Ltac Internals

Pierre-Marie Pédrot

INRIA

Coq Implementor Workshop
Disclaimer: what follows applies to trunk (next 8.6)

(And I don’t want to discuss history in this talk anyway)
1 Bird’s eye view

2 Engine

3 Tactics

4 Ltac

5 Future plans
Overall organization of the code

- Lower strata (engine folder)
- ML-defined tactics (tactics folder)
- Ltac itself (ltac folder)

Some folders also of interest: pretyping, proofs
1. Bird’s eye view

2. Engine

3. Tactics

4. Ltac

5. Future plans
This part defines the basic blocks upon which Ltac relies.

- The `Evd.evar_map` proof state
- The $\alpha$ `Proofview.tactic` monad
- The $\alpha$ `Ftactic.t` monad (or is it?)
The evar map (evd.ml)

“The one proof state to rule them all”

type Evd.evar_map

It contains many things defining the proof term being built.

- A map from evars to partial terms
- The current universe unification graph
- Some ugly stuff from the past (the infamous metas)
- More stuff I don’t want to talk about
- Extensible state for clever hacks
The evar map (continued)

Relevant files:
- Low-level definitions: evd.ml
- Statically monotonous variant: sigma.ml
- High-level interaction: evarutil.ml

Note that I’m actively promoting the use of Sigma to get static guarantees, but the API is not entirely ported, so your mileage may vary. You may have to use glue code that will eventually disappear.
“I would like backtrack. And state. And IO.”

type \( \alpha \) tactic

Monadically defines the core effects of the proof engine.

- *Tarte à la crème* (tclUNIT, tclBIND)
- Backtrack (tclZERO, tclOR)
- Backtracking state (tclEVARs, tclEVARMAP, ...)
  - Contains an evar map, but not only
- IO (*NonLogical*, I am not too fond of this API)

(See my CoqHoTT-minute blog post for semantics)
Correct mental model of tactics:

From a state, produce a list of results that have a local state

where State ≡ evar map + goals + focus

and Goals ≡ hypothesis + conclusion

tclZERO ≡ nil, tclPLUS ≡ app
Emulate the historical engine: Proofview.Goal.enter and variants

\[ \text{type } (\alpha, \rho) \text{ Proofview.Goal.t} \]
\[ \text{val enter : } \ldots \text{ enter } \rightarrow \text{ unit tactic} \]

- Indexed by a phantom normalization type + a stage just as Sigma
- Can be projected to recover data (concl, hyps, evar map, ...)
- enter apply a continuation on each focussed goal
- Two orthogonal flags
  1. \( \text{nf}_* \): Do we normalize the goal w.r.t evars?
  2. \( \text{s}_* \): Do we change the current state?
Ftactic (motivation)

From 8.5 onwards, tactics may act on several goals.

This conflicts with Ltac (lack of) semantics! E.g.

```
let t := constr:(x) in ...
```

- Is \( x \) a variable local to a goal (i.e. hypothesis)?
- Is \( x \) a global variable (i.e. definition or section variable)?

Ltac says: the former.

We need to focus on the fly!
type \( \alpha \) Ftactic.t

- Built upon Proofview.tactic
- Monadic API as well
- Two modes: global vs. focussed
- Once focussed, this is forever
- Currently incorrect implementation (not a monad)
1. Bird’s eye view

2. Engine

3. Tactics

4. Ltac

5. Future plans
Not much to say here.

- Many files that implement Coq core tactics
- The kind of code that breaks from being looked at

Have a look at tactics/tactics.ml for 5 kloc of joyful code!

(Everything mentioning clenv not to be looked at)
Essentially, the complete, most basic primitives you can use:

- `Proofview.Goal.enter` to focus on goals
- `Evarutil.new_evar` to introduce holes
- `Refine.refine` to solve a goal
1 Bird’s eye view

2 Engine

3 Tactics

4 Ltac

5 Future plans
Ltac overview

Same three-level steps as terms, with a bit of variations

\[
\cdots \\
\downarrow \\
\text{Tacexpr.raw_tactic_expr} \\
\downarrow \\
\text{Tacexpr.glob_tactic_expr} \\
\downarrow \\
\text{Geninterp.Val.t Ftactic.t} \\
\downarrow \\
\text{unit tactic}
\]

\textit{parsing}

\text{Tacintern.glob_tactic}

\text{Tacinterp.value_interp}

\text{Tacinterp.tactic_of_value}
raw_tactic_expr and glob_tactic_expr share the same skeleton.

