Interactive Theorem Proving (ITP) Course
Parts VII, VIII

Thomas Tuerk (tuerk@kth.se)

KTH

Academic Year 2016/17, Period 4

version 5056611 of Wed May 3 09:55:18 2017



# Part VII

# Backward Proofs



## Motivation I

- let's prove `!A B. A /\ B <=> B /\ A`

```
(* Show |- A /\ B ==> B /\ A *)
val thm1a = ASSUME ``A /\ B``;
val thm1b = CONJ (CONJUNCT2 thm1a) (CONJUNCT1 thm1a);
val thm1  = DISCH ``A /\ B`` thm1b

(* Show |- B /\ A ==> A /\ B *)
val thm2a = ASSUME ``B /\ A``;
val thm2b = CONJ (CONJUNCT2 thm2a) (CONJUNCT1 thm2a);
val thm2  = DISCH ``B /\ A`` thm2b

(* Combine to get |- A /\ B <=> B /\ A *)
val thm3  = IMP_ANTISYM_RULE thm1 thm2

(* Add quantifiers *)
val thm4  = GENL [``A:bool``, ``B:bool``] thm3
```

- this is how you write down a proof
- for finding a proof it is however often useful to think **backwards**



## Motivation II - thinking backwards

- we want to prove
  - `!A B. A /\ B <=> B /\ A`
- all-quantifiers can easily be added later, so let's get rid of them
  - `A /\ B <=> B /\ A`
- now we have an equivalence, let's show 2 implications
  - `A /\ B ==> B /\ A`
  - `B /\ A ==> A /\ B`
- we have an implication, so we can use the precondition as an assumption
  - using `A /\ B` show `B /\ A`
  - `A /\ B ==> B /\ A`

## Motivation III - thinking backwards

- we have a conjunction as assumption, let's split it
  - using `A` and `B` show `B /\ A`
  - `A /\ B ==> B /\ A`
- we have to show a conjunction, so let's show both parts
  - using `A` and `B` show `B`
  - using `A` and `B` show `A`
  - `A /\ B ==> B /\ A`
- the first two proof obligations are trivial
  - `A /\ B ==> B /\ A`
- ...
- we are done



## Motivation IV

- common practise
  - think backwards to find proof
  - write found proof down in forward style
- often switch between backward and forward style within a proof
  Example: induction proof
  - backward step: induct on ...
  - forward steps: prove base case and induction case
- whether to use forward or backward proofs depend on
  - support by the interactive theorem prover you use
    - HOL 4 and close family: emphasis on backward proof
    - Isabelle/HOL: emphasis on forward proof
    - Coq : emphasis on backward proof
  - your way of thinking
  - the theorem you try to prove



## HOL Implementation of Backward Proofs

- in HOL
  - proof tactics / backward proofs used for most user-level proofs
  - forward proofs used usually for writing automation
- backward proofs are implemented by **tactics** in HOL
  - decomposition into subgoals implemented in SML
  - SML datastructures used to keep track of all open subgoals
  - forward proof used to construct theorems
- to understand backward proofs in HOL we need to look at
  - `goal` — SML datatype for proof obligations
  - `goalStack` — library for keeping track of goals
  - `tactic` — SML type for functions performing backward proofs



## Goals

- goals represent proof obligations, i. e. theorems we need/want to prove
- the SML type `goal` is an abbreviation for `term list * term`
- the goal `([asm_1, ..., asm_n], c)` records that we need/want to prove the theorem `{asm_1, ..., asm_n} |- c`

**Example Goals**

| **Goal** | **Theorem** |
|---|---|
| `([``A``, ``B``], ``A /\ B``)` | `{A, B} |- A /\ B` |
| `([``B``, ``A``], ``A /\ B``)` | `{A, B} |- A /\ B` |
| `([``B /\ A``], ``A /\ B``)` | `{B /\ A} |- A /\ B` |
| `([], ``(B /\ A) ==> (A /\ B)``)` | `|- (B /\ A) ==> (A /\ B)` |

