A new type for tactics

Andrea Asperti Wilmer Ricciotti Claudio Sacerdoti Coen Enrico Tassi
Department of Computer Science, University of Bologna
asperti@cs.unibo.it, ricciott@cs.unibo.it, sacerdot@cs.unibo.it, tassi@cs.unibo.it

Abstract

The type of tactics in all (procedural) proof assistants is (a variant of) the one introduced in LCF. We discuss why this is inconvenient and we propose a new type for tactics that 1) allows the implementation of more clever tactics; 2) improves the implementation of declarative languages on top of procedural ones; 3) allows for better proof structuring; 4) improves proof automation; 5) allows tactics to rearrange and delay the goals to be proved (e.g. in case of side conditions or PVS subtyping judgements).

Categories and Subject Descriptors F [*4*]: 1—Mechanical theorem proving

General Terms tactics, type

Keywords implementation, tactics, proof assistants

1. Overview

A proof assistant is a system that allows the user to interactively prove a theorem by entering commands, called *tactics*, that allow to reduce the initial conjecture to new, simpler ones until all conjectures are trivially proved. Conjectures, which are also called *goals*, can be described as sequents, i.e. pairs formed by the list of hypotheses to be used and the local thesis to be proved. A proof assistant must provide a data type to represent on-going proofs and a type for tactics.

Tactics were introduced for the first time in the LCF theorem prover [Gordon et al. 1979], historically one of the first and one of the most influential proof assistant ever developed. While every system implements its own data type for proofs and many alternative solutions have been proposed, most current systems like HOL-Light [HOL-Light], NuPRL [NuPRL], MetaPRL [Hickey et al. 2003], Coq [Barras B. et al. 1997, Coq] and Matita 0.x [Asperti et al. 2007, Matita] still use minor variants of the LCF representation for tactics. In Section 2 we discuss several limitations of the LCF representation: some of them have been addressed in most of the cited systems, but others have not been addressed so far, possibly with the exception of Isabelle [Isabelle]. In Section 3 we propose and discuss a new type for tactics that solves the previous limitations and that will be used in the next major release of the Matita interactive theorem prover. We draw conclusions in Section 4.

2. LCF tactics and their limitations

The representation of tactics in the LCF proof assistant is well known and it can be briefly described as follows (see, for instance, [Gordon et al. 1979], page 210):

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

A goal is a pair formed by a context (a list of formulas that are the hypothesis) and a formula that is the thesis. A tactic can be applied to just one goal and returns both a list of new goals to be proved and a "proof". Intuitively, the tactic reduces the goal to a possibly empty list of simpler goals and asserts the existence of a "procedure" to build a proof object (represented in LCF by the type `thm`) of the goal from the proofs of the subgoals. This procedure has type `proof`, i.e. it is an actual ML function from a list of proofs (`thms`) to a proof `thm`. The `thm` data type is abstract: only functions (i.e. tactics) defined in the ML module can directly construct inhabitants of `thm`, while functions defined outside the module can only combine tactics to build proofs. Thus, if the tactics defined in the module are correct, i.e. they implement sound logical rules, all the system is granted to be correct. This robustness property is so attractive that the LCF approach has become pretty standard.

What is actually stored in the `thm` data type is unspecified in the "LCF approach": it could range from just the proposition that is proved (if we are only interested in provability) to a proof term that is a trace of the proof (if we are interested in inspecting and manipulating the proof, e.g. for proof extraction or independent checking). Nevertheless, the `thm` data type can only represent completed proofs (hence the name `thm` that stands for "theorem"). During interactive proof construction, an ongoing proof will be represented only by an ML function from a list of `thms` to a `thm`. Such function is obtained by composing together the second component of the return type of the tactics used so far. Being a function, it cannot be inspected or modified in any way.

The LCF data types we have presented are not sufficient alone to fully represent the state of the system between tactics application, i.e. when further input is required to the user. The system needs to store somewhere the set of goals currently open and the function that represents the on-going proof. Moreover, since a tactic can be applied by design only to a single goal, it must also single out one of the opened goals, called the *focused goal*, that will be the argument of the next tactic.

LCF also introduced the notion of *tactical*, which is an higher-order tactic. Tacticals are used to build complex, ramified proofs from given tactics by applying tactics according to some strategy. Since the first tactic application can open more than one sequent, during a tactical application we also have the notion of *current goals*, which usually are the new goals recently opened by tactics application during the execution of the tactical. In particular, a

tactical must decide the order in which current goals get the focus and the way goals opened by different focused goals are merged together in the set of all current goals. Since the LCF types do not allow to represent these intermediate states, the implementation of the different systems either record this information in the `thm` data type, or leave this information implicit in the control flow data structures (e.g. the stack) of the code that implements the tacticals. In his PhD thesis [**?**], Kirchner has described an elegant monad, called proof monad, that allows to lift LCF tactics to tactics and tacticals working on the enriched representation.

