Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 125–142
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Tinycals: Step by Step Tacticals

Claudio Sacerdoti Coen[1] Enrico Tassi[2] Stefano Zacchiroli[3]

Department of Computer Science, University of Bologna
Mura Anteo Zamboni, 7 — 40127 Bologna, ITALY

Abstract

Most of the state-of-the-art proof assistants are based on procedural proof languages, scripts, and rely on LCF tacticals as the primary tool for tactics composition. In this paper we discuss how these ingredients do not interact well with user interfaces based on the same interaction paradigm of Proof General (the *de facto* standard in this field), identifying in the coarse-grainedness of tactical evaluation the key problem. We propose *Tinycals* as an alternative to a subset of LCF tacticals, showing that the user does not experience the same problem if tacticals are evaluated in a more fine-grained manner. We present the formal operational semantics of tinycals as well as their implementation in the Matita proof assistant.

Keywords: Interactive Theorem Proving, Small Step Semantics, Tacticals

1 Introduction

Several state-of-the-art interactive theorem provers are based on procedural proof languages; the user interacts with the system mainly via a textual *script* that records the executed commands. The commands that allow progress during a proof are called *tactics* and are executed atomically. NuPRL [10], Isabelle [6], Coq [13], and Matita[4] (the proof assistant under development by our team at the University of Bologna) are a few examples of those systems.

The best known proof assistant that provides only a declarative proof language is Mizar [8], while a few others superpose a declarative proof language on top of a procedural core. The most notable system in this category is Isabelle, which in its Isabelle/Isar variant offers to users the declarative Isar proof language [14].

With the exception of Mizar, both kind of systems share the same user interface paradigm, inspired by the pioneering work on CtCoq [2] and now incarnated by

[1] sacerdot@cs.unibo.it
[2] tassi@cs.unibo.it
[3] zacchiro@cs.unibo.it
[4] http://matita.cs.unibo.it

```
theorem lt_O_defactorize_aux:
 \forall f:nat_fact.
 \forall i:nat.
 O < defactorize_aux f i.
intro. elim f.
simplify. unfold lt.
rewrite > times_n_SO.
apply le_times.
change with (O < \pi _ i).
apply lt_O_nth_prime_n.
change with (O < (\pi _ i)^n).
apply lt_O_exp.
apply lt_O_nth_prime_n.
simplify.unfold lt.
rewrite > times_n_SO.
apply le_times.
change with (O < (\pi _ i)^n).
apply lt_O_exp.
apply lt_O_nth_prime_n.
change with
  (O < defact n1 (S i)).
apply H.
```

```
theorem lt_O_defactorize_aux:
 \forall f:nat_fact.
 \forall i:nat.
 O < defactorize_aux f i.
intro; elim f;
[1,2:
  simplify; unfold lt;
  rewrite > times_n_SO;
  apply le_times;
  [ change with (O < \pi _ i);
    apply lt_O_nth_prime_n
  |2,3:
    change with (O < (\pi _ i)^n);
    apply lt_O_exp;
    apply lt_O_nth_prime_n
  | change with
      (O < defact n1 (S i));
    apply H ] ].
```

Fig. 1. The same proof with (on the right) and without (on the left) tacticals.

Proof General [1]. In this paradigm, the smallest amount of code that can be executed atomically is the *statement*, which during a proof is either a tactic (in the procedural world) or a single proof step (in the declarative world).

Scripts can be understood only by step by step execution, getting feedback on the proof status from the system. Since feedback is given only between atomic steps (at the so called *execution points*), it is important to have atomic steps as small as possible for the sake of understanding but, also of debugging and proof maintenance. This is in contrast with *tacticals*, higher order constructs which can be used to combine tactics together.

In this paper we propose a replacement for tacticals in order to obtain smaller atomic execution steps. Our work is not relevant in the context of declarative proof languages. However, those few systems where it is possible to embed procedural scripts inside declarative proof steps may already provide the functionality we suggest.

Tacticals first appeared in the LCF theorem prover [5] in 1979. Paradigmatic examples of tacticals are *sequential composition* and *branching.*[5] The former, usually written as “t_1 ; t_2”, takes two tactics t_1 and t_2 and apply t_2 to each of the conjectures resulting from the application of t_1 to the current conjecture (of course its application can be repeated to obtain *pipelines* of tactics “t_1 ; t_2 ; t_3 ; $\cdots$”). The latter, “t ; [t_1 | $\cdots$ | t_n]”, takes $n + 1$ tactics, applies t to the current conjecture and, requiring t to return exactly n conjectures, applies t_1 to the first returned conjecture, t_2 to the second, and so forth.

Tacticals improve procedural proof languages providing concrete advantages, that we illustrate with Figure 1. The concrete syntax used in the figure is that of the Matita proof assistant.

Proof structuring. Using branching, the script representation of proofs can mimic the structure of the proof tree (the tree having conjectures as nodes and tactic-labeled arcs). Since proof tree branches usually reflect conceptual parts of the

[5] In this paper the term “branching” is used to refer to LCF’s `THENL` tactical

pen and paper proof, the branching tactical helps in improving scripts readability (on the average very poor, if compared with declarative proof languages). Even maintainability of proof scripts is improved by the use of branching, for example when hypothesis are added, removed or permuted.

For instance, in the right hand side of Figure 1 it is now clear that `elim f` splits the proof in two branches; both of them (selected by "`[1,2:`") begin with the same tactics until each branch is split again by the application of the `le_times` lemma. Of the four branches, the second and third one (selected by "`|2,3:`") are proved by the same tactics, being proofs of the same fact. All the tactics that are not followed by branching do not introduce ramifications in the proof.

In practice, the proof on the left hand side would be written by using indentation and blank lines to understand where branches start and end. This way readability is improved, but a lesser effect is achieved for proof maintenance. Moreover, the system does not verify in any way the layout of the proof and does not guarantee consistency when the script is changed. We expect that users will abandon this behaviour as soon as an alternative without drawbacks — not the case of LCF tacticals — will surface.

