Interactive Theorem Proving in Twelf

Amy Felty
School of Information Technology and Engineering
University of Ottawa
Canada
afelty@site.uottawa.ca

1 Introduction

In recent work, we showed how to implement tactic-style theorem proving in Twelf [2]. Tactics and tacticals are a mechanism used in a variety of theorem provers such as LCF [5], HOL [4], and Coq [8]. They provide flexible control for goal-directed proof search. Tactics provide the basic search procedures, while tacticals are used to compose tactics in various ways to form more complex operations or proof search strategies. Our goal was to add the power of tactic-style proof search to our proof-carrying code system, which is implemented in Twelf [1].

The Twelf system [7] is an implementation of the Logical Framework (LF) [6] which provides logic programming capabilities. LF is a dependently-typed language, and our implementation showed how to use the dependent types to guarantee partial correctness of tactics. Our implementation in Twelf was adapted from our earlier work implementing tactics and tacticals in the higher-order logic programming language λProlog [3]. In λProlog, there are no dependent types and so we did not have the partial correctness guarantee. On the other hand, λProlog provides programming support such as control primitives and I/O, while Twelf does not, which allowed us to implement a more flexible theorem prover. The emphasis in Twelf has been on proving properties of logics rather than building theorem provers for them, and so there are no plans to provide this kind of support.

Here, we describe how to do interactive theorem proving in Twelf without control or I/O primitives, thus allowing us to provide the benefits of both dependent types and interactive proof in the same setting. The lack of I/O primitives, however, forces a particular style of theorem proving. This style is similar to the one adopted in HOL. Both of these theorem provers have the property that the language that is used to implement the tactics and tacticals is the same as the language used to interact with the system to build proofs. In our case this language is Twelf, while the programming language for HOL is ML.

2 A Twelf Theorem Prover for First-Order Logic

As an example, we will consider a theorem prover for first-order logic. The implementation of the logic and the basic tactics and tacticals is exactly the same as in our previous work [2], so we just summarize here. We show just enough of the code to handle the simple example used in the next section to illustrate interactive proof. The full code can be found in the appendix.

Figure 1 introduces the Twelf declarations for first-order logic. Only the connectives and rules for implication and conjunction are shown. The rules given are the introduction and

```
i       : type.
o       : type.
pf      : o -> type.

imp     : o -> o -> o.        %infix right 10 imp.
imp_i    : (pf A -> pf B) -> pf (A imp B).
imp_e    : pf (A imp B) -> pf A -> pf B.

and : o -> o -> o.   %infix right 12 and.
and_i    : pf A -> pf B -> pf (A and B).
and_e1   : pf (A and B) -> pf A.
and_e2   : pf (A and B) -> pf B.
```

Figure 1: Example declarations for first-order logic in Twelf.

elimination rules for natural deduction. Dependent types are used so that a term of type (pf A) represents a proof in natural deduction of formula A.

Figure 2 introduces the data structures for proof goals. Each goal has a list of hypotheses associated with it. The first 4 declarations in the figure introduce these lists. An infix comma is used to separate items in the list. An individual hypothesis pairs a formula with its proof using the by constructor. Dependent types are used to guarantee that the second argument is a proof of the first argument. The turnstile is used to separate the hypotheses from the conclusion, and forms the primitive goal of our prover. After the primitive goal declaration, 3 constructors are introduced for compound goals. Their use is illustrated in the implementation of the 4 tactics that follow, which implement basic proof steps. A tactic has 3 arguments. The first is the tactic name, the second is the input goal, and the third is the output goal which contains the subgoal(s) that still must be solved to complete the proof.

As the first tactic illustrates, the output goal is tt whenever the proof has been completed, indicating there is nothing more to be done. This tactic completes the proof by finding the conclusion among the hypotheses using the memb program. The clauses for list processing predicates such as memb are straightforward and we do not show them here. The second tactic implements the implication introduction rule. In the output goal, allp is used to introduce a bound variable p to represent a proof of A. The third tactic implements the conjunction introduction rule. Note that the output goal contains the two subgoals that must be completed, separated by the & goal constructor. The fourth tactic illustrates the implementation of elimination rules as tactics. Here, nth_item is used to find the Nth hypothesis in Gamma and add two new hypotheses in the output subgoal. The nth_item program is implemented so that if the first argument is 0, it acts like memb.

