Formal Security Proofs with Quasi-Interpretations

David Nowak

CRIStAL, CNRS & Lille 1 University

Joint work with:
Hugo Férée, Samuel Hym,
Micaela Mayero and Jean-Yves Moyen

DICE 2016

April 2, 2016

Outline

Motivations: security proofs

Previous work: Bellantoni and Cook

Quasi-interpretation

Quasi-interpretation, formally

Outline

Motivations: security proofs

Previous work: Bellantoni and Cook

Quasi-interpretation

Quasi-interpretation, formally

The need for formal proof

- Wrong proofs by eminent cryptographers find their way into top-level cryptography conferences.

  An infamous example: RSA-OAEP
  Industry-wide standard (PKCS#1 V2, IEEE P1363)

  - It was proved secure (Bellare and Rogaway, Eurocrypt'94).
  - In fact, the proof had important flaws (Shoup, Crypto'01).
  - Those flaws were finally(?) fixed by Pointcheval in 2005.
  - Formal security proof for RSA-OAEP [Barthe et al., RSA'11]

- Sometimes there are hidden assumptions.

- Some subtle points may be swept under the carpet.

- "*Many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor.*" (Bellare and Rogaway, 2004).

The need for a proof assistant

- Human being
  - + Creative
  - – Make mistakes
- Computer
  - – Stupid: just follows instructions
  - + Thorough: make no oversight
- A proof assistant allows to combine their strengths.

The Coq proof assistant

- Based on a *kernel* which checks that:
  a given proof term p is really a proof of a given statement H.
- On top of the kernel
  - A tactic language for building proofs incrementically
  - Decision procedures and heuristics
  - Notations, implicit parameters, coercions. . . .
  - A standard library:
    arithmetic, analysis, polymorphic lists. . .
- The kernel is the only critical part:
  it will reject wrong proof terms.

Coq, concretely

```
Require Import Arith.

Lemma mult_plus_distr_r :
  forall n m p, (n + m) * p = n * p + m * p.
Proof.
induction n as [ | n IH ].

- trivial.

- intros m p.
  simpl.
  rewrite IH.
  apply plus_assoc.
Qed.
```

```
1 subgoals
n : nat
IH : forall m p : nat, (n + m) * p = n * p + m * p
m : nat
p : nat
____________________________________(1/1)
p + (n + m) * p = p + n * p + m * p
```

The need for a characterization of polynomial time

- An adversary is a function computable in probabilistic polynomial time (PPT),

  i.e., 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.

- A cryptographic scheme is a set of PPT functions.

- A security property is modeled as a challenge that is to be solved in polynomial time by the adversary.

- Security proof: Start from an adversary A that can solve the challenge in polynomial time, and use it to build another adversary that can solve in polynomial time a mathematical problem believed not to be in P.

Example: ElGamal public-key encryption scheme

- ElGamal consists of the three following algorithms:

$$\text{keygen}() \;=\; x \xleftarrow{R} \mathbb{Z}_q \;;\; pk \leftarrow \gamma^x \;;\; sk \leftarrow x \;;\; \text{return } (sk, pk)$$

$$\text{encrypt}(pk, m) \;=\; y \xleftarrow{R} \mathbb{Z}_q \;;\; c \leftarrow (\gamma^y, pk^y * m) \;;\; \text{return } c$$

$$\text{decrypt}(sk, (c_1, c_2)) \;=\; m \leftarrow \frac{c_2}{c_1^{sk}} \;;\; \text{return } m$$

- Correctness is obvious: decryption indeed undoes encryption.
- Security is not so obvious.
  In fact, what do we mean by security?

Example: Semantic security

- **In English:** The challenger says to the adversary "*Give me two plaintexts; I will select one by flipping a coin, encrypt it, and give you the resulting cyphertext; You must then guess which of the two plaintexts I have encrypted.*"

- **In pseudocode:**

$$
\begin{aligned}
&(pk, sk) \leftarrow \text{keygen}() \; ; \\
&r \xleftarrow{R} R \; ; \\
&(m_1, m_2) \leftarrow A_1(r, pk) \; ; \\
&b \xleftarrow{R} \{1, 2\} \; ; \\
&c \leftarrow \text{encrypt}(pk, m_b) \; ; \\
&\hat{b} \leftarrow A_2(r, pk, c) \; ; \\
&\textit{return } \hat{b} \stackrel{?}{=} b
\end{aligned}
$$

- The cryptographic scheme is said "semantically secure" if for any adversary (A_1, A_2), the probability that this game returns true is negligibly close to $\frac{1}{2}$.

Decisional Diffie-Hellman

Semantic security for ElGamal depends on a problem that is believed not to be in P.

> *No algorithm can distinguish in polynomial time between triples of the form* $(\gamma^x, \gamma^y, \gamma^{xy})$ *and* $(\gamma^x, \gamma^y, \gamma^z)$ *where* x*,* y *and* z *are chosen randomly in* $\mathbb{Z}_q$*.*

The complexity class FP

- A function problem:
  Given an input x, output y such that $x \; R \; y$.

- A function problem is solvable in polynomial time if there exists a deterministic Turing machine M and a polynomial p such that:
  - On an input x, machine M halts after at most $p(|x|)$ steps, and
  - $M(x) = y$ iff $x \; R \; y$

- FP is the set of function problems that can be solved by a deterministic Turing machine in polynomial time.

Formal proof with Turing machines?

+ It is easy to define Turing machines in a proof assistant.

– But it is difficult to find a definition that will be usable.

– Even on paper, authors adapt the definition to their purpose
  - Moving head: $\{L, R\}$ or $\{L, R, N\}$?
  - One or more tapes?
  - ...

The need for implicit computational complexity

- We do not want to count explicitly the number of steps in a precise execution model such as a Turing machine.
- We are interested in the complexity class, independently of the execution model.
- The right approach is ICC.

