Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A NOTE ON INTERACTIVE THEOREM PROVING WITH THEOREM CONTINUATION FUNCTIONS

C.-T. Chou

June 1992
CSD-920025

A Note on Interactive Theorem Proving with Theorem Continuation Functions

*Ching-Tsun Chou**
chou@cs.ucla.edu

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024, U.S.A.

June 7, 1992

Abstract

A simple technique for using theorem continuation functions interactively with HOL's subgoal package is presented. An interesting aspect of the technique is that it hinges on the availability of assignable variables in ML.

1 An Example

An important notion in tactic-based theorem proving in HOL [1, 2] is that of *theorem continuations*[1]. A theorem continuation is an ML function of type `thm -> tactic` (abbreviated as `thm_tactic`), which transforms a theorem into a tactic for continuing the proof (hence the name). A simple example of theorem continuations is:[2]

```
#let ttac : thm_tactic =                                                    1
#  \ t . let (l,r) = CONJ_PAIR t in
#        CONJ_TAC THENL [ACCEPT_TAC r; ACCEPT_TAC l] ;;
ttac = - : thm_tactic
```

Theorem continuation `ttac` splits the input theorem `t` (which is assumed to be conjunctive) into its left conjunct `l` and right conjunct `r` using `CONJ_PAIR`, then reduces the goal (which is also assumed to be conjunctive) into two subgoals corresponding to its two conjuncts using `CONJ_TAC`, and finally solves the left (right) subgoal with theorem `r` (`l`) using `ACCEPT_TAC`. Schematically, `ttac` can be depicted thus:

*Supported by IBM Graduate Fellowship.

[1] There is a resemblance between theorem continuations in HOL and continuations in Denotational Semantics, but the reader need not know the latter in order to understand the former or the technique presented in this note.

[2] HOL sessions are displayed in rectangular windows, each of which is labelled by a sequence number shown at the upper-right corner. Side-effects produced in lower-numbered windows persist into higher-numbered ones until the number is reset to 1, which marks the beginning of a new session.

```
          ?- r /\ l
==================================  ttac (|- l /\ r)
   ?- r              ?- l
========  |- r    ========  |- l
```

An important method of using theorem continuations is to use them as arguments of *theorem continuation functions*, an example of which is:

```
#DISCH_THEN ;;                                                    2
- : (thm_tactic -> tactic)
```

When applied to an implicative goal, DISCH_THEN removes the antecedent from the goal, creates a theorem by assuming the antecedent, produces a tactic by applying its first argument (which is a theorem continuation) to the theorem, and reduces the succedent of the original goal using the resulting tactic. Schematically, if

```
  ?- u
========  ttac (t |- t)
  ?- v
```

(read: under the assumption t, to prove u, it suffices to prove v), then

```
  ?- t ==> u
==============  DISCH_THEN ttac
     ?- v
```

(read: to prove t ==> u, it suffices to prove v). The following session shows how DISCH_THEN and ttac work together:

```
#g "a /\ b ==> b /\ a" ;;                                         3
"a /\ b ==> b /\ a"

() : void

#expand (DISCH_THEN ttac) ;;
OK..
goal proved
|- a /\ b ==> b /\ a

Previous subproof:
goal proved
() : void
```

Notice how closely the pattern of inference effected by (DISCH_THEN ttac) corresponds to the intuitive argument one uses to prove a /\ b ==> b /\ a. Indeed, if we substitute the definition of ttac for ttac, we get:

```
DISCH_THEN \ t .
  let (l,r) = CONJ_PAIR t in
  CONJ_TAC THENL
  [ ACCEPT_TAC r ;
    ACCEPT_TAC l ]
```

which can be read line-for-line as expressing the following informal proof of a /\ b ==> b /\ a:

- Assume the antecedent `t = a /\ b` is true.
  1. Hence both `l = a` and `r = b` are true.
  2. To prove `b /\ a`, it suffices to prove both `b` and `a`.
     (a) Assumption `r` proves `b`.
     (b) Assumption `l` proves `a`.

All built-in theorem continuation functions in HOL88 (*viz.*, those ML functions with names ending with '`_THEN`', '`_THENL`' or '`_THEN2`') afford equally intuitive and natural interpretations.

2 The Technique

In the above example the goal `?- a /\ b ==> b /\ a` is so simple that it is trivial to figure out what the theorem continuation `ttac` should be, once the meaning of `DISCH_THEN` is understood. But what if the goal is more complex?