Notice that the selection of multiple branches at a time we propose in this paper is an improvement over the standard branching tactical.

Conciseness. As code factorization is a good practice in programming, proof factorization is in theorem proving. The use of tacticals like sequential composition reduce the need of copy-and-paste in proof scripts helping in factorizing common cases in proofs (so frequent in formal proofs pertaining to the computer science field). Conciseness is evident in Figure 1.

In all the proof assistants we are aware of, tacticals are evaluated atomically and placing the execution point in the middle of complex tacticals (for example at occurrences of "**;**" in tactic pipelines) is not allowed. In Figure 1, this means that having the execution point at the beginning of the proof and asking the system to move it forward (i.e. to execute the next statement), the user will result in a "proof completed" status, without having any feedback of the inner proof status the system passed through. The only way for the user to inspect those status — a frequent need, for instance for script maintenance or proof presentation — is to manually de-structure the complex tacticals.

The big step evaluation of tacticals has also drawbacks on how proof authors develop their proofs. Since it is not always possible to predict the outcome of complex tactics, the following is common practice:

(i) evaluate the next tactic of the script;

(ii) inspect the set of returned conjectures;

(iii) decide whether the use of "**;**" or "[" is appropriate;

(iv) if it is: retract the last statement, add the tactical, go to step (i).

Last, but not less important, is the imprecise error reporting of big step evaluation of tacticals. Consider the frequent case of a script breaking and the user having to fix it. The error message returned by the system may concern an inner status unknown to the user, since the whole tactical is evaluated at once. Moreover, the error message will probably concern terms that do not appear verbatim in the script. Finding the statement that need to be fixed is usually done replacing tactics with identity tactic proceeding outside-in, until the single failing tactic is found. This technique is not only error prone, but is even not reliable in presence of *side-effects* (tactics closing conjectures other than that on which they are applied), since the identity tactic has no side-effects and branches of the proof may be affected by their absence.

We claim that the tension between tacticals and Proof General like interfaces can be broken. In this paper we present a tiny language of tacticals — the so called *tinycals* — which solves this issue. Tinycals can be evaluated in small steps, enabling the execution point to be placed inside complex structures like pipelines or branching constructs. This goal is achieved by de-structuring the syntax of tacticals and stating the semantics as a transition system over *evaluation status*, that are structures richer than the proof status tactics act on. Note that de-structuring does not necessarily mean changing the concrete syntax of tacticals, but rather enabling parsing and immediate evaluation of tactical fragments like "[" alone.

The paper is organized as follows. Section 2 describes the abstract syntax of tinycals together with their small-step operational semantics. Other advantages of tinycals with respect to LCF tacticals are discussed there as well. Section 3 presents the tinycals implementation in Matita. Section 4 deals with tacticals not covered by tinycals. Section 5 discusses related work and Section 6 concludes the paper.

2 Tinycals: syntax and semantics

The grammar of tinycals is reported in Table 1, where $\langle L\rangle$ is the top-level nonterminal generating the script language. $\langle L\rangle$ is a sequence of statements $\langle S\rangle$. Each statement is either an atomic tactical $\langle B\rangle$ (marked with "`tactic`") or a tinycal.

Note that the part of the grammar related to the tinycals themselves is completely de-structured. The need for embedding the structured syntax of LCF tacticals (nonterminal $\langle B\rangle$) in the syntax of tinycals will be discussed in Section 4. For the time being, the reader can suppose the syntax to be restricted to the case $\langle B\rangle ::= \langle T\rangle$.

We will now describe the semantics of tinycals which is parametric in the proof status tactics act on and also in their semantics (see Table 2).

A *proof status* is the logical status of the current proof. It can be seen as the current proof tree, but there is no need for it to actually be a tree. Matita for instance just keeps the set of conjectures to prove, together with a proof term where meta-variables occur in place of missing components. From a semantic point of view the proof status is an abstract data type. Intuitively, it must describe at least the set of conjectures yet to be proved. A *Goal* is another abstract data type used to index conjectures.

The function *apply_tac* implements tactic application. It consumes as input

Table 1
Abstract syntax of tinycals and core LCF tacticals.

$\langle S \rangle$ ::=	(**statements**)	$\langle L \rangle$ ::=	(**language**)
"`tactic`" $\langle B \rangle$	(tactic)	$\langle S \rangle$	(statement)
		$\mid$ $\langle S \rangle$ $\langle S \rangle$	(sequence)
$\mid$ "."	(dot)		
$\mid$ ";"	(semicolon)	$\langle B \rangle$ ::=	(**tacticals**)
		$\langle T \rangle$	(tactic)
$\mid$ "["	(branch)		
		$\mid$ "`try`" $\langle B \rangle$	(recovery)
$\mid$ "$\vert$"	(shift)		
		$\mid$ "`repeat`" $\langle B \rangle$	(looping)
$\mid$ $i_1, \ldots, i_n$ ":"	(projection)		
		$\mid$ $\langle B \rangle$ ";" $\langle B \rangle$	(composition)
$\mid$ "$*$:"	(wild card)		
		$\mid$ $\langle B \rangle$ ";["	(branching)
$\mid$ "`accept`"	(acknowledge)		
		$\langle B \rangle$ "$\vert$" ... "$\vert$" $\langle B \rangle$ "]"	
$\mid$ "]"	(merge)		
		$\langle T \rangle$ ::= ...	(**tactics**)
$\mid$ "`focus`" $[g_1; \cdots; g_n]$	(selection)		
$\mid$ "`done`"	(de-selection)		

Table 2
Semantics parameters.

proof status:	ξ
proof goal:	*goal*
tactic application:	$\mathit{apply\_tac} : T \to \xi \to \mathit{goal} \to \xi \times \mathit{goal}\ \texttt{list} \times \mathit{goal}\ \texttt{list}$

a tactic, a proof status, and a goal (the conjecture the tactic should act on), and returns as output a proof status and two lists of goals: the set of newly opened goals and the set of goals which have been closed. This choice enables our semantics to account for *side-effects*, that is: tactics can close goals other than that on which they have been applied, a feature implemented in several proof assistants via existential or meta-variables [4,9]. The proof status was not directly manipulated by tactics in LCF because of the lack of meta-variables and side effects.