Extensional vs intensional completeness

ICC provides decidable syntactic criteria for complexity classes

Soundness
If a program satisfies the syntatic criterion,
then it implements a function in the associated complexity class.

Extensional completeness
If a function is in a certain complexity class,
then there is a program satisfying the associated syntactic criterion.

Intensional completeness
If a program implements a function in a certain complexity class,
then it satisfies the associated syntactic criterion.

Outline

Motivations: security proofs

Previous work: Bellantoni and Cook

Quasi-interpretation

Quasi-interpretation, formally

The characterization of FP by Bellantoni and Cook (1992)

- A programming language BC that characterizes the complexity class FP
  - + It is sound.
  - + It is extensionally complete.
  - – It lacks expressivity.
- We formalized it in Coq [Heraud and Nowak, 2011].

  The proof consists in a translation back and forth between BC and the characterization of FP by Cobham (1965).

The characterization of FP by Bellantoni and Cook (1992)

i. **Constant** 0

ii. **Projection** $\pi_j^{m,n}(x_1, \ldots, x_m; x_{m+1}, \ldots, x_{m+n}) = x_j$

iii. **Successors** $s_i(; a) = ai \quad$ for $i \in \{0, 1\}$

iv. **Predecessor** $p(; 0) = 0 \quad$ and $\quad p(; ai) = a$

v. **Recursion** $\quad f(0, \overline{x}; \overline{a}) = g(\overline{x}; \overline{a})$
$f(yi, \overline{x}; \overline{a}) = h_i(y, \overline{x}; \overline{a}, f(y, \overline{x}; \overline{a})) \quad$ for $yi \neq 0$
where g, h_0 and h_1 are in this class

vi. **Composition** $\quad f(\overline{x}; \overline{a}) = h(\overline{r}(\overline{x};); \overline{t}(\overline{x}; \overline{a}))$
where h, $\overline{r}$ and $\overline{t}$ are in this class

There are two kind of variables separated by a semicolon:

$$f(\underbrace{x_1, \ldots, x_n}_{\text{normal}}; \underbrace{a_1, \ldots, a_s}_{\text{safe}})$$

Binary addition, by H. E. Rose (1984), page 127

give a recursive characterization $\bar{P}$ of P_f. $\bar{P}$ was described, briefly, by Cobham (1965), but no details were given.

The system $\bar{P}$ is a recursion theory on 'notation', that is a theory which describes the operations on strings of zeros and ones performed by the computing machine. Note that $\bar{P}$ itself is based on the usual sequence of natural numbers.

DEFINITION The system $\bar{P}$. The initial functions of $\bar{P}$ are o, i_k^n, s_1, s_2 and $\bar{D}$, where $s_1x = 2x$, $s_2x = 2x + 1$, $\bar{d}(x)$ denotes the number of digits in the binary representation of x (called the length of x) and

$$\bar{D}(x, y) = (2^{\bar{d}(x)})^{\bar{d}(y)}.$$

The operations of $\bar{P}$ are composition and limited recursion on notation defined as follows: given g, h_1, h_2, and j, f is given by limited recursion on notation if it satisfies

$$f(x_1,\ldots,x_n,0) = g(x_1,\ldots,x_n),$$
$$f(x_1,\ldots,x_n,s_1y) = h_1(x_1,\ldots,x_n,y,f(x_1,\ldots,x_n,y)) \quad \text{provided } y \neq 0,$$
$$f(x_1,\ldots,x_n,s_2y) = h_2(x_1,\ldots,x_n,y,f(x_1,\ldots,x_n,y)),$$
$$f(x_1,\ldots,x_n,y) \leqslant j(x_1,\ldots,x_n,y).$$

It is intuitively clear that all the operations and initial functions of $\bar{P}$ are 'polynomial time' bounded. For instance the calculation of $f(x_1,\ldots,x_n,y)$ by the scheme above takes $\bar{d}(y)$ steps plus the number of steps in the calculation of the auxiliary functions.

We shall show now that $\bar{P}$ contains all polynomial functions. The definition of addition is complicated because of the 'carry', but once we have this function the other arithmetic functions can be defined easily.

THEOREM 4.1 The addition function belongs to $\bar{P}$.

Proof. We shall give a sequence of definitions in $\bar{P}$ which will lead to addition.

1. Define f_1 by limited recursion on notation by

$$f_1(0) = s_2(0)$$
$$f_1(s_1x) = i_1^2(s_2x, f_1(x)) \qquad (x \neq 0)$$
$$f_1(s_2x) = s_1(i_2^2(x, f_1(x)))$$
$$f_1(x) \leqslant s_2(x).$$

Clearly f_1 defines the usual successor function s, $f_1(x) = x + 1$. In the remaining definitions we shall use a more informal notation.

