Technical Report

UCAM-CL-TR-39
ISSN 1476-2986

Number 39

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Tactics and tacticals in Cambridge LCF

Lawrence Paulson

July 1983

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Tactics and Tacticals in Cambridge LCF

Lawrence Paulson

University of Cambridge

July 1983

The tactics and tacticals of Cambridge LCF are described. Tactics reason about logical connectives, substitution, and rewriting; tacticals combine tactics into more powerful tactics. LCF's package for managing an interactive proof is discussed. This manages the subgoal tree, presenting the user with unsolved goals and assembling the final proof.

While primarily a reference manual, the paper contains a brief introduction to goal-directed proof. An example shows typical use of the tactics and subgoal package.

Table of Contents

1. Introduction

Cambridge LCF is a proof assistant derived from Edinburgh LCF [4]. Among its changes are a revised version of the logic PPLAMBDA, as already documented [7]. Tactics and tacticals are also different in Cambridge LCF. This paper is more a reference manual than a tutorial; I expect that you have already read an introduction to LCF [4,5].

Since I am one of the main implementors of Cambridge LCF, its tactics reflect my personal preferences, based on experience in several case studies. While other researchers may differ in the exact approach, the tactics described below are basic and widely applicable.

I would like to thank Mike Gordon for his comments on the paper.

2. Fundamentals of Tactics

Tactics accomplish goal-directed proof -- you begin with a statement of the desired theorem and reduce it to simpler ones. LCF uses a natural deduction logic [6], called PPLAMBDA; a theorem

$$[A_1; \ldots; A_n] \vdash B$$

states that the conclusion B is true whenever the assumptions A_1, ..., A_n are true. A goal is a pair

$$([A_1; \ldots; A_n] , B)$$

expressing the assumptions and conclusion of the desired theorem.

2.1. Validations

Let AL stand for a list of assumptions. A tactic is a function that maps a goal (AL,B) to a pair

$$([(AL_1,B_1); \ldots; (AL_n,B_n)], \text{ proof\_function })$$

The pair consists of a list of subgoals and a proof function, called the validation. The validation maps theorems asserting the subgoals,

$$AL_1 \vdash B_1 \quad \ldots \quad AL_n \vdash B_n$$

to the theorem asserting the original goal, AL⊢B. This proves the goal from theorems asserting the subgoals. If you prove the subgoals using other tactics, then you must compose their validations with the current one to reconstruct the entire proof. LCF composes the validations automatically when you perform a proof using tacticals and the subgoal package, described later in this paper.

There is no guarantee that the validation will prove the original goal. A faulty tactic could return a validation that fails or loops, or that produces a theorem with unwanted assumptions or the wrong conclusion. Such a tactic is called invalid, and is worse than useless because it can lead you down an incorrect path in a proof. Fortunately it is impossible to prove a false theorem in LCF.

Even standard tactics are valid only for goals that satisfy conditions documented below. Tactics do not always check these conditions in advance, for efficiency.

LCF includes the predefined ML types

```
lettype proof = thm list -> thm;;
lettype goal = form list # form;;
lettype tactic = goal -> ((goal list) # proof);;
```

Note that the function dest_thm has type "thm->goal".

2.2. Programming with tactics

Tactics may be combined into more powerful ones using operators called tacticals, described later. While a proof may require a special-purpose tactic, it is almost never necessary to program one from scratch, explicitly constructing the subgoal list and validation. New tactics are generally expressed in terms of standard tactics and tacticals; these may be regarded as a language for proof strategies.

Many standard tactics are mainly intended as building blocks for constructing other tactics. Tactics built from simpler ones are more readable and more likely to be correct than tactics built from scratch.

2.3. Notation

A tactic that reduces a goal B to subgoals B_1, ..., B_n, with the same assumptions, is written

$$\frac{B}{B_1 \quad \ldots \quad B_n}$$

A tactic typically passes all the original assumptions on to the subgoals, and perhaps adds new assumptions. Changes to the assumption list are noted in square brackets. A tactic that reduces a goal B to C and introduces assumptions A_1 and A_2 is written

```
        B
==============
[A1; A2] C
```

Most tactics apply only to goals of a certain form, and fail on all others. For a tactic such as

```
  A /\ B
==========
  A    B
```

it is implicit that only conjunctions are acceptable as the input goal.