In the rest of this section we will define the semantics of tinycals as a transition (denoted by $\longrightarrow$) on evaluation status. *Evaluation status* are defined in Table 3.

The first component of the status (*code*) is a list of statements of the tinycals grammar. The list is consumed, one statement at a time, by each transition. This choice has been guided by the un-structured form of our grammar and is the heart of the fine-grained execution of tinycals.

The second component is the proof status, which we enrich with a *context stack* (the third component). The context stack, a representation of the proof history so

Table 3
Evaluation status.

task	$= \texttt{int} \times (\texttt{Open}\ \mathit{goal} \mid \texttt{Closed}\ \mathit{goal})$	(task)
Γ	$= \mathit{task}\ \texttt{list}$	(context)
τ	$= \mathit{task}\ \texttt{list}$	("todo" list)
κ	$= \mathit{task}\ \texttt{list}$	(dot's continuations)
$\mathit{ctxt\_tag}$	$= \texttt{B} \mid \texttt{F}$	(stack level tag)
$\mathit{ctxt\_stack}$	$= (\Gamma \times \tau \times \kappa \times \mathit{ctxt\_tag})\ \texttt{list}$	(context stack)
code	$= \langle S \rangle\ \texttt{list}$	(statements)
status	$= \mathit{code} \times \xi \times \mathit{ctxt\_stack}$	(evaluation status)

far, is handled as a stack: levels get pushed on top of it either when the branching tinycal "[" is evaluated, or when "`focus`" is; levels get popped out of it when the corresponding closing tinycals are ("]" for "[" and "`done`" for "`focus`"). Since the syntax is un-structured, we can not ensure statically proper nesting of tinycals, therefore each stack level is equipped with a *tag* which annotates it with the creating tinycal (`B` for "[" and `F` for "`focus`"). In addition to the tag, each stack level has three components Γ, τ and κ respectively for active tasks, tasks postponed to the end of branching and tasks posponed by "`.`". The role of these componenets will be explained in the description of the tinycals that acts on them. Each component is a sequence of numbered tasks. A *task* is an handler to either a conjecture yet to be proved, or one which has been closed by a side-effect. In the latter case the user will have to confirm the instantiation with "`accept`".

Each evaluation status is meaningful to the user and can be presented by slightly modifying already existent user interfaces. Our presentation choice is described in Section 3. The impatient reader can take a sneak preview of Figure 2, where the interesting part of the proof status is presented as a notebook of conjectures to prove, and the conjecture labels represent the relevant information from the context stack by means of: 1) bold text (for conjectures in the currently selected branches, targets of the next tactic application; they are kept in the Γ component of the top of the stack); 2) subscripts (for not yet selected conjectures in sibling branches; they are kept in the Γ component of the level below the top of the stack). The rest of the information hold in the stack does not need to be shown to the user since it does not affect immediate user actions.

We describe first the semantics of the tinycals that do not involve the creation of new levels on the stack. The semantics is shown in Table 4, where some utility functions (described in Appendix A) are used.

Tactic application

Consider the first case of the tinycals semantics of Table 4. It makes use of the first component (denoted Γ) of a stack level, which represent the "current" goals,

Table 4
Basic tinycals semantics.

$$\langle \text{“}\texttt{tactic}\text{”}\ \langle T\rangle :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi_n, S'\rangle \qquad n \geq 1$$

where $[g_1;\cdots;g_n] = \mathit{get\_open\_goals\_in\_tasks\_list}(\Gamma)$

$$\text{and} \begin{cases} \langle \xi_0, G^o_0, G^c_0\rangle = \langle \xi, [\,], [\,]\rangle & \\ \langle \xi_{i+1}, G^o_{i+1}, G^c_{i+1}\rangle = \langle \xi_i, G^o_i, G^c_i\rangle & g_{i+1} \in G^c_i \\ \langle \xi_{i+1}, G^o_{i+1}, G^c_{i+1}\rangle = \langle \xi', (G^o_i \setminus G^c) \cup G^o, G^c_i \cup G^c\rangle & g_{i+1} \notin G^c_i \\ \quad \text{where } \langle \xi', G^o, G^c\rangle = \mathit{apply\_tac}(T, \xi_i, g_{i+1}) & \end{cases}$$

and $S' = \langle \Gamma', \tau', \kappa', t\rangle :: \mathit{close\_tasks}(G^c_n, S)$

and $\Gamma' = \mathit{mark\_as\_handled}(G^o_n)$

and $\tau' = \mathit{remove\_tasks}(G^c_n, \tau)$

and $\kappa' = \mathit{remove\_tasks}(G^c_n, \kappa)$

$$\langle \text{“;”} :: c, \xi, S\rangle \longrightarrow \langle c, \xi, S\rangle$$

$$\langle \text{“}\texttt{accept}\text{”} :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $\Gamma = [\langle j_1, \texttt{Closed}\ g_1\rangle; \cdots ; \langle j_n, \texttt{Closed}\ g_n\rangle] \qquad n \geq 1$

and $G^c = [g_1;\cdots;g_n]$

and $S' = \langle [\,], \mathit{remove\_tasks}(G^c, \tau), \mathit{remove\_tasks}(G^c, \kappa), t\rangle$
$\qquad :: \mathit{close\_tasks}(G^c, S)$

$$\langle \text{“.”} :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, \langle [l_1], \tau, [l_2;\cdots;l_n] \cup \kappa, t\rangle :: S\rangle \qquad n \geq 1$$

where $\mathit{get\_open\_tasks}(\Gamma) = [l_1;\cdots;l_n]$

$$\langle \text{“.”} :: c, \xi, \langle \Gamma, \tau, l :: \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, \langle [l], \tau, \kappa, t\rangle :: S\rangle$$

where $\mathit{get\_open\_tasks}(\Gamma) = [\,]$

that is the set of goals to which the next tactic evaluated will be applied.