Binary addition, by H. E. Rose (1984), page 128

2. $x \dot{-} 1$: $0 \dot{-} 1 = 0$, $s_1 x \dot{-} 1 = s_2(x \dot{-} 1)$, $s_2 x \dot{-} 1 = s_1 x$, and $x \dot{-} 1 \leqslant x$.
3. $1 \dot{-} x$: $1 \dot{-} 0 = 1$, $1 \dot{-} s_i x = 0$ $(i = 1, 2)$, and $1 \dot{-} x \leqslant 1$.
4. Quotient by 2, $q(x, 2)$: $q(0, 2) = 0$, $q(s_i x, 2) = x$ $(i = 1, 2)$, and $q(x, 2) \leqslant x$.
5. Remainder by 2, $r(x, 2)$: $r(0, 2) = 0$, $r(s_1 x, 2) = 0$, $r(s_2 x, 2) = 1$, and $r(x, 2) < 2$.
6. The length of x, $\bar{d}(x)$: $\bar{d}(0) = 0$, $\bar{d}(s_i x) = \bar{d}(x) + 1$ $(i = 1, 2)$, and $\bar{d}(x) \leqslant x$, by (1).
7. $x \cdot (1 \dot{-} y)$: $x \cdot (1 \dot{-} 0) = x$ and $x \cdot (1 \dot{-} s_i y) = 0$ $(i = 1, 2)$.
8. $x + (1 \dot{-} y)$: $x + (1 \dot{-} 0) = x + 1$ and $x + (1 \dot{-} s_i y) = x$ $(i = 1, 2)$, by (1).
9. $(1 \dot{-} x) \cdot y + x \cdot (1 \dot{-} (1 \dot{-} x))$: $(1 \dot{-} 0) \cdot y + 0 \cdot (1 \dot{-} (1 \dot{-} 0)) = y$, for $i = 1, 2$ $(1 \dot{-} s_i x) \cdot y + s_i x \cdot (1 \dot{-} (1 \dot{-} s_i x)) = s_i x$, and $(1 \dot{-} x) \cdot y + x \cdot (1 \dot{-} (1 \dot{-} x)) \leqslant \bar{D}(y + 1, s_2 x)$.
10. $x + r(y, 2)$: $x + r(0, 2) = x$, $x + r(s_1 y, 2) = x$, and $x + r(s_2 y, 2) = x + 1$.
11. $x + (1 \dot{-} y) \cdot r(z, 2)$: $x + (1 \dot{-} y) \cdot r(0, 2) = x$, $x + (1 \dot{-} y) \cdot r(s_1 z, 2) = x$, and $x + (1 \dot{-} y) \cdot r(s_2 z, 2) = x + (1 \dot{-} y)$, by (8).
12. Concatenation in $\bar{\mathrm{P}}$, $2^{\bar{d}(x)} \cdot y + x$: $2^{\bar{d}(0)} \cdot y + 0 = y$, $2^{\bar{d}(s_i x)} \cdot y + s_i x = s_i(2^{\bar{d}(x)} \cdot y + x)$ $(i = 1, 2)$, and $2^{\bar{d}(x)} \cdot y + x \leqslant \bar{D}(x, y + 1)$.
13. $x \cdot 2^{\bar{d}(y)}$: $x \cdot 2^{\bar{d}(0)} = x$, $x \cdot 2^{\bar{d}(s_i y)} = s_1(x \cdot 2^{\bar{d}(y)})$ $(i = 1, 2)$, and $x \cdot 2^{\bar{d}(y)} \leqslant \bar{D}(x, y)$.
14. $q(x, 2^{\bar{d}(y)})$, the first $(\bar{d}(x) \dot{-} \bar{d}(y))$ digits of x: $q(x, 2^{\bar{d}(0)}) = x$ and $q(x, 2^{\bar{d}(s_i y)}) = q(q(x, 2^{\bar{d}(y)}), 2)$ by (4).
15. $q(x, q(2^{\bar{d}(y)}, 2^{\bar{d}(z)}))$: this is similar to (14).

We shall define now $r(x, 2^{\bar{d}(y)})$, the last $\bar{d}(y)$ digits of x, in two stages. First we define S, then the required function is given by composition. Using definitions (5), (11), (14), and (15) we define

16. $S(x, y, 0) = 0$,
    for $i = 1, 2$ $S(x, y, s_i z) =$
    $s_1 S(x, y, z) + (1 \dot{-} q(q(2^{\bar{d}(x)}, 2^{\bar{d}(z)}), 2^{\bar{d}(y)}) \cdot r(q(x, q(2^{\bar{d}(x)}, 2^{\bar{d}(z)})), 2)$
    $S(x, y, z) \leqslant x$.

In this definition note first that as $\bar{d}(z)$ increases, $r(q(x, q(2^{\bar{d}(x)}, 2^{\bar{d}(z)})), 2)$ enumerates the digits (0 or 1) of the binary representation of x and second the factor multiplying this term is non-zero only when $\bar{d}(z) > \bar{d}(x) \dot{-} \bar{d}(y)$.

17. $r(x, 2^{\bar{d}(y)}) = S(x, y, x)$ by (16).

Binary addition, by H. E. Rose (1984), page 129

To deal with the 'carry' we shall define f so that $f(x)$ gives the position of the last zero in the binary representation of x. As above we define an auxiliary function g working digit by digit now from right to left. Using definitions (3), (5), (9), (13), and (14) we define

18. $g(x, 0) = 0$,
    for $i = 1, 2$ $g(x, s_i y) =$
    $(1 \dot{-} g(x, y)) \cdot 2^{\bar{d}(s_i y)} \cdot (1 \dot{-} r(q(x, 2^{\bar{d}(y)}), 2)) + g(x, y) \cdot (1 \dot{-} (1 \dot{-} g(x, y)))$
    $g(x, y) \leqslant x$.

19. $f(x) = g(x, x)$ by (18).

(For example, if x is represented by 10010111, then $f(x)$ is represented by 1000.)

We shall define now $x + 2^{\bar{d}(y)}$; addition can be defined from this by iteration. First consider an example. Suppose x is represented by 110100111101 and y by 110, so that $2^{\bar{d}(y)}$ is represented by $1000 = b_0$. We split x into three parts b_1, b_2, and b_3 where $b_1 = 11010$, $b_2 = 0111$, and $b_3 = 101$. Note that the representation of x is the concatenation of b_1, b_2, and b_3. Now it is easy to see that $x + 2^{\bar{d}(y)}$ is represented by the concatenation of b_1, b_0, and b_3 which equals 110101000101.

20. $x + 2^{\bar{d}(y)}$ can now be defined by the following program: construct in turn
    $q(x, 2^{\bar{d}(y)}) = b_4$, $r(x, 2^{\bar{d}(y)}) = b_3$ using (14) and (17),
    $f(b_4) = b_0 = 2^{\bar{d}(f(b_4)) \dot{-} 1}$ using (19) (Note: $x = 2^{\bar{d}(y)}$ implies $2^{\bar{d}(x)} = 2x$.),
    $q(b_4, b_0) = b_1$ using (14), and
    $2^{\bar{d}(b_2)} \cdot (2^{\bar{d}(b_0)} \cdot b_1 + b_0) + b_2 = x + 2^{\bar{d}(y)}$ using (12) twice.

