Cryptography needs to prove security of its protocols.
- "Paper" proofs are often discovered wrong a few year later. Formal proofs are needed.
- Security:
  Adversary gives me 2 plain texts. I randomly chose one and encrypt it. Adversary guesses which one.
  Protocol is secure if Adversary guesses correctly with probability 0.5

The need for formal proofs

- Cryptography needs to prove security of its protocols.
- "Paper" proofs are often discovered wrong a few year later. Formal proofs are needed.
- Security:
  Adversary gives me 2 plain texts. I randomly chose one and encrypt it. Adversary guesses which one.
  Protocol is secure if Adversary guesses correctly with probability 0.5
- Against all Adversaries? No! PTIME Adversaries are enough.
- Formal proof of security needs to quantify "For all adversaries running in polynomial time".

The need for ICC

"For all adversaries running in polynomial time"

- Formalise polynomial Turing Machines?

The need for ICC

"For all adversaries running in polynomial time"

- Formalise polynomial Turing Machines?
  - Not that uniform definition (number of tapes, heads, ...)
  - Need to handle clocks and bounds.
  - Extremely hard to actually program an adversary (or the algorithm).
  - ICC provides nice, machine-free, somewhat expressive characterisations of PTIME.

A long term Work

- Formalisation of Bellantoni&Cook system [Heraud&Nowak, 2011].

A long term Work

- Formalisation of Bellantoni&Cook system [Heraud&Nowak, 2011].
  - Still very hard to program. . . (Binary addition takes 3 pages in H. Rose's classical book on Primitive recursion).

A long term Work

- Formalisation of Bellantoni&Cook system [Heraud&Nowak, 2011].
  - Still very hard to program. . . (Binary addition takes 3 pages in H. Rose's classical book on Primitive recursion).
- Shonan meeting organised by D. Nowak (2013).
  - Quasi-Interpretations offer a good mix between expressivity and simplicity.

A long term Work

- Formalisation of Bellantoni&Cook system [Heraud&Nowak, 2011].
  - Still very hard to program... (Binary addition takes 3 pages in H. Rose's classical book on Primitive recursion).
- Shonan meeting organised by D. Nowak (2013).
  - Quasi-Interpretations offer a good mix between expressivity and simplicity.
- Presentation of a preliminary version of this work [Dice 2016].
  - Only soundness was proved.
  - No really interesting example, poor tool (little automation).

A long term Work

- Formalisation of Bellantoni&Cook system [Heraud&Nowak, 2011].
  - Still very hard to program... (Binary addition takes 3 pages in H. Rose's classical book on Primitive recursion).
- Shonan meeting organised by D. Nowak (2013).
  - Quasi-Interpretations offer a good mix between expressivity and simplicity.
- Presentation of a preliminary version of this work [Dice 2016].
  - Only soundness was proved.
  - No really interesting example, poor tool (little automation).
- Conference paper in the formal proofs community [CPP 2018].

The tool

Motivating example

Modular exponentiation: $\texttt{expmod}(x, y, m) = x^y[m]$

- Central in many modern cryptographic algorithms.
- Polynomiality is not trivial and needs careful use of the modulo.
- 30 function symbols, 110 rules.

Modular exponentiation: the good parts

- Interface module, writing TRS is (arguably) easy.
- Rules are mostly obtained by extraction from (proven) Coq code. Very little adhoc code.
- Code is proven semantically correct.
- Termination by PPO is entirely automatic.
- Finding QIs is incremental, guided, and supported by the proof assistant (use of full Arithmetic to prove inequalities between polynomials). Most rules are solved by a couple of tactics leaving only the burden of finding the QI to the user.

Modular exponentiation: the bad parts

- Need to add some "clock" arguments, with no computational use, just to ensure bounds.
- These arguments are only needed when the function is re-used. Without them, a QI can be found but is too big for re-use.
- Thus, compositionality is not really achieved.

Demo time!

The proof

From Paper to Formal: ICC

Usually, in ICC:

- Soundness proofs are hard (need to carefully craft a bound on the programs). Thus somewhat detailed.
- Completeness proofs are easy (need to show that a previous system is contained in the new one). Thus often extremely short.

From Paper to Formal: ICC

Usually, in ICC:

- Soundness proofs are hard (need to carefully craft a bound on the programs). Thus somewhat detailed.
- Completeness proofs are easy (need to show that a previous system is contained in the new one). Thus often extremely short.