2.1 Limitations

We now present several limitations of the LCF data type that have consequences on the class of tactics that can be implemented, on the proof language and on the user interface.

(i) Lack of metavariables Most modern proof assistants allow existentially quantified metavariables to occur in formulae. Metavariables stand for terms that are currently unknown and that will be instantiated later on, usually by means of unification. They arise in three different situations. The first one is when they correspond to implicit, not fully constrained information in a formula, e.g. when the infix notation "$\_ + \_$" is used for the operation of an unknown semi-group in the expression $\forall x, y.x + y = y + x$, that is interpreted as $\forall x, y : ?_G.x +_{?_G} y = y +_{?_G} x$ (where $?_G$ is a metavariable to be instantiated later). The second one is when the user apply backwards a deduction rule, like $\exists$-introduction, but prefers to delay the choice of the witnesses as much as possible, in the spirit of constraint programming. The third situation generalizes the previous one and is obtained when a deduction rule, e.g. transitivity, is matched (or unified) against the goal, and some metavariables remain free.

Metavariables are not compatible with the LCF data type, since a metavariable can be instantiated by one tactic and the instantiation must be applied to every formula in every goal. The latter operation cannot be performed by the tactic, since it takes in input only the focused goal and not the set of all goals. The observation is not novel and can be explicitly found, for instance, in [**?**] where Paulson writes "the validation model above does not handle unification. Goals may not contain unknowns to be instantiated later. As a consequence, the LCF user must supply an explicit term at each $\exists$:right step".

In Section 2.2 we will see how all the major proof assistants have extended the LCF types to add metavariables. In the meantime, we notice how the already cited work by Kirchner [**?**] fails to explicitly take metavariables in account.

(ii) All tactics are local A direct consequence of existential metavariables is that a "wrong" instantiation of one of them can make a different goal false, hence not provable. As an example, consider two goals generated by a transitivity law: $\Gamma \vdash a \leq ?_x$ and $\Gamma \vdash ?_x \leq b$. Here $?_x$ stands for the intermediate (still unknown) term c that makes proving $\Gamma \vdash a \leq c$ and $\Gamma \vdash c \leq b$ easier than proving $\Gamma \vdash a \leq b$. If a tactic (especially an automatic one) has a local view over the set of open conjectures (i.e. knows just the goal $\Gamma \vdash a \leq ?_x$), it is unlikely to find an instantiation for $?_x$ that makes proving $\Gamma \vdash ?_x \leq b$ simpler or even possible (think for example to the trivial but useless solution $?_x := a$ that is obtained proving the first goal with the reflexive property of $\leq$). More to the point, restricting the view of tactics to a single input sequent, gives them not enough information to detect valid but pointless instantiation of metavariables. With the exception of Isabelle, all other major proof assistants that have accommodated metavariable still see a tactic as a function whose input is just one focused goal and thus does not allow the implementation of non-local tactics in the spirit of constraint programming.

A different motivation for introducing non-local tactic is given by the wish to extend the user-level $\mathcal{L}$-tac language of Coq [Delahaye and di Cosmo 1999] with a pattern matching construct over the set of all goals, in the spirit of what is allowed over contexts. $\mathcal{L}$-tac allows the lightweight definition in a script file of ad-hoc tactics that match certain configurations and proceed in the proofs exploiting the domain-knowledge. It would be useful to detect global configurations in order to look for goals that have certain shapes (e.g. can be closed using decision procedures or are more likely to be false or are all instances of a more general conjecture).

(iii) No partial code extraction Opaque proofs (i.e. ML functions) prevent many interesting elaborations the system can perform even if they are incomplete. For instance, proofs made in constructive logic do have an algorithmic content, that is hard to grasp without a proper cleanup procedure (that essentially amounts to the erasure of non computationally relevant parts). This procedure can, at some extent, be performed even on incomplete proofs, giving the user a feedback on the extracted code at early stages of the proof development. This is important if the user is looking for a particular implementation (i.e. an efficient one) of the specification.

(iv) On-going proofs cannot be rendered Similarly to the lack of partial code extraction, it is impossible to render a proof in progress to the user, e.g. as a derivation tree (or DAG) or in pseudo-natural language.

(v) Unstructured scripts Most modern interactive theorem provers do offer a user interface based on a textual script, input by the user, that is step-by-step checked by the system. The checked part is locked: no edit can be performed on that part without retracting the checked commands (i.e. an undo operation affecting the status of the ongoing proof is performed).