Figure 3 implements some standard tacticals. The last 5 declarations are the tacticals themselves, while the remaining code is the Twelf version of the λProlog code which is needed to implement tacticals in a logic programming language.

The goalreduce code simplifies compound goals by removing all unnecessary occurrences of tt. The deterministic directive tells the logic programming engine that all queries involving the goalreduce and remove_tt predicates should succeed at most once. This directive is not as powerful as the Prolog cut (!) operator, which we use in the λProlog version of this code, but it is sufficient for this program.

The maptac tactical applies tactics to compound goals, reducing them to basic goals before

```
hyp: type.
hyps: type.
nil : hyps.
, : hyp -> hyps -> hyps.     %infix right 4 , .

by: {A} pf A -> hyp.         %infix none 5 by.
goal: type.
|- : hyps -> hyp -> goal.    %infix none 3 |- .

&: goal -> goal -> goal.     %infix right 2 &.
allp: (pf A -> goal) -> goal.
tt: goal.

tac: type.
initial_tac: tac.
imp_r_tac: tac.
and_r_tac: tac.
and_l_tac: rational -> tac.

tactic: tac -> goal -> goal -> type.
t1: tactic initial_tac (Gamma |- A by P) tt <- memb (A by P) Gamma.
t2: tactic imp_r_tac (Gamma |- (A imp B) by (imp_i P1))
                     (allp [p: pf A](A by p , Gamma |- B by (P1 p))).
t3: tactic and_r_tac (Gamma |- (A and B) by (and_i P1 P2))
                     (Gamma |- A by P1 & Gamma |- B by P2).
t4: tactic (and_l_tac N) (Gamma |- C by P)
           ((A by (and_e1 Q)) , (B by (and_e2 Q)) , Gamma |- C by P) <-
         nth_item N ((A and B) by Q) Gamma.
```

Figure 2: Goals and some tactics for first-order logic.

```
goalreduce: goal -> goal -> type.
remove_tt: goal -> goal -> type.
%deterministic goalreduce.
%deterministic remove_tt.

gr1: goalreduce (G1 & G2) RG <-
       goalreduce G1 RG1 <- goalreduce G2 RG2 <- remove_tt (RG1 & RG2) RG.
gr2: goalreduce (allp G) RG <-
       ({p} goalreduce (G p) (RG1 p)) <- remove_tt (allp RG1) RG.
gr3: goalreduce G G.

rt1: remove_tt (G & tt) G.
rt2: remove_tt (tt & G) G.
rt3: remove_tt (allp [p]tt) tt.
rt4: remove_tt G G.

maptac: tac -> goal -> goal -> type.
m1: maptac T tt tt.
m2: maptac T (InG1 & InG2) (OutG1 & OutG2) <-
     maptac T InG1 OutG1 <- maptac T InG2 OutG2.
m3: maptac T (allp InG) (allp OutG) <- {p} maptac T (InG p) (OutG p).
m4: maptac T (Gamma |- A by P) OutG <- tactic T (Gamma |- A by P) OutG.

idtac: tac.
then: tac -> tac -> tac.    %infix left 2 then.
orelse: tac -> tac -> tac.  %infix left 2 orelse.
repeat: tac -> tac.

tactical1: tactic idtac G G.
tactical2: tactic (T1 then T2) InG OutG <-
           tactic T1 InG MidG <- maptac T2 MidG OutG.
tactical3: tactic (T1 orelse T2) InG OutG <- tactic T1 InG OutG.
tactical4: tactic (T1 orelse T2) InG OutG <- tactic T2 InG OutG.
tactical5: tactic (repeat T) InG OutG <-
           tactic ((T then (repeat T)) orelse idtac) InG OutG.
```

Figure 3: Tacticals in Twelf.

passing them on to other tacticals and tactics. Of the remaining 5 tacticals, note in particular the `then` tactical, which performs the composition of tactics by first applying the first tactic, and then mapping the application of the second tactic (using `maptac`) so that the second tactic gets applied to each of the primitive subgoals within the (possibly) compound goal `MidG`.