When a tactic is evaluated, the set Γ of current goals is inspected (expecting to find at least one of them), and the tactic is applied in turn to each of them in order to obtain the final proof status. At each step i the two sets C^o_i and G^c_i of goals opened and closed so far are updated. This process is atomic to the user (i.e. no feedback is given while the tactic is being applied to each of the current goals in turn), but she is free to cast off atomicity using branching. After the tactic has been applied to all goals, the new set of current goals is created containing all the goals which have been opened during the applications, but not already closed. They are marked (using the *mark_as_handled* utility) so that they do not satisfy the *unhandled* predicate, indicating that some tactic has been applied to them. Goals closed by side effects are removed from τ and κ and marked as `Closed` in S. The reader can find a datailed description of this procedure in Appendix A.

Sequential composition

Since sequencing is handled by Γ, the semantics of "**;**" is simply the identity function. We kept it in the syntax of tinycal for preserving the parallelism with LCF tacticals.

Side-effects handling

"`accept`" (third case in Table 4) is a tinycal used to deal with side-effects. Consider for instance the case in which there are two current goals on which the user branches. It can happen that applying a tactic to the first one closes the second, removing the need of the second branch in the script. Using tinycals the user will never see branches she was aware of disappear without notice. Cases like the above one are thus handled marking the branch as `Closed` (using the *close_tasks* utility) on the stack and requiring the user to manually acknowledge what happened on it using the "`accept`" tinycal, preserving the correspondence among script structure and proof tree.

Example 2.1 Consider the following script:

```
apply trans_eq; [ apply H | apply H1 | accept ]
```

where the application of the transitivity property of equality to the conjecture $L = R$ opens the three conjectures $?_1 : L = ?_3$, $?_2 : ?_3 = R$ and $?_3 : nat$. Applying the hypothesis H instantiates $?_3$, implicitly closing the third conjecture, that thus has to be acknowledged.

Local de-structuring

Structuring proof scripts enhances their readability as long as the script structure mimics the structure of the intuition behind the proof. For this reason, authors do not always desire to structure proof scripts down to the most far leaf of the proof tree.

Example 2.2 Consider for instance the following script snippet template:

```
tac1;
[ tac2. tac3.
| tac4; [ tac5 | tac6 ] ]
```

Here the author is trying to mock-up the structure of the proof (two main branches, with two more branches in the second one), without caring about the structure of the first branch.

Tacticals do not allow un-structured scripts to be nested inside branches. In the example, they would only allow to replace the first branch with the identity tactic, continuing the un-structured snippet "tac2. tac3." at the end of the snippet, but this way the correspondence among script structure and proof tree would be completely lost. The semantics of the tinycal "**.**" (last two cases of Table 4) accounts for local use of un-structured script snippets.

When "**.**" is applied to a non-empty set of current goals, the first one is selected and become the new singleton current goals set Γ. The remaining goals are remembered in the third component of the current stack level (*dot's continuations*, denoted κ), so that when the "**.**" is applied again on an empty set of goals they can be recalled in turn. The locality of "**.**" is inherited by the locality of dot's continuation κ to stack levels.

Table 5 describes the semantics of tinycals that require a stack discipline.

Branching

Support for branching is implemented by "**[**", which creates a new level on the stack for the first of the current goals. Remaining goals (the current *branching context*) are stored in the level just below the freshly created one. There are three different ways of selecting them. Repeated uses of "**|**" consume the branching context in sequential order. $i_1,\ldots,i_n$"**:**" enables multiple positional selection of goals from the branching context. "$*$**:**" recall all goals of the current branching context as the new set of current goals. The semantics of all these branching tacticals is shown in the first five cases of Table 5.

Each time the user finishes working on the current goals and selects a new goal from the branching context, the result of her work (namely the current goals in Γ) needs to be saved for restoring at the end of the branching construct. This is needed to implement the LCF semantics that provides support for snippets like the following:

Example 2.3

```
tac1; [ tac2 | tac3 ]; tac4
```

where the goals resulting by the application of `tac2` *and* `tac3` are re-flowed together to create the goals set for `tac4`.

The place where we store them is the second component of stack levels (*todo list*, denoted τ). Each time a branching selection tinycal is used the current goals set (possibly empty) is appended to the todo list for the current stack level.

When "**]**" is used to finish branching (fifth rule of Table 5), the todo list τ is used to create the new set of current goals Γ, together with the goals not handled during the branching (note that this is a small improvement over LCF tactical semantics, where leaving not handled branches is not allowed).