This interaction paradigm suffers the big step execution semantics of LCF tacticals, that are still today the primary tool to combine together tactics and give a structure to proof scripts. The big step semantics of tacticals is forced by their type. Being higher-order tactics, they can be executed only when all their arguments are provided. Or better, there is no semantics for the tactical if some of its arguments are unknown.

For example, when a tactic opens heterogeneous goals the user may want to use the branching tactical (`[ ... | ... | ... ]`) to run appropriate tactics on every branch. Since it is unlikely that he is able to fill all the blanks (i.e. `...`) in a row, he is forced by the system to continuously refine its compound command, execute it to see the result, and retract to able to further refine it. This loop is not only annoying, requiring additional key-presses or mouse clicks: it also forces the user to type the refined command in a blind way, since in order to edit the script he must ask the system to retract the last command, and this operation also changes the displayed proof status. Thus the user has to type the next command step looking at the goal in the state it was many steps before.

Structuring the proof script makes it more easy to fix when it breaks, since the structure of the proof is more explicit. For example failures are detected early since new goals coming from the application of modified lemmas pop up in the right part of the proof (i.e. they are no accidentally delayed). If the user interface refrains the user to give a proper structure to the proof script, it is unlikely that he would be happy to perform a major redesign of the axioms or basic definitions he is using, since this would break proof scripts, and fixing them would be a very expensive operation.

Another strong point against a big step evaluation semantics of operators to combine commands and structure scripts is that, unless the interaction language is declarative, just reading the proof script is not enough to re-read a proof: single commands have to be executed step-by-step to understand what is going on. In system equipped with standard (big-step executed) tacticals, what

is usually done is (in the rare case in which the proof is structured) to de-structure the proof on the fly, modifying the proof script in such a way that only a part of every compound command is executed. Re-reading a proof script is not only necessary during talks or demos, but is the main activity a team member performs when fixing a script he is not the author of (i.e. that he is not supposed to deeply understand). Given that the cost of writing a formal, mechanically checkable, proof is very high, we believe that every design choice that makes collaboration on the formalization activity harder is to be avoided.

The execution of a partially given tactical can be performed if a semantics is given to a de-structured language for tacticals as in [Sacerdoti Coen et al. 2006]. For example, a data structure (similar to the stack that is used to execute functions in a regular programming language) can be explicitly adopted to allow the user to type just the command '`[`' and see its result, then use a tactic, then move to the following goal typing the command '`|`' and after he is done with the proof branches type the command '`]`'.

(vi) Poor implementation of declarative languages The languages of proof assistants are often classified between declarative and procedural ones.

In procedural languages, the user uses tactics that specify how the goal must be manipulated, but not what is expected from the manipulation. Intuitively, it corresponds to the information that remains in a derivation tree by erasing from the premises of each rule all (sub)-formulas that also occur in the rule conclusion. Most tactics for procedural languages are used to find proofs in a top-down way, since the amount of information that can be omitted is maximized in this way. However, tactics for bottom-up reasoning (like tactics to generate logical cuts) can also be present, but are usually more verbose.

In declarative languages, the user uses commands (that we identify with tactics) to build proofs by specifying what is proved at each step, usually omitting how it is proved. Automation supplies the missing justifications. Intuitively, it corresponds to the information that remains in a derivation tree when the name of every rule is omitted and only the tree structure and the formula are kept. Most tactics for declarative languages are used to describe proofs in a bottom-up way. Not every declarative language has tactics for top-down proof steps, but at least case analysis and induction are better captured in this way.

A long standing line of research [Harrison 1996, Wiedijk 2003] has tried to implement declarative languages, i.e. declarative tactics, on top of procedural ones. However, the results so far have never been completely satisfactory for two different reasons.

The first has to do with goal selection: when the user needs to prove multiple goals (e.g. the branches of a proof by induction, or the components of a conjuction), declarative languages like Isar [**?**] allow to dynamically select the goal to be proved, or even to prove something that is matched with the opened goal set only later, and possibly up to some easy deductions. Procedural tactics, together with the LCF limitation of just one focused goal, do not allow to implement properly this behaviour. Note that this is exactly the type of control over the proof history that tinycals allow.

The second has to do with information flow: in languages like Isar and Mizar a forward reasoning tactic can prove some fact and at the same time schedule it for usage by the tactic that ends the subproof. The latter tactic can only use the facts explicitly listed by the user in addition to those accumulated by previous tactics. The LCF data type does not allow to pass information around from one tactic to the next ones.