3 Building Proofs Interactively

In λProlog, we were able to implement a query tactic that asked the user to input the next tactic to be applied, and we provided a top loop that repeatedly executed this query tactic. The lack of I/O prevents us from implementing a top loop in Twelf. Instead, we show how to build proofs by "editing" tactics. To do so, the user starts with a simple tactic and adds to it at each step. To illustrate, we prove the simple theorem $((p_1 \wedge p_2) \Rightarrow (p_2 \wedge p_1))$ which expresses the symmetry of conjunction. The following Twelf query applies the implication introduction rule.

```
%query * 1 tactic imp_r_tac
                  (nil |- ((p1 and p2) imp (p2 and p1)) by P) OutG.
```

We use logic variables `P` and `OutG` for the proof and the output subgoal containing the subgoals still to be proved after applying `imp_r_tac`, respectively. Twelf responds with the output:

```
---------- Solution 1 ----------
OutG = allp ([p4:pf (p1 and p2)] p1 and p2 by p4 , nil |- p2 and p1 by X1 p4);
P = imp_i ([x:pf (p1 and p2)] X1 x).
____________________________________________
```

`P` contains the proof constructed so far, and `OutG` tells us that there is one subgoal with hypothesis `(p1 and p2)` and conclusion `(p2 and p1)`, where `p4` is a bound variable representing a proof of the hypothesis. From this information, we can conclude that we need to apply either and-introduction or and-elimination. We choose and-introduction and edit our original query to become:

```
%query * 1 tactic (imp_r_tac then and_r_tac)
                  (nil |- ((p1 and p2) imp (p2 and p1)) by P) OutG.
```

We now obtain the output:

```
---------- Solution 1 ----------
OutG =
   allp
      ([x:pf (p1 and p2)]
          p1 and p2 by x , nil |- p2 by X1 x
             & p1 and p2 by x , nil |- p1 by X2 x);
P = imp_i ([x:pf (p1 and p2)] and_i (X1 x) (X2 x)).
____________________________________________
```

This output contains all of the information that we need to proceed, but it is getting difficult to read. We can ignore `P` because we don't need to read the proof as it is constructed, but we must be able to read `OutG` in order to choose a next step. In this case, `OutG` contains two subgoals, both within the scope of an `allp` constructor. Although this example is simple, it is clear that more complex proofs could result in complex goal structure with many subgoals hidden inside.

```
extractgoal : rational -> rational -> rational -> goal -> goal -> type.
%deterministic extractgoal.
e1: extractgoal N N (N + 1) (Gamma |- A by P) (Gamma |- A by P).
e2: extractgoal N M (M + 1) (Gamma |- A by P) tt.
e3: extractgoal N M M tt tt.
e4: extractgoal N NIn NOut (InG1 & InG2) (OutG1 & OutG2) <-
      extractgoal N NIn NMid InG1 OutG1 <-
      extractgoal N NMid NOut InG2 OutG2.
e5: extractgoal N NIn NOut (allp InG) (allp OutG) <-
      {p} extractgoal N NIn NOut (InG p) (OutG p).

extract_one_goal : rational -> tac.
e7: tactic (extract_one_goal N) InG OutG <-
      extractgoal N 1 M InG MidG <-
      goalreduce MidG OutG.
```

Figure 4: A tactic for extracting a single subgoal.

```
thenc: tac -> tac -> tac.    %infix left 2 thenc.
tactical6: tactic (T1 thenc T2) InG OutG <-
           tactic T1 InG MidG <- tactic T2 MidG OutG.

step: rational -> tac -> goal -> goal -> type.
step1 : step N T InG OutG <- tactic (T thenc (extract_one_goal N)) InG OutG.
```

Figure 5: A tactic for interaction.

