Formal Proof of Polynomial-Time Complexity with Quasi-Interpretations*

Hugo Férée
School of Computing
University of Kent
United Kingdom

Samuel Hym
CRIStAL
University of Lille
France

Micaela Mayero
LIPN
University of Paris 13
France

Jean-Yves Moyen
Department of Computer Science
University of Copenhagen
Denmark

David Nowak
CRIStAL
CNRS & University of Lille
France

Abstract

We present a Coq library that allows for readily proving that a function is computable in polynomial time. It is based on quasi-interpretations that, in combination with termination ordering, provide a characterisation of the class FP of functions computable in polynomial time. At the heart of this formalisation is a proof of soundness and extensional completeness. Compared to the original paper proof, we had to fill a lot of not so trivial details that were left to the reader and fix a few glitches. To demonstrate the usability of our library, we apply it to the modular exponentiation.

CCS Concepts • **General and reference** → *Verification*; • **Theory of computation** → *Complexity classes*; *Logic and verification*; *Program verification*; Complexity theory and logic; Semantics and reasoning;

Keywords Coq formal proof, implicit complexity, polynomial time

ACM Reference Format:
Hugo Férée, Samuel Hym, Micaela Mayero, Jean-Yves Moyen, and David Nowak. 2018. Formal Proof of Polynomial-Time Complexity with Quasi-Interpretations. In *CPP '18: CPP '18: Certified Proofs and Programs , January 8–9, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3167097

*Part of the work presented in this paper was done while the first author was affiliated with CRIStAL, University of Lille, the other part was supported by EPSRC Grant JEP/N028759/1, UK. The fourth author was supported by the Marie Skłodowska–Curie action "Walgo", program H2020-MSCA-IF-2014, number 655222.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CPP'18, January 8–9, 2018, Los Angeles, CA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5586-5/18/01…$15.00
https://doi.org/10.1145/3167097

1 Introduction

Most of the Internet protocols used nowadays rely on cryptographic primitives that are used to ensure security properties for communication such as secrecy, authenticity, or integrity. The security of most of those primitives rely on a particular mathematical problem that is believed not to be solvable by a feasible adversary. Cryptographers follow Cobham's thesis that asserts that being feasible is the same as being computable in polynomial time [10]. More precisely, they assume that the adversary is computable in probabilistic polynomial time (PPT), i.e. it is executable on a Turing machine extended with a read-only tape that has been filled with random bits, and working in worst-case polynomial time [17, 23]. It is unfortunately not uncommon for cryptographic primitives to be published with a mathematical proof of security in a top-level peer-reviewed conference, followed by an attack a few months or years later [7]. Cryptography is a field that is clearly in dire need of formal proofs and that has been addressed since the end of the previous decade [1, 5, 6, 12, 13, 22, 27–29, 33]. However some of those approaches were not implemented. And in those that were implemented, the complexity of the adversary was either not taken into account or was entirely manual such as with the cost monad used in [6].

A way to take complexity issues into account in a formal proof of security would be to formalise a precise execution model (e.g., a Turing machine) and to explicitly count the number of steps necessary for the execution of the adversary. Not only would it be tedious, but it would provide results depending on the particular execution model in use whereas we are here interested in the complexity class independently of the execution model. Implicit computational complexity (ICC) is a more convenient approach for our purpose: it relates programming languages with complexity classes

without relying on a specific execution model nor an explicit counting of execution steps.

In [10], Cobham gave a characterisation of FP in terms of bounded recursion: the primitive recursion scheme is further restricted by requiring that calls are bounded. This is a characterisation of FP in the sense that it defines a set of programs, BR, which is both sound and extensionally complete for FP: any program in BR is computable in polynomial time; and any function in FP can be computed by a program in BR. This characterisation is, however, somewhat inelegant in the sense that it requires explicit bounds to be built. Moreover, it is based on primitive recursion which is known to be of very limited expressive power [11].

In [8], Bellantoni and Cook got rid of the explicit bounds by introducing the tiering mechanism: some values (results of recursion) are tagged as being "safe" and may not be used to control another recursion. This prevents too many nesting of recursion and thus prevents exponential time computation. The class of programs, BC, that they define, still characterises FP in the same sense (soundness and extensional completeness).

However, BC suffers from the same expressivity problem as BR or the primitive recursion. It is extremely difficult for programmers to write a program that respect the syntactical constraints and thus it makes any practical use of the characterisation clumsy at best, as experienced while formalising it in [18].

Since the breakthrough of Bellantoni and Cook, ICC has focused on improving the expressivity of its characterisations of FP. A first step was to follow the road that was taken for primitive recursion. It is known in the Term Rewriting community that the Multiset Path Ordering (MPO) characterises the Primitive Recursive functions [19]. This characterisation enjoys the soundness and extensional completeness properties; and notably extends the expressivity of the language (e.g. it is possible to compute the maximum of two unary integers x and y in time $\min(x, y)$). By introducing the tiering discipline into MPO, Marion defined LMPO [24] which characterises FP and is more expressive than BC as it allows more complex forms of recursion.

Unfortunately, LMPO still had huge expressivity limitations. Since it mixes the termination analysis (via MPO) and the bound analysis (via the tiering mechanism), some programs require extra arguments to be added just to get the machinery going. Typically, a binary `max` function may need a third argument just to be used as a bound during the proofs. Even worse, the need for such extra (and counter-intuitive) arguments only arise when the function is further reused as part of a larger code, thus breaking any hopes of true compositionality of the code.

This problem has been partially solved by splitting the termination and bounds analysis and the introduction of *Quasi-Interpretations* (QIs) [9]. The termination (and Primitive Recursive bound) is still guaranteed by MPO, or, rather, a simpler but equivalent variant called *Product Path Ordering* (PPO); the bounds are externalised through QIs. PPO alone only characterises Primitive Recursion. QIs alone do not ensure termination. The composition of both characterises FP. We formalise here a detailed version of the proof as presented in [4].

Compared to Cobham's system, QIs are an external bound rather than an internal one. They do not need to be written within the system and can thus use some external additional proof power for computing bounds (i.e. the full Arithmetic). The characterisation is thus of relative simple use and expressive enough to allow the certification of program computing, for example, the modular exponentiation.

Of course, QIs are still lacking in expressivity and writing certified programs still requires a lot of work. ICC has since made some progress in that direction. However, having an intensionally complete characterisation (that is, one that admits all the programs computing in polynomial time, not just one program for each function) is impossible [26]. We believe that QIs offer a good enough compromise between expressivity and easiness of writing the formal proofs (which are already quite long). Moreover, the techniques developed in this article can be reused to write formal proofs of more advanced ICC systems, should the need for more expressivity arise.

Contributions Our main contribution is a Coq library that allows for readily proving that a function is computable in polynomial time. At the heart of this library is the formalisation of Quasi-Interpretations along with the associated soundness and completeness theorems.

Not surprisingly, the formalisation of the soundness theorem required us to deal with a lot of details that were left implicit in the paper proof [4, 9], and to make some slight modifications in order to make the paper proof amenable to a formal proof.

On the other hand, the formal proof of the completeness theorem was done from scratch. We proceed by reduction from the characterisation of FP by Bellantoni and Cook [8, 18].