Customarily, a tactic for solving a complex goal is constructed by *interactively* building and traversing the proof tree using HOL's *subgoal package* [1, 3]. It is crucial to be able to perform the proof search interactively, for theorem proving is computationally too hard to be fully automated, and to have a tool like the subgoal package to do all the bookkeeping, for it is too tedious and error-prone for a human to keep track of all the details. But can we construct complex theorem continuations, not just tactics, interactively using the subgoal package?

A theorem continuation `ttac` is an ML function of type `thm -> tactic`. Obviously, without knowing the value of its theorem argument (call it `t`), `ttac` (more precisely, the part of `ttac` that has been constructed) cannot be interactively tested. The problem is: How does the user generate the correct value of `t` during an interactive session?

When a theorem continuation `ttac` is used as an argument of a theorem continuation function `tcl`, `ttac`'s input theorem `t` is produced by `tcl` either from the goal to be proved via the inverse of an introduction rule (*e.g.*, when `tcl` is `DISCH_THEN`), or from an already generated theorem via an elimination rule (*e.g.*, when `tcl` is `CHOOSE_THEN`; see the example in Section 3), or from a combination of both. It is possible, in principle at least, to generate the correct value of `t` by mimicking `tcl` manually. But this approach is as tedious and error-prone as doing interactive proofs manually without the benefit of the subgoal package. The contribution of this note is to present a technique for overcoming this difficulty.

The basic idea behind the technique is very simple: Have `ttac` assign the value of its theorem argument to an *assignable* variable which is *not* local to, and hence can be accessed from outside, `ttac`.

```
#letref t = ARB_THM ;;    % Initialize t to some arbitrary theorem %          I
t = |- $= = $=

#let ttac : thm_tactic = ( \t' . t := t' ; ALL_TAC ) ;;
ttac = - : thm_tactic
```

It is essential that `t` be an assignable variable, since non-assignable variables (*i.e.*, those variables declared with `let` instead of `letref`) cannot be re-assigned a new value. Let us examine the behavior of `ttac` by re-doing the previous example:

```
#g "a /\ b ==> b /\ a" ;;                                                    2
"a /\ b ==> b /\ a"

() : void

#expandf (DISCH_THEN ttac) ;;
OK..
"b /\ a"

() : void

#t ;;
a /\ b |- a /\ b
```

Thus the theorem that DISCH_THEN feeds ttac with has been 'captured' and stored in t, which can be accessed globally. The proof can now be finished as in the previous example:

```
#let (l,r) = CONJ_PAIR t ;;                                                 3
l = a /\ b |- a
r = a /\ b |- b

#expandf (CONJ_TAC THENL [ACCEPT_TAC r; ACCEPT_TAC l]) ;;
OK..
goal proved
. |- b /\ a
|- a /\ b ==> b /\ a

Previous subproof:
goal proved
() : void
```

Notice that we must use expandf instead of expand in the last step:

```
#backup () ;;                                                               4
"b /\ a"

() : void

#expand (CONJ_TAC THENL [ACCEPT_TAC r; ACCEPT_TAC l]) ;;
OK..
evaluation failed     Invalid tactic
```

The reason why expand fails is that since we have used DISCH_THEN, the last goal has no assumption at all. But theorems l and r both have the assumption a /\ b, which causes the validity check of expand to fail. Since our technique results in a proof style which is often incompatible with the default validity check of expand, we will in the sequel use expandf exclusively during interactive construction of theorem continuations and adopt the following abbreviation:

```
#let f = expandf ;;                                                         1
f = - : (tactic -> void)
```

But we will continue to use expand when a single, final tactic for solving the top-level goal has been constructed.

The basic technique demonstrated above can be refined. The assignable variable to which `ttac` assigns the value of its theorem argument does *not* have to be global. It is sufficient to have a local (hence anonymous) assignable variable to hold the 'captured' theorem, which is then returned as a function value. Furthermore, instead of writing a special piece of code for (the initial skeleton of) each theorem continuation that we might want to plug into `DISCH_THEN`, we can define a *uniform* transformation for all theorem continuation functions of type `thm_tactic -> tactic`. The prefix '`f_`' (obviously) arises from our abbreviating `expandf` as `f`.