21. $x + 2^{\bar{d}(y)} \cdot r(z, 2)$: use (20) and limited recursion on notation.

22. $x + y$: use (21) and iteration on notation to define A

$$A(x, y, 0) = x$$
$$A(x, y, s_i z) = A(x, y, z) + 2^{\bar{d}(z)} \cdot r(q(y, 2^{\bar{d}(z)}), 2) \quad (i = 1, 2)$$
$$A(x, y, z) \leqslant \bar{D}(x, y).$$

Note: by (5) and (14), $r(q(y, 2^{\bar{d}(z)}), 2)$ is the zth digit of y counting from right to left.)

Finally we have

$$x + y = A(x, y, x). \square$$

LEMMA 4.2 Multiplication and the function $x^{\bar{d}(y)}$ belong to $\bar{\mathrm{P}}$.

Extending Bellantoni-Cook

- SLR (Hofmann, 1997) adds higher order and linear types: It allows to program binary addition in a natural way.
- OSLR (Mitchell et al., 1998) adds a 0,1-valued oracle.
- CSLR (Zhang, 2009) adds monadic types to OSLR in order to build a logic upon the language.
- CSLR applied to security proof (Nowak and Zhang, 2010)

Outline

Motivations: security proofs

Previous work: Bellantoni and Cook

Quasi-interpretation

Quasi-interpretation, formally

Term Rewriting System

A program is a list of rewriting rules of the form

$$f\ p_1\ \ldots\ p_n \quad \longrightarrow \quad t$$

with

$$\begin{array}{rcll} p & ::= & c\ p_1\ \ldots\ p_n \mid x & \text{pattern} \\ t & ::= & c\ t_1\ \ldots\ t_n \mid x \mid f\ t_1\ \ldots\ t_n & \text{term} \\ v & ::= & c\ v_1\ \ldots\ v_n & \text{value} \end{array}$$

where x is a variable, c a constructor and f a function symbol.

Call-by-value semantics

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

$$\frac{\exists j, t_j \text{ is not a value} \quad t_i \downarrow v_i \quad \mathtt{f}(v_1, \ldots, v_n) \downarrow v}{\mathtt{f}(t_1, \ldots, t_n) \downarrow v} \text{(Split)}$$

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

Call-by-value semantics with memoisation

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

$$\frac{\begin{array}{c}\exists j, t_j \text{ is not a value} \\ \langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, v_i\rangle \quad \langle C_n, \mathtt{f}(v_1, \ldots, v_n)\rangle \Downarrow \langle C, v\rangle\end{array}}{\langle C_0, \mathtt{f}(t_1, \ldots, t_n)\rangle \Downarrow \langle C, v\rangle}\ \text{(Split)}$$

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

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

Assignment

An assignment $[\![.]\!]$ maps each constructor or function symbol of arity n to a function from $\mathbb{R}^n$ to $\mathbb{R}$ such that:

- **Additivity** $[\![c]\!](X_1, \ldots, X_n)$ is of the form

$$r_c \ + \ \sum_{i=1}^{n} X_i$$

  where $r_c \geq 1$ is a constant depending on c.
- **Subterm**

$$\forall i, \ \ X_i \ \leq \ [\![f]\!](X_1, \ldots, X_n)$$

- **Weak monotonicity**

$$(\ \forall i, \ \ X_i \leq Y_i\ ) \ \ \Rightarrow \ \ [\![f]\!](X_1, \ldots, X_n) \leq [\![f]\!](Y_1, \ldots, Y_n)$$

- **Polynomial** An assignment is bounded by a polynomial.

Quasi-interpretation

An assignment is extended to closed terms in the canonical way:

$$\begin{array}{rcl} [\![c\ t_1\ \ldots\ t_n]\!] & = & [\![c]\!]([\![t_1]\!], \ldots, [\![t_n]\!]) \\ [\![f\ t_1\ \ldots\ t_n]\!] & = & [\![f]\!]([\![t_1]\!], \ldots, [\![t_n]\!]) \end{array}$$

An assignment is a quasi-interpretation iff for each rewriting rule $l \longrightarrow r$, the assignment decreases (not necessarily strictly), i.e., for all substitution σ

$$[\![l\sigma]\!] \geq [\![r\sigma]\!]$$

Termination ordering

- A quasi-interpretation ensures a size bound but does not ensure termination (because it does not strictly decrease).
- We need an additional criterion:
  - We give an order for function symbols;
  - We extend it to terms: Product Path Ordering (PPO);
  - We check that it strictly decreases for each rule.

A criterion for FP

Theorem (Bonfante, Marion, Moyen (2011))

Soundness: *A program equipped with a quasi-interpretation and a compatible PPO implements a function in FP.*
Extensional completeness: *Any function in FP can be implemented as a program equipped with a quasi-interpretation and a compatible PPO.*

Example: binary addition

```
data Word = Nil | C0 Word | C1 Word

suc Nil     = C1 Nil
suc (C0 x) = C1 x
suc (C1 x) = C0 (suc x)

addF Nil     y       = y
addF x       Nil     = x
addF (C0 x) (C0 y) = C0 (addF x y)
addF (C0 x) (C1 y) = C1 (addF x y)
addF (C1 x) (C0 y) = C1 (addF x y)
addF (C1 x) (C1 y) = C0 (addT x y)

addT Nil     y       = suc y
addT x       Nil     = suc x
addT (C0 x) (C0 y) = C1 (addF x y)
addT (C0 x) (C1 y) = C0 (addT x y)
addT (C1 x) (C0 y) = C0 (addT x y)
addT (C1 x) (C1 y) = C1 (addT x y)

add x y = addF x y
```