Then, by providing an interface together with various decision procedures and tactics to help proving the hypotheses of the soundness theorem on a given program, we turned this theoretical results into a verified tool for writing programs as term-rewriting systems and ensuring that they implement polynomial-time computable functions. Altogether, this provides an environment to prove running time bounds as well as other formal properties (like functional correctness) about programs, all in the Coq proof environment.

In order to emphasise the usability of this machinery, we apply it successfully to a small yet non-trivial program. For this, we chose the modular exponentiation as it is broadly

used in cryptographic primitives — one of our initial motivations — and is a good example of program where the polynomial running-time bound cannot be proved by a straightforward argument.

Related work A formal proof that BC is a characterisation of FP was given in [18] and integrated in frameworks [6] and [27] for formal security proofs in cryptography. However, because of the limited expressivity of BC, the encoding of adversaries is difficult. In particular, BC does not allow for transmitting the carry bit to a recursive call. This thus makes the encoding of a simple function like binary addition tricky. A solution for binary addition without using a carry bit can however be found in pages 127–129 of [30] but it involves 21 intermediate definitions.

Researchers in the field of ICC have proposed more expressive characterisations of FP suitable for formal proof in cryptography. There is in particular SLR by Hofmann [21], that was later extended to OSLR by Mitchell et al. [25], and then to CSLR by Zhang [33]. The latter was used in [29] to model security proofs of cryptographic primitives in a way that takes into account complexity issues. But, since SLR and its successors rely on quite involved mathematics that have not yet found their way into proof assistants' libraries, it would be heavy work to formalise them into a proof assistant.

More recently, a type system has been defined to analyse the complexity of higher-order stateful programs, and was used to prove that the constructed adversary for the Goldreich-Levin theorem is polytime [3]. However, it has not been implemented.

Such tools have been developed to analyse the complexity of programs [2, 16, 20, 31, 32]. Such a tool however consists of thousands of lines of code that are not proved correct.

Outline After a brief overview of quasi-interpretations in Section 2, we present the interface of our library in Section 3. The next two sections present two major and novel aspects of our formal proof: Section 4 is about the proof of completeness and Section 5 is about automation. We then demonstrate the usability of our library in Section 6 with the example of modular exponentiation. Finally, we conclude in section 7.

2 Quasi-interpretations

In this section, we briefly present Quasi-Interpretations. Any reader interested in detailed definitions and proofs should refer to [4] or to our fully detailed technical report [15].

We consider first-order term rewriting systems (TRS) with disjoint sets $\mathcal{X}$, $\mathcal{F}$, $\mathcal{C}$ resp. of variables, function symbols and constructor symbols. A program will then be defined by a finite set of rewriting rules (or equations) of the form $\mathtt{f}(p_1, \ldots, p_n) \to t$ where $f \in \mathcal{F}$, p_i is a pattern (a term with no function symbol) and t is a term whose free variables appear in the (p_i).

We equip such programs with a call-by-value semantics with memoisation. In other words, the execution of a program uses and updates a cache which maps every already evaluated activations (i.e. terms of the form $f\ v_1\ \ldots\ v_n$ where v_i is a value) with its value. We will not give the details of this semantics here.

The first part of the QI technique consists in ensuring that the program is ordered according to an ordering from the PPO family. In order to define an order on terms, we first need to choose one on function symbols and constructors.

Definition 2.1 (Precedence)**.** A *precedence* $\preceq_{\mathcal{F}}$ is a preorder over $\mathcal{F} \bigcup \mathcal{C}$ which verifies the following properties:

- (*Compatible*) Function calls are done toward smaller functions: for each equation $\mathtt{f}(p_1, \ldots, p_n) \to r$ and each function symbol g appearing in r, $\mathtt{g} \preceq_{\mathcal{F}} \mathtt{f}$.
- (*Separating*) Constructors are strictly smaller than any function symbol: for each $\mathbf{c} \in \mathcal{C}$, $\mathtt{f} \in \mathcal{F}$, $\mathbf{c} \prec_{\mathcal{F}} \mathtt{f}$.
- (*Strict*) Constructors are mutually incomparable: for each distinct constructors $\mathbf{c} \neq \mathbf{d}$, $\mathbf{c}$ and $\mathbf{d}$ are incomparable.

We denote by $\approx_{\mathcal{F}}$ the associated equivalence: $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \Leftrightarrow \mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$ and $\mathtt{g} \preceq_{\mathcal{F}} \mathtt{f}$.

The following extension of an order on the elements of a set S to the tuples of elements of S is the last definition we need to define a Product Path Ordering on the terms of our program.

Definition 2.2 (Product Extension)**.** Let $\prec$ be a binary relation over a set S and $\preceq$ be its reflexive closure. Its *product extension* is the relation $\prec^p$ over tuples of elements of S such that: $(m_1, \ldots, m_k) \prec^p (n_1, \ldots, n_k)$ if and only if (i) $\forall i, m_i \preceq n_i$ and (ii) $\exists j$ such that $m_j \prec n_j$.

Definition 2.3 (PPO)**.** The strict Product Path Ordering — denoted by $\prec_{ppo}$ and called PPO from now on — is the relation defined in Figure 1. The Product Path Ordering $\preceq_{ppo}$ is the reflexive closure of $\prec_{ppo}$.

Definition 2.4 (Ordered by PPO)**.** An equation $l \to r$ is *strictly decreasing* if we have $r \prec_{ppo} l$. A program is *ordered by PPO* if each equation is strictly decreasing.

Note that a program ordered by PPO always terminates (see [14]). For this, we have defined the notion of precedence (Definition 2.1) on function symbols, which is related to the notion of rank.

Definition 2.5 (Rank)**.** A precedence naturally induces a notion of *rank* of function symbols defined as:

$$\mathrm{rk}(\mathtt{f}) = \max_{\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}} \mathrm{rk}(\mathtt{g}) + 1$$

Especially, $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$ iff $\mathrm{rk}(\mathtt{g}) < \mathrm{rk}(\mathtt{f})$.

We now need a second and last criterion on our program to guarantee that it computes a polynomial time computable

$$\frac{s \preceq_{ppo} t_i}{s \prec_{ppo} \mathbf{c}(\ldots, t_i, \ldots)}\ \text{(Subterm Cons)} \qquad \frac{s \preceq_{ppo} t_i}{s \prec_{ppo} \mathsf{f}(\ldots, t_i, \ldots)}\ \text{(Subterm Fun)} \qquad \frac{\forall i, s_i \prec_{ppo} \mathsf{f}(t_1, \ldots, t_n)}{\mathbf{c}(s_1, \ldots, s_m) \prec_{ppo} \mathsf{f}(t_1, \ldots, t_n)}\ \text{(Cons Strict)}$$

$$\frac{\mathsf{g} \prec_{\mathcal{F}} \mathsf{f} \quad \forall i, s_i \prec_{ppo} \mathsf{f}(t_1, \ldots, t_n)}{\mathsf{g}(s_1, \ldots, s_m) \prec_{ppo} \mathsf{f}(t_1, \ldots, t_n)}\ \text{(Fun Strict)} \qquad \frac{\mathsf{f} \approx_{\mathcal{F}} \mathsf{g} \qquad (s_1, \ldots, s_n) \prec^{p}_{ppo} (t_1, \ldots, t_n)}{\mathsf{g}(s_1, \ldots, s_n) \prec_{ppo} \mathsf{f}(t_1, \ldots, t_n)}\ \text{(Fun Equiv)}$$

Figure 1. Definition of $\prec_{ppo}$

function, or more precisely that it has a reduction proof of polynomial size in the memoised call-by-value semantics.