2
```
#let f_ttac_tac (ttac_tac : thm_tactic -> tactic) : void -> thm =
#  letref th = ARB_THM in
#  let ttac : thm_tactic = ( \ th' . th := th' ; ALL_TAC ) in
#  ( \ () . f (ttac_tac ttac) ; th )
#;;
f_ttac_tac = - : ((thm_tactic -> tactic) -> void -> thm)

#let f_DISCH_THEN = f_ttac_tac DISCH_THEN ;;
f_DISCH_THEN = - : (void -> thm)
```

Now we can have an interactive proof which is almost identical to the previous one:

3
```
#g "a /\ b ==> b /\ a" ;;
"a /\ b ==> b /\ a"

() : void

#let t = f_DISCH_THEN () ;;
OK..
"b /\ a"

t = a /\ b |- a /\ b

#let (l,r) = CONJ_PAIR t ;;
l = a /\ b |- a
r = a /\ b |- b

#f (CONJ_TAC THENL [ACCEPT_TAC r; ACCEPT_TAC l]) ;;
OK..
goal proved
. |- b /\ a
|- a /\ b ==> b /\ a

Previous subproof:
goal proved
() : void
```

3 Another Example

The theorem continuation function

```
CHOOSE_THEN : thm_tactic -> thm  -> tactic
```

can be described schematically as follows. If

```
  ?- u
========  ttac (t[x'/x] |- t[x'/x])
  ?- v
```

then

```
  ?- u
========  CHOOSE_THEN ttac (|- ?x.t)
  ?- v
```

where x' is a variant of x chosen not to be free in the assumption list of the goal. In other words, CHOOSE_THEN uses an existentially quantified theorem by instantiating it to a particular but arbitrary witness. Analogous to f_ttac_tac and f_DISCH_THEN, we can define:

```
#let f_ttac_ttac (ttac_ttac : thm_tactic -> thm -> tactic) : void -> thm -> thm =
#  letref th = ARB_THM in
#  let ttac : thm_tactic = ( \ th' . th := th' ; ALL_TAC ) in
#  ( \ () t . f (ttac_ttac ttac t) ; th )
#;;
f_ttac_ttac = - : (thm_tactical -> void -> thm -> thm)

#let f_CHOOSE_THEN = f_ttac_ttac CHOOSE_THEN ;;
f_CHOOSE_THEN = - : (void -> thm -> thm)
```

4

In the following we suppress the printing of the assumption lists of theorems, since they can be very long:

```
#top_print print_thm ;;
- : (thm -> void)
```

5

Now consider the goal:

```
#g "(?n1. !n. n >= n1 ==> P1 n) /\ (?n2. !n. n >= n2 ==> P2 n) ==>
#   (?n3. !n. n >= n3 ==> P1 n /\ P2 n)"
#;;
"(?n1. !n. n >= n1 ==> P1 n) /\ (?n2. !n. n >= n2 ==> P2 n) ==>
 (?n3. !n. n >= n3 ==> P1 n /\ P2 n)"

() : void
```

6

Since the goal is implicative, the obvious thing to do is to strip and assume the antecedent:

```
#let p = f_DISCH_THEN () ;;
OK..
"?n3. !n. n >= n3 ==> P1 n /\ P2 n"

p = . |- (?n1. !n. n >= n1 ==> P1 n) /\ (?n2. !n. n >= n2 ==> P2 n)

#let (p1,p2) = CONJ_PAIR p ;;
p1 = . |- ?n1. !n. n >= n1 ==> P1 n
p2 = . |- ?n2. !n. n >= n2 ==> P2 n
```

7

Now we have two existentially quantified theorems p1 and p2. The next thing to do is to use them by means of CHOOSE_THEN:

```
#let p1' = f_CHOOSE_THEN () p1 ;;                                    8
OK..
"?n3. !n. n >= n3 ==> P1 n /\ P2 n"

p1' = . |- !n. n >= n1 ==> P1 n

#let p2' = f_CHOOSE_THEN () p2 ;;
OK..
"?n3. !n. n >= n3 ==> P1 n /\ P2 n"

p2' = . |- !n. n >= n2 ==> P2 n
```

