A New Type For Tactics

Andrea Asperti <asperti@cs.unibo.it>
Wilmer Ricciotti <ricciott@cs.unibo.it>
Claudio Sacerdoti Coen <sacerdot@cs.unibo.it>
Enrico Tassi <tassi@cs.unibo.it>

University of Bologna

PLLMS, August 2009

Outline

1. LCF Tactics and their Limitations
2. State of the art
3. Our Proposal

Outline

LCF Tactics

```
type thm
type proof = thm list -> thm
type goal = form list * form
type tactic = goal -> (goal list * proof)
```

Lack of Metavariables

The validation model above does not handle metavariables and unification.
Goals may not contain unknowns to be instantiated later.

$$[E_1 \leq E_2] \xrightarrow{\text{transitivity}} [E_1 \leq ? \,;\, ? \leq E_2] \xrightarrow{\text{successor}} [S(E_1) \leq E_2]$$

```
type thm
type proof = thm list -> thm
type goal = form list * form
type tactic = goal -> (goal list * proof)
```

Locality

No automated global reasoning (à la constraint programming):

$[?_y \leq d\ ;\ a \leq ?_x + ?_y \leq b\ ;\ c \leq ?_x - ?_y]$

Coq's $\mathcal{L}$-tac:

- a high-level language to exploit domain specific knowledge
- can figure out a local strategy pattern matching the sequent
- can not pattern match all the goals at once to figure out the global strategy

No Partial Code Extraction / Proof Rendering

```
type thm
type proof = thm list -> thm
```

Code extraction

- possible for complete proofs (maps $[] \mapsto \pi$ where π : `thm`)
- not possible for partial proofs (maps $l \mapsto \pi[l]$)

Difficult to check if the proof is following the wanted (e.g. computationally efficient) algorithm:

Code extraction for complete proofs is too late!

Unstructured Scripts

```
thens_tactical: tactic -> tactic list -> tactic

intro n; elim n;
 [ simplify; reflexivity;
 | intro H; rewrite > H; auto; ]
```

- requires multiple undo-redo to be written
- fragile, difficult to fix when it breaks
- difficult to understand
- leads to unstructured scripts

Implementation of Declarative Languages

```
we proceed by induction on n to prove P(n)
 case S m: ...
 case O: ...
```

The tactic (here `case S m:`) chooses the goal.

```
P_0 by ... (H)              []
P_1 by ...                  []
then P_2 by H               [P_1]
and P_3                     []::[P_2]
hence P_4                   [P_2, P_3]
```

An accumulator is used to chain forward reasoning steps, passing information to tactics applied next.

Unclassified Goals

No way to tag goals:

- goals that are side conditions
- goals to be proved automatically
- goals to be postponed (e.g. PVS subtyping judgements)
- goals subject to a rippling procedure
- ...

The tag needs to carry informations, e.g.:

- a rippled goal needs to carry the inductive hypothesis and the rippling direction
- a goal to be proved automatically may carry the set of facts to be used

Outline

1. LCF Tactics and their Limitations
2. State of the art
3. Our Proposal

HOL-Light

```
type thm =
  Sequent of (term list * term)  (* hyps, concl *)
type justification =
  instantiation -> thm list -> thm
type goalstate =
   (term list * instantiation)
  * goal list * justification
type tactic = goal -> goalstate
```

Metas	Global reasoning	Proof inspection	Structured scripts	Declarative language	Tagged goals
√					

Coq

```
type tactic =
  goal sigma -> (goal list sigma * validation)
and validation = (proof_tree list -> proof_tree)

type proof_tree = {
  open_subgoals : int;
  goal : goal;
  ref : (rule * proof_tree list) option }
and rule = ...
```

? = using additional data structures

Metas	Global reasoning	Proof inspection	Structured scripts	Declarative language	Tagged goals
√					

MetaPRL

```
type tactic =
  sentinal -> msequent -> msequent list * ext_just
type msequent_so_vars =
  SOVarsDelayed | SOVars of SymbolSet.t
type msequent = {
  mseq_goal : term;
  mseq_assums : term list;
  mseq_so_vars : msequent_so_vars ref;
}
type ext_just =
  | RuleJust of ...
  | RewriteJust of ...
  ...
```

Metas	Global reasoning	Proof inspection	Structured scripts	Declarative language	Tagged goals
√		√			√