Example: assignment

- Size of constructors

$$r_{\mathsf{Nil}} = r_{\mathsf{C0}} = r_{\mathsf{C1}} = 1$$

- Assignment for constructors

$$\begin{aligned} [\![\mathsf{Nil}]\!] &= r_{\mathsf{Nil}} \\ [\![\mathsf{C0}]\!](X) &= r_{\mathsf{C0}} + X \\ [\![\mathsf{C1}]\!](X) &= r_{\mathsf{C1}} + X \end{aligned}$$

- Assignment for function symbols

$$\begin{aligned} [\![\mathsf{suc}]\!](X) &= 1 + X \\ [\![\mathsf{addF}]\!](X, Y) &= 1 + X + Y \\ [\![\mathsf{addT}]\!](X, Y) &= 1 + X + Y \\ [\![\mathsf{add}]\!](X, Y) &= 1 + X + Y \end{aligned}$$

Example: quasi-interpretation

We check that it decreases:

$$[\![\text{add x y}]\!] \;=\; 1 + X + Y \;=\; [\![\text{addF x y}]\!]$$

$$\begin{aligned}
[\![\text{addF (C1 x) (C1 y)}]\!] &= 1 + [\![\text{C1 x}]\!] + [\![\text{C1 y}]\!] \\
&= 1 + (1 + X) + (1 + Y) \\
&> 1 + (1 + X + Y) \\
&= [\![\text{C0 (addT x y)}]\!]
\end{aligned}$$

$$\begin{aligned}
[\![\text{suc (C1 x)}]\!] &= 1 + [\![\text{C1 x}]\!] \\
&= 1 + (1 + X) \\
&= [\![\text{C0 (suc x)}]\!]
\end{aligned}$$

Example: strictly decreasing by PPO

- The order for function symbols:

$$rank(\text{suc}) = 0 \quad rank(\text{addF}) = rank(\text{addT}) = 1 \quad rank(\text{add}) = 2$$

$$\text{suc} \prec_{\mathcal{F}} \text{addF} \approx_{\mathcal{F}} \text{addT} \prec_{\mathcal{F}} \text{add}$$

- Each rule strictly decreases by PPO:
  - addF x y $\prec_{ppo}$ add x y by (Fun Strict) because
    - addF $\prec_{\mathcal{F}}$ add
    - x $\prec_{ppo}$ add x y by (Subterm Fun)
    - y $\prec_{ppo}$ add x y by (Subterm Fun)
  - C0 (suc x) $\prec_{ppo}$ suc (C1 x) by (Cons Strict) because suc x $\prec_{ppo}$ suc (C1 x) by (Fun Equiv) because
    - suc $\approx_{\mathcal{F}}$ suc
    - x $\prec^{p}_{ppo}$ C1 x by (Subterm Cons)

Example: conclusions

+ Our program for binary addition is natural.

+ The quasi-interpretation is easy to find because, although it must decrease, it does not have to strictly decrease (as opposed to an interpretation).

– To deal with the carry bit:
  - We cannot use an additional argument.
  - This is because we must respect PPO.
  - Instead we resort to mutually recursive functions.

Outline

Motivations: security proofs

Previous work: Bellantoni and Cook

Quasi-interpretation

Quasi-interpretation, formally

The point of making a formal proof

- Fix glitches that always remain in paper proofs
- Make the theorem available for formal proofs that need it
- Obviously true does not necessarily imply easily provable.
  - Obviously true lemmas might not even be stated.
  - But you can't get away from it with a proof assistant.
  - Two extreme examples as an aside (in the next three slides)

An aside: 1+1 = 2 in Principia Mathematica (1910)

***110·632.** $\vdash : \mu \in \mathrm{NC} . \supset . \mu +_c 1 = \hat{\xi}\{(\exists y) . y \in \xi . \xi - \iota`y \in \mathrm{sm}``\mu\}$

Dem.

$\vdash . *110{\cdot}631 . *51{\cdot}211{\cdot}22 . \supset$

$\vdash : \mathrm{Hp} . \supset . \mu +_c 1 = \hat{\xi}\{(\exists \gamma, y) . \gamma \in \mathrm{sm}``\mu . y \in \xi . \gamma = \xi - \iota`y\}$

$[*13{\cdot}195] \qquad = \hat{\xi}\{(\exists y) . y \in \xi . \xi - \iota`y \in \mathrm{sm}``\mu\} : \supset \vdash . \mathrm{Prop}$

***110·64.** $\vdash . 0 +_c 0 = 0 \qquad [*110{\cdot}62]$

***110·641.** $\vdash . 1 +_c 0 = 0 +_c 1 = 1 \qquad [*110{\cdot}51{\cdot}61 . *101{\cdot}2]$

***110·642.** $\vdash . 2 +_c 0 = 0 +_c 2 = 2 \qquad [*110{\cdot}51{\cdot}61 . *101{\cdot}31]$

***110·643.** $\vdash . 1 +_c 1 = 2$

Dem.

$\vdash . *110{\cdot}632 . *101{\cdot}21{\cdot}28 . \supset$

$\vdash . 1 +_c 1 = \hat{\xi}\{(\exists y) . y \in \xi . \xi - \iota`y \in 1\}$

$[*54{\cdot}3] \quad = 2 . \supset \vdash . \mathrm{Prop}$

The above proposition is occasionally useful. It is used at least three times, in *113·66 and *120·123·472.