Definition 2.6 (Assignment)**.** An *assignment* of a constructor $\mathbf{c}$ of arity n is a function $(\!|\mathbf{c}|\!) : \mathbb{N}^n \to \mathbb{N}$ which has the *(Additivity)* property, that is, which is of the form:

$(\!|\mathbf{c}|\!)(X_1, \ldots, X_n) = \sum_{i=1}^{n} X_i + c_{\mathbf{c}}$ with $c_{\mathbf{c}} \geq 1$ a constant depending on $\mathbf{c}$. An *assignment* of a function symbol f whose arity is n is a function $(\!|\mathsf{f}|\!) : \mathbb{N}^n \to \mathbb{N}$ with the following properties:

- (*Subterm*) $X_i \leq (\!|\mathsf{f}|\!)(X_1, \ldots, X_n)$ for all $1 \leq i \leq n$.
- (*Weak Monotonicity*) $(\!|\mathsf{f}|\!)$ is increasing with respect to each variable:

$$\forall 1 \leq i \leq n, X_i \leq Y_i \Rightarrow (\!|\mathsf{f}|\!)(X_1, \ldots, X_n) \leq (\!|\mathsf{f}|\!)(Y_1, \ldots, Y_n)$$

Once the assignment function has been defined over functions and constructors, it can be extended to all terms by composition: each program variable is interpreted as a function variable, and the interpretation of a term is then an integer function with as many variables as the term has.

Definition 2.7 (Quasi-interpretation)**.** An assignment $(\!|.|\!)$ is a quasi-interpretation for a program if for each equation $l \to r$ of the program and each substitution σ mapping variables to values and whose domain contains the variables of l, we have $(\!|r\sigma|\!) \leq (\!|l\sigma|\!)$.

These definitions are now enough to define a criterion for the FP class.

Definition 2.8 (P-criterion)**.** A program is said to satisfy the *P-criterion* if and only if it terminates by PPO and has a quasi-interpretation that is pointwise bounded by a polynomial.

Theorem 2.9 (Soundness)**.** *If a program satisfies the P-criterion, then it computes a polynomial-time function, i.e. the size of its reduction proofs are bounded by a polynomial in the size of its inputs.*

We also have the corresponding completeness theorem.

Theorem 2.10 (Completeness)**.** *Every polynomial-time computable function can be implemented by a program terminating by PPO and equipped with a quasi-interpretation that is pointwise bounded by a polynomial.*

Then, our tool consists in applying the verified implementation of the soundness theorem to a user-provided program. We describe in the following sections how to do so in practice. In particular in Section 3 we explain the interface one can use to define some program as a list of rules and in Section 5 we describe the tooling we provide to help with proving the hypotheses of the P-criterion: proving termination by PPO is fully automated and we provide tactics to prove interactively that a program has a quasi-interpretation.

Our main application is the modular exponentiation and has been chosen for two main reasons: it is widely used in cryptography and proving that it runs in polynomial time is not completely straightforward. We will provide more details about it in Section 6.

3 The interface module

We provide an Interface module to simplify the definition of a program (i.e. a set of rewriting rules) without having to dig into every module. This interface is structured into:

- module types, to describe the various parameters that should be defined for a given application,
- functors, to instantiate our definitions and theorems to that application.

In order to define a program, one must first define the types that will be used for the various kinds of symbols: the variables, the functions and the constructors. One must also provide a way to decide the equality for all those symbol kinds along with defaults for functions and constructors; we use those default values to handle error cases, just like the function of the standard library which returns the `nth` element of a list.

So, for instance, our implementation of the modular exponentiation begins with:

```
Inductive variable := x | y | p | ... .
Inductive function := pred_doubleF | ... .
Inductive constructor := TrueC | FalseC | ... .
```

Then we define the decision procedures for equality over these types automatically:

```
Scheme Equality for variable.
Scheme Equality for constructor.
Scheme Equality for function.
```

They are in turn used to define an instance `EMSyntax` of the `SYNTAX` module type.

```
Module EMSyntax <: Interface.SYNTAX.
  Definition variable := variable.
```

```
  Definition function := function.
  Definition constructor := constructor.

  Definition function_default := errF.
  Definition constructor_default := errC.

  Definition variable_eq_dec := variable_eq_dec.
  Definition function_eq_dec := function_eq_dec.
  Definition constructor_eq_dec :=
                        constructor_eq_dec.
End EMSyntax.
```

Those basic parameters are enough to build the full syntax for an application: values, terms, rewriting rules, reduction proofs (which are called cbv, since they correspond to call-by-value reductions), etc. This can be performed using the MkSyn functor.

```
Module Import Syn := Interface.MkSyn(EMSyntax).
```

Once the complete syntax is set up, one can define the program itself, as a list of rewriting rules:

```
Module Type PROGRAM.
  Parameter prog : list rule.
End PROGRAM.
```

The Interface provides some handy notations to help with getting readable rewriting rules. To get the best results, they should be completed with notations specific to the application.

```
Import Syn.ProgramNotations.
Definition em_prog : list rule := [ ...
  condF (TRUE, x, y) --> x;
  condF (FALSE, x, y) --> y;
  ... ].
```

Finally it is possible to instantiate our MkProg functor to access all the definitions and theorems depending on the program.

```
Module EMProg <: Syn.PROGRAM.
  Definition prog := em_prog.
End EMProg.

Module Import Prog := Syn.MkProg (EMProg).
Import Prog.QI Prog.Evaluator.
```

In particular, once we have proved that our program fulfills its hypotheses, we can instantiate the P_criterion theorem (i.e. Theorem 2.9):

```
Theorem polytime: ∀ i s p c f lv d v,
  let t := fapply f lv in
  let pi := cbv_update i s p c t d v in
  wf pi → cache_bounded qic qif c →
  size pi ≤ global_bound mcs qif f lv c.
```

where t is the initial term, pi is a reduction proof — i.e. encodes a tree of reduction rules defined by the call-by-value memoisation semantics —, wf is a predicate checking that this reduction proof is well-formed, and size is the size of the reduction proof which accounts for the number of rules that the proof uses, but also the size of all the terms it contains. The function global_bound is then a polynomial depending on the maximum constant used to interpret constructors (often 1), the quasi-interpretation qif, the size of the initial term t, and the size of the initial cache c, which will often be considered as empty.

The Syn.MkProg functor also gives us access to a simple Evaluator needed to prove that the rewriting rules of the program do indeed define the intended functions. To do so, it defines an inductive relation on terms and values:

```
Inductive evaluates : term → value → Prop :=
| CAPPLY: ∀ lt lv c, evaluates_list lt lv →
           evaluates (capply c lt) (c_capply c lv)
| FAPPLY: ∀ lt lv f v i s t,
           evaluates_list lt lv →
           first_rule (f, lv) = Some (i, s, t) →
           evaluates t v →
           evaluates (fapply f lt) v
```

where evaluates_list is the obvious corresponding mutually defined function on lists and first_rule (f, lv) = Some (i,s,t) essentially means that the first rule which matches the term f lv produces a term t.

