Proof Assistants
Proofs by Reflection, Tactic Language

Guillaume Melquiond

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

2011-01-25
Outline

1. Proofs by Reflection
   - Example: Peano’s Arithmetic
   - Type Theory and $\beta$-Conversion
   - Handling Expressions
   - Reflecting Propositions
   - Example: the \texttt{ring} Tactic
   - Typing and Oracle

2. Tactic Language
   - Syntactic Analysis
   - Building Terms
   - Tactics for Computing
Definition (Inductive Type for Integers)

\textbf{Inductive} \ \texttt{nat} := \texttt{O} : \texttt{nat} \mid \texttt{S} : \texttt{nat} \to \texttt{nat}.

(* 5 = \texttt{SSSSSO} *)

Axiom (Relating Addition to \texttt{O} and \texttt{S})

\( + : \texttt{nat} \to \texttt{nat} \to \texttt{nat} \)

\( \texttt{addO} : \forall b, \texttt{O} + b = b \)

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

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 “$4 + (2 + 3) = 9$”)

\[
\begin{align*}
9 &= 9 & \text{refl_equal} \\
0 + 9 &= 9 & \text{add0} \\
\vdots & & \text{addS \times 4} \\
4 + 5 &= 9 & \text{add0} \\
4 + (0 + 5) &= 9 & \text{addS} \\
4 + (1 + 4) &= 9 & \text{addS} \\
4 + (2 + 3) &= 9 & \text{addS}
\end{align*}
\]

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)**

```coq
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 = plus a b.
```

Note: most formal systems define arithmetic operators directly as functions, hence avoiding the need for soundness theorems.
Example (Proof of “4 + (2 + 3) = 9”)

\[
\begin{align*}
9 &= 9 \quad \text{refl\_equal} \\
\text{plus } 4 \ (\text{plus } 2 \ 3) &= 9 \quad \text{??}
\end{align*}
\]

\[
\begin{align*}
4 + (\text{plus } 2 \ 3) &= 9 \quad \text{plus\_xlate} \\
4 + (2 + 3) &= 9 \quad \text{plus\_xlate}
\end{align*}
\]

Note: one could consider \(\lambda\)-calculus as a rewriting system and iteratively reduce “\text{plus } 4 \ (\text{plus } 2 \ 3) = 9” to 9. This is no less costly than applying Peano’s axioms.
Type Theory and \( \beta \)-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.

\[
\frac{p : A}{p : B} \quad A \equiv B
\]

\( \beta \)-conversion: \((\lambda x.t)u \equiv t[x \leftarrow u]\).
Example (Proof of “4 + (2 + 3) = 9”)

\[
\begin{align*}
  & p : 9 = 9 \quad \text{refl\_equal} \\
  & p : \text{plus} 4 \ (\text{plus} 2 3) = 9 \quad \text{\textbeta-conversion} \\
  & 4 + (\text{plus} 2 3) = 9 \quad \text{plus\_xlate} \\
  & 4 + (2 + 3) = 9 \quad \text{plus\_xlate}
\end{align*}
\]

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

Note: \(\text{\textbeta-conversion} \) is necessarily *implicit* when typechecking, so term “\text{refl\_equal} 9” has also type “\text{plus} 4 \ (\text{plus} 2 3) = 9”.

G. Melquiond (INRIA, LRI)
At worst, $\beta$-reducing has the same complexity than applying rewriting rules, but the constant factor is smaller:

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

Moreover, $\beta$-reduction is amenable to optimizations:

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

### Example (Coq’s Virtual Machine)

$\lambda$-terms are compiled to OCaml bytecode, executed (call-by-value evaluation strategy), decompiled to $\lambda$-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 $\mathbb{N}$)

\[
\text{Inductive } \text{expr} := \\
\quad \mid \text{Cst} : \text{nat} \rightarrow \text{expr} \\
\quad \mid \text{Add} : \text{expr} \rightarrow \text{expr} \rightarrow \text{expr}.
\]

Definition (Interpretation of Reified Expressions)

\[
\text{Fixpoint } \text{interp_expr} \ e : \text{nat} := \\
\quad \text{match } e \ \text{with} \\
\quad \mid \text{Cst} \ n \Rightarrow n \\
\quad \mid \text{Add} \ x \ y \Rightarrow (\text{interp_expr} \ x) + (\text{interp_expr} \ y) \\
\quad \text{end}.
\]