In order to help with the complexity of too many subgoals, we write a tactic that allows the user to choose one subgoal to work on at each step and forces the output to show only that subgoal. This tactic, called `extract_one_goal` is implemented in Figure 4. It takes a number argument to indicate which subgoal should be displayed. The auxiliary deterministic `extractgoal` program is used to first replace all subgoals other than the chosen one with `tt`, and then `goalreduce` is used to remove these subgoals. If the user chooses a subgoal out of range, the result is just `tt`.

We then implement a `step` tactic (in Figure 5) which applies a specified tactic and then uses `extract_one_goal` to extract a specific subgoal. This tactic uses the new `thenc` tactical, which is similar to `then` but allows the second tactic to be applied to a compound subgoal directly instead of mapping its application to each primitive subgoal.

Using the new tactic, we continue our example, and extract the second subgoal with the following query.

```
%query * 1 tactic (step 2 (imp_r_tac then and_r_tac))
                  (nil |- ((p1 and p2) imp (p2 and p1)) by P) OutG.
---------- Solution 1 ----------
OutG = allp ([x:pf (p1 and p2)] p1 and p2 by x , nil |- p1 by X1 x);
P = imp_i ([x:pf (p1 and p2)] and_i (X2 x) (X1 x)).
____________________________________________
```

This particular example is simple enough that we can complete the proof by applying and-elimination in both subgoals at the same time using the following query.

```
%query * 1 tactic (step 1 (imp_r_tac then and_r_tac then
                           (and_l_tac 1) then initial_tac))
                  (nil |- ((p1 and p2) imp (p2 and p1)) by P) OutG.
---------- Solution 1 ----------
OutG = tt;
P = imp_i ([x:pf (p1 and p2)] and_i (and_e2 x) (and_e1 x)).
____________________________________________
```

The fact that we there is no subgoal `1` after applying this tactic indicates that there are no remaining subgoals, and we can see that the proof is a complete proof.

The `extract_one_subgoal` and `step` tactics are just two examples of tactics that help provide interaction in Twelf. A variety of others can and should be implemented in order to have effective interaction. For instance, unlike our simple example, it will often be desirable to apply different tactics to different subgoals. We could, for instance, implement a tactic which applies a specified tactic to one subgoal and leaves the others untouched. We could also implement, as is done in HOL, a version of `then` which takes a tactic to apply first and a list of tactics to apply after that. The list must be the same length as the number of primitive subgoals generated after applying the first tactic, and the first tactic in the list would be applied to the first primitive subgoal and so on.

4 Conclusion

We have shown how to implement tactics which help with interactive proof in Twelf, where control and I/O primitives are limited. Many of the proofs in our proof-carrying code system were done without the benefit of this prover. Instead the proof `P` in each case was built by hand using only constants like those in Figure 1. This prover, especially with its interactive component, provides quite a bit more flexibility in constructing such proofs.

In our λProlog prover, we did not need as many interactive primitives. Instead, the interactive interpreter was implemented as a top loop which asked for an input tactic at each step. The list version of `then`, for example, was not needed in the implementation of the interactive interpreter. Note also that in the Twelf prover, we must re-execute the entire tactic every time we add a step to it. This re-execution is not necessary in an interactive prover with a top loop.

References

[1] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, 2000.

[2] Andrew W. Appel and Amy P. Felty. Dependent types ensure partial correctness of theorem provers. *Journal of Functional Programming*, 2002. To appear.

[3] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):43–81, August 1993.

[4] M. J. C. Gordon and T. F. Melham. *Introduction to HOL—A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[5] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[6] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[7] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Sixteenth International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206. Springer-Verlag, 1999.

[8] The Coq Development Team. The Coq Proof Assistant reference manual: Version 7.3. Technical report, INRIA, 2002.

