The Dark Side of $\mathcal{L}_{tac}$

Bruno Barras

ADT - Coq

30/06/09
Why $\mathcal{L}_{\text{tac}}$?

LCF vs $\mathcal{L}_{\text{tac}}$:

- LCF tactics
  
  ```plaintext
  type tactic =
    goal → (goal list * (proof list → proof))
  apply : term → tactic
  ```

- Tacticals
  
  ```plaintext
  then : tactic → tactic → tactic
  ```

- meta-language support vs toplevel
  
  term API vs what?

$\mathcal{L}_{\text{tac}}$ provides term manipulation facilities
Features of $\mathcal{L}_{\text{tac}}$

- Untyped
- Functional (exceptions, no mutable variables)
- Call-by-Value
- Toplevel side-effects ($\text{Ltac ::=}$)
- Dedicated operators (match with backtrack)
- Goal passed implicitly
- Dynamic link (goal context)
Evaluation strategy

Example (A tactic that proves \texttt{True})

\begin{verbatim}
Ltac prove_true :=
  let H := fresh in assert (H:=I).
\end{verbatim}
Evaluation strategy

**Example (A tactic that proves True)**

```
Ltac prove_true :=
  let H := fresh in assert (H:=I).
```

**But**

```
let f := prove_true in f; f ⊸ name clash!
(prove_true; prove_true is OK)
```
Evaluation strategy

Example (A tactic that proves True)

Ltac prove_true :=
  
  let H := fresh in assert (H:=I).

But let f := prove_true in f; f ↭ name clash!
(prove_true; prove_true is OK)

Both tactics work with:

Ltac prove_true :=
  
  (let H := fresh in assert (H:=I));idtac.
Ltac prove_true _ :=
  
  let H := fresh in assert (H:=I).

B. Barras

The Dark Side of $\mathcal{L}_{\text{Ltac}}$
Mixing tactics and expressions

A tactic cannot both transform the goal and return a value.

Example (A robust intro tactic)

```coq
let H := intro in ...
more elegant than
let H := fresh in intro H; ...
```

Fixes:

- **CPS**: `Ltac myintro :=
  fun k ⇒ let H := fresh in intro H; k H`
- ugly hacks (encoding result within the goal)
fail and \mid\mid is not for error handling

Example (Applying transitivity with helpful error message)

Ltac trans :=
    match goal with
    [ H1:_=?x, H2:?x=_ |- _| ] =>
        constr:(eq_trans H1 H2) ||
        fail 1 "anomaly: ill-typed transitivity"
    end.

Always fails... (Value is a term. Expected a tactic.)
Extendability

- LCF tactics can be extended by arbitrary ML code
- Ltac accepts only macros
Expressions and tactics

\[ E ::= x \mid \lambda x.e \mid e_1 e_2 \mid \text{fresh} \mid T \mid tac \mid \text{match } e \text{ with } p_i \Rightarrow e_i \text{ end} \]

\[ tac ::= \text{idtac} \mid \text{fail} \mid e_1 || e_2 \mid e_1 ; e_2 \mid \ldots \text{ (i.e. all LCF tactics)} \]

Values

\[ v ::= \text{TRM}(T) \mid \text{FUN}(\rho, x, E) \mid \text{TAC}(\rho, tac) \mid \text{SGL}(\text{goal}^*) \]

Two semantics

- \( \text{Val}^\rho_G(E) \) (evaluation as an argument: tactics delayed)
- \( \text{Eval}^\rho_G(E) \) (head evaluation: tactics applied to goal)
Evaluation of expressions

\( \lambda \) core:

- \( \text{Val}_G^\rho(x) = \rho(x) \)
- \( \text{Val}_G^\rho(\lambda x.e) = \text{FUN}(\rho, x, e) \)
- \( \text{Val}_G^\rho(e_1 e_2) = \text{Val}_G^{\rho';x'=\text{Val}_G^\rho(e_2)}(e') \) \text{ if } \text{Val}_G^\rho(e_1) = \text{FUN}(\rho', x, e') \)

NB: dynamic linking of term variables

let f _ := constr:x in
  clear x; intro x; let g := f() in apply g
Evaluation of expressions

Terms and tactics:

- \( \text{Val}_G^\rho(T) = \text{TRM}(\rho(T)) \) (term typed in \( G \))
- \( \text{Val}_G^\rho(\text{fresh}) = \text{TRM}(x) \) (\( x \notin G \))
- \( \text{Val}_G^\rho(\text{tac}) = \text{TAC}(\rho, \text{tac}) \)
- \( \text{Val}_G^\rho(\text{match } e \text{ with } p_i \Rightarrow e_i \text{ end}) = \begin{cases} \text{Val}_G^{\rho;\sigma}(e_i) & \text{if lazy} \\ \text{Eval}_G^{\rho;\sigma}(e_i) & \text{otherwise} \end{cases} 
\)
  where \( i, \sigma \text{ s.t. } \text{Val}_G^\rho(e) = \text{TRM}(\sigma(p_i)) \)
Evaluation of expressions

Head evaluation:
\[
\text{Eval}^\rho_G(E) = \begin{cases} 
\text{SGL}([\text{tac}]^\rho' \ G) & \text{if Val}^\rho_G(E) = \text{TAC}(\rho', \text{tac}) \\
\text{Val}^\rho_G(E) & \text{otherwise}
\end{cases}
\]

Execution of tactics:
\[
\begin{align*}
[e_1; e_2]^\rho &= \text{then } [e_1]^\rho \ [e_2]^\rho \\
[e_1||e_2]^\rho &= \text{orelse } [e_1]^\rho \ [e_2]^\rho \\
[\text{apply } T]^\rho &= \text{apply } \rho(T)
\end{align*}
\]

Toplevel evaluation:
\[
[E]^\rho(G) = \vec{g} \quad \text{if } \text{Eval}^\rho_G(E) = \text{SGL}(\vec{g})
\]
Example

Proving True twice:

- let f := let H := fresh in assert(H:=I) in f; f

Semantics:

fun g → let f = let h = fresh g in
  fun g → assert(h,I) g in
  then f f g

We’d rather have:

fun g → let f g = let h = fresh g in
  assert(h,I) g in
  then f f g

B. Barras

The Dark Side of $L_{\text{tac}}$
Summary of issues

- Error handling
- Executing a tactic *and* returning a result
- Controlling *when* a tactic is executed
Error handling

Promote to the expression level:

- fail, ||, first
- idtac
Tactics with an output

Several choices:

- cf Arnaud Spiwack’s new proof engine ($\approx$)
  
  ```
  type +'a tactic = goal list -> ’a * goal list
  ```

  It’s a (state) monad

- Subgoals as **threads**:
  
  ```
  type +'a tactic = goal list -> (’a * goal) list
  ```

  Tactics: side-effect on a local variable (goal)
  (Shared memory: evars)
Subgoals = Threads

```coq
case (l:list); intros;
[ tac
| fun x l' => tac' ].
```

- Separation of logical and naming aspects of \texttt{intro}.
- Implementation of a non-logical stack of arguments.
Now, executing tactics in argument position makes sense. So we need a way to freeze execution of tactics:

```
let H := intro in ...
```

vs

```
let H := 'intro in ...
```

(We also need a syntax to force the execution)
Conclusions

- $\mathcal{L}_{\text{tac}}$ has surprising (though simple) semantics
- Dichotomy LCF/$\mathcal{L}_{\text{tac}}$ awkward
- Control of execution returned to the user
- Tactics with a result are flexible
- New paradigm for passing non-logical arguments