(vii) Unclassified goals For a formal system, every conjecture is the same: a set of hypotheses and a conclusion. This is reflected by the LCF type for tactics, where the only distinction between newly generated goals is their position in the output list. For example, when we proceed by induction, we know that some of the new conjectures will need the application of the inductive hypothesis to be solved, while other goals do not. This information is lost in the coarse LCF tactic type, but could be exploited by the system, for example, automatically running procedures like rippling on all inductive cases.

Another example where some new goals deserve a special treatment is generalized rewriting (rewriting with setoids). In that case, a rewriting step generates goals of two kinds: the rewritten conclusion, and a proof that the context under which the rewriting took place is made of morphisms. The latter class of goals can usually be solved automatically, once the user proved that every elementary functional symbol is a morphism.

Some interactive provers, most notably PVS and ACL2, collect sets of side conditions (like subtyping judgements) the user is *not* expected to immediately solve. Most of these side conditions become trivial when the user enriches the context with additional facts or assumptions and are thus temporarily put apart by these systems.

2.2 Tactics in current proof assistants

Coq adopts a type for tactics which is very close to the LCF one. Here `sigma` is the type for metavariables indexed by the type of focused goals (i.e. a tactic runs on a goal and generates a list of new ones). The `validation` type is opaque (being a function) and is aimed at producing a proof-tree like representation of the proof.

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

The proof tree type is complicated by the fact that Coq implements metavariables, thus a proof may be incomplete. Note the number of open subgoals and the optionally applied rule equipped with the list of sub-trees.

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

Coq solves the limitation (i) but not (ii), (v), (vi). Limitations (iii), (iv) and (vii) are solved by working on additional data structures.

Isabelle-Pure adopts the most complex, but flexible, type for tactics, which is quite far away from the LCF data type since a "proof context" for the whole proof is passed around. A small part of the complexity is needed to cope with the logical-framework approach (i.e. it is abstracted over the theory selected by the user).

```
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 conv = cterm -> thm;

type tactic = thm -> thm Seq.seq
```

Note that a tactic returns a (possibly infinite) sequence of (not yet proved) theorems. This allows an elegant and lightweight implementation of backtracking, for example associated to impossibility of finding a most general unifier but the possibility of enumerating all second-order ones. Also note that tactics do not take in input a focused goal: every tactic must know which goal(s) to work on.

Isabelle solves the limitation (i), (ii), (iii), (iv) and (vi). Limitation (v), (vii) are solved by working on additional data structures.

HOL-Light extends the LCF type for tactics keeping track of (meta)variables instantiations. Note that a tactic is allowed to generate new variables (the `term list` component of the goal state).

```
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
```

HOL-Light solves (i) but not (ii), (iii), (iv), (v), (vi), (vii).

MetaPRL adopts a concrete representation of proof steps (here called `ext_just`) but is more conservative about tactics input, that is a single sequent as in LCF.

```
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 ...
  ...
```

MetaPRL solves (i), (iii), (iv), (vii) but not (ii), (v), (vi).

Matita 0.x adopts a type similar to the LCF one, but the proof object `proof` for the whole proof is always passed around and can be concretely inspected. Metavariable instantiation is done lazily by recording the instantiation in the `substitution` data type, which is a map from instantiated metavariables to their values. Tinycals are implemented by means of an additional data structure, called context stack [Sacerdoti Coen et al. 2006], which is not accessible by tactics. Thus tinycals are not tactics and the proof object `proof` does not capture the whole proof status.

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

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

Matita 0.x solves (i), (iii) and (iv) but not (ii), (vi) and (vii). It also solves (v) thanks to additional data structures.

3. A new type for tactics

We propose a new type for tactics that is a function from and to a `tac_status` defined as follows:

```
type proof_object
type metasenv = goal list

type proof_status = metasenv * proof_object

type tac_status = {
  pstatus : proof_status;
  gstatus : context_stack;
}

type tactic = tac_status -> tac_status
```

A `tac_status` is made of a proof status `pstatus` and a context stack `gstatus`. The proof status component carries a (partial) proof object made of a set of open goals and existentially quantified metavariables (`metasenv`), and a data type for partial proofs. In our proposal, goals and existentially quantified metavariables are handled uniformly, for instance by showing all of them to the user as goals and by allowing tactics to either instantiate a metavariable (with a term) or a goal (with a proof).

The context stack is meant to keep track of proof structuring commands, like reordering, focusing, postponing or tagging goal sets. The `context_stack` data structure was introduced in [Sacerdoti Coen et al. 2006] to give a semantics to tinycals, which are a syntactically de-structured version of some LCF tacticals equipped with a small step semantics. The context stack that equips the proof object plays the same role of the indexed proof tree in [**?**].