3. Predicate Calculus Tactics

Many predicate calculus rules are well-suited for backwards proof. A tactic whose validation uses a single inference rule is said to *invert* that rule. A tactic is often named after the rule that it inverts, having the reverse effect of what its name suggests. For example, the tactic SPEC_TAC generalizes its goal. Its validation uses the inference rule SPEC, which specializes a theorem.

3.1. Tactics for quantifiers

Forall -- proving a generalization for an arbitrary variable

```
GEN_TAC: tactic

  !x.A(x)
 =========
   A(x')
```

chooses x' as a variant of x not free in goal or assumptions

Forall -- proving a generalization for a particular variable
(for use in tactical programming)

```
X_GEN_TAC: term -> tactic
           y

  !x.A(x)
 =========
   A(y)
```

valid if the variable y is not free in goal or assumptions

Forall -- proving a more general goal
(related to Boyer and Moore's "generalization" [1])

```
SPEC_TAC: (term # term) -> tactic
            t      x

         A(t)
     ============
       !x.A(x)
```

valid if x is not free in A(t), except within t;
equivalently, if x is not free in A(UU)

Exists -- proving existence by stating a value

```
EXISTS_TAC: term -> tactic
              t

       ?x.A(x)
    ============
       A(t)
```

3.2. Tactics for connectives

Conjunction -- proving the conjuncts separately

CONJ_TAC: tactic

```
      A /\ B
   ============
      A    B
```

Disjunction -- selecting which disjunct to prove

DISJ1_TAC: tactic

```
     A \/ B
   ==========
       A
```

DISJ2_TAC: tactic

```
     A \/ B
   ==========
       B
```

Implication -- assuming the antecedent

DISCH_TAC: tactic

```
       A ==> B
   ==============
      [ A ]  B
```

Implication -- taking a theorem as an antecedent

MP_TAC: thm -> tactic

```
       |-A

         B
   ===============
       A ==> B
```

If-and-only-if -- proving each direction separately

IFF_TAC: tactic

```
            A <=> B
    ============================
       A ==> B          B ==> A
```

4. Additional Tactics and Functions

These include tactics for substitution, induction, manipulating assumptions, and performing case splits. Tactics that take theorem parameters, such as SUBST_TAC, CONTR_TAC, and MP_TAC, should be supplied with theorems that depend on no assumptions other than those of the goal. Any additional assumptions will crop up when the validation is applied, rendering the tactic invalid.

Substitution in the goal (at specified occurrence numbers)

SUBST_TAC: (thm list) -> tactic
SUBST_OCCS_TAC: ((int list) # thm) list -> tactic
for theorems [$\vdash t_1 == u_1$; ...; $\vdash t_n == u_n$]
with occurrence lists [o_1; ...; o_k] for each theorem

```
      A(t1, ..., tn)
    ====================
      A(u1, ..., un)
```

Simple substitution (useful with tacticals)

SUBST1_TAC: thm -> tactic
$\vdash t == u$

```
        B(t)
    ============
        B(u)
```

Substitution in the goal and its assumptions

```
SUBST_ALL_TAC: thm -> tactic
               |-t==u

      [ AL(t) ] B(t)
    ==================
      [ AL(u) ] B(u)
```

The symbol AL stands for the entire assumption list. SUBST_ALL_TAC breaches the style of natural deduction, where the assumptions are kept fixed. However, the tactic is valid and occasionally useful.

Fixed-point induction
(uses type thms and TR_CASES in testing admissability [7])

```
INDUCT_TAC: (thm list) -> (term # term)list -> tactic
            type thms      fun_i   f_i

            A(FIX fun_1, ..., FIX fun_n)
=========================================================
                          A(UU)
!f_1...f_n. A(f_1, ..., f_n)  ==>  A(fun_1 f_1, ..., fun_n f_n)
```

Case split on a conditional expression
(searches goal for a conditional and expands the three cases)

```
COND_CASES_TAC: tactic

        A(p=>t|u)
    =================
      [p==UU] A(UU)
      [p==TT] A(t)
      [p==FF] A(u)
```

