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

Bruno Barras

ADT - Coq

30/06/09

Why $\mathcal{L}_{tac}$?

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

- LCF tactics

```
type tactic =
   goal → (goal list * (proof list → proof))
apply : term → tactic
```

- Tacticals

```
then : tactic → tactic → tactic
```

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

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

Features of $\mathcal{L}_{tac}$

- Untyped
- Functional (exceptions, no mutable variables)
- Call-by-Value
- Toplevel side-effects (Ltac $::=$)

- Dedicated operators (match with backtrack)
- Goal passed implicitely
- Dynamic link (goal context)

Evaluation strategy

Example (A tactic that proves True)

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

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` $\leadsto$ 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` $\rightsquigarrow$ 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)).
```

Mixing tactics and expressions

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

Example (A robust intro tactic)

```
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 || 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

Semantics

Expressions and tactics

$$E ::= x \mid \lambda x.e \mid e_1\, e_2 \mid \text{fresh} \mid T \mid tac \mid \text{match}\; e \;\text{with}\; p_i \Rightarrow e_i \;\text{end}$$
$$tac ::= \text{idtac} \mid \text{fail} \mid e_1 || e_2 \mid e_1 ; e_2 \mid \ldots \quad \text{(i.e. all LCF tactics)}$$

Values

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

Two semantics

- $\text{Val}_G^\rho(E)$ (evaluation as an argument: tactics delayed)
- $\text{Eval}_G^\rho(E)$ (head evaluation: tactics applied to goal)

Evaluation of expressions

λ core:

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

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:

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

  where i,σ s.t. $\mathsf{Val}_G^{\rho}(e) = \mathsf{TRM}(\sigma(p_i))$

Evaluation of expressions

Head evaluation:

- $\text{Eval}^{\rho}_{G}(E) = \begin{cases} \text{SGL}([tac]^{\rho'}\ G) & \text{if } \text{Val}^{\rho}_{G}(E) = \text{TAC}(\rho', tac) \\ \text{Val}^{\rho}_{G}(E) & \text{otherwise} \end{cases}$

Execution of tactics:

- $[e_1; e_2]^{\rho} = \texttt{then}\ [e_1]^{\rho}\ [e_2]^{\rho}$
- $[e_1 || e_2]^{\rho} = \texttt{orelse}\ [e_1]^{\rho}\ [e_2]^{\rho}$
- $[\text{apply}\ T]^{\rho} = \texttt{apply}\ \rho(T)$

Toplevel evaluation:

- $[E]^{\rho}(G) = \vec{g} \qquad \text{if } \text{Eval}^{\rho}_{G}(E) = \text{SGL}(\vec{g})$

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
```

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

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

- Separation of logical and naming aspects of `intro`.
- Implementation of a non-logical stack of arguments.

Quote

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

- $\mathcal{L}_{tac}$ has surprising (though simple) semantics
- Dichotomy LCF/$\mathcal{L}_{tac}$ awkward

- Control of execution returned to the user
- Tactics with a result are flexible
- New paradigm for passing non-logical arguments