Table 5
Branching tinycals semantics.

$$\langle \text{“[”} :: c, \xi, \langle [l_1;\cdots;l_n], \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle \qquad n \geq 2$$

where $renumber\_branches([l_1;\cdots;l_n]) = [l'_1;\cdots;l'_n]$

and $S' = \langle [l'_1], [\,], [\,], \mathtt{B}\rangle :: \langle [l'_2;\cdots;l'_n], \tau, \kappa, t\rangle :: S$

$$\langle \text{“|”} :: c, \xi, \langle \Gamma, \tau, \kappa, \mathtt{B}\rangle :: \langle [l_1;\cdots;l_n], \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle \qquad n \geq 1$$

where $S' = \langle [l_1], \tau \cup get\_open\_tasks(\Gamma) \cup \kappa, [\,], \mathtt{B}\rangle :: \langle [l_2;\cdots;l_n], \tau', \kappa', t'\rangle :: S$

$$\langle i_1,\ldots,i_n \text{“:”} :: c, \xi, \langle [l], \tau, [\,], \mathtt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $unhandled(l)$

and $\forall j = 1\ldots n, \quad \exists l_j = \langle j, s_j\rangle, \quad l_j \in l :: \Gamma'$

and $S' = \langle [l_1;\cdots;l_n], \tau, [\,], \mathtt{B}\rangle :: \langle (l :: \Gamma') \setminus [l_1;\cdots;l_n], \tau', \kappa', t'\rangle :: S$

$$\langle \text{“} * \text{:”} :: c, \xi, \langle [l], \tau, [\,], \mathtt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $unhandled(l)$

and $S' = \langle l :: \Gamma', \tau, [\,], \mathtt{B}\rangle :: \langle [\,], \tau' \cup get\_open\_tasks(\Gamma) \cup \kappa, \kappa', t'\rangle :: S$

$$\langle \text{“]”} :: c, \xi, \langle \Gamma, \tau, \kappa, \mathtt{B}\rangle :: \langle \Gamma', \tau', \kappa', t'\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $S' = \langle \tau \cup get\_open\_tasks(\Gamma) \cup \Gamma' \cup \kappa, \tau', \kappa', t'\rangle :: S$

$$\langle \text{“}\mathtt{focus}\text{”}\ [g_1;\cdots;g_n] :: c, \xi, \langle \Gamma, \tau, \kappa, t\rangle :: S\rangle \longrightarrow \langle c, \xi, S'\rangle$$

where $g_i \in get\_open\_goals\_in\_status(S)$

and $S' = \langle mark\_as\_handled([g_1;\cdots;g_n]), [\,], [\,], \mathtt{F}\rangle$

$\quad :: close\_tasks(\langle \Gamma, \tau, \kappa, t\rangle :: S)$

$$\langle \text{“}\mathtt{done}\text{”} :: c, \xi, \langle [\,], [\,], [\,], \mathtt{F}\rangle :: S\rangle \longrightarrow \langle c, \xi, S\rangle$$

Focusing

The pair of tinycals “`focus`”... “`done`” is similar in spirit to the pair “[”... “]”, but is not required to work on the current branching context. With “`focus`”, goals located everywhere on the stack can be recalled to form a new set of current goals. On this the user is then free to work as she prefer, for instance branching, but is required to close all of them before invoking "`done`".

The intended use of "`focus`"... "`done`" is to deal with meta-variables and side effects. The application of a tactic to a conjecture with meta-variables in the conclusion or hypotheses can instantiate the meta-variables making other conjectures false. In other words, in presence of meta-variables conjectures are no longer independent and it becomes crucial to consider and close a bunch or dependent conjectures together, even if in far away branches of the proof. In these cases "`focus`"... "`done`" is used to select all the related branches for immediate work on them. Alternatively, "`focus`"... "`done`" can be used to jump on a remote branch of the tree in order to instantiate a meta-variable by side effects before resuming proof search from the current position.

Note that using "`focus`"... "`done`", no harm is done to the proper structuring of scripts, since all goals the user is aware of, if closed, will be marked as `Closed` requiring her to manually "`accept`" them later on in the proof.

3 Implementation issues

Tinycals have been implemented in the Matita proof assistant. This section describes the issues faced in their implementation.

Encoding of tacticals

Tacticals play two different roles in a proof assistant. They can be used both in scripts and in tactic implementations. As a matter of fact at least one tactical among sequential composition and branching is used in the implementation of each derived tactic.

In this paper we propose the replacement of tacticals with tinycals. Tacticals operate on proof status, while tinycals operate on evaluation status. This is welcome when tinycals are used in scripts, since the additional information kept in the evaluation status is the rich intermediate state we want to present to the user. On the contrary, this datatype change does not allow the replacement of tacticals with tinycals in the implementation of derived tactics. Thus we are immediately led to consider if it is possible to express tacticals in terms of tinycals, in order to avoid an independent re-implementation of related operations.

The answer is positive under additional assumptions on the abstract data type of proof status. Intuitively, we need to define two "inverse" functions to embed a proof status, a goal, and a code in an evaluation status (let it be *embed*) and to project an evaluation status to a proof status and two lists of opened and closed goals (let it be *proj*). Once the two functions are implemented, we can express sequential composition and branching as follows:

$$(t_1; t_2)(\xi, g) = proj(eval(embed([t_1; \text{``;''}; t_2], \xi, g))) \tag{1}$$

$$(t; [t_1|\ldots|t_n])(\xi, g) = proj(eval(embed([t; \text{``[''}; t_1; \text{``|''}; \ldots; \text{``|''}; t_n; \text{``]''}], \xi, g))) \tag{2}$$

where *eval* is the transitive closure of $\longrightarrow$. For each status S the code of the status $eval(S)$ is empty.

The *embed* function is easily defined as:

$$embed(c,\xi,g) = \langle c,\xi,[\langle g,[\,],[\,],\mathtt{F}\rangle]\rangle$$

To define the *proj* function, however, we need to be able to compute the set of goals opened and closed by $eval(embed(c,\xi,g))$ for any given code c, proof status ξ and selected goal g. The formers are easily computed by the *get_open_goals_in_status* utility of Appendix A. However, to compute the latter the information stored in an evaluation context is not enough.

We say that tactics *do not reuse goals* whenever closed goals cannot be re-opened (remember that a goal is just an handle to a conjecture, not the conjecture itself). Concretely, it is possible to respect this property in the implementation by keeping a global counter that represents the highest goal index already used. When a tactic opens a new goal it picks the successor of the counter, that is also incremented. When tactics do not reuse goals it is possible to determine the goals closed by a sequence of evaluation steps by comparing the set of open goals at the two extremes of the sequence. To make this comparison it is possible to add to the proof status abstract data type a method that returns the set of opened goals.