Contradiction

```
CONTR_TAC:  thm  -> tactic
        ¦-FALSITY()

          A
    ============
          -
```

Accepting a theorem that states the goal

```
ACCEPT_TAC:  thm  -> tactic
             ¦-A

          A
    ============
          -
```

Taking a theorem as an assumption

```
ASSUME_TAC: thm -> tactic
            ¦-A

         C
    ============
       [A] C
```

4.1. Stripping connectives from a theorem

```
STRIP_ASSUME_TAC: thm -> tactic
```

ASSUME_TAC always puts the theorem, exactly as given, onto the assumption list. STRIP_ASSUME_TAC first breaks the theorem apart, stripping off certain outer connectives, then puts the resulting pieces on the assumption list. It adds a conjunction as separate conjuncts, validated by the inference rules CONJUNCT1 and CONJUNCT2. It causes a case split given a disjunction, validated by the rule DISJ_CASES. It eliminates an existential quantifier by choosing an arbitrary variable, validated by the rule CHOOSE.

It also attempts to solve the goal using ACCEPT_TAC and CONTR_TAC.

The tactic is implemented using theorem continuations [8]. Its effect is suggested by the following diagrams, except that the new assumptions cause recursive calls to STRIP_ASSUME_TAC whenever possible. Thus it does not add contradictions, conjunctions, disjunctions, or existential formulas to the assumption list.

for |-A/\B

```
        C
   =============
     [A; B] C
```

for |-A\/B

```
             C
   ==================
    [A] C     [B] C
```

for |-?x.A(x) (chooses x' as a variant of x)

```
          C
   ================
     [ A(x') ] C
```

4.2. Stripping connectives from a goal

STRIP_TAC: tactic

This is for breaking a goal apart. STRIP_TAC removes one outer connective from the goal, using CONJ_TAC, DISCH_TAC, or GEN_TAC. If the goal is an implication A==>B, it applies DISCH_TAC — then uses STRIP_ASSUME_TAC to break up the antecedent A, putting the pieces in the assumption list.

Like STRIP_ASSUME_TAC, it is implemented using theorem continuations.

4.3. Functions involving tactics

Proving a goal using a tactic

```
TAC_PROOF: (goal # tactic) -> thm
```

Proving and saving a theorem using a tactic

```
prove_thm: (token # form # tactic) -> thm
```

Testing a validation on dummy theorems

```
chktac: ((goal list) # proof) -> thm
```

The function chktac "proves" the subgoals using mk_fthm, producing theorems that include the additional assumption FALSITY(). It applies the validation to these theorems. If the tactic is valid, chktac returns a theorem stating the goal, but with the additional assumption. This validity test is fairly reliable; most validations do not notice the assumption FALSITY(). It is helpful for debugging new tactics interactively:

```
NEW_TAC test_goal;;
chktac it;;
```

Now compare the resulting theorem with the test_goal.

4.4. Rewriting tactics

```
REWRITE_TAC: (thm list) -> tactic
ASM_REWRITE_TAC: (thm list) -> tactic
```

Most proofs involve the rewriting tactics REWRITE_TAC and ASM_REWRITE_TAC, which are described in detail elsewhere [9]. REWRITE_TAC transforms the goal using the given list of theorems as rewrite rules; ASM_REWRITE_TAC adds the assumptions to the input list of theorems. The inference rule IMP_CANON puts each theorem into the form

$$A_1 \texttt{==>} \ldots \texttt{==>} A_n \texttt{==>} \quad t\texttt{==}u$$
$$A_1 \texttt{==>} \ldots \texttt{==>} A_n \texttt{==>} \quad (B \texttt{<=>} C)$$

A rewriting step replaces either a term or a formula. An instance of the left side (t or B) of a rule is replaced by the corresponding instance of the right side (u or C), if recursive invocation of rewriting succeeds in proving the instances of the antecedents A_i.

Besides explicit formula rewrites such as B<=>C, any theorem asserting a predicate P causes rewriting of P to TRUTH(). Any theorem asserting the negation ~P causes rewriting of P to FALSITY().