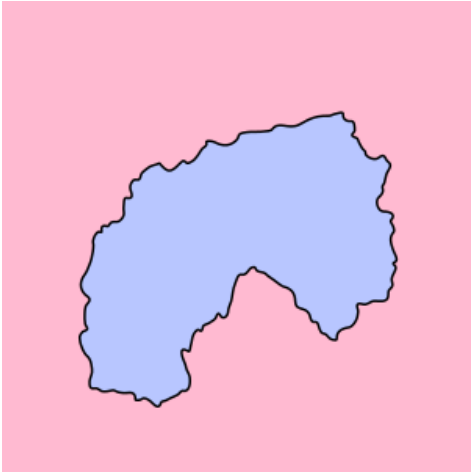
*110·7·71 are required for proving *110·72, and *110·72 is used in

An aside:1+1=2 in Coq (nowadays)

```
Goal 1+1=2.
Proof. trivial. Qed.
```

An aside: The Jordan curve theorem

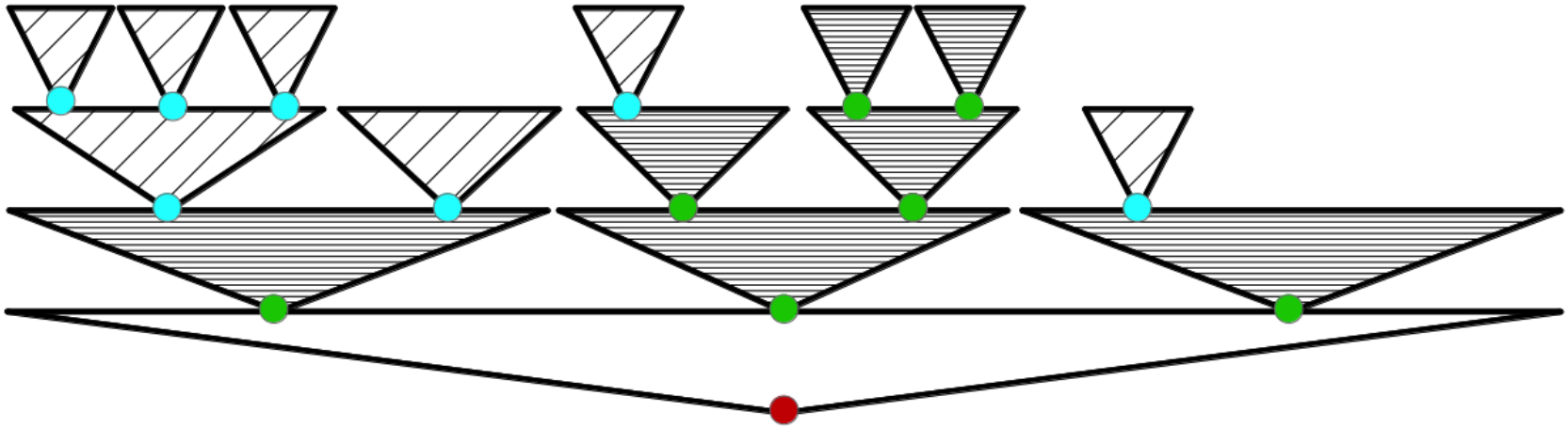
Every non-self-intersecting continuous loop curve divide the plane into an inside region and an outside region.



- This is obviously true, but not easily provable.
- The first formal proof of this theorem by Hales in 2005 contains about 60,000 lines of code.

Obviously true, but not so easily provable

- Jean-Yves Moyen re-wrote for us the paper proof with details.
- However, we had to prove many lemmas that are so obviously true that Jean-Yves did not even state. They are implicit in the paper proof.
- **Example of such lemma**:
  For any first activation a_1 (except the root) at a given rank $rank(a_1)$, there exists an activation a_2 such that:
  - a_1 is a first activation of a_2, and
  - $rank(a_1) < rank(a_2)$.



A small change of the semantics

The existential quantifier in the Split rule is to avoid an unbounded number of split.

$$\frac{\exists j, t_j \text{ is not a value} \quad \langle C_{i-1}, t_i \rangle \Downarrow \langle C_i, v_i \rangle \quad \langle C_n, \mathtt{f}(v_1, \ldots, v_n) \rangle \Downarrow \langle C, v \rangle}{\langle C_0, \mathtt{f}(t_1, \ldots, t_n) \rangle \Downarrow \langle C, v \rangle} \text{(Split)}$$

A small change of the semantics

The existential quantifier in the Split rule is to avoid an unbounded number of split.

$$\frac{\cancel{\exists j, t_j \text{ is not a value}} \quad \langle C_{i-1}, t_i\rangle \Downarrow \langle C_i, v_i\rangle \quad \langle C_n, \mathtt{f}(v_1, \ldots, v_n)\rangle \Downarrow \langle C, v\rangle}{\langle C_0, \mathtt{f}(t_1, \ldots, t_n)\rangle \Downarrow \langle C, v\rangle}\ (\text{Split})$$

We instead force a Split to be followed by a Read or an Update.

```
Fixpoint wf (proof_tree : cbv) : Prop :=
match proof_tree with
.
.
.
| cbv_split l (( cbv_update _ _ _ C' (fapply f lv) C'' v)) C (fapply f' lt) C'''
    ...
| cbv_split l (( cbv_read C' (fapply f lv) v)) C (fapply f' lt) C'' v' ⇒
    ...
.
.
.
end.
```

Structural induction: natural number

- An inductive definition such as

```
Inductive nat : Type :=
| O : nat
| S : nat → nat.
```

  comes with a structural induction principle:

```
∀ P : nat → Prop,
P O                                  →
(∀ n : nat, P n → P (S n)) →
∀ n : nat, P n
```

- You might want to prove your own induction principle:

```
Lemma strong_induction:
∀ P : nat → Prop,
(∀ n : nat, (∀ k : nat, (k < n → P k)) → P n) →
∀ n : nat, P n.
```

Structural induction: term

- An inductive definition such as

```
Inductive term : Type :=
| var : variable → term
| capply : constructor → list term → term
| fapply : function → list term → term.
```

comes with a too weak structural induction principle:

```
∀ P : term → Prop,
(∀ x,     P (var x))          →
(∀ c lt, P (capply c lt))  →
(∀ f lt, P (fapply f lt))  →
∀ t : term, P t
```