Let *diff* be the function that given two proof status ξ and ξ' returns the set of goals that were open in ξ and are closed in ξ'. For each proof status ξ the $proj_\xi$ function is defined as:

$$proj_\xi([\,],\xi',S) = (\xi', get\_open\_goals\_in\_status(S), diff(\xi,\xi'))$$

The function $proj_\xi$ must be used in Equation (1) and Equation (2) in place of *proj*.

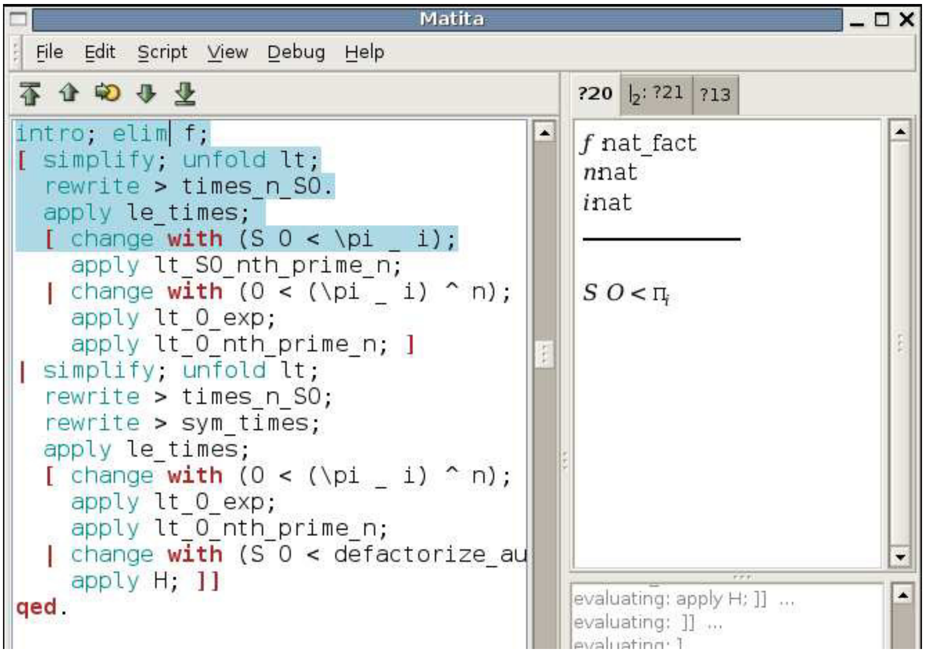
Tinycals user interface

Tinycals would be worthless without a way to present evaluation status to the user. Our current solution for the Matita user interface is shown in Figure 2.

We already had a Proof General like user interface with script and execution point (on the left of Figure 2) and a tabbed representation of the set of open conjectures (on the right) as sequents, using meta-variable indexes as labels. What the user was missing to work with tinycals was a visual representation of the stack. Our choice has been to represent the current branching context as tab label annotations: all goals in the current goals set have their labels typeset in boldface, goals of the current branching context have labels prepended by $|_n$ (where n is their positional index), and goals already closed by side-effects have strike-through labels like: ~~?n~~.

For instance in Figure 2, the only goal (in bold-face) the next tactic will be applied to is 20 (i.e. $\Gamma = [\langle 1, \mathtt{Open}\ 20\rangle]$), while goal 21 will be selected by the next "|" tinycal.

This choice makes the user aware of which goals will be affected by a tactic evaluated at the execution point, and of all the indexing information she might need there. She indeed can see all meta-variable indexes (in case she wants to "`focus`") and all the positional indexes of goals in the current branching context



```
intro; elim f;
[ simplify; unfold lt;
  rewrite > times_n_SO.
  apply le_times;
  [ change with (S O < \pi _ i);
    apply lt_SO_nth_prime_n;
  | change with (O < (\pi _ i) ^ n);
    apply lt_O_exp;
    apply lt_O_nth_prime_n; ]
| simplify; unfold lt;
  rewrite > times_n_SO;
  rewrite > sym_times;
  apply le_times;
  [ change with (O < (\pi _ i) ^ n);
    apply lt_O_exp;
    apply lt_O_nth_prime_n;
  | change with (S O < defactorize_au
    apply H; ]]
qed.
```

```
f: nat_fact
n: nat
i: nat
________
S O < π_i
```

```
evaluating: apply H; ]] ...
evaluating: ]] ...
evaluating: ]
```

Fig. 2. Evaluation status representation in the Matita user interface.

(for $i_1,\ldots,i_n$ ":" and "$*$:"). Yet, this user interface choice minimizes the drift from the usual way of working with Proof General like interfaces.

4 A digression on the remaining tacticals

Of the basic LCF tacticals, we have considered so far only sequential composition and branching. It is worth discussing the remaining ones, in particular *try*, $||$ (or-else) and *repeat*.

The *try* T tactical, that never fails, applies the tactic T, behaving as the identity if T fails. It is a particular case of the or-else tactical: $T_1||T_2$ behaves as T_1 if T_1 does not fail, as T_2 otherwise. Thus *try* T is equivalent to $T||id$.

The try and or-else tacticals occur in a script with two different usages. The most common one is after sequential composition: T_1; *try* T_2 or $T_1; T_2||T_3$. Here the idea is that the user knows that T_2 can be applied to some of the goals generated by T_1 (and T_3 to the others in the second case). So she is faced with two possibilities: either use branching and repeat T_2 (or T_3) in every branch, or use sequential composition and backtracking (encapsulated in the two tacticals). Tinycals offer a better solution to either choice by means of the projection and wild card tinycals: $T_1; [i_1,\ldots,i_n : T_2|* : T_3]$. The latter expression is not also more informative to the reader, but it is also computationally more efficient since it avoids the (maybe costly) application of T_2 to several goals.