Tautologous formulas such as A/\TRUTH() are simplified. Beta-conversion occurs whenever possible. Disjunctions are expanded wherever they appear as conjuncts or antedecents. In particular, (A\/B)==>C becomes (A==>C)/\(B==>C), resulting in a case split. Existential formulas are similarly expanded; thus (?x.A(x))==>C becomes !x'.(A(x')==>C).

The tactics introduce local assumptions. For a conjunction A/\B or an implication A==>B, the formula A is assumed true when rewriting B.

Rewriting continues as long as rules apply. This may result in looping -- for instance if a term t gets rewritten, in one or more steps, to a

term containing t. In particular you must never use commutative rules such as x+y==y+x. An implicative rewrite A ==> t==u will probably loop if t occurs in A. Looping is tricky because it may involve several rewrite rules.

Functions for use with rewriting:

Reversing the orientation of a rule (term or formula rewrite)

```
REV_REWRITE: thm -> thm
```

Given a theorem stating t==u or A<=>B, possibly with quantifiers and antecedents, REV_REWRITE returns a logically equivalent theorem stating u==t or B<=>A.

Printing the rewrites in canonical form (including assumptions)

```
used_rewrites: (thm list) -> (thm list # thm list)
asm_used_rewrites: (thm list) -> goal -> (thm list # thm list)
```

These functions are helpful for predicting the effect of the rewriting tactics on a goal. They process the list of theorems as the rewriting tactics do. They put the theorems into canonical form, and classify them as term rewrites, formula rewrites, or unsuitable for rewriting. The output consists of the list of term rewrites, paired with the list of formula rewrites.

4.5. Resolution tactics

```
IMP_RES_TAC: thm -> tactic
RES_TAC: tactic
```

I am hesitant to discuss the resolution tactics, having experimented with many different kinds without being fully satisifed. However, resolution plays an important role. This section describes the basic resolution tactics, omitting any details that are likely to change.

LCF resolution is trivial compared with full-blown resolution [2]. Suppose we have a list of facts, $[\vdash A_1; \ldots; \vdash A_m]$, and an implication

$$B_1 \Longrightarrow \ldots \Longrightarrow B_n \Longrightarrow C \ .$$

Resolution consists of matching the antecedents against the facts, attempting to prove some instance of C by Modus Ponens. Currently LCF does not provide unification, only matching -- the facts are not instantiated, only the implication is. If some but not all antecedents can be proved, then resolution produces results that are implications.

The tactic IMP_RES_TAC impth resolves the theorem impth, which should be an implication, against the assumption list. It adds the results to the assumption list using STRIP_ASSUME_TAC; in particular, it solves the goal if it finds a contradiction. RES_TAC calls IMP_RES_TAC for all the implications it finds in the assumption list, introducing the results from each invocation.

5. Tacticals

LCF's theorem-proving power derives from passing tactics as arguments and returning tactics as results. A proof composed of many tactic steps can always be expressed as a single compound tactic. Tacticals are operators for combining tactics into larger tactics.

As in Edinburgh LCF, many tacticals have more general types than those given below. They can work with other forms of goal-directed programming, such as the prime number example [4]. The fully polymorphic types are too complex to print here, but can be obtained by invoking LCF.

Most tacticals have simple definitions in ML, which are given here as additional documentation. The ML definitions actually running in LCF may be different.

5.1. Basic tacticals

Cambridge LCF includes the tacticals THEN, ORELSE, and REPEAT, which originated in Edinburgh LCF [4].

Sequencing (infix operator)

```
THEN  : tactic -> tactic -> tactic
```

The tactic (tac_1 THEN tac_2) applies tac_1 to the goal, then applies tac_2 to the resulting subgoals, and returns a flattened list of the subgoals of the subgoals. Its validation composes tac_1's validation with those returned by tac_2 for each of the subgoals. If tac_1 returns an empty subgoal list, then tac_2 is never invoked. The tactic fails if tac_1 or tac_2 does.

Alternation (infix operator)

```
ORELSE : tactic -> tactic -> tactic
```

The tactic (tac_1 ORELSE tac_2) applies tac_1 to the goal, returning the subgoals and validation. If tac_1 fails then the tactic calls tac_2. If tac_2 also fails, then the entire tactic fails.