At this stage we are ready to solve the existential goal. A suitable witness for n3 is n1 + n2:

```
#f (EXISTS_TAC "n1 + n2") ;;                                         9
OK..
"!n. n >= (n1 + n2) ==> P1 n /\ P2 n"

() : void

#f (GEN_TAC) ;;
OK..
"n >= (n1 + n2) ==> P1 n /\ P2 n"

() : void

#let q = f_DISCH_THEN () ;;
OK..
"P1 n /\ P2 n"

q = . |- n >= (n1 + n2)
```

Suppose the following theorems have already been proved:

```
#(th1,th2) ;;                                                        10
(|- !n1 n2 n. n >= (n1 + n2) ==> n >= n1,
 |- !n1 n2 n. n >= (n1 + n2) ==> n >= n2)
: (thm # thm)
```

Then some forward reasoning would generate suitable theorems to finish the proof:

```
#let q1 = itlist MATCH_MP [p1'; th1] q                               11
#and q2 = itlist MATCH_MP [p2'; th2] q ;;
q1 = .. |- P1 n
q2 = .. |- P2 n
```

```
#f (ACCEPT_TAC (CONJ q1 q2)) ;;                                          12
OK..
goal proved
... |- P1 n /\ P2 n
.. |- n >= (n1 + n2) ==> P1 n /\ P2 n
.. |- !n. n >= (n1 + n2) ==> P1 n /\ P2 n
.. |- ?n3. !n. n >= n3 ==> P1 n /\ P2 n
.. |- ?n3. !n. n >= n3 ==> P1 n /\ P2 n
. |- ?n3. !n. n >= n3 ==> P1 n /\ P2 n
|- (?n1. !n. n >= n1 ==> P1 n) /\ (?n2. !n. n >= n2 ==> P2 n) ==>
   (?n3. !n. n >= n3 ==> P1 n /\ P2 n)

Previous subproof:
goal proved
() : void
```

Finally, we can condense the whole proof session into one single tactic, which we use **expand** to test:

```
#g "(?n1. !n. n >= n1 ==> P1 n) /\ (?n2. !n. n >= n2 ==> P2 n) ==>          13
#   (?n3. !n. n >= n3 ==> P1 n /\ P2 n)"
#;;
"(?n1. !n. n >= n1 ==> P1 n) /\ (?n2. !n. n >= n2 ==> P2 n) ==>
 (?n3. !n. n >= n3 ==> P1 n /\ P2 n)"

() : void

#expand (
#  DISCH_THEN \ p .
#    let (p1,p2) = CONJ_PAIR p
#    in
#    CHOOSE_THEN ( \ p1' .
#    CHOOSE_THEN ( \ p2' .
#      EXISTS_TAC "n1 + n2" THEN
#      GEN_TAC THEN
#      DISCH_THEN \ q .
#        let q1 = itlist MATCH_MP [p1'; th1] q
#        and q2 = itlist MATCH_MP [p2'; th2] q
#        in
#        ACCEPT_TAC (CONJ q1 q2)
#    ) p2
#    ) p1
#) ;;
OK..
goal proved
|- (?n1. !n. n >= n1 ==> P1 n) /\ (?n2. !n. n >= n2 ==> P2 n) ==>
   (?n3. !n. n >= n3 ==> P1 n /\ P2 n)

Previous subproof:
goal proved
() : void
```