## Tactics

- the SML type `tactic` is an abbreviation for the type `goal -> goal list * validation`
- `validation` is an abbreviation for `thm list -> thm`
- given a goal, a tactic
  - decides into which subgoals to decompose the goal
  - returns this list of subgoals
  - returns a validation that
    - given a list of theorems for the computed subgoals
    - produces a theorem for the original goal
- special case: empty list of subgoals
  - the validation (given `[]`) needs to produce a theorem for the goal
- notice: a tactic might be invalid



## Tactic Example — `CONJ_TAC`

$$\frac{\Gamma \vdash p \qquad \Delta \vdash q}{\Gamma \cup \Delta \vdash p \ \wedge \ q}\,\mathrm{CONJ} \qquad\qquad \frac{\begin{array}{c}\mathtt{t} \equiv \mathtt{conj1\ /\backslash\ conj2}\\ \mathtt{asl} \vdash \mathtt{conj1} \qquad \mathtt{asl} \vdash \mathtt{conj2}\end{array}}{\mathtt{asl} \vdash \mathtt{t}}$$

```
val CONJ_TAC: tactic = fn (asl, t) =>
  let
    val (conj1, conj2) = dest_conj t
  in
    ([(asl, conj1), (asl, conj2)],
     fn [th1, th2] => CONJ th1 th2 | _ => raise Match)
  end
  handle HOL_ERR _ => raise ERR "CONJ_TAC" ""
```



## Tactic Example — `EQ_TAC`

$$\frac{\begin{array}{c}\Gamma \vdash p \Longrightarrow q\\ \Delta \vdash q \Longrightarrow p\end{array}}{\Gamma \cup \Delta \vdash p = q}\,\mathrm{IMP\_ANTISYM\_RULE} \qquad\qquad \frac{\begin{array}{c}\mathtt{t} \equiv \mathtt{lhs\ =\ rhs}\\ \mathtt{asl} \vdash \mathtt{lhs\ ==>\ rhs}\\ \mathtt{asl} \vdash \mathtt{rhs\ ==>\ lhs}\end{array}}{\mathtt{asl} \vdash \mathtt{t}}$$

```
val EQ_TAC: tactic = fn (asl, t) =>
  let
    val (lhs, rhs) = dest_eq t
  in
    ([(asl, mk_imp (lhs, rhs)), (asl, mk_imp (rhs, lhs))],
     fn [th1, th2] => IMP_ANTISYM_RULE th1 th2
      | _          => raise Match)
  end
  handle HOL_ERR _ => raise ERR "EQ_TAC" ""
```



## proofManagerLib / goalStack

- the `proofManagerLib` keeps track of open goals
- it uses `goalStack` internally
- important commands
  - **g** — set up new goal
  - **e** — expand a tactic
  - **p** — print the current status
  - **top_thm** — get the proved thm at the end

## Tactic Proof Example I

**Previous Goalstack**

```
-
```

**User Action**

```
g `!A B. A /\ B <=> B /\ A`;
```

**New Goalstack**

```
Initial goal:

!A B. A /\ B <=> B /\ A

: proof
```



## Tactic Proof Example II

**Previous Goalstack**

```
Initial goal:

!A B. A /\ B <=> B /\ A

: proof
```

**User Action**

```
e GEN_TAC;
e GEN_TAC;
```

**New Goalstack**

```
A /\ B <=> B /\ A

: proof
```



## Tactic Proof Example III

**Previous Goalstack**

```
A /\ B <=> B /\ A

: proof
```

**User Action**

```
e EQ_TAC;
```

**New Goalstack**

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

A /\ B ==> B /\ A

: proof
```



## Tactic Proof Example IV

**Previous Goalstack**

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

A /\ B ==> B /\ A : proof
```