Identities

```
ALL_TAC: tactic
NO_TAC: tactic
FAIL_TAC: token -> tactic
```

As identities for THEN and ORELSE, LCF provides the tactics ALL_TAC and NO_TAC. ALL_TAC accepts all goals, passing the goal unchanged. NO_TAC accepts no goals: it always fails. FAIL_TAC is like NO_TAC but expects you to supply the failure token.

Repetition

```
REPEAT: tactic -> tactic
```

The tactic (REPEAT tac) applies tac to the goal, and to all resulting subgoals, returning the goals for which tac fails. REPEAT never fails, but may loop.

ML definitions:

```
let (tac1 ORELSE tac2) g =  tac1 g ? tac2 g ;;
let ALL_TAC g = [g],hd;;
let FAIL_TAC tok g = failwith tok;;
let NO_TAC = FAIL_TAC `NO_TAC`;;
letrec REPEAT tac g = ((tac THEN REPEAT tac) ORELSE ALL_TAC) g ;;
```

5.2. List tacticals

In compound tactics, it is often useful operate on the assumption list of the goal. The tactical

```
ASSUM_LIST: (thm list -> tactic) -> tactic
```

maps ASSUME over the assumptions, and supplies them to a tactic function:

```
ASSUM_LIST thltac ([A1; ...; An], B)  --->
        thltac ["A1|-A1";  ...;  "An|-An"]
```

The basic tacticals have been generalized to operate on lists of tactics. It is often useful to map a parametric tactic, such as X_GEN_TAC or EXISTS_TAC, over a list. An important special case is mapping a function of type thm->tactic, such as CONTR_TAC or SUBST1_TAC, over the assumptions of the goal.

Applying every tactic in sequence

```
EVERY: tactic list -> tactic
EVERY [tac1; ...; tacn]   --->   tac1 THEN ... THEN tacn
```

```
MAP_EVERY: (* -> tactic) -> (* list) -> tactic
MAP_EVERY tacf [x1; ...; xn]  --->  EVERY [tacf x1; ...; tacf xn]
```

```
EVERY_ASSUM: (thm -> tactic) -> tactic
EVERY_ASSUM thtac ([A1; ...; An], B)  --->
        EVERY [thtac "A1|-A1";  ...;  thtac "An|-An"]
```

Applying the first successful tactic

FIRST: tactic list -> tactic

FIRST [tac_1; ...; tac_n] ---> tac_1 ORELSE ... ORELSE tac_n

MAP_FIRST: (* -> tactic) -> (* list) -> tactic

MAP_FIRST tacf [x_1; ...; x_n] ---> FIRST [tacf x_1; ...; tacf x_n]

FIRST_ASSUM: (thm -> tactic) -> tactic

FIRST_ASSUM thtac ([A_1; ...; A_n], B) --->
FIRST [thtac "A_1|-A_1"; ...; thtac "A_n|-A_n"]

EVERY and FIRST construct compound tactics using begin/end blocks, in Algol style. MAP_EVERY and MAP_FIRST provide a concise notation for iteration:

MAP_EVERY EXISTS_TAC [t;u;v]

instead of

EXISTS_TAC t THEN EXISTS_TAC u THEN EXISTS_TAC v

EVERY_ASSUM and FIRST_ASSUM produce tactics that search the assumption list, such as those of Cohn [3]. For instance, the tactic

FIRST_ASSUM (\asm. CONTR_TAC asm ORELSE ACCEPT_TAC asm)

searches the assumptions for either a contradiction or the desired conclusion. This style leads to theorem continuations [8].

ML definitions:

```
let ASSUM_LIST aslfun (asl,w) =  aslfun (map ASSUME asl) (asl,w);;

let EVERY tacl = itlist $THEN tacl ALL_TAC;;
let MAP_EVERY tacf lst  =  EVERY (map tacf lst);;
let EVERY_ASSUM = ASSUM_LIST o MAP_EVERY;;

let FIRST tacl = itlist $ORELSE tacl NO_TAC;;
let MAP_FIRST tacf lst =  FIRST (map tacf lst);;
let FIRST_ASSUM = ASSUM_LIST o MAP_FIRST;;
```

5.3. Making a tactic valid

VALID: tactic -> tactic

The tactical VALID constructs (usually) valid tactics. It applies tac to the goal, then tests the resulting validation on dummy theorems. If the resulting theorem differs from the goal, or contains additional assumptions, then VALID fails; otherwise it returns the goal list and validation. VALID uses chktac, documented above, which is imperfect but fairly reliable.