Matita 0.x

```
type proof =
 uri option * metasenv * substitution *
   term Lazy.t * term * attribute list
type goal = int
type metasenv = (goal * term list * term) list
type substitution=(goal * term list * term * term) list
type status = proof * goal

type tactic
val mk_tactic: (status -> proof * goal list) -> tactic
```

Metas	Global reasoning	Proof inspection	Structured scripts	Declarative language	Tagged goals
√		√			

Isabelle-Pure

```
datatype thm = Thm of
 deriv *                                    (*derivation*)
 {thy_ref: theory_ref,             (*reference to theory*)
  tags: Properties.T,         (*additional annotations*)
  maxidx: int,                     (*max idx of Var TVar*)
  shyps: sort OrdList.T,             (*sort hypotheses*)
  hyps: term OrdList.T,                   (*hypotheses*)
  tpairs: (term * term) list,    (*flex-flex pairs*)
  prop: term}                             (*conclusion*)
and deriv = Deriv of
 {max_promise: serial,
  open_promises: (serial * thm future) OrdList.T,
  promises: (serial * thm future) OrdList.T,
  body: Pt.proof_body};
type tactic = thm -> thm Seq.seq
```

Metas	Global reasoning	Proof inspection	Structured scripts	Declarative language	Tagged goals
√	√	√		√	

Outline

Our Proposal

```
type proof_object

type goal
type metasenv = (goal * term list * term) list

type proof_status = metasenv * proof_object
type tac_status = {
  pstatus : proof_status;
  gstatus : context_stack;
}
type tactic = tac_status -> tac_status
```

Metas	Global reasoning	Proof inspection	Structured scripts	Declarative language	Tagged goals
√	√	?	√	√	√

The Context Stack

```
type task =
  int * [ `Open | `Closed ] * goal * [> `No_tag ]
type context = task list * task list
type context_stack = context list
```

Metas	Global reasoning	Proof inspection	Structured scripts	Declarative language	Tagged goals
√	√	?	√	√	√

Example

tactic	focused	postponed	tail
	[(0,O,?22)]	[]	[]
exists	[(0,O,?38);(0,O,?39)]	[]	[]
[	[(1,O,?38)]	[]	[([(2,O,?39)],[])]
2:	[(2,O,?39)]	[[(1,O,?38)]]	[([],[])]
assumption	[]	[[(1,C,?38)]]	[([],[])]
]	[(1,C,?38)]	[]	[]

Embedding LCF tactics

Most tactics operates on a single goal.

```
type lcf_tactic =
  proof_status -> goal -> proof_status

distribute_tac: lcf_tactic -> tactic
exec: tactic -> lcf_tactic

exec (distribute_tac lcf_tac) s g = lcf_tac s g
```

Distribute_tac

```
let distribute_tac tac status =
 match status.gstatus with
 | [] -> assert false
 | (g, t) :: s ->
    let rec aux s go gc = function
     | [] -> s, go, gc
     | (_,_,n,_) :: loc_tl ->
        let s, go, gc =
         (* a metavariable could have been closed by side effect *)
         if n \in gc then s, go, gc
         else
           let sn = tac s n in
           let go',gc' = compare_statuses s sn in
           sn,((go \cup [n]) \setminus gc') \cup go',gc \cup gc'
        in
        aux s go gc loc_tl
    in
    let s0, go0, gc0 = status.pstatus, [], [] in
    let sn, gon, gcn = aux s0 go0 gc0 g in
    (* deep_close set all instantiated metavariables to `Close *)
    let stack = (gon, t \setminus gcn) :: deep_close gcn s
    in { gstatus = stack; pstatus = sn }
```

Exec

```
let exec tac pstatus g =
  let stack = [ [0, `Open, g, `No_tag ], [] ] in
  let status =
   tac { gstatus = stack ; pstatus = pstatus }
  in
   status.pstatus
```

The block tactic

```
let block_tac l status =
 fold_left (fun status tac -> tac status) status l
```

The LCF tactical `thens` is simply implemented as:

```
let thens_tac t tl =
 block_tac (t :: `[` :: separate `|` tl @ `]`)
```

where `separate `|` [ t_1 ; ... ; t_n ]` is
`[ t_1 ; `|` ; ... ; `|` ; t_n ].`

Conclusions

- few literature
- common misconception about LCF data types
- studying an overcoming their limitations
- our proposal for Matita 1.0