Here, on paper, completeness is usually: "*It is easy to see that each function in B is ordered by MPO'. Then Lemma 4.1 of [2] provides a quasi-interpretation*" (LPAR'01).

- Obviously far from Formal proof.
- Actually not really correct.
- Still easier than proof of soundness.

From Paper to Formal: the long way

- Original proof (LPAR'01): around 3 pages of proof (reduction to LMPO).

From Paper to Formal: the long way

- Original proof (LPAR'01): around 3 pages of proof (reduction to LMPO).
- Independent proof (TCS'11): around 6 (soundness) + 2 (completeness) pages of proof (reduction from parallel register machines).

From Paper to Formal: the long way

- Original proof (LPAR'01): around 3 pages of proof (reduction to LMPO).
- Independent proof (TCS'11): around 6 (soundness) + 2 (completeness) pages of proof (reduction from parallel register machines).
- With improved semantics (MSCS'12): around 4 pages of proof (soundness), many details skipped (not the main goal).

From Paper to Formal: the long way

- Original proof (LPAR'01): around 3 pages of proof (reduction to LMPO).
- Independent proof (TCS'11): around 6 (soundness) + 2 (completeness) pages of proof (reduction from parallel register machines).
- With improved semantics (MSCS'12): around 4 pages of proof (soundness), many details skipped (not the main goal).
- "Master 2 level" detailed proof: around 35 (soundness) + 3 (completeness) pages of proof (several "obvious" results still have no proof).

From Paper to Formal: the long way

- Original proof (LPAR'01): around 3 pages of proof (reduction to LMPO).
- Independent proof (TCS'11): around 6 (soundness) + 2 (completeness) pages of proof (reduction from parallel register machines).
- With improved semantics (MSCS'12): around 4 pages of proof (soundness), many details skipped (not the main goal).
- "Master 2 level" detailed proof: around 35 (soundness) + 3 (completeness) pages of proof (several "obvious" results still have no proof).
- Coq proof: around 16000 (non-empty) lines of code (completeness: around 3000 lines)...

From Paper to Formal: the long way

- Original proof (LPAR'01): around 3 pages of proof (reduction to LMPO).
- Independent proof (TCS'11): around 6 (soundness) + 2 (completeness) pages of proof (reduction from parallel register machines).
- With improved semantics (MSCS'12): around 4 pages of proof (soundness), many details skipped (not the main goal).
- "Master 2 level" detailed proof: around 35 (soundness) + 3 (completeness) pages of proof (several "obvious" results still have no proof). (less than 6000 lines of LaTeX)
- Coq proof: around 16000 (non-empty) lines of code (completeness: around 3000 lines)...

The formal result

- Soundness:

- Completeness:

The formal result

- Soundness:
  - bound on the size of *the full derivation tree of the evaluation of a term, including all caches at all levels*.
  - QIs are not bounded *a priori* but the bound depends on the QI (hence polynomial bound with polynomial QI).
- Completeness:

The formal result

- Soundness:
  - bound on the size of *the full derivation tree of the evaluation of a term, including all caches at all levels.*
  - QIs are not bounded *a priori* but the bound depends on the QI (hence polynomial bound with polynomial QI).
- Completeness:
  - Reduction from the previous proof of BC.
  - Every BC program can be translated into a TRS that satisfies the P-criterion.
  - Still missing a proof of (semantic) correctness of the translation.

Some difficulties in the proof

CBV

$$\frac{t_i \downarrow \qquad v_i}{\mathbf{c}(t_1, \ldots, t_n) \downarrow \qquad \mathbf{c}(v_1, \ldots, v_n)}\ (\text{Constructor})$$

$$\frac{\begin{array}{c}\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \\ t_i \downarrow \qquad v_i \qquad \mathtt{f}(v_1, \ldots, v_n) \downarrow \qquad v\end{array}}{\mathtt{f}(t_1, \ldots, t_n) \downarrow \qquad v}\ (\text{Split})$$

$$\frac{\begin{array}{c}\mathtt{f}(p_1, \ldots, p_n) \to r \in \mathcal{E} \\ \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \qquad r\sigma \downarrow \qquad v\end{array}}{\mathtt{f}(v_1, \ldots, v_n) \downarrow \qquad v}\ (\text{Update})$$