**User Action**

```
e STRIP_TAC;
```

**New Goalstack**

```
B /\ A
------------------------------------
  0.  A
  1.  B
```

## Tactic Proof Example V

### Previous Goalstack

```
B /\ A
------------------------------------
  0.  A
  1.  B
```

### User Action

```
e CONJ_TAC;
```

### New Goalstack

```
A
------------------------------------
  0.  A
  1.  B

B
------------------------------------
  0.  A
  1.  B
```



## Tactic Proof Example VI

### Previous Goalstack

```
A
------------------------------------
  0.  A
  1.  B

B
------------------------------------
  0.  A
  1.  B
```

### User Action

```
e (ACCEPT_TAC (ASSUME ``B:bool``));
e (ACCEPT_TAC (ASSUME ``A:bool``));
```

### New Goalstack

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

: proof
```



## Tactic Proof Example VII

### Previous Goalstack

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

: proof
```

### User Action

```
e STRIP_TAC;
e (ASM_REWRITE_TAC[]);
```

### New Goalstack

```
Initial goal proved.
|- !A B. A /\ B <=> B /\ A:
   proof
```



## Tactic Proof Example VIII

### Previous Goalstack

```
Initial goal proved.
|- !A B. A /\ B <=> B /\ A:
   proof
```

### User Action

```
val thm = top_thm();
```

### Result

```
val thm =
   |- !A B. A /\ B <=> B /\ A:
   thm
```

## Tactic Proof Example IX

### Combined Tactic

```
val thm = prove (``!A B. A /\ B <=> B /\ A``,
  GEN_TAC >> GEN_TAC >>
  EQ_TAC >| [
    STRIP_TAC >>
    STRIP_TAC >| [
      ACCEPT_TAC (ASSUME ``B:bool``),
      ACCEPT_TAC (ASSUME ``A:bool``)
    ],

    STRIP_TAC >>
    ASM_REWRITE_TAC[]
  ]);
```

### Result

```
val thm =
   |- !A B. A /\ B <=> B /\ A:
   thm
```



## Tactic Proof Example X

### Cleaned-up Tactic

```
val thm = prove (``!A B. A /\ B <=> B /\ A``,
  REPEAT GEN_TAC >>
  EQ_TAC >> (
    REPEAT STRIP_TAC >>
    ASM_REWRITE_TAC []
  ));
```

### Result

```
val thm =
   |- !A B. A /\ B <=> B /\ A:
   thm
```



## Summary Backward Proofs

- in HOL most user-level proofs are tactic-based
  - automation often written in forward style
  - low-level, basic proofs written in forward style
  - nearly everything else is written in backward (tactic) style
- there are **many** different tactics
- in the lecture only the most basic ones will be discussed
- **you need to learn about tactics on your own**
  - good starting point: `Quick` manual
  - learning finer points takes a lot of time
  - exercises require you to read up on tactics
- often there are many ways to prove a statement, which tactics to use depends on
  - personal way of thinking
  - personal style and preferences
  - maintainability, clarity, elegance, robustness
  - ...



# Part VIII

# Basic Tactics

## Syntax of Tactics in HOL

- originally tactics were written all in capital letters with underscores
  Example: `ALL_TAC`
- since 2010 more and more tactics have overloaded lower-case syntax
  Example: `all_tac`
- sometimes, the lower-case version is shortened
  Example: `REPEAT`, `rpt`
- sometimes, there is special syntax
  Example: `THEN`, `\\`, `>>`
- which one to use is mostly a matter of personal taste
  - all-capital names are hard to read and type
  - however, not for all tactics there are lower-case versions
  - mixed lower- and upper-case tactics are even harder to read
  - often shortened lower-case name is not *speaking*

