Proof by computation

Benjamin Grégoire[1]

[1]INRIA Sophia Antipolis - Méditerranée, France

EPIT 2015

Examples of proof by computation

- 4-colors theorem
- Kepler conjecture
- Goldbach's conjecture
- Primality proofs

Daily life examples: SAT/SMT solver, ...

A proof of $2 + 2 = 4$ in Peano

Simplified version with axioms:

$$\begin{gathered}(n+1)+m=(n+m)+1 \\ 0+m=m\end{gathered}$$

The proof will look like:

$$\dfrac{\dfrac{\dfrac{\overline{4=4}}{0+4=4}}{1+3=4}}{2+2=4}$$

Think of the size of a proof of $100000 + 100000 = 200000 \ldots$

A first proof of $2 + 2 = 4$

```
Lemma tptf_rw : 2 + 2 = 4.
Proof.
  rewrite !plus_Sn_m,plus_O_n;reflexivity.
Qed.

Print tptf_rw.
tptf_r =
eq_ind_r (fun n : nat => n = 4)
  (eq_ind_r (fun n : nat => S n = 4)
     (eq_ind_r (fun n : nat => S (S n) = 4)
        eq_refl
        (plus_O_n 2))
     (plus_Sn_m 0 2))
  (plus_Sn_m 1 2)
     : 2 + 2 = 4
```

A second proof of $2 + 2 = 4$

```
Lemma tptf_s : 2 + 2 = 4.
Proof.
  simpl;reflexivity.
Qed.

Print tptf_s.
tptf_s = @eq_refl nat 4
     : 2 + 2 = 4

Check (@eq_refl nat 4).
@eq_refl nat 4
     : 4 = 4
```

Why does it works?

- The proposition $2 + 2 = 4$ depends on the function $+$
- The program $2 + 2$ evaluates (reduces) to 4
- So the proposition $2 + 2 = 4$ evaluates to $4 = 4$ (they are convertible)
- Any proof of $4 = 4$ is also a proof of $2 + 2 = 4$ (conversion rule)

$$\frac{\Gamma \vdash t : P \qquad P \equiv Q}{\Gamma \vdash t : Q}[\text{Conv}]$$

Conversion rule allows:

- Small proof (in memory)
- Transparent step (Assia's talk)

Can we use this property to automatize proofs ?

Idea

We want to prove a (semi)decidable property $P\ z$

1. Write a function C that checks the property
2. Prove the correctness :

$$\text{Lemma } C_{cool} : \forall n, C\ n = true \rightarrow P\ n.$$

3. Apply the correctness lemma $C_{cool}\ z$
4. We have to prove $C\ z = true$
5. This can be done by computation

Example: Proving primality

Definition prime $p := 1 < p \wedge \forall n,\ 1 < n < p \to \sim(n \mid p)$.

Definition isprime $p :=$
  $1 < p$ && forallb (fun $n \Rightarrow (p \bmod n) \,!= 0)\ 2\ p$.

Lemma isprime_correct p :
  isprime $p\ =\ $ true $\to$ prime p.

Some primality proofs

```
Lemma prime17 : prime 17.
Proof.
  apply (isprime_correct 17).
  (* we should prove isprime 17 = true *)
  compute.
  (* we should prove true = true *)
  reflexivity.
Qed.

Print prime17.
isprime_correct 17 (@eq_refl bool true).
```

Some primality proofs

```
Lemma prime1069 : prime 1069.
Proof.
  apply (isprime_correct 1069).
  (* we should prove isprime 1069 = true *)
  compute.
  (* we should prove true = true *)
  reflexivity.
Qed.

Print prime1069.
isprime_correct 1069 (@eq_refl bool true).
```

Benchmarks

The size of the proof term is almost constant:

isprime_correct n (@eq_refl bool true)

But not the time for checking the proof:

n	time
17	0.003
1069	0.205
7919	2.189

Can we improve this?

Reduction strategy

During the verification of the proof Coq should check that:

$$(\text{isprime } 1069 = \text{true}) \equiv (\text{true} = \text{true})$$

To do that it should reduce (isprime 1069).

What is the reduction strategy used by Coq ?

Lazy reduction

- By default Coq uses a Lazy comparison strategy.
- It is a good default strategy

Checking

$$\text{fact } (100 + 1) \equiv \text{fact } 101$$

just requires to check that $(100 + 1) \equiv 101$

No need to compute fact (100 + 1) nor fact 101.