The major difference of our type for tactics with the standard LCF one is that the input is no longer a single goal, but a global view of the ongoing proof which can be altered. Moreover, it is possible to focus simultaneously on a set of goals. For example a tactic could make no progress (in terms of closing open goals) but it could change the focus to the set of goals in which an existentially quantified metavariable occurs; then another tactic performing automatic proof search could be run on the focused goal set to find an instantiation for the metavariable that allows to solve all goals simultaneously.

We describe now an implementation of the `context_stack` which is both a simplification and an extension of the one in [Sacerdoti Coen et al. 2006] and that allows to implement the most interesting tinycals described there. The code excepts are in (pseudo) OCaml syntax.

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

The context stack is a stack of contexts, which are pairs of focused (Γ) and locally postponed (τ) tasks. A task is a numbered and tagged goal that can be either open or already closed. The tag is used to associate arbitrarily information to a goal, e.g. to mark it for automation. The number will be discussed in a while. A goal can be present in the stack only once.

The intended semantics of the stack is that a tactic should normally act only on the focused task in the context at the top of the stack, replacing the focused tasks with the new tasks opened by the tactic.

This behaviour is illustrated by the following pseudo-tactic:

```
let pure_tac status =
 let focused, t, s =
  match status.gstatus with
  | [] -> assert false
  | (g, t) :: s -> map (fun _,_,x,_ -> x) g, t, s
 in
 let newpstatus, newgoals =
   do_something_useful status.pstatus focused
 in
 let g =
  map (fun x -> 0, `Open, x, `No_tag) newgoals
 in
  { pstatus = newpstatus; gstatus = (g, t) :: s }
```

The initial stack is the following, where g is the original goal stated by the user.

```
[ [0, `Open, g, `No_tag ], [] ]
```

A new context is pushed on the stack by the branching tinycal ([) that is used in order to be able to re-focus only on subsets of the focused tasks Γ_o, for instance to apply different tactics to each goal. When the branching tinycal is used, the tasks in Γ_o are re-numbered with their position in the list Γ_o. Initially, the new stack is made of just one focused task, which was the first previously focused one (i.e. the task in Γ_o numbered with 1). The new focused goal is removed from Γ_o.

```
let branch_tac status =
 let new_gstatus =
  match status.gstatus with
  | [] -> assert false
  | (g, t) :: s ->
     match init_pos g with (* numbers goals *)
     | [] | [ _ ] -> fail
     | task :: tl -> ([task], []) :: (tl, t) :: s
 in
  { status with gstatus = new_gstatus }
```

The user can stop working on that task and move to the next one using the shift tinycal (|) that moves what is currently focused (i.e. is in Γ) on the postponed list τ at the top of the stack, and moves the next previously focused task in Γ_o to Γ. Alternatively, he can use the focusing tinycal (`n_1,\ldots,n_m:`) to stop working on the currently focused tasks and focusing on the task in Γ_0 numbered by $n_1, \ldots, n_m$. A shortcut is the remaining tinycal (`*:`) that focus on all remaining tasks in Γ_o.

```
let shift_tac status =
  let new_gstatus =
    match status.gstatus with
    | (g, t) :: (g', t') :: s ->
          (match g' with
          | [] -> fail
          | loc :: loc_tl ->
                (([ loc ], t ∪ filter_open g ∪ k)
                :: (loc_tl, t') :: s))
     | _ -> fail
  in
    status with gstatus = new_gstatus

let pos_tac i_s status =
  let new_gstatus =
    match status.gstatus with
    | [] -> assert false
    | ([ loc ], t) :: (g', t') :: s
      when is_fresh loc ->
        let l_js =
          filter (fun i,_ -> i ∈ i_s) ([loc] ∪ g')
        in
         ((l_js, t)
           :: (([ loc ] ∪ g') \ l_js, t') :: s)
    | _ -> fail
  in
    status with gstatus = new_gstatus

let wildcard_tac status =
  let new_gstatus =
    match status.gstatus with
    | [] -> assert false
    | ([ g ] , t) :: (g', t') :: s ->
            (([g] ∪ g', t) :: ([], t') :: s)
    | _ -> fail
  in
    status with gstatus = new_gstatus
```

The merge tinycal (]) pops the context at the top of the stack by appending all tasks in the popped context to the focused list Γ at the new top of the stack.

```
let merge_tac status =
  let new_gstatus =
    match status.gstatus with
    | [] -> assert false
    | (g, t) :: (g', t') :: s ->
        ((t ∪ filter_open g ∪ g' ∪ k, t') :: s)
    | _ -> fail
  in
    status with gstatus = new_gstatus
```

Note that the composition of '[', multiple '|'s and ']' is semantically equivalent to the "thens" LCF tactical.