The second usage of try and or-else is inside a repeat tactical. The *repeat* T tactical applies T once, failing if T fails; otherwise the tactical recursively applies T again on every goal opened by T until T fails, in which case it behaves as the identity tactic.

Is it possible to provide an un-structured version of *try* T, $T||T'$, and *repeat* T in the spirit of tinycals in order to allow the user to write and execute T step by step inspecting the intermediate evaluation status? The answer is negative as we can easily see in the simplest case, that of *try* T. Consider the statement $T; \mathit{try}\ (T_1; T_2)$ where sequential composition is supposed to be provided by the corresponding tinycal. Let T open two goals and suppose that "*try*" is executed atomically so that the evaluation point is just before T_1. When the user executes T_1, T_1 can be applied as expected to both goals in sequence. Let ξ be the proof status after the application of T and let ξ_1 and ξ_2 be those after the application of T_1 to the first and second goal respectively. Let now the user execute the identity tinycal ";" followed by T_2 and let T_2 fail over the first goal. To respect the intended semantics of the tactical, the status ξ_2 should be *partially* backtracked to undo the changes from ξ to ξ_1, preserving those from ξ_1 to ξ_2.

If the system has side effects the latter operation is undefined, since T_1 applied to ξ could have instantiated meta-variables that controlled the behavior of T_1 applied to ξ_1. Thus undoing the application of T_1 to the first goal also invalidates the previous application of T_1 to the second goal.

Even if the system has no side effects, the requirement that proof status can be partially backtracked is quite restrictive on the possible implementations of a proof status. For instance, a proof status cannot be a simple proof term with occurrences of meta-variables in place of conjectures, since backtracking a tactic would require the replacement of a precise subterm with a meta-variable, but there would be no information to detect which subterm.

As a final remark, the simplest solution of implementing partial backtracking by means of a full backtrack to ξ followed by an application of T_1 to the second goal only does not conform to the spirit of tinycals. With this implementation, the application of T_1 to the second goal would be performed twice, sweeping the waste of computational resources under the rug. The only honest solution consists of keeping all tacticals, except branching and sequential composition, fully structured as they are now. The user that wants to inspect the behavior of $T; \mathit{try}\ T_1$ before that of $T; \mathit{try}\ (T_1; T_2)$ is obliged to do so by executing atomically *try* T_1, backtracking by hand and executing *try* $(T_1; T_2)$ from scratch. A similar conclusion is reached for the remaining tacticals. For this reason in the syntax given in Table 1 the production $\langle B \rangle$ lists all the traditional tacticals that are not subsumed by tinycals. Notice that atomic sequential composition and atomic branching (as implemented in the previous section) are also listed since tinycals cannot occur as arguments of a tactical.

5 Related work

Different presentations of the semantics of tacticals has been given in the past. The first presentation has been given in [5] by Gordon et al. Although a larger set of tacticals than that considered here was described in their work, the problem of inspection of inner proof status was not considered. Proof General-like interfaces were not available at the time, as well as meta-variables and tactics with side-effects.

In [7], Kirchner described a small step semantics of Coq tacticals. Despite the minor expressive advantages offered by tinycals over the corresponding Coq tacticals (like "`focus`", "`*:`", $i_1,\ldots,i_n$"`:`", the less constrained use of "[", and the structuring facilities implemented by "`.`" and "`accept`"), the formalization of tinycals is more general and we believe that it can be applied to a large class of proof assistants. In particular our semantics only assume an abstract proof status and a very general type for tactic applications, while in [7] a very detailed API for proof trees was assumed.

Delahaye in [3] described $\mathcal{L}_{tac}$, a powerful meta-language which can be used both by users and tactics implementors to write small automations at the proof language level. $\mathcal{L}_{tac}$ is way more powerful than tinycals, featuring constructs typical of high-level programming and defining their reduction semantics. However, since its aim was different, $\mathcal{L}_{tac}$ fails to address the interaction problem that tinycals do address.

Two alternative approaches for authoring structured HOL scripts have been proposed in [11] and [12]. The first approach, implemented in Syme's TkHOL, is similar to the one presented in this paper but lacks a formal description. Moreover, unlike HOL, we consider a logic with meta-variables which can be closed by side effects. Therefore the order in which branches are closed by tactics is relevant and must be made explicit in the script. For this reason we support tinycals like "`focus`" and $i_1,\ldots,i_n$"`:`" which were not needed in TkHOL. The second approach, by Takahashi et al., implements syntax directed editing by automatically claiming lemmata for each goal opened by the last executed tactic. This technique breaks down with meta-variables because they are not allowed in the statements of lemmata.

6 Conclusions

In this paper we presented the syntax and semantics of tinycals, a tactical language able to mimic some of the LCF tacticals so widespread in state-of-the-art proof assistants. Tinycals advantages over LCF tacticals is that their syntax is un-structured and their evaluation proceeds step by step, enabling the user to start execution of a structured script before its completion. Intermediate proof status can be inspected and tactics with side effects are supported as well. The neat result is better integration with user interfaces based on the CtCoq/Proof General paradigm. Some implementative issues have also been discussed, and the extension of the approach to other tacticals has been considered with negative results.

Tinycals have been implemented and are used in the Matita proof assistant for the ongoing development of its standard library. Users experienced with other proof assistants, in particular Coq, consider them a serious improvement in the proof authoring interface. This is not a big figure (our users are just the member of our research team at the time of writing), but is enough to motivate our work on them, hoping to see them adopted soon in other systems.

References

[1] Aspinall, D., *Proof General: A generic tool for proof development*, in: *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000*, Lecture Notes in Computer Science **1785** (2000).

[2] Bertot, Y., *The CtCoq system: Design and architecture*, Formal Aspects of Computing **11** (1999), pp. 225–243.