Other reduction strategies

In some examples it is better to direclty compute the value:

$$
\begin{gathered}
\text{fact } 101 \\
\equiv \\
9425947759838359420851623124482936749562312794 \\
7025437683278893534169775993162214765030878615 \\
9180834691162349000354959958336970630260326400 \\
0000000000000000000000
\end{gathered}
$$

Coq provides two reduction strategies to do that efficiently:

- vm_compute (based on an OCaml like virtual machine)
- native_compute (based on OCaml native compiler), since 8.5 (Maxime Dénès)

Selecting the strategy

How can we select the strategy that should be used?

```
Lemma prime7919_vm : prime 7919.
Proof.
  apply (isprime_correct 7919).
  vm_compute. reflexivity.
Time Qed.
(* Finished transaction in 0. secs (0.184u,0.s)) *)
Print prime7919_vm.
isprime_correct 7919 (@eq_refl bool true<:isprime 7919 = true)
```

Native compute

How can we select the strategy that should be used?

```
Lemma prime7919_nc : prime 7919.
Proof.
  apply (isprime_correct 7919).
  native_compute. reflexivity.
Time Qed.
(* Finished transaction in 0. secs (0.03u,0.s)) *)
Print prime7919_nc.
isprime_correct 7919 (@eq_refl bool true<<:isprime 7919 = true)
```

Benchmarks

n	lazy	vm	native
17	0.003	0.001	0.001
1069	0.205	0.019	0.004
7919	2.189	0.183	0.030
65761	26.010	2.114	0.324

Can we do better ?

Improve the checker

Checking only the odd numbers between 3 and $\sqrt{p}$

Definition isprime' $p :=$
  $1 < p$ &&
  (p mod 2) $! = 0$ &&
  forallb (fun $n \Rightarrow (p \text{ mod } 2 * n + 1) \; ! = 0)$
    1 (sqrt $p/2 + 1$).

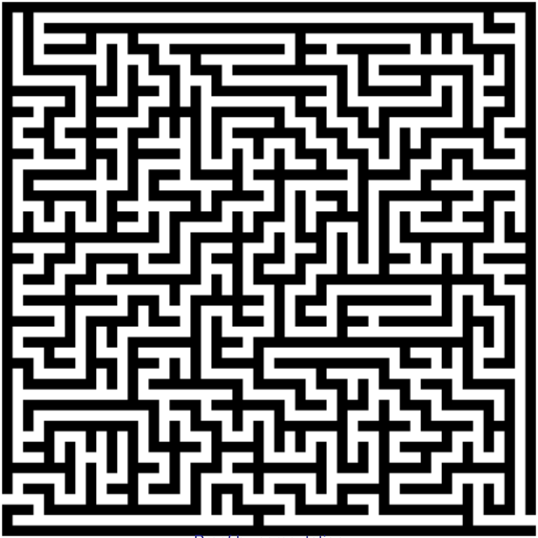
Benchmark:

n	native
65,761	0.002
100,000,007	0.063
1,234,567,891	0.263
3,912,839,611	0.521

Can we do better?

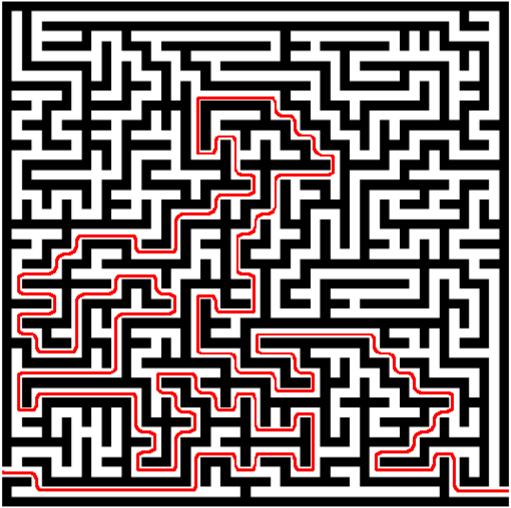
Using external certificate

Finding a path in a labyrinth (can be hard):



Using external certificate

Checking a path in a labyrinth (it is easy):



Using external certificate

Theorem (Pocklington's (1914))