We have not discussed so far the use of `Closed` tasks. Since we use goals (hence tasks) to represent also metavariables, a tactic can instantiate a metavariable which is not currently focused and thus can be anywhere in the context stack. In this case, we mark the task as `Closed` so that, when the user will later focus on it, he will be aware that the goal has already been automatically closed by side effects. The only tactic which works on closed tasks is the `skip` tinycal that just removes the task by leaving in the script an acknowledgement (the `skip` occurrence) of the automatic choice.

```
let skip_tac status =
 let new_gstatus =
  match status.gstatus with
  | [] -> assert false
  | (gl, t) :: s ->
     let gl = map (fun _,_,x,_ -> x) gl in
     if exists ((=) `Open) gl then fail
     else ([], t) :: s
 in
  { status with gstatus = new_gstatus }
```

Other tinycals described in [Sacerdoti Coen et al. 2006] require a slightly more elaborate context stack. The most interesting ones are the pair `focus`/`unfocus` that allows to focus on an arbitrary subset of the goals, whereas the focusing tinycals we have described only allows to focus on a subset of the tasks that were focused when `[` was most recently used.

3.1 LCF-like tactics

Tactics presented so far can freely manipulate the context stack. For instance, tinycals are just tactics that change the stack without changing the goals. However, the most frequent case for a tactic is still that acts locally on a single focused goal, and does not care

about focusing or postponement of the generated goals. For this reason we introduce a simplified type that corresponds to the LCF type extended to care for metavariables.

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

An `lcf_tactic` takes as input a proof status and the focused goal (that must belong to the proof status) and returns a new proof status. The list of new goals can be computed by comparing the metasenv in input and in output. Passing the metasenv (and proof object) around allows the tactic to instantiate metavariables all over the proof. The justification for the tactic is recorded in the proof object. Since we put no requirements on the latter, we are free to implement it either as an ML function or as a concrete, inspectable, data structure like a λ-term if our system is based on the Curry-Howard isomorphism.

An `lcf_tactic` can be lifted to a `tactic` by applying it in sequence to each focused goal, collecting all the opened goals and turning all of them into the new focused goals on top of the stack. This is implemented by the `distribute_tac` tactic:

```
(* distribute_tac: lcf_tactic -> tactic *)
let distribute_tac tac status =
 match status.gstatus with
 | [] -> assert false
 | (g, t) :: s ->
    (* aux [pstatus] [open goals] [close goals] *)
    let rec aux s go gc =
     function
     | [] -> s, go, gc
     | (_,switch,n,_) :: loc_tl ->
        let s, go, gc =
         (* a metavariable could have been closed
          * by side effect *)
         if n ∈ gc then s, go, gc
         else
           let sn = tac s n in
           let go',gc' = compare_statuses s sn in
           sn,((go ∪ [n]) \ gc') ∪ go',gc ∪ 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 \ gcn) :: deep_close gcn s
    in
      gstatus = stack; pstatus = sn
```

When implementing a `lcf_tactic`, it is sometimes useful to call a `tactic` on one goal but, because of lack of the context stack, an `lcf_tactic` can only directly call another `lcf_tactic`. Therefore, we introduce the `exec` function to turn a tactic into an `lcf_tactic` by equipping the proof status with a singleton `context_stack` and by forgetting the returned context stack:

```
(* exec: tactic -> lcf_tactic *)
let exec tac pstatus g =
  let stack = [ [0, `Open, g, `No_tag ], [] ] in
  let status =
   tac { gstatus = stack ; pstatus = pstatus }
  in
   status.pstatus
```

The functions `exec` and `distribute_tac` form a retraction pair: for each proof status s and goal i,

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

They are inverse functions when applied to just one focused goal or alternatively when restricted to LCF-like tactics, i.e. tactics that ignore the context stack and that they behave in the same way when applied at once to a set of focused goals and to each goal in turn. Thus, we can provide a semantic preserving embedding of any LCF tactic into our new data type for tactics. Moreover, as proved in [Sacerdoti Coen et al. 2006], we can also provide a semantic preserving embedding of all LCF tacticals. In the current presentation, this is achieved by means of the `block_tac` that allows to execute a list of tactics:

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

For instance, 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]`.

4. Conclusions

We presented a new, simple data type for tactics that extends the LCF data type and that admits a semantic preserving embedding of LCF tactics into it. In Sect. 2 we discussed several limitations of LCF tactics that are overcome in our solution. We review them here:

Lack of metavariables In LCF, a tactic only takes in input a single focused goal and thus cannot change the other ones by side effects. In our proposal the input and output types of a tactic include the proof status, allowing the instantiation of a metavariable all over the proof status.

