Proof Assistants
Proofs by Reflection, Tactic Language

Guillaume Melquiond

INRIA Saclay – Île-de-France
Laboratoire de Recherche en Informatique

2011-01-25

Outline

Example: Peano's Arithmetic

Definition (Inductive Type for Integers)

```
Inductive nat := O : nat | S : nat -> nat.
(* 5 = SSSSSO *)
```

Axiom (Relating Addition to O and S)

$+ : \texttt{nat} \to \texttt{nat} \to \texttt{nat}$

$\texttt{addO} : \quad \forall b,\ \texttt{O} + b = b$

$\texttt{addS} : \forall a\ b,\ (\texttt{S}\ a) + b = a + (\texttt{S}\ b)$

Note: formal systems tend to prefer definitional extensions for consistency, so they won't contain the above axioms.

Deductive Reasoning for Peano's Arithmetic

Example (Deductive Proof of "$4 + (2 + 3) = 9$")

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots \quad \dfrac{\dfrac{}{9 = 9}\,\texttt{refl\_equal}}{0 + 9 = 9}\,\texttt{add0}}{4 + 5 = 9}\,\texttt{addS} \times 4}{4 + (0 + 5) = 9}\,\texttt{add0}}{4 + (1 + 4) = 9}\,\texttt{addS}}{4 + (2 + 3) = 9}\,\texttt{addS}}{}
$$

9 steps

The bigger the natural numbers in the proof, the more theorems have to be instantiated to prove the statement. This growth has a nonnegligible cost.

- Time complexity: matching and applying theorems (any prover).
- Space complexity: storing proof terms (Coq-like provers).

Computing a bit inside Proofs

Definition (Addition as a Function)

```
Fixpoint plus x y : nat :=
  match x with
  | O => y
  | S x' => plus x' (S y)
  end.
```

Lemma (Soundness)

plus_xlate : $\forall a\ b,\ a + b = \texttt{plus}\ a\ b$.

Note: most formal systems define arithmetic operators directly as functions, hence avoiding the need for soundness theorems.

Computing a bit inside Proofs

Example (Proof of "$4 + (2 + 3) = 9$")

$$\dfrac{\dfrac{\dfrac{\overline{9 = 9}\ \texttt{refl\_equal}}{\texttt{plus}\ 4\ (\texttt{plus}\ 2\ 3) = 9}\ {\color{red}???}}{4 + (\texttt{plus}\ 2\ 3) = 9}\ \texttt{plus\_xlate}}{4 + (2 + 3) = 9}\ \texttt{plus\_xlate}$$

Note: one could consider λ-calculus as a rewriting system and iteratively reduce "$\texttt{plus}\ 4\ (\texttt{plus}\ 2\ 3) = 9$" to 9.
This is no less costly than applying Peano's axioms.

Type Theory and β-Conversion

Theorem (Curry-Howard Isomorphism)

Formula A^* is valid if and only if type A is inhabited.

Example : $(\Gamma \vdash_{\text{typing}} f : P \to Q)$ is equivalent to $(\Gamma^* \vdash_{\text{proving}} P^* \Rightarrow Q^*)$.

Property (Type Theory)

Convertible types have the same inhabitants.

$$\frac{p : A}{p : B}\ A \equiv B$$

β-conversion: $(\lambda x.t)u \equiv t[x \leftarrow u]$.

Type Theory and β-Conversion

Example (Proof of "$4 + (2 + 3) = 9$")

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\color{red}{p} : 9 = 9}\ \texttt{refl\_equal}}{\color{red}{p} : \texttt{plus}\ 4\ (\texttt{plus}\ 2\ 3) = 9}\ \color{red}{\beta\text{-conversion}}}{4 + (\texttt{plus}\ 2\ 3) = 9}\ \texttt{plus\_xlate}}{4 + (2 + 3) = 9}\ \texttt{plus\_xlate}$$

3 steps

Amount of theorem instantiations no longer depends on the constants, only on the number of arithmetic operators.

Note: β-conversion is necessarily implicit when typechecking, so term "`refl_equal 9`" has also type "$\texttt{plus}\ 4\ (\texttt{plus}\ 2\ 3) = 9$".

β-Reduction

At worst, β-reducing has the same complexity than applying rewriting rules, but the constant factor is smaller:

- no matching of theorems,
- only substitutions of terms.

Moreover, β-reduction is amenable to optimizations:

- normalization by evaluation (e.g. Isabelle),
- abstract machine (e.g. Coq).