Let $n > 1$ and natural numbers a, (p_1, α_1),...,(p_k, α_k); n is prime if:

$$
\begin{array}{rcll}
p_1 \ldots p_k & \textit{are} & \textit{prime numbers} & (0) \\
(p_1^{\alpha_1} \ldots p_k^{\alpha_k}) & | & n-1 & (1) \\
a^{n-1} & = & 1 \quad (\textit{mod}\ n) & (2) \\
gcd(a^{\frac{n-1}{p_i}}, n) & = & 1 & (3) \\
(p_1^{\alpha_1} \ldots p_k^{\alpha_k}) & > & \sqrt{n} & (4)
\end{array}
$$

- $a, p_1, \alpha_1, \ldots, p_k, \alpha_k$ is a Pocklington certificate for n
- Finding a, p_i, α_i is hard (partial factorization of $n-1$)
- Checking the certificate is easy

General scheme

- Use an external(dedicated) tool to find the certificate:

  $$a, p_1, \alpha_1, \ldots, p_k, \alpha_k$$

  do not need to be trusted/certified
- Write in Coq the checker, and prove its correctness;
- Apply the correctness lemma and compute the result of the checker;

Goldbach: about 25,040,013,776 prime numbers certified in Coq (relatively small prime between $4.10^{10}.2^{52}$ and 10^{29})

"Proving the primality of a number of about 300 decimal digits takes about an hour" (few years ago, using the VM)

Where we are?

Proof by computation allows:

- small proofs (important in Coq, where proof terms are keep)
- efficient verification (native compute)
- linking with external tools (certificates)
- proving only the correctness of the checker

Requires:

- An efficient checker
- A data type on which we can compute

What happens if we are not in this situation?

Deciding permutation of list

```
Inductive list (A:Type) : Type =
  | nil    : list A
  | cons : A → list A → list A.

Fixpoint app (l1 l2:list A) : list A :=
   match l1 with
   | nil          ⇒ l2
   | cons a l1' ⇒ cons a (app l1' l2)
   end.
```

Notation:

- a::l is a notation for cons a l
- l1++l2 is a notation for app l1 l2

Deciding permutation of list

We would like to have a tactic to solve the following problem:

Are two lists equal up to permutation?

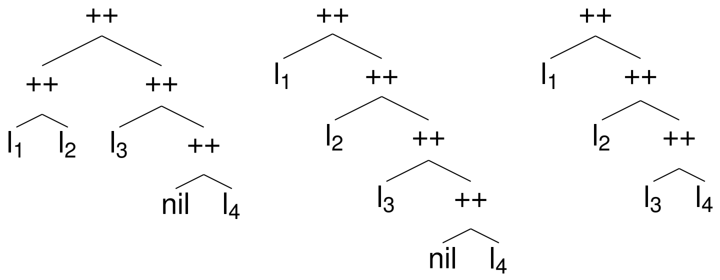
Example:

$$((l_1 ++ l_2) ++ (l_3 ++ (nil ++ l_4)))$$

and

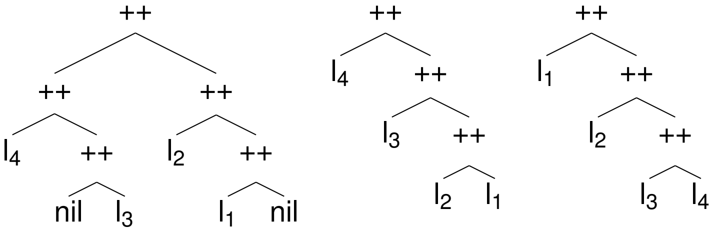
$$(l_4 ++ (nil ++ l_3)) ++ (l_2 ++ (l_1 ++ nil))$$

Main idea: Flattening



Associativity: $(l_1 ++ l_2) ++ l_3 = l_1 ++ (l_2 ++ l_3)$
Neutral : nil $++\ l = l$

Main idea: Flattening and Sorting



In Coq

We need to write the following program:

1. flatten the tree representation:
    - associativity: (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3)
    - neutral: nil ++ l = l   l ++ nil = l
2. sort the resulting tree: we need an order relation

Main difficulty: We can not write this program directly in Coq:

1. The "++" operator is a function (not a constructor)
2. So we cannot write a program of the form:

```
match l with
| l1 ++ l2 => ...
| ...
```

General idea

$$\begin{tikzcd}
E \arrow[r, "\mathrm{norm}"] \arrow[d, "\mathrm{eval}"] & E \arrow[d, "\mathrm{eval}"] \\
L \arrow[r, equal] \arrow[u, bend left, "\text{reify(not a Coq program)}"] & L
\end{tikzcd}$$