A The file logic.elf

```
% Copyright (C) 1999 Andrew Appel and Amy Felty.   All rights reserved.

i      : type.
o      : type.
pf     : o -> type.

%use inequality/rationals.
const  : rational -> i.

imp    : o -> o -> o.   %infix right 10 imp.
imp_i   : (pf A -> pf B) -> pf (A imp B).
imp_e   : pf (A imp B) -> pf A -> pf B.

and : o -> o -> o.   %infix right 12 and.
and_i   : pf A -> pf B -> pf (A and B).
and_e1  : pf (A and B) -> pf A.
and_e2  : pf (A and B) -> pf B.

or : o -> o -> o.    %infix right 11 or.
or_i1   : pf A -> pf (A or B).
or_i2   : pf B -> pf (A or B).
or_e    : pf (A or B) -> (pf A -> pf C) -> (pf B -> pf C) -> pf C.

forall : (i -> o) -> o.
forall_i : ({X:i} pf (A X)) -> pf (forall A).
forall_e : pf(forall A) -> {X:i} pf (A X).

exists : (i -> o) -> o.
exists_i : {X:i}pf (A X) -> pf(exists A).
exists_e : pf (exists A) -> ({X:i} pf (A X) -> pf B) -> pf B.

false: o.
false_e : pf false -> pf A.

not : o -> o = [A] A imp false.
not_i : (pf A -> pf false) -> pf (not A) = imp_i.
not_e : pf (not A) -> pf A -> pf false = imp_e.

true : o = not false.
true_i: pf (true) = not_i [P] P.
```