CBV with cache (memoisation)

$$\frac{\langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, v_i\rangle}{\langle C_0, \mathbf{c}(t_1, \ldots, t_n)\rangle \Downarrow \langle C_n, \mathbf{c}(v_1, \ldots, v_n)\rangle}\ \text{(Constructor)}$$

$$\frac{\begin{array}{c}\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \\ \langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, v_i\rangle \quad \langle C_n, \mathtt{f}(v_1, \ldots, v_n)\rangle \Downarrow \langle C, v\rangle\end{array}}{\langle C_0, \mathtt{f}(t_1, \ldots, t_n)\rangle \Downarrow \langle C, v\rangle}\ \text{(Split)}$$

$$\frac{(\mathtt{f}(v_1, \ldots, v_n), v) \in C}{\langle C, \mathtt{f}(v_1, \ldots, v_n)\rangle \Downarrow \langle C, v\rangle}\ \text{(Read)}$$

$$\frac{\begin{array}{c}\nexists u/\ (\mathtt{f}(v_1, \ldots, v_n), u) \in C \quad \mathtt{f}(p_1, \ldots, p_n) \to r \in \mathcal{E} \\ \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad \langle C, r\sigma\rangle \Downarrow \langle D, v\rangle\end{array}}{\langle C, \mathtt{f}(v_1, \ldots, v_n)\rangle \Downarrow \left\langle D \bigcup \{(\mathtt{f}(v_1, \ldots, v_n), v)\}, v\right\rangle}\ \text{(Update)}$$

How to represent a derivation proof tree?

Inductive type, but "side" conditions are hard to enforce directly and are checked *a posteriori* with a `well_formed` property.

How to represent a derivation proof tree?

Inductive type, but "side" conditions are hard to enforce directly and are checked *a posteriori* with a `well_formed` property.

$$\frac{t_i \downarrow v_i}{\mathbf{c}(t_1, \ldots, t_n) \downarrow \mathbf{c}(v_1, \ldots, v_n)} \text{(Constructor)}$$

How to represent a derivation proof tree?

Inductive type, but “side” conditions are hard to enforce directly and are checked *a posteriori* with a `well_formed` property.

$$\frac{t_i \downarrow v_i}{\mathbf{c}(t_1, \ldots, t_n) \downarrow \mathbf{c}(v_1, \ldots, v_n)} \text{ (Constructor)}$$

Is represented by the constructor

```
| cbv_constr : list cbv → term → value → cbv
```

How to represent a derivation proof tree?

Inductive type, but "side" conditions are hard to enforce directly and are checked *a posteriori* with a `well_formed` property.

$$\frac{t_i \downarrow v_i}{\mathbf{c}(t_1, \ldots, t_n) \downarrow \mathbf{c}(v_1, \ldots, v_n)} \text{ (Constructor)}$$

Is represented by the constructor

```
| cbv_constr : list cbv → term → value → cbv
```

corresponding to the "rule"

$$\texttt{cbv\_constr}\ [\ldots \pi_{\texttt{i}} \ldots]\ \texttt{t}\ \texttt{v} = \frac{\ldots \pi_i \ldots}{t \downarrow v} \text{ (Constructor)}$$

Semantic Derivations must be well-formed

$$\texttt{cbv\_constr}\ [\ldots \pi_{\texttt{i}} \ldots]\ \texttt{t}\ \texttt{v} = \frac{\ldots \pi_i \ldots}{t \downarrow v}\,(\text{Constructor})$$

Semantic Derivations must be well-formed

$$\texttt{cbv\_constr}\ [\ldots \pi_{\mathtt{i}} \ldots]\ \texttt{t}\ \texttt{v} = \frac{\ldots \pi_i \ldots}{t \downarrow v}\ (\text{Constructor})$$

Additional predicate:

$$|\ \texttt{cbv\_constr}\ \pi\texttt{l}\ (\texttt{capply c lt})\ (\texttt{c\_capply c}'\ \texttt{lv})\ \Rightarrow$$

$$\texttt{andl}\ (\texttt{map wf}\ \pi\texttt{l}) \wedge\ \texttt{c} = \texttt{c}' \wedge$$

$$\texttt{lt} = \texttt{map proj\_left}\ \pi\texttt{l}\ \wedge\ \texttt{lv} = \texttt{map proj\_right}\ \pi\texttt{l}$$