Note that evaluates is really simple because its goal is proving that the rewriting rules compute the intended function. So it does not have to be a function that evaluates a term in polynomial time and involving memoisation[1]. It just has to be proved correct in the sense that it implies the existence of a reduction proof tree from t to v:

```
Lemma evaluates_sound (t : term) (v : value):
  evaluates t v →
  max_arity_term t ≤ max_arity →
  exists p,
    wf p ∧ proj_right p = v ∧ proj_left p = t.
```

For instance, we can prove that the succF function defined by rewriting rules in our program computes the same result as the successor function of the standard library of Coq for all positive numbers:

```
Lemma succ_correct p: ∀ t,
  evaluates t (value_of_pos p) →
  evaluates (succF t) (value_of_pos (Pos.succ p)).
```

[1]We have actually developed an interpreter, i.e. a Coq function that, given a program and an initial term, builds its reduction tree. This interpreter is quite involved since Coq requires function definitions to come with a proof of termination when they are not structurally recursive. So the interpreter definition is tweaked so that some intermediate results of Theorem 2.9 can be used to show it always terminates (or returns an error). Beyond the great complexity of this interpreter (which would make it difficult to prove that it is indeed sound), evaluates is better suited to prove in practice that a program does compute the same result as a reference Coq function because it is much simpler (no failure to handle, for instance).

where `value_of_pos` is a function from some of the program values to the corresponding Coq datatype (here `Positive` for positive binary numbers).

4 Completeness

The soundness theorem proves that the set of programs validating the P-criterion is not "too big" (programs stay polynomial). However, it does not says anything about it being "big enough". Especially, if we want to certify safety of cryptographic protocols and replace the "for all adversaries computing in polynomial time" quantification by "for all adversaries satisfying the P-criterion", we need the two sets to be functionally equivalent (that is, to contain programs computing the exact same set of functions). So, we need to prove that any function of FP can be computed by a program satisfying the P-criterion. This property is called extensional completeness: the set of program is not complete (some polytime programs do not terminate by PPO or admit no QI) but its extension (the set of functions it computes) is complete. Since true completeness is undecidable (the set of polytime programs is Σ_2), extensional completeness is the best we can hope.

In Implicit Complexity, extensional completeness is usually easy to prove and almost overlooked. Indeed, new characterisations are usually built almost incrementally on old ones and the completeness proof usually only requires to show that every program accepted by the old criterion is also accepted by the new one (that is, the new set of accepted programs contains an older set already known to be extensionally complete).

The P-criterion is no exception. Its completeness proof boils down to showing that any BC program terminates by PPO and admits a QI. Since the shape of BC programs is extremely constrained, these properties are somewhat obvious and quickly stated in paper proofs. Since we do have an existing formal proof of the extensional completeness of BC [18], we decided to re-use it for the completeness of the P-criterion.

A more direct way could have been to directly simulate clocked Turing Machines by Term Rewriting Systems. States and Alphabet can be represented by finite types while the tape is classically represented by two lists (one for each half). Thus, the main step of the simulation is performed by a rule like:

$$\mathtt{eval}(c+1, st, l, hd, r) \to \mathtt{eval}(c, st', l', hd', r')$$

The decrease of the first argument (clock) ensures termination and, if it is initialised to a large enough value (polynomial), gives enough time to complete the execution.

However, such a simulation does not terminate by PPO. Indeed, in the case of, for example, a movement to the left, one gets something like $r' = hd :: r$, *i.e.* the right half of the tape increases (becomes longer). This is forbidden by PPO.

In [9], the proof of completeness is done by simulating *Register Machines*, for which this problem does not exist. However, the proof thus relies on the fact that polynomial Register Machines and polynomial Turing Machines do compute the same functions, another proof that would need formalisation.

Therefore, the reduction from BC programs looked like a reasonable idea. It nevertheless lead us into unexpected difficulties…

The formalisation of BC programs in [18] defines them in a way similar to Primitive Recursion (on notations). That is, for example, the recursion is defined as "if g, h_0 and h_1 are BC functions, so is $\mathtt{REC}(g, h_0, h_1)$" (with some constraints on arities). In order to apply the P-criterion, we first need to turn that into a term rewriting system (TRS), which is the syntax used (the semantic equivalence of this translation would need to be formally proved for a complete and robust result). It is quite easy to see that this definition corresponds to the rules:

$$f(0, \vec{n}; \vec{s}) \to g(\vec{n}; \vec{s})$$
$$f(S_i(x), \vec{n}; \vec{s}) \to h_i(x, \vec{n}; \vec{s}, f(x, \vec{n}; \vec{s}))$$

where $\vec{n}$ and $\vec{s}$ are the two kinds of variables used in BC. While easy to do and understand on paper, this translation needs to create a new function symbol (f) in order to work.

Creating fresh names is a standard issue in, typically, compilation. It is normally solved by using numbers (that is, the functions will be called f_1, f_2, …) and keeping a global counter of "first available number" that is incremented every time a new name is required.

Unfortunately, a global counter modified by side effects is not something that is easy to handle in Coq proofs. So we had to find another way. This result in quite complicated completeness proofs where most of the difficulty is located in the somewhat bureaucratic handling of function names.

The naive way to do it is to have the translation function, `BC_to_TRS`, being of type `nat → bc → nat × trs` where the `nat` are the first available number before and after the translation. Then, recursive calls can carry this information over and handle the issue. This is, however, not very convenient when the function needs to be mapped over a list of BC terms (as is the case for the composition rule, since primitive recursion allows composition of an arbitrarily large number of functions).

The solution to this is of course to abstract the side effect into a state monad where the state is here only a single `nat`, (the first available number). So, we have a type `monad A = nat → nat × A` and the translation can easily be written with type `bc → monad trs`. This function can now be mapped over a list, resulting in a list of monads that can then be turned into a monad of a list.

Another way to create fresh names could be to use the position in the syntactical tree of the BC term. That is, $f_{[]}$ would

represent the main function symbol, $f_{[0]}$ the function corresponding to the first subterm, $f_{[0,2,1]}$ the second subterm of the third subterm of the first subterm, ... However, we would still have needed to carry over the initial part of position (hence, keep a state), and the non-bounded composition rule implies that the syntactical tree can have arbitrary arity, thus the list of positions also has to contain arbitrary natural numbers (and not just booleans). Using position lists for names would definitely have simplified some intermediate Lemmas (like `BC_to_TRS_func_bounds` below) but without removing the main difficulty, and it might have create new difficulties just because lists of naturals are more complex than naturals.

In addition to building the rules, the translation also returns a lot of global parameters for the programs (embedded into the record type `trs_prog`). The rank (precedence) and QI are obvious extra info that needs to be built at translation time. We also carry over the range of function numbers used (first and last) as it makes the proofs somewhat less complicated afterwards.

Once the translation is done, we need to prove that every TRS obtained from a BC program satisfies the P-criterion (with the QI built during the translation).

The BC programs already have some sort of well-formedness check which looks at the arities of functions involved in recursion and composition. For this reason, most of the lemmas here have a `arities bc = ok_arities n s` hypothesis which states that only well-formed BC programs are considered. Note also that the complicated case in the proofs is usually the composition because of the unbounded number of recursive calls on $\vec{g_N}$ and $\vec{g_S}$.