**In the lecture we will use mostly the old-style names.**



## Some Basic Tactics

| | |
|---|---|
| `GEN_TAC` | remove outermost all-quantifier |
| `DISCH_TAC` | move antecedent of goal into assumptions |
| `CONJ_TAC` | splits conjunctive goal |
| `STRIP_TAC` | splits on outermost connective (combination of `GEN_TAC`, `CONJ_TAC`, `DISCH_TAC`, ...) |
| `DISJ1_TAC` | selects left disjunct |
| `DISJ2_TAC` | selects right disjunct |
| `EQ_TAC` | reduce Boolean equality to implications |
| `ASSUME_TAC` thm | add theorem to list of assumptions |
| `EXISTS_TAC` term | provide witness for existential goal |



## Tacticals

- tacticals are SML functions that combine tactics to form new tactics
- common workflow
  - develop large tactic interactively
  - using `goalStack` and editor support to execute tactics one by one
  - combine tactics manually with tacticals to create larger tactics
  - finally end up with one large tactic that solves your goal
  - use `prove` or `store_thm` instead of `goalStack`
- make sure to **clearly mark proof structure** by e. g.
  - use indentation
  - use parentheses
  - use appropriate connectives
  - ...
- goalStack commands like e or g should not appear in your final proof



## Some Basic Tacticals

| | | |
|---|---|---|
| tac1 >> tac2 | `THEN`, `\\` | applies tactics in sequence |
| tac >\| tacL | `THENL` | applies list of tactics to subgoals |
| tac1 >- tac2 | `THEN1` | applies tac2 to the first subgoal of tac1 |
| `REPEAT` tac | `rpt` | repeats tac until it fails |
| `NTAC` n tac | | apply tac n times |
| `REVERSE` tac | `reverse` | reverses the order of subgoals |
| tac1 `ORELSE` tac2 | | applies tac1 only if tac2 fails |
| `TRY` tac | | do nothing if tac fails |
| `ALL_TAC` | `all_tac` | do nothing |
| `NO_TAC` | | fail |

## Basic Rewrite Tactics

- (equational) rewriting is at the core of HOL's automation
- we will discuss it in detail later
- details complex, but basic usage is straightforward
  - given a theorem `rewr_thm` of form `|- P x = Q x` and a term `t`
  - rewriting `t` with `rewr_thm` means
  - replacing each occurrence of a term `P c` for some `c` with `Q c` in `t`
- **warning:** rewriting may loop
  Example: rewriting with theorem `|- X <=> (X /\ T)`

| | |
|---|---|
| `REWRITE_TAC` thms | rewrite goal using equations found in given list of theorems |
| `ASM_REWRITE_TAC` thms | in addition use assumptions |
| `ONCE_REWRITE_TAC` thms | rewrite once in goal using equations |
| `ONCE_ASM_REWRITE_TAC` thms | rewrite once using assumptions |



## Case-Split and Induction Tactics

| | |
|---|---|
| `Induct_on` ‘term‘ | induct on `term` |
| `Induct` | induct on all-quantor |
| `Cases_on` ‘term‘ | case-split on `term` |
| `Cases` | case-split on all-quantor |
| `MATCH_MP_TAC` thm | apply rule |
| `IRULE_TAC` thm | generalised apply rule |



## Assumption Tactics

| | |
|---|---|
| `POP_ASSUM` thm-tac | use and remove first assumption<br>common usage `POP_ASSUM MP_TAC` |
| `PAT_ASSUM` term thm-tac<br>also `PAT_X_ASSUM` term thm-tac | use (and remove) first<br>assumption matching pattern |
| `WEAKEN_TAC` term-pred | removes first assumption<br>satisfying predicate |



## Decision Procedure Tactics

- decision procedures try to solve the current goal completely
- they either succeed of fail
- no partial progress
- decision procedures vital for automation

| | |
|---|---|
| `TAUT_TAC` | propositional logic tautology checker |
| `DECIDE_TAC` | linear arithmetic for `num` |
| `METIS_TAC` thms | first order prover |
| `numLib.ARITH_TAC` | Presburger arithmetic |
| `intLib.ARITH_TAC` | uses Omega test |