- E a data type, representing our problem on which we can compute (AST)
- L the type of list
- here the notion of equality is the permutation on list

AST

```
Definition var := nat.

Inductive expr :=
  | Enil : expr
  | Eapp : expr -> expr -> expr
  | Evar : var -> expr.


Fixpoint eval (rho:valuation A) (e:expr):list A:=
  match e with
  | Enil       => nil
  | Eapp e1 e2 => eval rho e1 ++ eval rho e2
  | Evar v     => rho v
  end.
```

Example

Assuming that

$$\rho : 0 \mapsto l_0; 1 \mapsto f\ x$$

we have

$$\text{eval}\ \rho\ (\text{Lapp}\ (\text{Lvar}\ 0)\ (\text{Lvar}\ 1))$$

and

$$l_0 +\!\!+\ f\ x$$

are convertible

Flattening

Associativity and neutral element:

```
Fixpoint flatten (e:expr) : list var :=
  match e with
  | Enil        => nil
  | Evar v      => v :: nil
  | Eapp e1 e2 => flatten e1 ++ flatten e2
  end.
```

Correctness: $\forall \rho\ e,\ \text{eval}\ \rho\ e = \text{eval\_l}\ \rho\ (\text{flatten}\ e)$

$$\begin{tikzcd}
e \arrow[rr, "\text{flatten}"] \arrow[d, "\text{eval}"'] & & l_v \arrow[d, "\text{eval_l}"] \\
[\![e]\!]_\rho \arrow[rr, equal] & & [\![l_v]\!]_\rho
\end{tikzcd}$$

Normalisation

```
Definition norm e := sort (flatten e).
```

Remark: The order used for sort is the order provided by variables

Correctness: $\forall \rho\ e,$ Permutation (eval ρ e) (eval_l ρ (norm e))

A checker for permutation

```
Definition checker e1 e2 := norm e1 =? norm e2.
```

Correctness:

$\forall \rho\ e_1\ e_2,\ \text{checker}\ e_1\ e_2 = \text{true} \rightarrow \text{Permutation}\ (\text{eval}\ \rho\ e_1)\ (\text{eval}\ \rho\ e_2)$

Example

```
Lemma test1 (l1 l2 l3 l4:list A) :
   Permutation ((l1++l2)++(l3++(nil++l4)))
               (l1++(nil++l2)++l3++(l4++nil)).
Proof.
  apply (checker_correct
          (l1 :: l2 :: l3 :: l4 :: nil) (* rho *)
          (Eapp (Eapp (Evar 0) (Evar 1))
                (Eapp (Evar 2)
                      (Eapp Lnil (Evar 3))))
          (Eapp (Evar 0)
                (Eapp (Eapp Enil (Evar 1))
                      (Eapp (Evar 2)
                            (Eapp (Evar 3)
                                  Enil))))).
  native_compute. reflexivity.
Qed.
```

Remark

Checking the term :

$$(\text{checker\_correct}\ \rho\ e_1\ e_2\ (\text{eq\_refl true})) : l_1 = l_2$$

require two conversions:

$$\begin{array}{rcl} (\llbracket e_1 \rrbracket_\rho = \llbracket e_2 \rrbracket_\rho) & \equiv & (l_1 = l_2) \\ \text{true} = \text{true} & \equiv & (\text{checker}\ e_1\ e_2 = \text{true}) \end{array}$$

Only the second need fast evaluation (the first is linear).

Infering the corresponding AST

Providing manually the arguments to checker_correct is boring

This can be automatized by:

- writing OCaml (expert)
- using Ltac
- using type class (TP)
- using Mtac

```
Lemma test1_infer (l1 l2 l3 l4:list A):
   Permutation ((l1++l2)++(l3++(nil++l4)))
               (l1++(nil++l2)++l3++(l4++nil)).
Proof. permlist. Qed.
```

General scheme

$$
\begin{tikzcd}
E \arrow[r, "\text{norm}"] \arrow[d, "\text{eval}"] & E \arrow[d, "\text{eval}"] \\
L \arrow[r, equal] \arrow[u, bend left, "\text{reify(not a Coq program)}"] & L
\end{tikzcd}
$$

- Reification is not a Coq program
- The checker is a Coq program
- Only the correctness need to be proved (not the completness)