The first steps are purely bureaucratic and need to check that the result is indeed a well-formed TRS (e.g. variables in the RHS are present in the LHS), and that some of the global parameters built during the translation (e.g. the maximal arity) are indeed correct. We have, for example, the following lemma:

```
Proposition BC_to_TRS_rules_vars_defined bc n s st:
  let trs := snd (BC_to_TRS bc st) in
  arities bc = ok_arities n s →
  Forall rule_vars_defined trs.(rules).
```

stating that the variables appearing in RHS are all defined in the corresponding LHS. Most of the lemmas have this structure: first we build the TRS from the BC program (using the translation, evaluating the resulting monad and taking the part of the result which is interesting); next we have some pre-conditions (mostly that the BC program is meaningful, i.e. has correct arities); and then we state the property (here that all variables appearing in the rules are defined).

```
Proposition BC_to_TRS_arity bc n s st:
  let trs := snd (BC_to_TRS bc st) in
  arities bc = ok_arities n s →
  max_arity_prog trs.(rules) ≤ trs.(maxar).
```

states that the maximum arity computed by the translation is indeed a bound on the arities of symbols used in the program.

```
Theorem BC_to_TRS_wf bc n s st:
  let trs := snd (BC_to_TRS bc st) in
  arities bc = ok_arities n s →
  wf_prog trs.(maxar) trs.(rules).
```

finally states that the TRS we build are well-formed.

The next step is quite complicated but one of the most important to ease the actual proofs. We prove that the bounds on the numbers (names of function symbols) used are correct. We have:

```
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).
```

saying that every function appearing in the (LHS of) the TRS has a number within the bounds;

```
Proposition BC_to_TRS_infos_iff bc st f:
  let trs := snd (BC_to_TRS bc st) in
  f ∈ map fst trs.(infos) ⟷
  f ∈ all_lhs_funcs trs.
```

saying that functions have an `info` (i.e. a QI) if and only if they appear in LHS (i.e. are defined somewhere);

```
Lemma BC_to_TRS_rhs_funcs_defined bc st:
  let trs := snd (BC_to_TRS bc st) in
  incl (all_rhs_funcs trs.(rules))
       (all_lhs_funcs trs).
```

sayings that functions that appear in RHS are defined somewhere (appear in some LHS).

Knowing that the bounds on function numbers (`first` and `last`) are correct is crucial for the following proofs. Indeed, when we need to handle the recursive cases (recursion and composition), this allows us to separate cleanly in which subcall the function under consideration was defined and thus direct the proof toward this sub-call where we also know that we will find the correct `info` for the function.

To prove that the result of the translation terminates by PPO, we need to split the TRS following the recursive calls. In order to do so, we also need to split the rank function accordingly (the translation builds it incrementally by merging the rank functions of each sub-call). This is done by a series of lemmas such as:

```
Lemma same_rank_same_ppo_iff s t rk rk':
  (∀ f, f ∈ functions_of_term t → rk f = rk' f) →
  (∀ f, f ∈ functions_of_term s → rk f = rk' f) →
  PPO rk t s ⟷ PPO rk' t s.
```

which states that if two rank functions (`rk` and `rk'`) agree on the symbols actually appearing in the terms, then the ordering is the same.

Together with the precise bounds on the functions numbers established in the previous step, this allows us to split the rank function in the correct way to make use of induction, and we end up with:

```
Theorem BC_to_TRS_PPO bc st n s:
  let trs := snd (BC_to_TRS bc st) in
  arities bc = ok_arities n s →
  PPO_prog trs.(rules) (get_rank trs).
```

stating that the program obtained by translation is indeed ordered by PPO, using the rank function that was computed during the translation.

Lastly, we can turn to the QI. We also start by a QI substitution lemma, similar to the rank substitution lemma, and needed for the same reasons.

```
Lemma qif_swap t qic s:
  ∀ qif1 qif2,
  (∀ f, f ∈ (functions_of_term t) →
  qif1 f = qif2 f) →
  term_assignment qic qif1 (subst s t) =
  term_assignment qic qif2 (subst s t).
```

If `qif1` and `qif2` agree on the functions appearing in the term `t`, then ⟦t⟧ stays the same whether we use one or the other. Again, this allows to split the incrementally built QI into the correct components to use the induction.

Next, we can prove each property of the QI one by one, e.g.:

```
Lemma BC_to_TRS_subterm bc st n s:
  let trs := snd (BC_to_TRS bc st) in
  arities bc = ok_arities n s →
  subterm_QI_prog trs.
```

until we finally group all the results into saying that the computed QI are correct (Theorem 2.9):

```
Theorem BC_to_TRS_QI st (n s:nat) bc:
   let trs := snd (BC_to_TRS bc st) in
   arities bc = ok_arities n s →
   valid_QI_prog trs.
```

Having both termination by PPO and existence of a QI, we just need a final Theorem (Theorem 2.10):

```
Theorem BC_to_TRS_P_criterion bc st no sa:
  ∀ i s p c f lv d v,
  let trs := snd (BC_to_TRS bc st) in
  let t := fapply f lv in
  let pi := cbv_update i s p c t d v in
  arities bc = ok_arities no sa →
     f ∈ all_lhs_funcs trs →
  wf Nat.eq_dec Nat.eq_dec constructor_eq_dec
             rule_default
    trs.(rules) trs.(maxar) pi →
  cache_bounded variable function
            constructor qic (qif trs) c →
  size pi ≤ global_bound variable function
            constructor trs.(rules) trs.(maxar)
            trs.(maxrank) mcs (qif trs) f lv c.
```

Here, we first build `trs` by translating the argument (`bc`); then we build a semantic proof tree (`pi`) to evaluate a term inside this TRS; then come some sanity checks on the arities of the BC program, the fact that the term we evaluate calls a function existing in the TRS, that `pi` is well formed, and that the initial cache is polynomially bounded (we can't hide an exponential value in it); lastly, we can conclude that the size of `pi` is bounded by the (polynomial) `global_bound`.

5 Automation

We here describe the various steps one has to follow to apply the soundness theorem to a given program. We will underline which of them are automated and how, and which remain the user's responsibility.

Once the program has been written, we need to provide decidable equality lemmas for the `function`, `variable` and `constructor` datatypes for the `SYNTAX` module defined in Section 3. This is often easily achieved as they can be chosen to be simple sum types, whose equality properties can be automatically derived by Coq using `Scheme Equality`.

Then we have to ensure that the program is well formed, i.e. that for every rule $\mathtt{f}(p_1, \ldots, p_n) \to t$, every variable occurring in t occurs in one of the p_i. This is achieved by a simple tactic which could easily be turned into a decision function using the decidable equality function on variables.

5.1 Proving the PPO criterion

Recall that the first of the two criteria for our technique is to prove that our program terminates by PPO. For this, we need to find a precedence relation (see Definition 2.1) and the notion of rank (see Definition 2.5).

Functions of rank 0, or of lowest precedence, are those that do not call other function (except mutual recurrence) and thus can be defined "alone". Functions of rank 1 only require the functions of rank 0 to be defined, that is only call functions of rank 0 (plus possible mutual recurrence between functions of rank 1). Functions of rank 2 call functions of rank 0 or 1, and so on. The rank/precedence intuitively corresponds to the order in which functions need to be defined in an actual programming language.