B The file prover.elf

```
% Copyright (C) 2002 Andrew Appel and Amy Felty.   All rights reserved.

%%%%%%%%%%%%%%%%%%%%%%%%%%%
% goals of the interpreter

goal: type.
&: goal -> goal -> goal.  %infix right 2 &.
allp: (pf A -> goal) -> goal.
```

```
alltm: (i -> goal) -> goal.
tt: goal.

tac: type.
tactic: tac -> goal -> goal -> type.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% basic definitions and utilities

hyp: type.
hyps: type.

by: {A} pf A -> hyp.   %infix none 5 by.

nil : hyps.

, : hyp -> hyps -> hyps.   %infix right 4 , .

|- : hyps -> hyp -> goal.    %infix none 3 |- .

memb: hyp -> hyps -> type.
memb1: memb H (H , Gamma).
membN: memb H1 (H2 , Gamma) <- memb H1 Gamma.

memb_and_rest: hyp -> hyps -> hyps -> type.
memb_and_rest1: memb_and_rest H1 (H1 , Gamma) Gamma.
memb_and_restN: memb_and_rest H1 (H2 , Gamma) (H2 , Rest) <-
  memb_and_rest H1 Gamma Rest.

nth_item: rational -> hyp -> hyps -> type.
nth_item1: nth_item 0 H Gamma <- memb H Gamma.
nth_item1: nth_item 1 H1 (H1 , Gamma).
nth_itemN: nth_item N H1 (H2 , Gamma) <- nth_item (N - 1) H1 Gamma.

nth_and_rest: rational -> hyp -> hyps -> hyps -> type.
nth_and_rest1: nth_and_rest 0 H Gamma Rest <-
  memb_and_rest H Gamma Rest.
nth_and_rest1: nth_and_rest 1 H1 (H1 , Gamma) Gamma.
nth_and_restN: nth_and_rest N H1 (H2 , Gamma) (H2 , Rest) <-
  nth_and_rest (N - 1) H1 Gamma Rest.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Tactics implementing inference rules

initial_tac: tac.
initial_tacN: rational -> tac.
and_r_tac: tac.
imp_r_tac: tac.
or_r1_tac: tac.
or_r2_tac: tac.
neg_r_tac: tac.
forall_r_tac: tac.
exists_r_tac: tac.
and_l_tac: rational -> tac.
```

```
and_l_tacR: rational -> tac.
imp_l_tac: rational -> tac.
imp_l_tacR: rational -> tac.
or_l_tac: rational -> tac.
or_l_tacR: rational -> tac.
neg_l_tac: rational -> tac.
neg_l_tacR: rational -> tac.
forall_l_tac: rational -> tac.
forall_l_tacR: rational -> tac.
exists_l_tac: rational -> tac.
exists_l_tacR: rational -> tac.

true_r_tac: tac.
imp_r_initial_tac: tac.
true_imp_tac: tac.
false_imp_tac: tac.

t1a: tactic initial_tac (Gamma |- A by P) tt <- memb (A by P) Gamma.
t1b: tactic (initial_tacN N) (Gamma |- A by P) tt <- nth_item N (A by P) Gamma.

t2: tactic and_r_tac (Gamma |- (A and B) by (and_i P1 P2))
                     (Gamma |- A by P1 & Gamma |- B by P2).

t3: tactic imp_r_tac (Gamma |- (A imp B) by (imp_i P1))
                     (allp [p2: pf A](A by p2 , Gamma |- B by (P1 p2))).

t4: tactic or_r1_tac (Gamma |- A or B by (or_i1 P)) (Gamma |- A by P).
t5: tactic or_r2_tac (Gamma |- A or B by (or_i2 P)) (Gamma |- B by P).

t6: tactic neg_r_tac (Gamma |- (not A) by (not_i P1))
                     (allp [p2: pf A](A by p2 , Gamma |- false by (P p2))).

t7: tactic forall_r_tac (Gamma |- (forall A) by (forall_i P))
                        (alltm [t: i](Gamma |- (A t) by (P t))).

t8: tactic exists_r_tac (Gamma |- (exists A) by (exists_i X P))
                        (Gamma |- (A X) by P).

t9: tactic (and_l_tac N)
      (Gamma |- C by P)
      ((A by (and_e1 Q)) , (B by (and_e2 Q)) , Gamma |- C by P) <-
         nth_item N ((A and B) by Q) Gamma.

t10: tactic (and_l_tacR N)
      (Gamma1 |- C by P)
      ((A by (and_e1 Q)) , (B by (and_e2 Q)) , Gamma2 |- C by P) <-
         nth_and_rest N ((A and B) by Q) Gamma1 Gamma2.

t11: tactic (imp_l_tac N)
      (Gamma |- C by P)
      ((Gamma |- A by P2) &
       ((B by (imp_e P1 P2)) , Gamma |- C by P)) <-
         nth_item N ((A imp B) by P1) Gamma.
```

```
t12: tactic (imp_l_tacR N)
      (Gamma1 |- C by P)