Example (Coq's Virtual Machine)

λ-terms are compiled to OCaml bytecode, executed (call-by-value evaluation strategy), decompiled to λ-terms that are in weak normal form.

The OCaml interpreter is modified a bit to handle "accumulators", e.g. "`@eq nat 9 9`" (`@eq` is a function without body).

Encoding Arithmetic Expressions

Definition (Inductive Type for Expressions over ℕ)

```
Inductive expr :=
  | Cst : nat -> expr
  | Add : expr -> expr -> expr.
```

Definition (Interpretation of Reified Expressions)

```
Fixpoint interp_expr e : nat :=
  match e with
  | Cst n   => n
  | Add x y => (interp_expr x) + (interp_expr y)
  end.
```

Note: "+" is assumed to be an uninterpreted symbol in the code above (no computational content).

Encoding Arithmetic Expressions

Example (Proof of "$4 + (2 + 3) = 9$")

$$\dfrac{\dfrac{???}{\texttt{interp\_expr (Add (Cst 4) (Add (Cst 2) (Cst 3)))} = 9}}{4 + (2 + 3) = 9}\ \beta\text{-conversion}$$

Evaluating Arithmetic Expressions

Definition (Evaluation of Reified Expressions)

```
Fixpoint eval_expr e : nat :=
  match e with
  | Cst n    => n
  | Add x y => plus (eval_expr x) (eval_expr y)
  end.
```

Lemma (Soundness)

`expr_xlate`: $\forall e$, `interp_expr` $e =$ `eval_expr` e.

Evaluating Arithmetic Expressions

Example (Proof of "$4 + (2 + 3) = 9$")

$$\dfrac{\dfrac{\dfrac{\dfrac{}{9 = 9}\ \texttt{refl\_equal}}{\texttt{eval\_expr (Add (Cst 4)}\ldots)) = 9}\ \beta\text{-conversion}}{\texttt{interp\_expr (Add (Cst 4)}\ldots)) = 9}\ \texttt{expr\_xlate}}{4 + (2 + 3) = 9}\ \beta\text{-conversion}$$

2 steps

The proof structure no longer depends on the arithmetic expression, since rewriting with `plus_xlate` is no longer needed.

Example (Coq Script)

```
change (4 + (2 + 3)) with
  (interp_expr (Add (Cst 4) (Add (Cst 2) (Cst 3)))).
rewrite expr_xlate.
exact (refl_equal 9).
```

Comparison Operators

Definition (Less Than or Equal)

```
Inductive le (n : nat) : nat -> Prop :=
  | le_n : n <= n
  | le_S : forall m : nat, n <= m -> n <= S m.
```

Example (Deductive Proof of "$9 \le 18$")

$$\dfrac{\dfrac{\dfrac{}{9 \le 9}\ \texttt{le\_n}}{9 \le 10}\ \texttt{le\_S}}{\vdots} \qquad \texttt{le\_S} \times 7$$

$$\dfrac{9 \le 17}{9 \le 18}\ \texttt{le\_S}$$

10 steps

Comparison Operators

While equality proofs are structurally trivial, comparison proofs are not. So comparisons should be transformed into equalities first.

Definition (Comparing Natural Numbers)

```
Fixpoint leqb x y : bool =
  match x, y with
  | O   , _    => true
  | S _ , O    => false
  | S x', S y' => leqb x' y'
  end.
```

Lemma (Soundness)

`leq_spec` : $\forall a\ b,$ `leqb` $a\ b = \mathit{true} \Leftrightarrow a \leq b$

Encoding Propositions

Definition (Reified Propositions)

```
Inductive prop := Eq : prop | Le : prop.
Definition interp_prop : prop -> Prop := ...
Definition eval_prop : prop -> bool := ...
Lemma prop_xlate : forall p,
  eval_prop p = true -> interp_prop p.
```

Example (Proof of "$4+(2+3) \le (5+6)+7$")

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathit{true} = \mathit{true}}\ \texttt{refl\_equal}}{\texttt{eval\_prop (Le (Add}\ldots\texttt{) (Add}\ldots\texttt{))} = \mathit{true}}\ \beta\text{-conversion}}{\texttt{interp\_prop (Le (Add}\ldots\texttt{) (Add}\ldots\texttt{))}}\ \texttt{prop\_xlate}}{4+(2+3) \le (5+6)+7}\ \beta\text{-conversion}$$

2 steps

Example : the ring Tactic

Example (Polynomial Equality)

```
Goal forall x y,
  ((x + y) * (x - y) = x * x - y * y)%Z.
intros x y. ring.
```

Proof script generated by ring :

```
pose (hyp_list := nil);
pose (fv_list := x :: y :: nil);
apply (Zr_ring_lemma1 ring_subst_niter fv_list hyp_list
  (PEmul (PEadd (PEX Z 1) (PEX Z 2)) (PEsub (PEX Z 1) (PEX Z 2)))
  (PEsub (PEmul (PEX Z 1) (PEX Z 1)) (PEmul (PEX Z 2) (PEX Z 2)))).
exact I.
vm_compute; exact (refl_equal true).
```

Last line is the β-conversion followed by a boolean equality.