Conversely, finding such a rank function which is compatible with the program is enough to define a precedence relation. In other words, the rank function assigns an integer to each function symbol such that for every rule of the program $\mathtt{f}(p_1, \ldots, p_n) \to t$, $\mathrm{rk}(f) \geq \mathrm{rk}(g)$ for all functions g that occur in t.

This amounts to finding a topological sorting of a directed graph (the function dependency graph). We thus define such a sorting Coq function and wrap it into a tactic to compute the rank function once and for all, avoiding unnecessary simplifications or computation steps in later proofs. Note

that we do not need to formally prove the correctness of this function, as we will need to prove the compatibility of the program with the PPO order where any invalid ordering would be detected.

Then, we proved a decidability proposition for PPO, which we simply have to apply to each rule of the program:

```
Lemma PPO_dec t1 t2: {PPO t1 t2}+{¬PPO t1 t2}.
```

To summarize, the proof of the first criterion — i.e. finding a precedence relation and proving that the program terminates by PPO — is entirely automated, which means that some manual work has to be provided for finding a valid quasi-interpretation as belonging to FP would be decidable otherwise.

5.2 Finding a compatible quasi-interpretation

This final and most difficult part is to define an interpretation function `QI` over terms by providing the framework with an interpretation for each constructor and each function symbol, i.e. some functions

```
qic : constructor → list nat → nat
qif : function → list nat → nat
```

together with some proofs of properties about them. The earlier function is the simplest as it simply requires to be additive with a strictly positive additive constant. The function `fun c ⇒ 1 + suml args` (where `suml args` is the sum of the elements of the list `args`) turns out to be always sufficient.

Let's recall from Definition 2.7 that we need `qif f` to be a function from lists of integers (as many as the arity of `f`) which is bounded by a polynomial and is compatible with each rule `f lp → t` in the program, i.e. which satisfies for each substitution `subst s` from variables to values:

```
qif f (map (term_assignment qic qif) (map (subst s) lp)
≥ term_assignment qic qif (subst s t)
```

where `term_assignment` maps a term to an integer, given two functions `qic` and `qif`.

The subterm and monotonicity properties (see Definition 2.6) also have to be proved, but they can be handled quite easily with some generic tactics.

The difficult part is then to define this function. Already for an example as small as our modular exponentiation, we need to find about thirty polynomials satisfying a few hundred inequalities. Defining them all at once turns out to be non-scalable, especially as each inequality depends on the previous ones (in the program's precedence order). This is why we designed an approach allowing us to define each one separately as well as to prove the corresponding inequalities in an incremental manner.

For this we define partial quasi-interpretations, which may not be defined on each function symbol:

```
Definition p_assignment_function :=
  function → option(list nat → nat).
```

as well as the corresponding partial compatibility property:

```
Definition p_compatible_QI qic qif:= ∀ f lp t s,
 let ru := rule_intro f lp t in (ru ∈ prog) →
 p_term_assignment qic qif (subst s t) ≤p
 p_term_assignment qic qif
        (subst s (lhs_of_rule ru)).
```

where `p_term_assignment` and `≤p` and the respective liftings of `term_assignment` and `≤` in the option monad. We finally define the property `p_smc qic F` which states that given an interpretation `qic` for constructors, the partial interpretation `F` satisfies the partial equivalents of subterm, weak monotonicity and compatibility.

We will denote by `F;;G` the composition of the two partial QIs `F` and `G`: `(F;;G) f` is equal to `F f` if it is defined, otherwise to `G f` if it is defined.

Then, we use the following lemma every time we want to extend the definition of our partial interpretation to more function symbols.

```
Lemma p_smc_split F H qic:
  {G' | p_smc qic (F;;H;;G')} →
    {G | p_smc qic (F;;G)}.
```

Here, `F` represents the current state of our partial QI, and H what remains to be defined to obtain a compatible QI. `H` is then another partial function that we are willing to add to `F`, and `G'` is the new part that remains to be defined. The partial interpretation `H` will most of the time only define one additional function (or a few, in the case of mutual recursion).

Starting from the nowhere-defined partial QI, we will iterate this lemma until it is fully defined and interleave each application of the latter with the following one:

```
Lemma p_smc_QI_app F prog1 prog2 qic:
  p_compatible_QI prog1 qic F →
  {G | p_smc prog2 qic (F;;G)} →
  {G | p_smc (prog1 ++ prog2) qic (F;;G)}.
```

This way, we can prove the inequalities for a few rules of the program (usually the rules defining the function whose QI has just been defined by the latest `H`), i.e. `prog1`, and then get rid of them. `F` still has to remain, as it might appear in later equations, i.e. in `prog2`.

Altogether, these two lemmas are used to define some tactics machinery to help the library's user defining their QI. If the program is of the form `prog1 ++ prog2 ++ prog3` then using these two lemmas leads to a proof of this form:

- The initial goal is to find an interpretation for the whole program:

  ```
  {qif | subterm qif ∧ monotonic qif ∧
  compatible_QI qic qif (prog1 ++ prog2 ++ prog3)}
  ```

- Which amounts to finding a partial interpretation extending the nowhere-defined partial interpretation `E`:

  ```
  {G | p_smc (prog1 ++ prog2 ++ prog3) qic (E;;G)}
  ```

- After some steps, we have defined a partial interpretation $F1$ which is compatible with `prog1` and the current goal is now:

  ```
  {G | p_smc (prog2 ++ prog3) qic (E;;F1;;G)}
  ```

- If `prog2` is the set of rules defining a set S of functions by mutual induction, then we apply the first "split" lemma to extend the interpretation to theses symbols with some partial interpretation `F2` defined on S:

  ```
  {G | p_smc (prog2 ++ prog3) qic (E;;F1;;F2;;G)}
  ```

- If `F2` has been chosen correctly, we can prove that our current interpretation is compatible with `prog2` (otherwise, go back to the previous step) and apply the second lemma:

  ```
  {G | p_smc prog3 qic (E;;F1;;F2;;G)}
  ```

- After `prog3` has been handled in a similar way using a partial interpretation $F3$, we have:

  ```
  {G | p_smc [] qic (E;;F1;;F2;;F3;;G)}
  ```

  which is true with `G = E` as long as the whole interpretation is indeed subterm and monotonic, which is easily provable.

Note that an important aspect of this incremental feature is that it allows for errors: if one realises that an equation cannot be satisfied because of a previous erroneous assignment (most likely a "too big" polynomial), the error can be corrected without requiring to execute the proof script from the beginning of the program: multiplied by a large number of lines, the resolution time for each inequality could be a no-go. Also, this approach allows to deal with modular definitions of programs: a program's QI only needs to be defined from the QI of its dependencies.

6 Example: modular exponentiation

We have applied the various features of our library to an implementation of the modular exponentiation. It is composed of about 110 rules and 30 function symbols and has been mostly written by using Coq's code extraction feature to Haskell code on Coq's Standard Library binary arithmetic functions:

```
Extraction Language Haskell.
Recursive Extraction N.modulo.
```

The extracted code was then easily translated to our syntax for term rewriting systems, as it is quite close to Haskell's. Only a few functions using pattern matching needed to be split onto several functions, but this whole translation process could be easily automated.