6. The Subgoal Package

When conducting a proof that involves many subgoals and tactics, you must keep track of all the validations and compose them in the correct order. While this is feasible even in large proofs, it is tedious. LCF provides a package for building and traversing the tree of subgoals, stacking the validations and applying them properly.

The package implements a simple framework for interactive proof. You create and traverse the proof tree top-down. Using a tactic, you expand

the current goal into subgoals and validation, which are pushed onto the goal stack. You can consider these subgoals in any order. If the tactic solve the goal (returns an empty subgoal list), then the package proceeds to the next goal in the tree. It saves several preceeding states, to which you can return if you make a mistake in the proof.

Setting the initial goal

set_goal: goal -> void

Expanding the current goal

expand: tactic -> void

Applies the tactic, validated using VALID, to the goal. Prints resulting subgoals. If there are none, applies the validation, and those above it whose subgoals have been proved. Prints the resulting theorems.

Fast expand (does not apply VALID)

expandf: tactic -> void

Printing n levels of the goal stack

print_state: int -> void

Saving the topmost theorem (onto the theory file)

save_top_thm: token -> thm

Rotating the current subgoals (by n steps)

rotate: int -> void

Backing up from last state change

backup: void -> void

Getting the current state (for additional backup)

get_state: void -> goalstack

Restoring a previous state

set_state: goalstack -> void

Getting the top goal on the stack

top_goal: void -> goal

The assignable variable backup_min, initially 12, is the maximum number of proof states saved on the backup list. You may backup repeatedly until the list is exhausted; backing up discards the current state. You may save any desired state using get_state, and restore it later using set_state.

7. Example Proof

To show the tactics, tacticals, and subgoal package in use, I have contrived a proof that involves most of the logical connectives. Let us enter the goal to the subgoal package:

```
#set_goal
#   ([],
#    "!p. ~ p==UU  ==> !x y. (p=>x|y == y  <=>  (p==FF \/ x==y:*))");;

"!p. ~ p == UU  ==>  (!x y. (p => x | y) == y  <=>  p == FF  \/  x == y)"
```

To prove it, first strip off some quantifiers and connectives. Since GEN_TAC removes a universal quantifier, DISCH_TAC removes an implication, and IFF_TAC removes an if-and-only-if, the tactic

```
EVERY [GEN_TAC; DISCH_TAC; GEN_TAC; GEN_TAC; IFF_TAC; DISCH_TAC]
```

can break up the goal considerably. But it is tedious to list each step. Using standard tactics and tacticals we can break the goal completely apart:

```
#expand (REPEAT (STRIP_TAC ORELSE IFF_TAC));;
3 subgoals
"(p => x | y) == y"
    [ "~ p == UU" ]
    [ "x == y" ]

"(p => x | y) == y"
    [ "~ p == UU" ]
    [ "p == FF" ]

"p == FF  \/  x == y"
    [ "~ p == UU" ]
    [ "(p => x | y) == y" ]
```

There are three cases because STRIP_TAC broke the "<==" case into subcases for p==FF and x==y. Actually, it was a mistake to break up the "<=>" at all. The term (p=>x|y)==y requires considering separately the cases where p is UU, TT, or FF. Yet it already appears in an assumption, where it will be difficult to reason with. (Assumptions are printed on separate lines, enclosed in square brackets.) Let us back up and try again without IFF_TAC, then use COND_CASES_TAC to eliminate the conditional.

```
#backup();;
"!p. ~ p == UU  ==>  (!x y. (p => x | y) == y  <=>  p == FF  \/  x == y)"

#expand (REPEAT STRIP_TAC);;
"(p => x | y) == y  <=>  p == FF  \/  x == y"
    [ "~ p == UU" ]

#expand COND_CASES_TAC;;
3 subgoals
"y == y  <=>  FF == FF  \/  x == y"
    [ "~ p == UU" ]
    [ "p == FF" ]

"x == y  <=>  TT == FF  \/  x == y"
    [ "~ p == UU" ]
    [ "p == TT" ]

"UU == y  <=>  UU == FF  \/  x == y"
    [ "~ p == UU" ]
    [ "p == UU" ]
```