Typing and Oracle

All the proofs now have the same structure (automatization!).
Only the inductive object reifying the proposition is needed.

This object is the abstract syntax tree of the proposition
(with all the unknown subterms generalized away, possible).

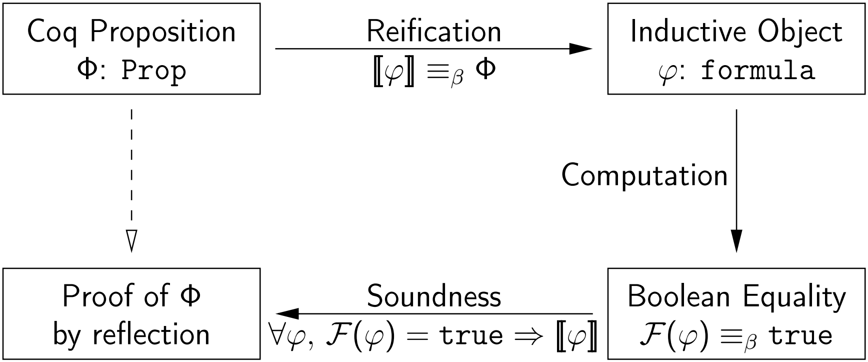
The reification is necessarily performed by an external oracle.
But β-conversion ensures that its result is correct.

$$\dfrac{\mathit{true} = \mathit{true}}{\texttt{eval\_prop}\ \ldots = \mathit{true}} \quad \left\langle \begin{array}{l}\text{typechecking fails}\\ \text{if the user asks too much}\end{array}\right.$$

$$\vdots$$

$$\dfrac{\texttt{interp\_prop}\ (\texttt{Le}\ldots)}{4 + (2 + 3) \leq 5 + 6} \quad \left\langle \begin{array}{l}\text{typechecking fails}\\ \text{if the oracle is wrong}\end{array}\right.$$

Reification and Reflection



Outline

Tactic Language

The structure of a λ-term cannot be obtained inside the logic, hence the need for an oracle, either written in OCaml or in Ltac.

Ltac allows to define new tactics from a Coq script.

Example

```
Ltac rewrite_clear K :=
  rewrite K ; clear K.

Goal forall x, x = 3 -> x + 5 = 8.
intros x H.
rewrite_clear H.
```

Ltac Characteristics

- Term notations are available.
- Recursive functions have no termination requirement.
- Matching can be performed on terms, hypotheses, goal.
- Matching backtracks if the executed branch fails.
- Only two types: Coq λ-terms and Coq tactics.
- Quite slow.

Syntactic Analysis of the Goal

Example (Finding a Particular Hypothesis)

```
Ltac find_rewrite v :=
  match goal with
  | H: v = ?x |- _ => rewrite H
  end.

Goal forall x, x = 3 -> x + 5 = 8.
intros.
find_rewrite x.
```

Note: if H_i were to match but "`rewrite` H_i" were to fail,
Coq would then try to find another hypothesis H_j with $j > i$.

Building Terms

Example (Replacing Values in a List)

```
Ltac replace l u v :=
  let rec aux l' :=
    match l' with
    | ?h :: ?t =>
      let t' := aux t in
      match h with
      | u => constr:(v :: t')
      | _ => constr:(h :: t')
      end
    | nil => l
    end in
  aux l.
```

Note: matching is purely syntactic, β-conversion is not used in Ltac.

- matching l' fails if its head is not a list constructor (e.g. `rev nil`);
- u matches h only if they have the exact same structure.

Tactics for Computing

During a proof, Coq does not strongly normalize terms, unless asked for.

- `compute` (call-by-value),
- `lazy` (call-by-name),
- `vm_compute` (call-by-value in the abstract machine).

Note: `vm_compute` decorates the proof term with special coercions, so that Coq remembers it should use the VM to typecheck at `Qed` time.

Inside a tactic, the reduction of a term can be obtained by

```
let t := eval compute u in ...
```

(or `lazy` or `simpl` or `hnf` or ...).