## Subgoal Tactics

- it is vital to structure your proofs well
  - improved maintainability
  - improved readability
  - improved reusability
  - saves time in medium-run
- therefore, use many small lemmata
- also, use many explicit subgoals

| | |
|---|---|
| ‘term-frag‘ by tac | show term with tac and add it to assumptions |
| ‘term-frag‘ `sufficies_by` tac | show it sufficies to prove term |



## Term Fragments / Term Quotations

- notice that `by` and `sufficies_by` take **term fragments**
- term fragments are also called **term quotations**
- they represent (partially) unparsed terms
- parsing takes time place during execution of tactic in context of goal
- this helps to avoid type annotations
- however, this means syntax errors show late as well
- the library **Q** defines many tactics using term fragments



## Importance of Exercises

- here many tactics are presented in a very short amount of time
- there are many, many more important tactics out there
- few people can learn a programming language just by reading manuals
- similar few people can learn HOL just by reading and listening
- you should write your own proofs and play around with these tactics
- solving the exercises is highly recommended
  (and actually required if you want credits for this course)



## Tactical Proof - Example I - Slide 1

- we want to prove `!l. LENGTH (APPEND l l) = 2 * LENGTH l`
- first step: set up goal on `goalStack`
- at same time start writing proof script

**Proof Script**

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
```

**Actions**

- run `g ``!l. LENGTH (APPEND l l) = 2 * LENGTH l```
- this is done by hol-mode
- move cursor inside term and press `M-h g`
  (menu-entry `HOL - Goalstack - New goal`)

## Tactical Proof - Example I - Slide 2

### Current Goal

```
!l. LENGTH (l ++ l) = 2 * LENGTH l
```

- the outermost connective is an all-quantor
- let's get rid of it via GEN_TAC

### Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (l ++ l) = 2 * LENGTH l``,
GEN_TAC
```

### Actions

- run e GEN_TAC
- this is done by hol-mode
- mark line with GEN_TAC and press M-h e
  (menu-entry HOL - Goalstack - Apply tactic)



## Tactical Proof - Example I - Slide 3

### Current Goal

```
LENGTH (l ++ l) = 2 * LENGTH l
```

- LENGTH of APPEND can be simplified
- let's search an appropriate lemma with DB.match

### Actions

- run DB.print_match [] ``LENGTH (_ ++ _)``
- this is done via hol-mode
- press M-h m and enter term pattern
  (menu-entry HOL - Misc - DB match)
- this finds the theorem listTheory.LENGTH_APPEND
  |- !l1 l2. LENGTH (l1 ++ l2) = LENGTH l1 + LENGTH l2



## Tactical Proof - Example I - Slide 4

### Current Goal

```
LENGTH (l ++ l) = 2 * LENGTH l
```

- let's rewrite with found theorem listTheory.LENGTH_APPEND

### Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
GEN_TAC >>
REWRITE_TAC[listTheory.LENGTH_APPEND]
```

### Actions

- connect the new tactic with tactical >> (THEN)
- use hol-mode to expand the new tactic



## Tactical Proof - Example I - Slide 5

### Current Goal

```
LENGTH l + LENGTH l = 2 * LENGTH l
```

- let's search a theorem for simplifying 2 * LENGTH l
- prepare for extending the previous rewrite tactic

### Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
GEN_TAC >>
REWRITE_TAC[listTheory.LENGTH_APPEND]
```

### Actions

- DB.match finds theorem arithmeticTheory.TIMES2
- press M-h b and undo last tactic expansion
  (menu-entry HOL - Goalstack - Back up)