As a side effect, depending on the actual implementation of the proof status, it is possible to implement tactics that alter the proof status not only by instantiating metavariables, but also by changing sub-terms or sub-proof-objects (e.g. to improve a proof object that was found by an automatic tactic or to share duplicated parts of a proof). This behaviour can easily be prevented when the proof object is an abstract data type that only provides the metavariable instantiation method.

All tactics are local Since our tactics are applied to the whole proof status, they can reason globally on it, e.g. by using constraint programming techniques to instantiate metavariables constrained by multiple goals. The context stack also provides a list of focused goals tactics are supposed to act on, favouring a kind of reasoning that is intermediate between the global one and the local one of LCF. As in the previous case, abstract data types can be used to prevent global reasoning in favour of the intermediate one.

No partial code extraction The proof of a goal is not required to be an ML function as in LCF and thus it can be implemented with a concrete type of λ-terms to allow code extraction. The λ-calculus needs to be extended with metavariables in the spirit of [Muoz 1997, Geuvers and Jojgov 2002] to allow the representation of partial proofs and code extraction from them.

On-going proofs cannot be rendered In the same way, it is now possible to render the proof object or to analyze it in alternative ways. For instance, it is possible to data mine the proof object in search of duplicate sub-proofs.

Unstructured scripts In [Sacerdoti Coen et al. 2006] we introduced tinycals with the precise aim of improving the user interface of proof assistants by letting the user effectively write structured

scripts. There, tinycals were not a special kind of tactics, but additional commands to be interleaved with tactics. In this paper we present a uniform data type for tactics that blurs the distinction between tactics and tinycals, allowing the formers to manipulate the context stack (as tinycals do).

Poor implementation of declarative languages An application of the previous observation is given by the implementation of declarative commands to perform case analysis or induction. Suppose that we want to implement a language with the following commands (tactics):

```
by induction on T we want to prove P
by cases on T we want to prove P
case X (arg1: T1) ... (argn: Tn):
by induction hypothesis we know P (H)
```

A proof by cases or induction is started using one of the first two tactics and continues by switching in turn to each case using the third tactic, as in the following example:

```
by induction on n we want to prove n + 0 = n
 case O:
   ....
 case S (m: nat):
   by induction hypothesis we know m+0 = m (IH)
   ...
```

The user should be free to process the cases in any order. Thus the `case` tactic should first focus on the goal that corresponds to the name of the constructor used. However, since an LCF tactic can only work on the focused goal, focusing must be performed outside the tactic and the `case` command cannot be implemented as a tactic and thus it cannot be easily re-used inside other tactics. In our approach, the `by cases/induction` tactics open several new goals that are all focused at once and the `case` tactic simply works on the wanted case only by focusing on it.

Moreover, the semantics of declarative tactics are often based on a working set of justifications that are incrementally accumulated to prove the thesis. E.g. in the Isar-like declarative script

```
  n = 2 * m by H
moreover
  m * 2 = x + 1 by K
hence
  n = x + 1 by sym_times
```

the third inference is justified by the first two propositions and `sym_times`. Thus the semantics of `moreover` and `hence` is that of accumulating the justifications in some set which must be passed around in the proof status. The LCF data type for tactics does not allow to implement this set, whereas in our proposal the proof object can store any information. In particular, this kind of information is better stored in the context stack, e.g. in the form of tags.

Unclassified goals Goals are freely tagged in the context stack to attach information to them. A typical application consists in marking proofs of side conditions that the system should try to solve automatically, for instance by resorting to a database of ad-hoc lemmas as in the implementation of Type Classes in Coq by Sozeau [Sozeau and Oury 2008].

Our data type, while remaining quite simple, is general enough to accommodate all the information stored by other systems in their data types for tactics and it is strictly more general than the one of all other systems, with the exception of Isabelle/Pure. On the one hand, the latter is very similar to (an instance) of ours, since a tactic maps `thms` (i.e. a proof status) to `thms`, like in our proposal, and the information stored in a `thm` roughly comprises the one we use. On the other hand, an Isabelle/Pure tactic returns a sequence of `thms` in order to capture the non determinism of some tactics. The same solution can be adopted with minor effort even in our proposal.

The forthcoming major release of Matita (1.x) is based on the type for tactics described in this paper with minor differences. The `context_stack` is slightly more complex as in [Sacerdoti Coen et al. 2006] to support a larger number of tinycals. For the `proof_object` we use the Calculus of (co)Inductive Constructions with metavariables as described in [**?**]. The most interesting example of non local tactic is a prolog style proof search procedure where metavariables are extensively used to delay instantiations and backtracking is performed w.r.t. the whole set of goals a metavariable occurs in (see [Asperti and Tassi 2009]). A declarative language of tactics is built on top of the procedural one thanks to the possibility of dynamically focusing on goals; it also benefits of goal tagging.