Semantic Derivations must be well-formed

$$\texttt{cbv\_constr}\ [\ldots \pi_{\texttt{i}} \ldots]\ \texttt{t v} = \frac{\ldots \pi_i \ldots}{t \downarrow v}\ (\text{Constructor})$$

Additional predicate:

$$|\ \texttt{cbv\_constr}\ \pi\texttt{l}\ (\texttt{capply c lt})\ (\texttt{c\_capply c}'\ \texttt{lv})\ \Rightarrow$$

$$\texttt{andl}\ (\texttt{map wf}\ \pi\texttt{l}) \wedge\ \texttt{c} = \texttt{c}' \wedge$$

$$\texttt{lt} = \texttt{map proj\_left}\ \pi\texttt{l}\ \wedge\ \texttt{lv} = \texttt{map proj\_right}\ \pi\texttt{l}$$

All theorems look like:

```
∀...
let pi := (cbv_update ...) in
wf pi →
...
```

Semantic Derivations must be well-formed

$$\texttt{cbv\_constr}\ [\ldots \pi_{\mathtt{i}} \ldots]\ \texttt{t v} = \frac{\ldots \pi_i \ldots}{t \downarrow v}\ \text{(Constructor)}$$

Additional predicate:

$$|\ \texttt{cbv\_constr}\ \pi\texttt{l}\ (\texttt{capply c lt})\ (\texttt{c\_capply c}'\ \texttt{lv})\ \Rightarrow$$

$$\texttt{andl}\ (\texttt{map wf}\ \pi\texttt{l}) \wedge\ \texttt{c} = \texttt{c}' \wedge$$

$$\texttt{lt} = \texttt{map proj\_left}\ \pi\texttt{l}\ \wedge\ \texttt{lv} = \texttt{map proj\_right}\ \pi\texttt{l}$$

All theorems look like:

```
∀...
let pi := (cbv_update ...) in
wf pi →
...
```

(Similar to Proof Structure vs Proof Nets)

Handling the existential quantifier

$$\frac{t_i \downarrow v_i \quad \mathtt{f}(v_1, \ldots, v_n) \downarrow v}{\mathtt{f}(t_1, \ldots, t_n) \downarrow v} \text{(Split)}$$

Handling the existential quantifier

$$\frac{\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \quad t_i \downarrow v_i \quad \mathtt{f}(v_1, \ldots, v_n) \downarrow v}{\mathtt{f}(t_1, \ldots, t_n) \downarrow v} \text{(Split)}$$

In effect, this means that (Split) must be followed by (Functions).

Handling the existential quantifier

$$\frac{\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \quad t_i \downarrow v_i \quad \mathtt{f}(v_1, \ldots, v_n) \downarrow v}{\mathtt{f}(t_1, \ldots, t_n) \downarrow v} \text{(Split)}$$

In effect, this means that (Split) must be followed by (Functions).
Adding the well-formed check:

$$\mid \mathtt{cbv\_split}\ \mathtt{l}\ (\mathtt{cbv\_function}\ \ldots)(\mathtt{fapply}\ \mathtt{f}'\ \mathtt{lt})\ \mathtt{v}' \Rightarrow \ \ldots$$

Handling the existential quantifier

$$\frac{\exists j, t_j \notin \mathcal{T}(\mathcal{C}) \quad t_i \downarrow v_i \quad \texttt{f}(v_1, \ldots, v_n) \downarrow v}{\texttt{f}(t_1, \ldots, t_n) \downarrow v} \text{(Split)}$$

In effect, this means that (Split) must be followed by (Functions).
Adding the well-formed check:

$$|\ \texttt{cbv\_split l (cbv\_function } \ldots)(\texttt{fapply f}'\ \texttt{lt})\ \texttt{v}' \Rightarrow \ \ldots$$

This corresponds to defining the semantics with a "double rule"

$$\cfrac{t_i \downarrow v_i \qquad \cfrac{\texttt{f}(p_1, \ldots, p_n) \to r \in \mathcal{E} \quad \sigma \in \mathfrak{S} \quad p_i\sigma = v_i \quad r\sigma \downarrow v}{\texttt{f}(v_1, \ldots, v_n) \downarrow v} \text{(F)}}{\texttt{f}(t_1, \ldots, t_n) \downarrow v} \text{(S)}$$

Big Step Induction

- Paper proofs rely on “call trees”, or a $\rightsquigarrow$ relation, which amounts to only keeping the (Update) and (Read) rules (the rest is bookkeeping for finding the leftmost outermost redex).
- Building these in Coq would be tedious (plus need an extra layer of correction lemmas).