These three cases are for the three possible values of p. The goals are hardly in simplest form. It is usually best to follow up a case split by a call to ASM_REWRITE_TAC. Rather than prove each case separately, we can back up and tackle them all at once:

```
#backup();;
"(p => x | y) == y  <=>  p == FF  \/  x == y"
    [ "~ p == UU" ]

#expand (COND_CASES_TAC  THEN  ASM_REWRITE_TAC[]);;
"UU == y  <=>  x == y"
    [ "~ p == UU" ]
    [ "p == UU" ]
```

Only the UU case remains. We can prove it using resolution to detect the contradiction in the assumptions. The subgoal package winds up the proof, printing intermediate results. Then we can save the theorem on the current theory file:

```
#expand RES_TAC;;
goal proved
..|-"UU == y  <=>  x == y"
.|-"(p => x | y) == y  <=>  p == FF  \/  x == y"
|-"!p.
    ~ p == UU  ==>  (!x y. (p => x | y) == y  <=>  p == FF  \/  x == y)"

Previous subproof:  goal proved

#save_top_thm `example`;;
|-"!p.
    ~ p == UU  ==>  (!x y. (p => x | y) == y  <=>  p == FF  \/  x == y)"
: thm
```

The complete proof is

```
EVERY [REPEAT STRIP_TAC; COND_CASES_TAC; ASM_REWRITE_TAC[]; RES_TAC]
```

We can obtain a different proof using MP_TAC and ASM_REWRITE_TAC to perform the case split on p. First back up past the previous case split. Then supply MP_TAC with an instance of the axiom TR_CASES, producing a goal that has a disjunctive antecedent. ASM_REWRITE_TAC solves the UU case, using the assumption ~p==UU, and expands the other cases:

```
#backup();;
"UU == y  <=>  x == y"
    [ "~ p == UU" ]
    [ "p == UU" ]


#backup();;
"(p => x | y) == y  <=>  p == FF  \/  x == y"
    [ "~ p == UU" ]


#expand (MP_TAC (SPEC "p" TR_CASES));;
"p == UU  \/  p == TT  \/  p == FF  ==>
 ((p => x | y) == y  <=>  p == FF  \/  x == y)"
    [ "~ p == UU" ]


#expand (ASM_REWRITE_TAC[]);;
"(p == TT  ==>  ((TT => x | y) == y  <=>  x == y))  /\
 (p == FF  ==>  (FF => x | y) == y)"
    [ "~ p == UU" ]
```

I deliberately omitted the rewrite rule COND_CLAUSES from the call to ASM_REWRITE_TAC in order to show the case split. Supplied with this rule, ASM_REWRITE_TAC solves all three cases at once. The proof is one step shorter than the previous one:

```
EVERY [ REPEAT STRIP_TAC;
        MP_TAC (SPEC "p" TR_CASES);
        ASM_REWRITE_TAC [COND_CLAUSES] ]
```

References

[1] R. Boyer and J. Moore, A Computational Logic (Academic Press, 1979).

[2] C.-L. Chang and R. Lee, Symbolic Logic and Mechanical Theorem Proving (Academic Press, 1973).

[3] A. Cohn, The equivalence of two semantic definitions: a case study in LCF, SIAM Journal of Computing 12 (May 1983) pages 267-285.

[4] M. Gordon, R. Milner, and C. Wadsworth, Edinburgh LCF (Springer-Verlag, 1979).

[5] M. Gordon, Representing a logic in the LCF metalanguage, in: D. Neel, editor, Tools and Notions for Program Construction (Cambridge University Press, 1982) pages 163-185.

[6] Z. Manna, Mathematical Theory of Computation (McGraw-Hill, 1974).

[7] L. Paulson, The revised logic PPLAMBDA: a reference manual, Report No. 36, Computer Laboratory, University of Cambridge (1983).

[8] L. Paulson, Tactics as theorem continuations, Technical Report (in preparation), Computer Laboratory, University of Cambridge (1983).

[9] L. Paulson, A higher-order implementation of rewriting, Science of Computer Programming (to appear).