Another recent attempt to provide a general data type for tactics is the one in Kirchner's PhD thesis [**?**] where the author introduces a proof monad on top of a generic data type for indexed proof trees, i.e. proof trees where goals are hierarchically structured and an index, which is a pointer in the tree, is used to specify the set of focused goal.

Kirchner's indexed proof trees play the same role of our context stack, with several differences. The first one is that Kirchner does not consider metavariables, that we assimilate to proof goals, and tactics with side effects. The second one is that it puts severe constraints on the set of focused goals, whereas, in the version of the context stack we describe in [Sacerdoti Coen et al. 2006] and we implemented in Matita, focusing is unconstrained. A final difference is the lack of goals tagging in indexed proof trees.

Kirchner's proof monad adds to the indexed proof trees additional information about the outcome of tactics application. In particular, a lifted tactic can either fail, close the goal or open new subgoals. Making this information explicit allows to give a compositional semantics to the LCF tacticals that deal with backtracking, in the same spirit as our context tree was used to give a compositional semantics to the LCF tacticals that we turned into tinycals. We claim that, in a system without metavariables and side effects, Kirchner's proof monad can be applied to our context proof trees with the same benefits. The study of the combination of the proof monad with our proposal in the case of metavariables and the development of appropriates tinycals that benefit from the obtained data type are left as future work.

Finding in the literature precise comparisons and discussions about the benefits of data types used in proof assistant implementation is very hard and a common misconception is that the LCF data type is still largely adopted in its original form. In reality, the LCF data type suffers from several limitations and some of them have been lifted in different ways by the existing systems. While our contribution is not groundbreaking, it paves the way to more detailed comparisons of the pros and cons of the different implementations and it provides a reference general data type for future implementations.

References

Special Issue on Interactive Theorem Proving and Verification. *Sadhana*, 34(1), 2009.

Andrea Asperti and Enrico Tassi. An interactive driver for goal directed proof strategies. In *Proc. of User Interfaces for Theorem Provers 2008. Montreal, CA, August 2008*, 2009. To be published.

Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.

Barras B., Boutin S., Cornes C., Courant J., Filliatre J. C., Giménez E., Herbelin H., Huet G., Munoz C., Murthy C., Parent C., Paulin-Mohring

C., Saibi A., and Werner B. The Coq Proof Assistant Reference Manual : Version 6.1. Technical Report RT-0203, Inria (Institut National de Recherche en Informatique et en Automatique), France, 1997.

Coq. The Coq proof-assistant. `http://coq.inria.fr`, 2009.

David Delahaye and Roberto di Cosmo. Information retrieval in a Coq proof library using type isomorphisms. In *Proceedings of TYPES 99*, volume Lecture Notes in Computer Science. Springer-Verlag, 1999.

Herman Geuvers and Gueorgui I. Jojgov. Open proofs and open terms: A basis for interactive logic. In J. Bradfield, editor, *Computer Science Logic: 16th International Workshop, CSL 2002*, volume 2471 of *Lecture Notes in Computer Science*, pages 537–552. Springer-Verlag, January 2002.

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. Edinburgh LCF: a mechanised logic of computation. volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

John Harrison. A Mizar Mode for HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *LNCS*, pages 203–220. Springer-Verlag, 1996.

Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, , and Xin Yu. Metaprl — a modular logical environment. In David Basin and Burkhart Wolff, editors, *TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

HOL-Light. The HOL Light proof-assistant. `http://www.cl.cam.ac.uk/users/jrh/hol-light/`.

Isabelle. The Isabelle proof-assistant. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.

Florent Kirchner and César Muñoz. Pvs#: Streamlined tacticals for pvs. *Electr. Notes Theor. Comput. Sci.*, 174(11):47–58, 2007.

Matita. The Matita interactive theorem prover. `http://matita.cs.unibo.it`.

Csar Muoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.

NuPRL. The NuPRL proof-assistant. `http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html`.

Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Tinycals: step by step tacticals. In *Proceedings of User Interface for Theorem Provers 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 125–142. Elsevier Science, 2006.

Matthieu Sozeau and Nicolas Oury. First-class type classes. In *TPHOLs*, pages 278–293, 2008.

Freek Wiedijk. Mmode, a mizar mode for the proof assistant Coq. Technical Report NIII-R0333, University of Nijmegen, 2003.