This approach presents three major advantages: it was quick to write; it prevented us from writing too much *ad hoc* code which illustrates that this technique requires a limited amount of modification on a traditional program; proving its correctness was quite easy as the translated functions are directly specified by the corresponding Coq function.

After this, two functions had to be modified to make the program terminate by PPO. Namely, Coq's `Pos.compare_cont` function, which compares two positive binary numbers according to one of the three relations $<$, $>$ and $=$, represented in a third argument as a constructor `Lt`, `Gt` or `Eq`.

The issue is that its definition would lead to rules of the form:

```
compare_contF (XI p, XO q, r) -->
    compare_contF (p, q, Gt)
```

where `XI` and `XO` are constructors for binary words. Only the rule (Fun Equiv) of Figure 1 could be applied in this case, which would require that `Gt` $\preceq_{ppo}$ `r`, which is false as only the subterms of a value can be smaller than the value itself. In other words, we cannot prove that the third argument is decreasing by PPO (in the non-strict sense). To circumvent this, we instead defined three functions: one function for each comparison relation, thus getting rid of the third argument. One of them is not recursive, and the other two are mutually recursive and all their arguments decrease on a recursive call (one of them strictly). Once again, this kind of situation could be automatically detected (an argument is from a finite type and doesn't decrease by PPO) and a program transformation applied automatically as well.

The second problematic function was `modulo` (more precisely one of the auxiliary function defining it) as one of its recursive call argument on some inputs x and y is of the form $x - y$, which is not comparable to x with respect to $\prec_{ppo}$. The solution was to use a simpler, more direct implementation of `modulo` defined by recursion on notation, i.e. the only recursion call in `modulo (C x) m` is `modulo x m`, where `C` $\in$ {`XO`,`XI`}. Note that this leads to a longer, yet reasonable, correctness proof for this function, as its definition pattern is quite different from its specification (i.e. Coq's standard library's `N.modulo`).

The only remaining functions are the modular multiplication and modular exponentiation, as they are not defined in Coq's standard library. We first implemented them in a straightforward way by recursion on notation, which allows it to terminate by PPO. It is worth noting that this implementation runs in polynomial time using the call-by-value semantics with memoisation but not without memoisation, as it does two identical recursive calls. This could be fixed using a slightly less naive implementation but this is not our concern here but rather finding a polynomial interpretation.

As expected, we found polynomially-bounded interpretations for all functions but the modular exponentiation. The issue is in fact related to the modular multiplication.

Indeed, the following rule

```
mul_mod_auxF (XI x, y, m) -->
 modF (addF (y, doubleF (mul_mod_auxF (x, y, m))), m)
```

for the modular multiplication implies (according to definition 2.7) finding a polynomial $P = (\!|\texttt{mul\_mod\_aux}|\!)$ satisfying at least the following inequality: $P(X+1,Y,M) \geq P(X+1,Y,M)+M+3$. Although this can be achieved, for example with $P(X,Y,M) = (X+1)\times(M+3)+Y$ (recall that it also has to satisfy the monotonicity and subterm properties), the monomial X has to be multiplied by a factor $C(Y,M)$ greater than 1. Then, as `x` is also the decreasing argument in the definition of the modular exponentiation, its interpretation would need to satisfy $Q(X+1,Y,M) \geq C(Y,M)\times Q(X,Y,M)$, whose solutions are all exponential.

This example shows that despite our efforts in the previous section to make the definition of quasi-interpretations incremental, there may not always be a modular way to define it. Indeed, there may be a program `prog1` (here all the program but modular exponentiation) with a QI interpretation, but which can't be extended with a program `prog2` defined on top of it (here, the modular exponentiation) such that a QI can be found, in which case `prog1`, or at least its QI has to be modified.

Our solution to this issue is to add a "clock" argument `c` to the modular multiplication function.

```
mul_mod_auxF (XI x, y, m, XO c) -->
 modF(addF (y, doubleF(mul_mod_auxF(x, y, m, c))), m)
```

Since the interpretation of this last argument decreases in the same way as $(\!|x|\!)$, then we can find a "smaller" interpretation:

$$(\!|\texttt{mul\_mod\_auxF}|\!)(X,Y,M,C) = (M+1)\times(C+1)+\max(X,Y)$$

This extra argument is also propagated to the modular exponentiation function, but is now constant:

```
exp_mod_auxF(x, XO y, m, c) -->
  mul_modF(exp_mod_auxF(x, y, m, c),
           exp_mod_auxF (x, y, m, p), m, p)
```

This now admits a polynomial interpretation:

$$(\!|\texttt{exp\_mod\_aux}|\!)(X,Y,M,C) = X+2\times(Y+1)\times(M+3)\times(C+1)$$

Finally, we only need to call this function with a large enough clock argument. For this, we need it to be larger (in terms of word length) than any first argument of the modular multiplication. However, all occurrences of `mul_modF` have a call to `exp_mod_auxF` as first argument, whose return values are always bounded by the modulus `m`. It is thus sufficient to build a term of size $M+1$ if the modulus is of size M, which leads to a modular exponentiation function with the following QI:

$$(\!|\texttt{exp\_mod}|\!)(X,Y,M) = X+2\times(Y+1)\times(M+3)\times(M+2)$$

which is indeed a polynomial.

Note that the correctness proofs for the intermediate functions with an extra clock argument are slightly more complex, as they have to ensure that it is large enough on each call.

7 Conclusions and future work

We have shown how a powerful theoretical result such as a characterisation of the complexity class FP can be turned into a powerful tool to prove that a function is in this class. We were led not only to deal with numerous details left implicit in the original proof, but to complete it with a new and fully detailed proof of the completeness theorem. We have then added a thick layer of proof engineering because the raw theorems by themselves were not usable in practice. In this process we were led to introduce the notion of partial QI that allows for incrementally defining a QI instead of having to define a whole QI in one go.

Our Coq library consists of about 3600 lines of specification and 12300 lines of proof, excluding the examples. The modular exponentiation example consists of about 500 lines of specification and 360 of proof. The whole formal development as well as the technical report with detailed paper proofs are available at http://www.cristal.univ-lille.fr/~nowakd/cecoa/.

One direction for future work is to improve automation by adding heuristics that help finding a QI. Another is to combine our library with an external tool that, starting from a program, would generate a TRS in our formalism with proof obligations in Coq for those of the properties that could not be proved automatically.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful and detailed comments.