## Tactical Proof - Example I - Slide 6

### Current Goal

```
LENGTH (l ++ l) = 2 * LENGTH l
```

- extend the previous rewrite tactic
- finish proof

### Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
GEN_TAC >>
REWRITE_TAC[listTheory.LENGTH_APPEND, arithmeticTheory.TIMES2]);
```

### Actions

- add TIMES2 to the list of theorems used by rewrite tactic
- use hol-mode to expand the extended rewrite tactic
- goal is solved, so let's add closing parenthesis and semicolon



## Tactical Proof - Example I - Slide 7

- we have a finished tactic proving our goal
- notice that GEN_TAC is not needed
- let's polish the proof script

### Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
GEN_TAC >>
REWRITE_TAC[listTheory.LENGTH_APPEND, arithmeticTheory.TIMES2]);
```

### Polished Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
REWRITE_TAC[listTheory.LENGTH_APPEND, arithmeticTheory.TIMES2]);
```



## Tactical Proof - Example II - Slide 1

- let's prove something slightly more complicated
- drop old goal by pressing `M-h d`
  (menu-entry `HOL - Goalstack - Drop goal`)
- set up goal on `goalStack` (`M-h g`)
- at same time start writing proof script

### Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (``!x1 x2 x3 l1 l2 l3.
  (MEM x1 l1 /\ MEM x2 l2 /\ MEM x3 l3) /\
  ((x1 <= x2) /\ (x2 <= x3) /\ x3 <= SUC x1) ==>
  ~(ALL_DISTINCT (l1 ++ l2 ++ l3))``,
```



## Tactical Proof - Example II - Slide 2

### Current Goal

```
!x1 x2 x3 l1 l2 l3.
  (MEM x1 l1 /\ MEM x2 l2 /\ MEM x3 l3) /\
  x1 <= x2 /\ x2 <= x3 /\ x3 <= SUC x1 ==>
  ~ALL_DISTINCT (l1 ++ l2 ++ l3)
```

- let's strip the goal

### Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (``!x1 x2 x3 l1 l2 l3.
  (MEM x1 l1 /\ MEM x2 l2 /\ MEM x3 l3) /\
  ((x1 <= x2) /\ (x2 <= x3) /\ x3 <= SUC x1) ==>
  ~(ALL_DISTINCT (l1 ++ l2 ++ l3))``,
REPEAT STRIP_TAC
```

## Tactical Proof - Example II - Slide 2

**Current Goal**

```
!x1 x2 x3 l1 l2 l3.
  (MEM x1 l1 /\ MEM x2 l2 /\ MEM x3 l3) /\
  x1 <= x2 /\ x2 <= x3 /\ x3 <= SUC x1 ==>
  ~ALL_DISTINCT (l1 ++ l2 ++ l3)
```

- let's strip the goal

**Proof Script**

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
REPEAT STRIP_TAC
```

**Actions**

- add `REPEAT STRIP_TAC` to proof script
- expand this tactic using hol-mode



## Tactical Proof - Example II - Slide 3

**Current Goal**

```
F
------------------------------------
  0.  MEM x1 l1      4.  x2 <= x3
  1.  MEM x2 l2      5.  x3 <= SUC x1
  2.  MEM x3 l3      6.  ALL_DISTINCT (l1 ++ l2 ++ l3)
  3.  x1 <= x2
```

- oops, we did too much, we would like to keep `ALL_DISTINCT` in goal

**Proof Script**

```
val NOT_ALL_DISTINCT_LEMMA = prove (``...``,
REPEAT GEN_TAC >> STRIP_TAC
```

**Actions**

- undo `REPEAT STRIP_TAC` (`M-h b`)
- expand more fine-tuned strip tactic



## Tactical Proof - Example II - Slide 4

**Current Goal**

```
~ALL_DISTINCT (l1 ++ l2 ++ l3)
------------------------------------
  0.  MEM x1 l1       3.  x1 <= x2
  1.  MEM x2 l2       4.  x2 <= x3
  2.  MEM x3 l3       5.  x3 <= SUC x1
```

- now let's simplify `ALL_DISTINCT`
- search suitable theorems with `DB.match`
- use them with rewrite tactic