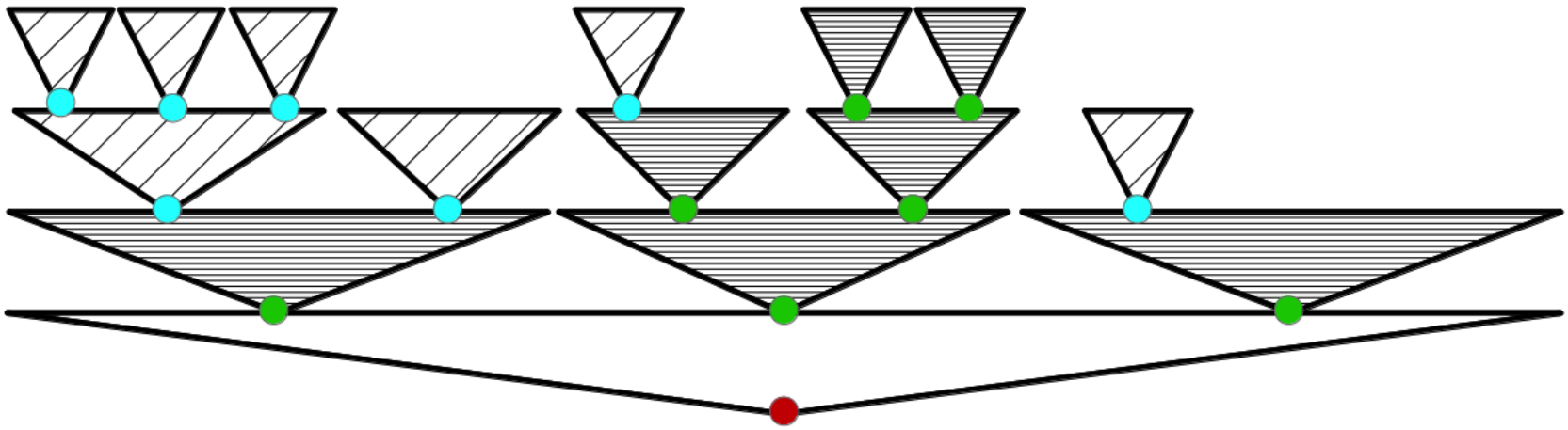
- One has to prove the right one:

```
Lemma term_ind2 : ∀ (P : term → Prop),
  (∀ x,                                P (var x))        →
  (∀ c lt, (∀ t, In t lt → P t) → P (capply c lt)) →
  (∀ f lt, (∀ t, In t lt → P t) → P (fapply f lt)) →
  ∀ t, P t.
```

Big induction

- We sometimes want the induction hypothesis to be about sub-activations.
  Example:
  In the proof that sub-activations are strictly smaller by PPO.

- In [Bonfante et al., 2011], reduction trees are transformed in call trees/dags by removing details between activations.



Big induction, formally

The type `cbv` is for reduction proofs:

```
Inductive cbv : Type :=
| cbv_constr :
    list cbv → cache → term → cache → value → cbv
| cbv_split :
    list cbv → cbv → cache → term → cache → value → cbv
| cbv_update :
    nat → (variable → value) → cbv →
    cache → term → cache → value → cbv
| cbv_read : cache → term → value → cbv.
```

A big induction principle, instead of transformation into call trees:

```
Lemma cbv_big_induction :
  ∀ (P : cbv → Prop),
  (∀ i s p c1 t c2 v,
   (∀ p', In p' (first_activations (cbv_update i s p c1 t c2 v)) → P p') →
   P (cbv_update i s p c1 t c2 v)) →
  ∀ i s p c1 t c2 v, P (cbv_update i s p c1 t c2 v).
```

Big induction: example

Sub-activations are strictly smaller by PPO:

```
Lemma PPO_activations i s p c1 t c2 v p' :
  let proof_tree := cbv_update i s p c1 t c2 v in
  PPO_prog → wf proof_tree →
  In p' (activations proof_tree) →
  p' = proof_tree ∨ PPO (proj_left p') (proj_left proof_tree).
Proof.
  revert p'.
  apply cbv_big_induction; try tauto.
  .
  .
  .
Qed.
```

Reverse induction: natural number

For natural numbers

```
Inductive nat : Type :=
| O : nat
| S : nat → nat.
```

you can prove a reverse induction principle:

```
Lemma rev_nat_ind :
  ∀ (P : nat → Prop) (m : nat),
  P m →
  (∀ k, k < m → P (S k) → P k) →
  ∀ n, n <= m → P n.
```

Reverse induction: reduction proof

```
Lemma cbv_reverse_induction : ∀ (P : cbv → Prop) proof_tree,
  P proof_tree →

  (∀ lp c1 t c2 v,
   InCBV (cbv_constr lp c1 t c2 v) proof_tree →
   P (cbv_constr lp c1 t c2 v) →
   ∀ p, In p lp → P p) →

  (∀ lp p c1 t c2 v,
   InCBV (cbv_split lp p c1 t c2 v) proof_tree →
   P (cbv_split lp p c1 t c2 v) →
   ∀ p', (p' = p ∨ In p' lp) → P p') →

  (∀ i s p c1 t c2 v,
   InCBV (cbv_update i s p c1 t c2 v) proof_tree →
   P (cbv_update i s p c1 t c2 v) →
   P p) →

  ∀ p, InCBV p proof_tree → P p.
```

Reverse induction: example

If a reduction proof is well-formed,
then all its sub-proofs are well-formed:

```
Lemma wf_InCBV_wf p proof_tree:
  wf proof_tree → InCBV p proof_tree → wf p.
Proof.
  intro H_proof_tree_wf.
  apply cbv_reverse_induction.
  .
  .
  .
Qed.
```