      ((Gamma2 |- A by P2) &
       ((B by (imp_e P1 P2)) , Gamma2 |- C by P)) <-
        nth_and_rest N ((A imp B) by P1) Gamma1 Gamma2.

t13: tactic (or_l_tac N)
      (Gamma |- C by (or_e P P1 P2))
      ((allp [p1: pf A](A by p1 , Gamma |- C by (P1 p1))) &
       (allp [p2: pf B](B by p2 , Gamma |- C by (P2 p2)))) <-
        nth_item N ((A or B) by P) Gamma.

t14: tactic (or_l_tacR N)
      (Gamma1 |- C by (or_e P P1 P2))
      ((allp [p1: pf A](A by p1 , Gamma2 |- C by (P1 p1))) &
       (allp [p2: pf B](B by p2 , Gamma2 |- C by (P2 p2)))) <-
        nth_and_rest N ((A or B) by P) Gamma1 Gamma2.

t15: tactic (neg_l_tac N)
      (Gamma |- C by P)
      ((Gamma |- A by P2) &
       ((false by (not_e P1 P2)) , Gamma |- C by P)) <-
        nth_item N ((not A) by P1) Gamma.

t16: tactic (neg_l_tacR N)
      (Gamma1 |- C by P)
      ((Gamma2 |- A by P2) &
       ((false by (not_e P1 P2)) , Gamma2 |- C by P)) <-
        nth_and_rest N ((not A) by P1) Gamma1 Gamma2.

t17: tactic (forall_l_tac N)
      (Gamma |- C by P)
      (((A X) by (forall_e Q X)) , Gamma |- C by P) <-
        nth_item N ((forall A) by Q) Gamma.

t18: tactic (forall_l_tacR N)
      (Gamma1 |- C by P)
      (((A X) by (forall_e Q X)) , Gamma2 |- C by P) <-
        nth_and_rest N ((forall A) by Q) Gamma1 Gamma2.

t19: tactic (exists_l_tac N)
      (Gamma |- C by (exists_e P1 P2))
      (alltm [t: i] allp [p: pf (A t)] (A t) by p ,
        Gamma |- C by (P2 t p)) <-
        nth_item N ((exists A) by P1) Gamma.

t20: tactic (exists_l_tacR N)
      (Gamma1 |- C by (exists_e P1 P2))
      (alltm [t:i] allp [p: pf (A t)] (A t) by p ,
        Gamma2 |- C by (P2 t p)) <-
        nth_and_rest N ((exists A) by P1) Gamma1 Gamma2.

t21: tactic true_r_tac (Gamma |- true by true_i) tt.
```

```
t22: tactic imp_r_initial_tac (Gamma |- (A imp A) by (imp_i [p] p)) tt.

t23: tactic true_imp_tac (Gamma |- true imp A by (imp_i [p] Q))
                       (Gamma |- A by Q).

t24: tactic false_imp_tac (Gamma |- false imp A by (imp_i false_e)) tt.



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% goal reduction

goalreduce: goal -> goal -> type.
remove_tt: goal -> goal -> type.
%deterministic goalreduce.
%deterministic remove_tt.

gr1: goalreduce (G1 & G2) RG <-
       goalreduce G1 RG1 <- goalreduce G2 RG2 <- remove_tt (RG1 & RG2) RG.
gr2: goalreduce (allp G) RG <-
       ({p} goalreduce (G p) (RG1 p)) <- remove_tt (allp RG1) RG.
gr3: goalreduce (alltm G) RG <-
       ({t} goalreduce (G t) (RG1 t)) <- remove_tt (alltm RG1) RG.
gr4: goalreduce G G.

rt1: remove_tt (G & tt) G.
rt2: remove_tt (tt & G) G.
rt3: remove_tt (allp [p]tt) tt.
rt4: remove_tt (alltm [t]tt) tt.
rt5: remove_tt G G.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% the tacticals (an interpreter for tactics)

maptac: tac -> goal -> goal -> type.

m1: maptac T tt tt.
m2: maptac T (InG1 & InG2) (OutG1 & OutG2) <-
     maptac T InG1 OutG1 <- maptac T InG2 OutG2.
m3: maptac T (allp InG) (allp OutG) <- {p} maptac T (InG p) (OutG p).
m4: maptac T (alltm InG) (alltm OutG) <- {t} maptac T (InG t) (OutG t).
m5: maptac T (Gamma |- A by P) OutG <- tactic T (Gamma |- A by P) OutG.

idtac: tac.
then: tac -> tac -> tac.    %infix left 2 then.
thenc: tac -> tac -> tac.    %infix left 2 thenc.
orelse: tac -> tac -> tac.  %infix left 2 orelse.
repeat: tac -> tac.
try: tac -> tac.
complete: tac -> tac.

tactical1: tactic idtac G G.
tactical2: tactic (T1 then T2) InG OutG <-
            tactic T1 InG MidG <- maptac T2 MidG OutG.
```

```
tactical3: tactic (T1 orelse T2) InG OutG <- tactic T1 InG OutG.