**Proof Script**

```
val NOT_ALL_DISTINCT_LEMMA = prove (``...``,
REPEAT GEN_TAC >> STRIP_TAC >>
REWRITE_TAC[listTheory.ALL_DISTINCT_APPEND, listTheory.MEM_APPEND]
```



## Tactical Proof - Example II - Slide 5

**Current Goal**

```
~((ALL_DISTINCT l1 /\ ALL_DISTINCT l2 /\ !e. MEM e l1 ==> ~MEM e l2) /\
  ALL_DISTINCT l3 /\ !e. MEM e l1 \/ MEM e l2 ==> ~MEM e l3)
------------------------------------
  0.  MEM x1 l1       3.  x1 <= x2
  1.  MEM x2 l2       4.  x2 <= x3
  2.  MEM x3 l3       5.  x3 <= SUC x1
```

- from assumptions 3, 4 and 5 we know `x2 = x1 \/ x2 = x3`
- let's deduce this fact by `DECIDE_TAC`

**Proof Script**

```
val NOT_ALL_DISTINCT_LEMMA = prove (``...``,
REPEAT GEN_TAC >> STRIP_TAC >>
REWRITE_TAC[listTheory.ALL_DISTINCT_APPEND, listTheory.MEM_APPEND] >>
`(x2 = x1) \/ (x2 = x3)` by DECIDE_TAC
```

## Tactical Proof - Example II - Slide 6

### Current Goals — 2 subgoals, one for each disjunct

```
~((ALL_DISTINCT l1 /\ ALL_DISTINCT l2 /\ !e. MEM e l1 ==> ~MEM e l2) /\
  ALL_DISTINCT l3 /\ !e. MEM e l1 \/ MEM e l2 ==> ~MEM e l3)
------------------------------------
  0.  MEM x1 l1        4.  x2 <= x3
  1.  MEM x2 l2        5.  x3 <= SUC x1
  2.  MEM x3 l3        6a. x2 = x1
  3.  x1 <= x2         6b. x2 = x3
```

- both goals are easily solved by first-order reasoning
- let's use `METIS_TAC[]` for both subgoals

### Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (``...``,
REPEAT GEN_TAC >> STRIP_TAC >>
REWRITE_TAC[listTheory.ALL_DISTINCT_APPEND, listTheory.MEM_APPEND] >>
`(x2 = x1) \/ (x2 = x3)` by DECIDE_TAC >> (
  METIS_TAC[]
));
```



## Tactical Proof - Example II - Slide 7

### Finished Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (
``!x1 x2 x3 l1 l2 l3.
  (MEM x1 l1 /\ MEM x2 l2 /\ MEM x3 l3) /\
  ((x1 <= x2) /\ (x2 <= x3) /\ x3 <= SUC x1) ==>
  ~(ALL_DISTINCT (l1 ++ l2 ++ l3))``,
REPEAT GEN_TAC >> STRIP_TAC >>
REWRITE_TAC[listTheory.ALL_DISTINCT_APPEND, listTheory.MEM_APPEND] >>
`(x2 = x1) \/ (x2 = x3)` by DECIDE_TAC >> (
  METIS_TAC[]
));
```

- notice that proof structure is explicit
- parentheses and indentation used to mark new subgoals



# Part IX

# Induction Proofs



## Mathematical Induction

- mathematical (a. k. a. natural) induction principle:
  If a property $P$ holds for 0 and $P(n)$ implies $P(n+1)$ for all n, then $P(n)$ holds for all $n$.
- HOL is expressive enough to encode this principle as a theorem.