Product Path Ordering

$$\frac{s \preceq_{ppo} t_i}{s \prec_{ppo} \mathbf{c}(\ldots, t_i, \ldots)} \text{(Subterm Cons)}$$

$$\frac{s \preceq_{ppo} t_i}{s \prec_{ppo} \mathtt{f}(\ldots, t_i, \ldots)} \text{(Subterm Fun)}$$

$$\frac{\forall i, s_i \prec_{ppo} \mathtt{f}(t_1, \ldots, t_n)}{\mathbf{c}(s_1, \ldots, s_m) \prec_{ppo} \mathtt{f}(t_1, \ldots, t_n)} \text{(Cons Strict)}$$

$$\frac{\mathrm{g} \prec_{\mathcal{F}} \mathtt{f} \quad \forall i, s_i \prec_{ppo} \mathtt{f}(t_1, \ldots, t_n)}{\mathrm{g}(s_1, \ldots, s_m) \prec_{ppo} \mathtt{f}(t_1, \ldots, t_n)} \text{(Fun Strict)}$$

$$\frac{\mathtt{f} \approx_{\mathcal{F}} \mathrm{g} \qquad (s_1, \ldots, s_n) \prec^{p}_{ppo} (t_1, \ldots, t_n)}{\mathrm{g}(s_1, \ldots, s_n) \prec_{ppo} \mathtt{f}(t_1, \ldots, t_n)} \text{(Fun Equiv)}$$

$(s_1, \ldots, s_n) \prec^{p}_{ppo} (t_1, \ldots, t_n)$ iff $\forall i, s_i \preceq_{ppo} t_i$ and $\exists j, s_j \prec_{ppo} t_j$.

Product Path Ordering, formally

```
Inductive product {A: Type} (R : A → A → Prop) : list A → list A → Prop :=
| product_conseq :
    ∀ x y xs ys, x = y → product R xs ys  → product R (x::xs) (y:: ys)
| product_consst :
    ∀ x y xs ys, R x y → Forall2 (clos_refl R) xs ys → product R (x::xs) (y:: ys).

Inductive PPO: term → term → Prop :=
| ppo_constr_in    : ∀ s c lt,    In s lt                → PPO s (capply c lt)
| ppo_fun_in       : ∀ s f lt,    In s lt                → PPO s (fapply f lt)
| ppo_constr_sub   : ∀ s t c lt,  In t lt → PPO s t → PPO s (capply c lt)
| ppo_fun_sub      : ∀ s t f lt,  In t lt → PPO s t → PPO s (fapply f lt)
| ppo_constr_split : ∀ c ls f lt, (∀ s, In s ls → PPO s (fapply f lt))
                                  → PPO (capply c ls) (fapply f lt)
| ppo_funlt_split  : ∀ g ls f lt, rank g < rank f →
    (∀ s, In s ls → PPO s (fapply f lt)) →
    PPO (fapply g ls) (fapply f lt)
| ppo_funeqv_split : ∀ g ls f lt, rank g = rank f →
    product PPO ls lt →
    PPO (fapply g ls) (fapply f lt).
```

- The automatically generated induction principle is useless.
- We were not able to state a useful induction principle.
- instead we resort to induction in the size of the terms to prove that PPO is transitive and asymmetric.

Statement of the theorem: syntax

```
(** sets of variables, functions and constructors used in the program *)
Variables variable function constructor : Type.

(** the program, i.e., a list of rewriting rules *)
Variable prog : list rule.

(** Variables on the right side of a rule also occur in the left side. *)
Hypothesis prog_is_well_formed : wf_prog prog.

(** A maximal arity *)
Variable max_arity : nat.
```

Statement of the theorem: termination ordering

```
(** the rank of each function *)
Variable rank : function → nat.

(** a maximal rank *)
Variable max_rank : nat.

(** [max_rank] is maximal. *)
Hypothesis max_rank_is_maximal : ∀ f : function, rank f <= max_rank.

(** The maximal rank is strictly positive.
    (Just to avoid having to write max(1, max_rank) in one of the bounds) *)
Hypothesis max_rank_is_strictly_positive : 0 < max_rank.

(** Each rule strictly decreases by PPO. *)
Hypothesis prog_respects_PPO : PPO_prog prog rank.
```

Statement of the theorem: quasi-interpretation

```
(** the size of each constructor *)
Variable cs : constructor → nat.

(** a maximal constructor size *)
Variable mcs : nat.

(** a quasi-interpretation *)
Variable qic : constructor → list nat → nat.
Variable qif : function → list nat → nat.

(** The quasi-interpretation satisfies the required properties. *)
Hypothesis qic_and_qif_are_valid : valid_QI prog mcs qic qif cs.
```

Statement of the theorem: bound

```
(** A well-formed reduction proof [pi] is polynomially bounded. *)
Theorem P_criterion :
∀
  (i : nat) (s : variable → value) (p : cbv)
  (c : cache) (f : function) (lv : list term)
  (d : cache) (v : value),
  let pi := cbv_update i s p c (fapply f lv) d v in
  wf prog max_arity pi →
  cache_bounded qic qif c →
  size pi <= global_bound prog max_arity max_rank mcs qif f lv c.
```

Interpreter

To prove that a Coq function f is in FP, we must:

- exhibit a program `prog` that respects our criterion for FP, and
- prove that `prog` implements f.

Idea 1: $\mathit{interp}(\texttt{f t1 ... tn}) = f(\mathit{interp}(\texttt{t1}), \ldots, \mathit{interp}(\texttt{tn}))$

Rejected by Coq because it does not structurally terminate!

Idea 2:

- An internal interpreter (up to sub-activations) that terminates structurally, and
- An external interpreter (from activation to activations) that terminates because well-founded PPO strictly decreases.

Non-structurally mutual recursion not supported by Coq!

Idea 3: An abstract machine equipped with a well-founded order that ensures termination

Conclusions and future work

- A sound FP criterion in Coq based on QI and PPO

```
$ wc -l *.v
    1686 CBV.v
    4254 CBV_cache.v
      42 ExampleCBV.v
     745 Examples.v
     156 Final.v
    1284 Interpret.v
    1920 Lib.v
    3464 Ordering.v
     834 QI.v
     764 Syntax.v
   15149 total
```

- Extensional completeness: in progress
  By translating BC into TRS+QI+PPO

- Could we replace PPO by another termination order?
  But PPO contributes significantly to the polynomial bound.