Note: “+” is assumed to be an uninterpreted symbol in the code above (no computational content).
Example (Proof of “4 + (2 + 3) = 9”)

(???)

\[
\text{interp_expr} \left( \text{Add} \ (\text{Cst} \ 4) \ (\text{Add} \ (\text{Cst} \ 2) \ (\text{Cst} \ 3)) \right) = 9
\]

\[
4 + (2 + 3) = 9
\]

\(\beta\)-conversion
Evaluating Arithmetic Expressions

**Definition (Evaluation of Reified Expressions)**

\begin{verbatim}
Fixpoint eval_expr e : nat :=
    match e with
    | Cst n => n
    | Add x y => plus (eval_expr x) (eval_expr y)
end.
\end{verbatim}

**Lemma (Soundness)**

\[ \forall e, \text{interp_expr} e = \text{eval_expr} e. \]
Proofs by Reflection
Handling Expressions

Evaluating Arithmetic Expressions

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

\[
\begin{align*}
9 &= 9 & \text{refl\_equal} \\
\text{eval\_expr (Add (Cst 4)\ldots))} &= 9 & \beta\text{-conversion} \\
\text{interp\_expr (Add (Cst 4)\ldots))} &= 9 & \beta\text{-conversion} \\
4 + (2 + 3) &= 9
\end{align*}
\]

2 steps

The proof structure no longer depends on the arithmetic expression, since rewriting with plus\_xlate is no longer needed.

Example (Coq Script)

```coq
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 <= 18")

\[
\begin{align*}
9 \leq 9 & \quad \text{le}_n \\
9 \leq 10 & \quad \text{le}_S \\
\vdots & \\
9 \leq 17 & \quad \text{le}_S \times 7 \\
9 \leq 18 & \quad \text{le}_S \\
\end{align*}
\]

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)

\[
\text{Fixpoint } \text{leqb } x \ y : \text{bool} = \\
\text{match } x, \ y \text{ with} \\
| 0, \ _ \Rightarrow \text{true} \\
| S \ _, \ 0 \Rightarrow \text{false} \\
| S \ x', \ S \ y' \Rightarrow \text{leqb } x' \ y' \\
\text{end.}
\]

Lemma (Soundness)

\[
\text{leq_spec} : \forall a \ b, \text{leqb } a \ b = \text{true} \iff a \leq b
\]
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) ≤ (5 + 6) + 7”)

\[
\begin{align*}
    & \text{true} = \text{true} \\
    \Rightarrow & \text{refl_equal} \\
    \Rightarrow & \text{eval_prop (Le (Add\ldots) (Add\ldots))} = \text{true} \\
    \Rightarrow & \text{interp_prop (Le (Add\ldots) (Add\ldots))} \\
    \Rightarrow & 4 + (2 + 3) \leq (5 + 6) + 7
\end{align*}
\]

β-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 $\beta$-conversion followed by a boolean equality.
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 $\beta$-conversion ensures that its result is correct.

\[
\begin{align*}
\text{true} = \text{true} & \quad \text{typechecking fails if the user asks too much} \\
\text{eval\_prop} \ldots = \text{true} & \\
\vdots & \\
\text{interp\_prop} (\text{Le} \ldots) & \quad \text{typechecking fails if the oracle is wrong} \\
4 + (2 + 3) \leq 5 + 6 &
\end{align*}
\]
Reification and Reflection

Coq Proposition

Φ: Prop

Reification

[φ] ≡ β Φ

Inductive Object

φ: formula

Proof of Φ by reflection

Soundness

∀φ, \mathcal{F}(φ) = true \implies [φ]

Computation

Boolean Equality

\mathcal{F}(φ) \equiv β true
Outline

1. Proofs by Reflection
   - Example: Peano’s Arithmetic
   - Type Theory and $\beta$-Conversion
   - Handling Expressions
   - Reflecting Propositions
   - Example: the ring Tactic
   - Typing and Oracle

2. Tactic Language
   - Syntactic Analysis
   - Building Terms
   - Tactics for Computing
Tactic Language

The structure of a $\lambda$-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 $\lambda$-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 “\( \text{rewrite } H_i \)” 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, $\beta$-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.
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 ...).