  `|- !P. P 0 /\ (!n. P n ==> P (SUC n)) ==> !n. P n`

- Performing mathematical induction in HOL means applying this theorem (e. g. via `HO_MATCH_MP_TAC`)
- there are many similarish induction theorems in HOL
- Example: complete induction principle

  `|- !P. (!n. (!m. m < n ==> P m) ==> P n) ==> !n. P n`

## Structural Induction Theorems

- **structural induction** theorems are an important special form of induction theorems
- they describe performing induction on the structure of a datatype
- Example: `|- !P. P [] /\ (!t. P t ==> !h. P (h::t)) ==> !l. P l`
- structural induction is used very frequently in HOL
- for each algabraic datatype, there is an induction theorem



## Other Induction Theorems

- there are many induction theorems in HOL
  - datatype definitions lead to induction theorems
  - recursive function definitions produce corresponding induction theorems
  - recursive relation definitions give rise to induction theorems
  - many are manually defined
- Examples

```
|- !P. P [] /\ (!l. P l ==> !x. P (SNOC x l)) ==> !l. P l

|- !P. P FEMPTY /\
     (!f. P f ==> !x y. x NOTIN FDOM f ==> P (f |+ (x,y))) ==> !f. P f

|- !P. P {} /\
     (!s. FINITE s /\ P s ==> !e. e NOTIN s ==> P (e INSERT s)) ==>
     !s. FINITE s ==> P s

|- !R P. (!x y. R x y ==> P x y) /\ (!x y z. P x y /\ P y z ==> P x z) ==>
     !u v. R+ u v ==> P u v
```



## Induction (and Case-Split) Tactics

- the tactic `Induct` (or `Induct_on`) usually used to start induction proofs
- it looks at the type of the quantifier (or its argument) and applies the default induction theorem for this type
- this is usually what one needs
- other (non default) induction theorems can be applied via `INDUCT_THEN` or `HO_MATCH_MP_TAC`
- similarish `Cases_on` picks and applies default case-split theorems



## Induction Proof - Example I - Slide 1

- let's prove via induction
  `!l1 l2. REVERSE (l1 ++ l2) = REVERSE l2 ++ REVERSE l1`
- we set up the goal and start and induction proof on `l1`

**Proof Script**

```
val REVERSE_APPEND = prove (
``!l1 l2. REVERSE (l1 ++ l2) = REVERSE l2 ++ REVERSE l1``,
Induct
```

## Induction Proof - Example I - Slide 2

- the induction tactic produced two cases
- base case:
  `!l2. REVERSE ([] ++ l2) = REVERSE l2 ++ REVERSE []`
- induction step:

```
!h l2. REVERSE (h::l1 ++ l2) = REVERSE l2 ++ REVERSE (h::l1)
------------------------------------------------------------
!l2. REVERSE (l1 ++ l2) = REVERSE l2 ++ REVERSE l1
```

- both goals can be easily proved by rewriting

**Proof Script**

```
val REVERSE_APPEND = prove (``
!l1 l2. REVERSE (l1 ++ l2) = REVERSE l2 ++ REVERSE l1``,
Induct >| [
  REWRITE_TAC[REVERSE_DEF, APPEND, APPEND_NIL],
  ASM_REWRITE_TAC[REVERSE_DEF, APPEND, APPEND_ASSOC]
]);
```



## Induction Proof - Example II - Slide 2

- let's prove via induction
  `!l. REVERSE (REVERSE l) = l`
- we set up the goal and start and induction proof on `l`

**Proof Script**

```
val REVERSE_REVERSE = prove (
``!l. REVERSE (REVERSE l) = l``,
Induct
```



## Induction Proof - Example II - Slide 2

- the induction tactic produced two cases
- base case:
  `REVERSE (REVERSE []) = []`
- induction step:

```
!h. REVERSE (REVERSE (h::l1)) = h::l1
---------------------------------------------
REVERSE (REVERSE l) = l
```

- again both goals can be easily proved by rewriting

**Proof Script**

```
val REVERSE_REVERSE = prove (
``!l. REVERSE (REVERSE l) = l``,
Induct >| [
  REWRITE_TAC[REVERSE_DEF],
  ASM_REWRITE_TAC[REVERSE_DEF, REVERSE_APPEND, APPEND]
]);
```