Big Step Induction

- Paper proofs rely on “call trees”, or a $\rightsquigarrow$ relation, which amounts to only keeping the (Update) and (Read) rules (the rest is bookkeeping for finding the leftmost outermost redex).
- Building these in Coq would be tedious (plus need an extra layer of correction lemmas).
- Instead, we prove a “big step” induction lemma:

$$\left[\forall \widetilde{J}, \left((\forall \widetilde{H}, \widetilde{J} \rightsquigarrow \widetilde{H} \Rightarrow P(\widetilde{H})) \Rightarrow P(\widetilde{J})\right)\right] \Rightarrow \forall \widetilde{I}, P(\widetilde{I})$$

```
Lemma cbv_big_induction :
∀ (P : cbv → Prop),
  (∀ J,
    (∀ H, H ∈ (first_activations J) → P H) → P J) →
∀ I, P I.
```

Replacing automatic induction

Coq automatically generates structural induction principle for inductive types, but these were often not suitable for our needs.

Replacing automatic induction

Coq automatically generates structural induction principle for inductive types, but these were often not suitable for our needs.

```
Inductive Term: Type := ...
| capply: constructor → list term → term
```

Replacing automatic induction

Coq automatically generates structural induction principle for inductive types, but these were often not suitable for our needs.

```
Inductive Term: Type := ...
| capply: constructor → list term → term
```

Coq's induction:

```
∀ P, ...
(∀ c lt, P (capply c lt)) →
∀ t, P t
```

Quantifying on **all** possible subterms is way too much, it is sufficient to quantify on the "good" ones (that validate the predicate):

Replacing automatic induction

Coq automatically generates structural induction principle for inductive types, but these were often not suitable for our needs.

```
Inductive Term: Type := ...
| capply: constructor → list term → term
```

Coq's induction:

```
∀ P, ...
(∀ c lt, P (capply c lt)) →
∀ t, P t
```

Quantifying on **all** possible subterms is way too much, it is sufficient to quantify on the "good" ones (that validate the predicate):

```
∀ P, ...
(∀ c lt, (∀ t, t ∈ lt → P t) → P (capply c lt)) →
∀ t, P t
```

Completeness: from PR to TRS

- The class BC, both in paper and in previous formal proof, is defined with a "Primitive Recursive" syntax:

$$\mathtt{REC}(\mathtt{PROJ}_{0,1,1},\ \mathtt{COMP}(\mathtt{SUCC}_{\mathtt{ff}},\ [\,],\ [\mathtt{PROJ}_{1,2,3}]),$$

$$\mathtt{COMP}(\mathtt{SUCC}_{\mathtt{tt}},\ [\,],\ [\mathtt{PROJ}_{1,2,3}]))$$

Completeness: from PR to TRS

- The class BC, both in paper and in previous formal proof, is defined with a "Primitive Recursive" syntax:

$$\mathtt{REC}(\mathtt{PROJ}_{0,1,1},\ \mathtt{COMP}(\mathtt{SUCC}_{\mathtt{ff}},\ [\,],\ [\mathtt{PROJ}_{1,2,3}]),$$

$$\mathtt{COMP}(\mathtt{SUCC}_{\mathtt{tt}},\ [\,],\ [\mathtt{PROJ}_{1,2,3}]))$$

- We need to turn that into a Term Rewriting System (7 function symbols and 10 rules).
- Main (Coq) difficulty: create new function symbols. Using integers is nice but need to keep a global "first available integer".
  - Solution: use a state monad for translation.

Completeness: handling induction

- Hard case in inductive proofs is composition, because it's unbounded composition (need to handle list of subterms).
- We need delicate lemmas to ensure that we correctly handle the premises.

```
Proposition BC_to_TRS_func_bounds bc st f:
let trs := snd (BC_to_TRS bc st) in
f ∈ all_lhs_funcs trs →
trs.(first) ≤ f ≤ trs.(last).
```

Conclusion

Conclusion

- Formal proof is a lot of work.
  - Filling in many, many small gaps.
  - Stating and proving some “obvious but hard to prove” lemmas.
  - Correcting errors in the proof.
- Hopefully, ideas or some proofs can be reused by others.
- Tool with a good level of automation.

Questions? . . . or Cake

(hopefully inclusive)