References

[1] Reynald Affeldt, Miki Tanaka, and Nicolas Marti. 2007. Formal Proof of Provable Security by Game-Playing in a Proof Assistant. In Provable Security, First International Conference, ProvSec 2007, Wollongong, Australia, November 1-2, 2007, Proceedings (Lecture Notes in Computer Science), Willy Susilo, Joseph K. Liu, and Yi Mu (Eds.), Vol. 4784. Springer, 151–168. https://doi.org/10.1007/978-3-540-75670-5_10

[2] Martin Avanzini, Georg Moser, and Michael Schaper. 2016. TcT: Tyrolean Complexity Tool. In Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science), Marsha Chechik and Jean-François Raskin (Eds.), Vol. 9636. Springer, 407–423. https://doi.org/10.1007/978-3-662-49674-9_24

[3] Patrick Baillot, Gilles Barthe, and Ugo Dal Lago. 2015. Implicit Computational Complexity of Subrecursive Definitions and Applications to Cryptographic Proofs. In Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings (Lecture Notes in Computer Science), Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.), Vol. 9450. Springer, 203–218. https://doi.org/10.1007/978-3-662-48899-7_15

[4] Patrick Baillot, Ugo Dal Lago, and Jean-Yves Moyen. 2012. On quasi-interpretations, blind abstractions and implicit complexity. Mathematical Structures in Computer Science 22, 4 (2012), 549–580.

https://doi.org/10.1017/S0960129511000685

[5] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. 2012. Computer-Aided Cryptographic Proofs. In Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings (Lecture Notes in Computer Science), Lennart Beringer and Amy P. Felty (Eds.), Vol. 7406. Springer, 11–27. https://doi.org/10.1007/978-3-642-32347-8_2

[6] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 90–101. https://doi.org/10.1145/1480881.1480894

[7] Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. 2011. Beyond Provable Security Verifiable IND-CCA Security of OAEP. In Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings (Lecture Notes in Computer Science), Aggelos Kiayias (Ed.), Vol. 6558. Springer, 180–196. https://doi.org/10.1007/978-3-642-19074-2_13

[8] Stephen Bellantoni and Stephen A. Cook. 1992. A New Recursion-Theoretic Characterization of the Polytime Functions. Computational Complexity 2 (1992), 97–110. https://doi.org/10.1007/BF01201998

[9] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. 2011. Quasi-interpretations a way to control resources. Theor. Comput. Sci. 412, 25 (2011), 2776–2796. https://doi.org/10.1016/j.tcs.2011.02.007

[10] Alan Cobham. 1965. The Intrinsic Computational Difficulty of Functions. In Logic, Methodology and Philosophy of Science: Proceedings of the 1964 International Congress (Studies in Logic and the Foundations of Mathematics), Yehoshua Bar-Hillel (Ed.). North-Holland Publishing, 24–30.

[11] Loïc Colson. 1998. The Logic in Computer Science Column Functions versus Algorithms. Bulletin of the EATCS 65 (1998), 98–117.

[12] Judicaël Courant, Marion Daubignard, Cristian Ene, Pascal Lafourcade, and Yassine Lakhnech. 2011. Automated Proofs for Asymmetric Encryption. J. Autom. Reasoning 46, 3-4 (2011), 261–291. https://doi.org/10.1007/s10817-010-9186-x

[13] Jerry den Hartog. 2008. Towards mechanized correctness proofs for cryptographic algorithms: Axiomatization of a probabilistic Hoare style logic. Sci. Comput. Program. 74, 1-2 (2008), 52–63. https://doi.org/10.1016/j.scico.2008.09.006

[14] Nachum Dershowitz. 1982. Orderings for Term-Rewriting Systems. Theor. Comput. Sci. 17 (1982), 279–301. https://doi.org/10.1016/0304-3975(82)90026-3

[15] Hugo Férée, Samuel Hym, Micaela Mayero, Jean-Yves Moyen, and David Nowak. 2017. A detailed proof of the P-criterion. (Oct. 2017). http://www.cristal.univ-lille.fr/~nowakd/cecoa/

[16] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. 2017. Analyzing Program Termination and Complexity Automatically with AProVE. J. Autom. Reasoning 58, 1 (2017), 3–31. https://doi.org/10.1007/s10817-016-9388-y

[17] Oded Goldreich. 2001. The Foundations of Cryptography - Volume 1, Basic Techniques. Cambridge University Press.

[18] Sylvain Heraud and David Nowak. 2011. A Formalization of Polytime Functions. In Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings (Lecture Notes in Computer Science), Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.), Vol. 6898. Springer, 119–134. https://doi.org/10.1007/978-3-642-22863-6_11

[19] Dieter Hofbauer. 1992. Termination Proofs by Multiset Path Orderings Imply Primitive Recursive Derivation Lengths. Theor. Comput. Sci. 105, 1 (1992), 129–140. https://doi.org/10.1016/0304-3975(92)90289-R

[20] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 359–373. https://doi.org/10.1145/3009837

[21] Martin Hofmann. 2000. Safe recursion with higher types and BCK-algebra. Ann. Pure Appl. Logic 104, 1-3 (2000), 113–166. https://doi.org/10.1016/S0168-0072(00)00010-5

[22] Russell Impagliazzo and Bruce M. Kapron. 2006. Logics for reasoning about cryptographic constructions. J. Comput. Syst. Sci. 72, 2 (2006), 286–320. https://doi.org/10.1016/j.jcss.2005.06.008

[23] Jonathan Katz and Yehuda Lindell. 2014. Introduction to Modern Cryptography, Second Edition. CRC Press.

[24] Jean-Yves Marion. 2003. Analysing the implicit complexity of programs. Inf. Comput. 183, 1 (2003), 2–18. https://doi.org/10.1016/S0890-5401(03)00011-7

[25] John C. Mitchell, Ajith Ramanathan, Andre Scedrov, and Vanessa Teague. 2006. A probabilistic polynomial-time process calculus for the analysis of cryptographic protocols. Theor. Comput. Sci. 353, 1-3 (2006), 118–164. https://doi.org/10.1016/j.tcs.2005.10.044

[26] J.-Y. Moyen and J. G. Simonsen. 2016. More intensional versions of Rice's Theorem. In Developments in Implicit Computational Complexity, DICE'16, D. Mazza (Ed.). Eindhoven, Netherlands.

[27] David Nowak. 2007. A Framework for Game-Based Security Proofs. In Information and Communications Security, 9th International Conference, ICICS 2007, Zhengzhou, China, December 12-15, 2007, Proceedings (Lecture Notes in Computer Science), Sihan Qing, Hideki Imai, and Guilin Wang (Eds.), Vol. 4861. Springer, 319–333. https://doi.org/10.1007/978-3-540-77048-0_25

[28] David Nowak. 2008. On Formal Verification of Arithmetic-Based Cryptographic Primitives. In Information Security and Cryptology - ICISC 2008, 11th International Conference, Seoul, Korea, December 3-5, 2008, Revised Selected Papers (Lecture Notes in Computer Science), Pil Joong Lee and Jung Hee Cheon (Eds.), Vol. 5461. Springer, 368–382. https://doi.org/10.1007/978-3-642-00730-9_23

[29] David Nowak and Yu Zhang. 2015. Formal security proofs with minimal fuss: Implicit computational complexity at work. Inf. Comput. 241 (2015), 96–113. https://doi.org/10.1016/j.ic.2014.10.008

[30] H. E. Rose. 1984. Subrecursion: Functions and Hierarchies. Oxford University Press.

[31] Johannes Waldmann. 2015. Matrix Interpretations on Polyhedral Domains. In 26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland (LIPIcs), Maribel Fernández (Ed.), Vol. 36. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 318–333. https://doi.org/10.4230/LIPIcs.RTA.2015.318

[32] Harald Zankl and Martin Korp. 2014. Modular Complexity Analysis for Term Rewriting. Logical Methods in Computer Science 10, 1 (2014). https://doi.org/10.2168/LMCS-10(1:19)2014

[33] Yu Zhang. 2010. The computational SLR: a logic for reasoning about computational indistinguishability. Mathematical Structures in Computer Science 20, 5 (2010), 951–975. https://doi.org/10.1017/S0960129510000265