- Defined in Tacexpr
- Essentially reflect the syntax
- Parameterized by the inner arguments
- Globalization is functorial

Mutually defined with tactic arguments and atomic tactics.
Type \texttt{Val.t} is a dynamic extensible type.

- You can create new arguments (unique name)
- You can inject and project from this dynamic type

Interpretation function of Ltac parameterized by an environment

\[
\text{type interp\_sign} \sim \text{Val.t Id.Map.t}
\]
The great catastrophe of Ltac:

When are things evaluated?

Answer: Do I look like I know?
Some constructs are evaluated upfront:

- closures
- let, let rec
- the various match
- tactic arguments

The remaining is thunked, and evaluated according to heuristics.

A lot to say and to fix here, but time is running. See value_interp.
Another problem: lack of variables

- Many hacks relying on dynamic typing
- TeX-like confusion between quoted code and meta
  
  Tactic Notation "foo" ident_list(l) := intros l.
- No quotation feature, everything uses heuristics
  
  intro x; let x := constr:(0) in exact x
- Horrendous parsing tricks to counter this
  
  do int_or_var(x) tactic(t) := ...

See tacinterp.ml and taccoerce.ml for gory details.
Extensible language

Atomic tactics are historical remnants and should die.

The recommended way of adding tactics is through the generic extension mechanism.

- **ARGUMENT EXTEND** (for arguments, see TacGeneric)
- **TACTIC EXTEND** (for tactics, see TacML)
Generic arguments (genarg.ml)

Those are dynamic types that implement some primitives.

\[
\text{type } (\alpha, \beta, \gamma) \text{ Genarg.genarg_type}
\]

As for every Coq stuff, three levels

- The \textit{raw} level (user facing)
- The \textit{glob} level (internalized)
- The \textit{typed} level (ML-side typing)

A few hardwired genargs are defined in Stdarg and Constrarg.

By convention, they are named \textit{wit}_*.
Required operations

We can declare extensible operations on genargs.

```coq
module Genarg.Register
```

Important ones in the Coq codebase:
- Parsing to `raw` (`pcoq.ml`)
- Printing from `raw`, `glob`, `typed` (`genprint.ml`)
- Internalization from `raw` to `glob` (`genintern.ml`)
- Substitution from `glob` to `glob` (`genintern.ml`)
- Interpretation from `glob` to `Val.t` (`geninterp.ml`)
- Toplevel representation from `Val.t` to `typed` (`geninterp.ml`)
ARGUMENT EXTEND

There is a CAMLPX macro to generate such boilerplate.

ARGUMENT EXTEND auto_using
TYPED AS uconstr_list
PRINTED BY pr_auto_using
| [ "using" ne_uconstr_list_sep(l, ",","") ] -> [ l ]
| [ ] -> [ [ ] ]
END

Simple example, there is a more complicated variant.
(See extraargs.ml4)
Extending tactics (tacenv.ml)

One can register ML code to use as tactics.

\[
\text{type ml_tactic} = \text{Val.t list } \rightarrow \text{interp_sign } \rightarrow \text{unit tactic}
\]

Such tactics are referred by a ml_tactic_name:

- A ML plugin name (DECLARE PLUGIN foo)
- A ML tactic name
- An integer corresponding to the entry number

No way to directly refer to those primitives from Coq side!
Once again a CAMLPX macro to generate boilerplate.

TACTIC EXTEND econstructor
| [ "econstructor" ] -> [ Tactics.econstructor ]
| [ "econstructor" int_or_var(i) ] -> [ Tactics.econstructor_n i ]
END

This macro

- registers an ML tactic (with automatic casts from Val.t)
- adds a tactic notation referring to the TacML node.
1. Bird’s eye view

2. Engine

3. Tactics

4. Ltac

5. Future plans
Future plans

General guideline: turn Ltac into a ML.

- Fix the evaluation order (ouch!)
- Add static typing (see above)
- Add datatypes
- Fix tactic notations
- Generic quoting mechanism