4 The Code

Analogous to `f_ttac_tac` and `f_ttac_ttac`, we can define a uniform transformation for each type of built-in theorem continuation functions in HOL88. Notice that these definitions are needed only during interactive construction of theorem continuation arguments of theorem continuation functions. Once a proof is completed, the record of interaction can be packaged into a single tactic containing no '`f_...`' functions, as demonstrated in the previous example. Also notice that our technique applies, *mutatis mutandis*, to other LCF-style systems, such as Cambridge LCF [3], as well.

```
let f = expandf ;;


let f_ttac_tac (ttac_tac : thm_tactic -> tactic)
    : void -> thm =
  letref th = ARB_THM
  in
  let ttac : thm_tactic = ( \ th' . th := th' ; ALL_TAC )
  in
  ( \ () . f (ttac_tac ttac) ; th )
;;

let f_DISCH_THEN              = f_ttac_tac DISCH_THEN
and f_INDUCT_THEN (th : thm)  = f_ttac_tac (INDUCT_THEN th)
and f_RES_THEN                = f_ttac_tac RES_THEN
and f_STRIP_GOAL_THEN         = f_ttac_tac STRIP_GOAL_THEN
and f_SUBGOAL_THEN (t : term) = f_ttac_tac (SUBGOAL_THEN t)
;;

let f_ttac_ttac (ttac_ttac : thm_tactic -> thm -> tactic)
    : void -> thm -> thm =
  letref th = ARB_THM
  in
  let ttac : thm_tactic = ( \ th' . th := th' ; ALL_TAC )
  in
  ( \ () t . f (ttac_ttac ttac t) ; th )
;;

let f_ALL_THEN                          = f_ttac_ttac ALL_THEN
and f_ANTE_RES_THEN                     = f_ttac_ttac ANTE_RES_THEN
and f_CHOOSE_THEN                       = f_ttac_ttac CHOOSE_THEN
and f_CONJUNCTS_THEN                    = f_ttac_ttac CONJUNCTS_THEN
and f_DISJ_CASES_THEN                   = f_ttac_ttac DISJ_CASES_THEN
and f_FREEZE_THEN                       = f_ttac_ttac FREEZE_THEN
and f_IMP_RES_THEN                      = f_ttac_ttac IMP_RES_THEN
and f_NO_THEN                           = f_ttac_ttac NO_THEN
and f_STRIP_THM_THEN                    = f_ttac_ttac STRIP_THM_THEN
and f_X_CASES_THEN (xll: term list list) = f_ttac_ttac (X_CASES_THEN xll)
and f_X_CHOOSE_THEN (x : term)          = f_ttac_ttac (X_CHOOSE_THEN x)
;;
```

```
let f_ttac_ftac (ttac_ftac : thm_tactic -> term -> tactic)
    : void -> term -> thm =
  letref th = ARB_THM
  in
  let ttac : thm_tactic = ( \ th' . th := th' ; ALL_TAC )
  in
  ( \ () x . f (ttac_ftac ttac x) ; th )
;;

let f_FILTER_DISCH_THEN = f_ttac_ftac FILTER_DISCH_THEN
and f_FILTER_STRIP_THEN = f_ttac_ftac FILTER_STRIP_THEN
;;

let f_ttac_ttac_ttac (ttac_ttac_ttac : thm_tactic -> thm_tactic -> thm -> tactic)
    : void -> void -> thm -> (thm # thm) =
  letref th1 = ARB_THM and th2 = ARB_THM
  in
  let ttac1 : thm_tactic = ( \ th1' . th1 := th1' ; ALL_TAC )
  and ttac2 : thm_tactic = ( \ th2' . th2 := th2' ; ALL_TAC )
  in
  ( \ () () t . f (ttac_ttac_ttac ttac1 ttac2 t) ; (th1,th2) )
;;

let f_CONJUNCTS_THEN2  = f_ttac_ttac_ttac CONJUNCTS_THEN2
and f_DISJ_CASES_THEN2 = f_ttac_ttac_ttac DISJ_CASES_THEN2
;;

let f_ttacl_ttac (ttacl_ttac : thm_tactic list -> thm -> tactic)
    : void list -> thm -> thm list =
  letref thl = [ ] : thm list
  in
  let ttacl : int -> thm_tactic list =
      letrec ttacl' (m : int) =
        if (m = 0) then [ ]
        else ( \ th' . thl := thl @ [th'] ; ALL_TAC ).(ttacl' (m - 1))
      in
      ( \ n . thl := [ ] ; ttacl' n)
  in
  ( \ vl t . f (ttacl_ttac (ttacl (length vl)) t) ; thl )
;;

let f_CASES_THENL                          = f_ttacl_ttac CASES_THENL
and f_DISJ_CASES_THENL                     = f_ttacl_ttac DISJ_CASES_THENL
and f_X_CASES_THENL (xll: term list list) = f_ttacl_ttac (X_CASES_THENL xll)
;;
```

Acknowledgements

Tom Melham reminded me of the availability of assignable variables in ML. Ray Toal prompted me to write this note and also read an early draft. Peter Homeier, Sara Kalvala and Phil Windley

each made suggestions which greatly improved the presentation of this note. I am grateful to all of them.

References

[1] DSTO and SRI International, *The HOL System: DESCRIPTION*, (1991).

[2] M. J. C. Gordon, "HOL: A Proof Generating System for Higher-Order Logic", in *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P. A. Subrahmanyam, (Kluwer Academic Publishers, 1988), pp. 73–128.

[3] L. C. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge Tracts in Theoretical Computer Science 2 (Cambridge University Press, 1987).