[3] Delahaye, D., "Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve: une étude dans le cadre du système Coq," Ph.D. thesis, Université Pierre et Marie Curie (Paris 6) (2001).
URL http://cedric.cnam.fr/~delahaye/publications/these-delahaye.ps.gz

[4] Geuvers, H. and G. I. Jojgov, *Open proofs and open terms: A basis for interactive logic*, in: J. Bradfield, editor, *Computer Science Logic: 16th International Workshop, CLS 2002*, Lecture Notes in Computer Science **2471** (2002), pp. 537–552.

[5] Gordon, M. J. C., R. Milner and C. P. Wadsworth, *Edinburgh LCF: a mechanised logic of computation*, Lecture Notes in Computer Science **78** (1979).

[6] *The Isabelle proof-assistant*,
http://www.cl.cam.ac.uk/Research/HVG/Isabelle/.

[7] Kirchner, F., *Coq Tacticals and PVS Strategies: A Small-Step Semantics*, in: *Design and Application of Strategies/Tactics in Higher Order Logics*, 2003.

[8] *The Mizar proof-assistant*,
http://mizar.uwb.edu.pl/.

[9] Muñoz, C., "A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory," Ph.D. thesis, INRIA (1997).

[10] *The NuPRL proof-assistant*,
http://www.cs.cornell.edu/Info/Projects/NuPrl/nuprl.html.

[11] Syme, D., *A new interface for hol - ideas, issues and implementation*, in: *Proceedings of Higher Order Logic Theorem Proving and Its Applications, 8th International Workshop, TPHOLs 1995*, Lecture Notes in Computer Science **971** (1995), pp. 324–339.

[12] Takahashi, K. and M. Hagiya, *Proving as editing HOL tactics*, Formal Aspects of Computing **11** (1999), pp. 343–357.

[13] The Coq Development Team, *The Coq proof assistant reference manual*,
http://coq.inria.fr/doc/main.html (2005).

[14] Wenzel, M., *Isar - a generic interpretative approach to readable formal proof documents*, in: *Theorem Proving in Higher Order Logics*, 1999, pp. 167–184.

A Utility functions

The goal automatically selected by "[" or "|" is called *unhandled* until a tactic is applied to it. Unhandled goals are just postponed (not moved into the todo list τ) by $i_1,\ldots,i_n$"**:**". Goals opened by a tactic are marked with *mark_as_handled* to distinguishing them from unhandled goals. The function *renumber_branches* is used by "[" to name branches.

$$unhandled(l) = \begin{cases} true \;\text{ if } l = \langle n, \texttt{Open}\ g\rangle \wedge n > 0 \\ false \;\text{ otherwise} \end{cases}$$

$$mark\_as\_handled([g_1; \cdots; g_n]) = [\langle 0, \texttt{Open}\ g_1\rangle; \cdots; \langle 0, \texttt{Open}\ g_n\rangle]$$

$$renumber\_branches([\langle i_1, s_1\rangle; \cdots; \langle i_n, s_n\rangle]) = [\langle 1, s_1\rangle; \cdots; \langle n, s_n\rangle]$$

The next three functions returns open goals or tasks in the status or parts of it. Open goals are those corresponding to conjectures still to be proved.

$$get\_open\_tasks(l) = \begin{cases} [\,] & \text{if } l = [\,] \\ \langle i, \texttt{Open } g\rangle :: get\_open\_tasks(tl) & \text{if } l = \langle i, \texttt{Open } g\rangle :: tl \\ get\_open\_tasks(tl) & \text{if } l = hd :: tl \end{cases}$$

$$get\_open\_goals\_in\_tasks\_list(l) = \begin{cases} [\,] & \text{if } l = [\,] \\ g :: get\_open\_goals\_in\_tasks\_list(tl) & \text{if } l = \langle \_, \texttt{Open } g\rangle :: tl \\ get\_open\_goals\_in\_tasks\_list(tl) & \text{if } l = \langle \_, \texttt{Closed } g\rangle :: tl \end{cases}$$

$$get\_open\_goals\_in\_status(S) = \begin{cases} [\,] & \text{if } S = [\,] \\ get\_open\_goals\_in\_tasks\_list(\Gamma @ \tau @ \kappa) & \\ \quad @ get\_open\_goals\_in\_status(tl) & \text{if } S = \langle \Gamma, \tau, \kappa, \_\rangle :: tl \end{cases}$$

To keep the correspondence between branches in the script and ramifications in the proof, goals closed by side-effects are marked as `Closed` if they are in Γ (that keeps track of open branches). Otherwise they are silently removed from postponed goals (in todo list τ or dot continuation κ). `Closed` branches have to be accepted by the user with "`accept`".

$$close\_tasks(G, S) = \begin{cases} [\,] & \text{if } S = [\,] \\ \langle close_{aux}(G, \Gamma), \tau', \kappa', t\rangle :: close\_tasks(G, tl) & \text{if } S = \langle \Gamma, \tau, \kappa, t\rangle :: tl \\ \quad \text{where } \tau' = remove\_tasks(G, \tau) & \\ \quad \text{and } \kappa' = remove\_tasks(G, \kappa) & \end{cases}$$

$$close_{aux}(G, l) = \begin{cases} [\,] & \text{if } l = [\,] \\ \langle i, \texttt{Closed } g\rangle :: close_{aux}(G, tl) & \text{if } l = \langle i, \texttt{Open } g\rangle :: tl \wedge g \in G \\ hd :: close_{aux}(G, tl) & \text{if } l = hd :: tl \end{cases}$$

$$remove\_tasks(G, l) = \begin{cases} [\,] & \text{if } l = [\,] \\ remove\_tasks(G, tl) & \text{if } l = \langle i, \texttt{Open } g\rangle :: tl \wedge g \in G \\ hd :: remove\_tasks(G, tl) & \text{if } l = hd :: tl \end{cases}$$