tactical4: tactic (T1 orelse T2) InG OutG <- tactic T2 InG OutG.
tactical5: tactic (repeat T) InG OutG <-
            tactic ((T then (repeat T)) orelse idtac) InG OutG.
tactical6: tactic (try T) InG OutG <- tactic (T orelse idtac) InG OutG.
tactical7: tactic (complete T) InG tt <-
            tactic T InG OutG <- goalreduce OutG tt.
tactical8: tactic (T1 thenc T2) InG OutG <-
            tactic T1 InG MidG <- tactic T2 MidG OutG.

extractgoal : rational -> rational -> rational -> goal -> goal -> type.
%deterministic extractgoal.
e1: extractgoal N N (N + 1) (Gamma |- A by P) (Gamma |- A by P).
e2: extractgoal N M (M + 1) (Gamma |- A by P) tt.
e3: extractgoal N M M tt tt.
e4: extractgoal N NIn NOut (InG1 & InG2) (OutG1 & OutG2) <-
      extractgoal N NIn NMid InG1 OutG1 <-
      extractgoal N NMid NOut InG2 OutG2.
e5: extractgoal N NIn NOut (allp InG) (allp OutG) <-
      {p} extractgoal N NIn NOut (InG p) (OutG p).
e6: extractgoal N NIn NOut (alltm InG) (alltm OutG) <-
      {t} extractgoal N NIn NOut (InG t) (OutG t).

extract_one_goal : rational -> tac.
e7: tactic (extract_one_goal N) InG OutG <-
      extractgoal N 1 M InG MidG <-
      goalreduce MidG OutG.

step: rational -> tac -> tac.
step1 : tactic (step N T) InG OutG <-
         tactic (T thenc (extract_one_goal N)) InG OutG.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% example compound tactics

repeatall: tac =
  (repeat (initial_tac orelse true_r_tac orelse and_r_tac orelse imp_r_tac
           orelse or_r1_tac orelse or_r2_tac orelse imp_r_initial_tac
           orelse true_imp_tac orelse false_imp_tac)).

fo_auto: tac =
  (repeat (initial_tac orelse (and_l_tacR 0) orelse imp_r_tac orelse
           (exists_l_tacR 0) orelse forall_r_tac orelse
           (or_l_tacR 0) orelse and_r_tac orelse (imp_l_tacR 0) orelse
           neg_r_tac orelse or_r1_tac orelse or_r2_tac orelse
           (neg_l_tacR 0) orelse exists_r_tac orelse
           (forall_l_tacR 0))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% example queries

p1: o.
p2: o.
p3: o.
```

```
% Example 1

%query * 1 tactic imp_r_tac
                  (nil |- ((p1 and p2) imp (p2 and p1)) by P) OutG.

%query * 1 tactic (imp_r_tac then and_r_tac)
                  (nil |- ((p1 and p2) imp (p2 and p1)) by P) OutG.

%query * 1 tactic (step 2 (imp_r_tac then and_r_tac))
                  (nil |- ((p1 and p2) imp (p2 and p1)) by P) OutG.

%query * 1 tactic (imp_r_tac then and_r_tac then (and_l_tac 1) then
     initial_tac)
                  (nil |- ((p1 and p2) imp (p2 and p1)) by P) OutG.

%query * 1 tactic fo_auto
                  (nil |- ((p1 and p2) imp (p2 and p1)) by P) OutG.

% Example 2

%query * 1 tactic (repeat and_r_tac) (nil |- (p1 and p2 and p3) by P) OutG.

%query * 1 tactic repeatall (nil |- (p1 and p2 and p3) by P) OutG.

%query * 1 tactic fo_auto (nil |- (p1 and p2 and p3) by P) OutG.

%query * 1 tactic (extract_one_goal 3)
           (nil |- p1 by X1 & nil |- p2 by X2 & nil |- p3 by X3) G.

%query * 1 tactic (step 3 (repeat and_r_tac))
                  (nil |- (p1 and p2 and p3) by P) G.

q: i -> o.
a: i.
b: i.

% Example 3

%query * 1 tactic imp_r_tac
                (nil |- (((q a) or (q b)) imp (exists [x](q x))) by P) G.

%query * 1 tactic (step 1 (imp_r_tac then (or_l_tacR 0) then
                           exists_r_tac then initial_tac))
                (nil |- (((q a) or (q b)) imp (exists [x](q x))) by P) G.

%query * 1 tactic fo_auto
                  (nil |- (((q a) or (q b)) imp (exists [x](q x))) by P) OutG.
```