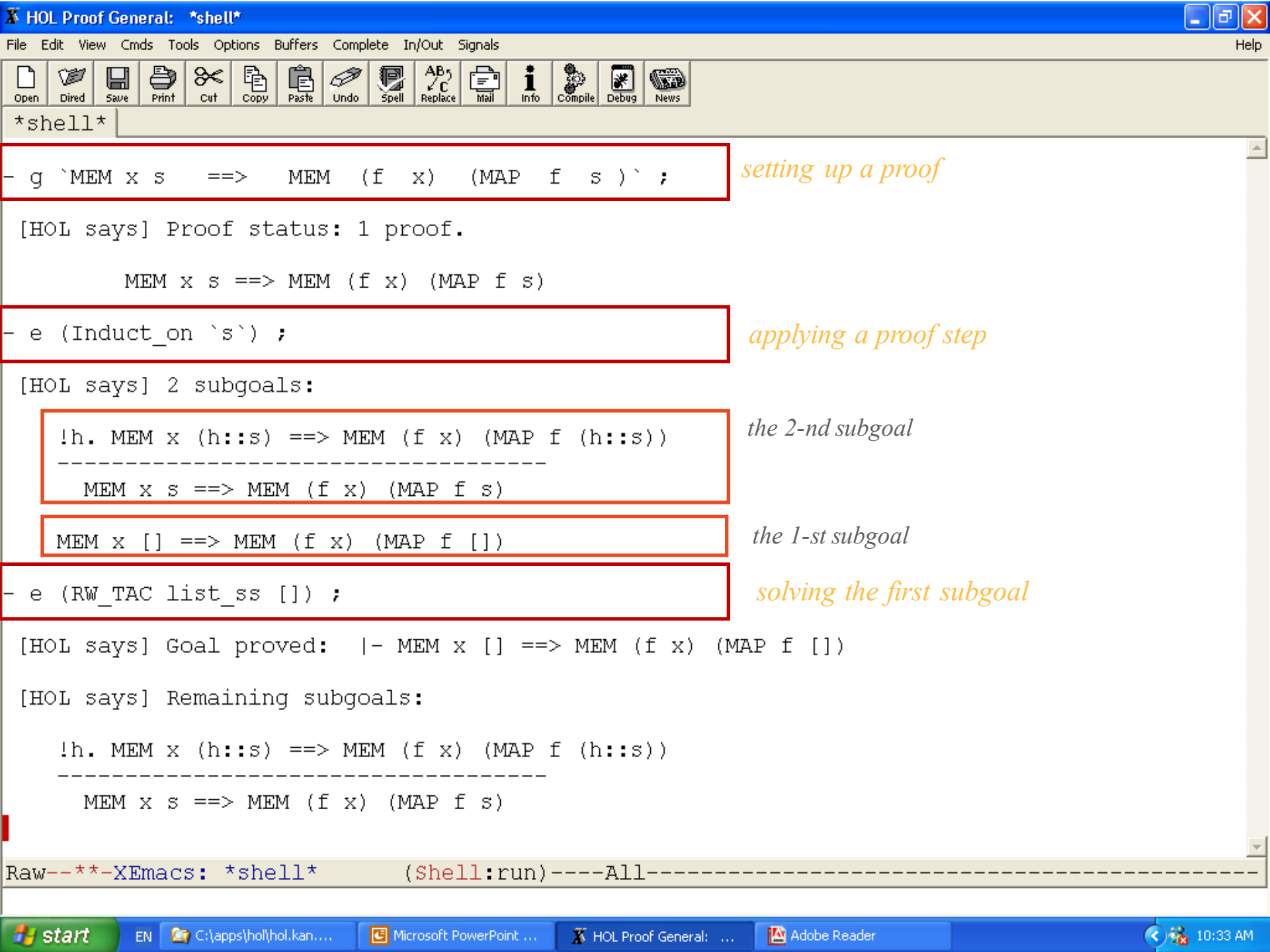
THE THEOREM PROVER HOL
OVERVIEW

Wishnu Prasetya

www.cs.uu.nl/docs/vakken/pv


HOL Proof General: *shell*
File Edit View Cmds Tools Options Buffers Complete In/Out Signals Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
shell
- g `MEM x s ==> MEM (f x) (MAP f s )` ;
setting up a proof
[HOL says] Proof status: 1 proof.
MEM x s ==> MEM (f x) (MAP f s)
- e (Induct_on `s`) ;
applying a proof step
[HOL says] 2 subgoals:
!h. MEM x (h::s) ==> MEM (f x) (MAP f (h::s))
MEM x s ==> MEM (f x) (MAP f s)
the 2-nd subgoal
MEM x [] ==> MEM (f x) (MAP f [])
the 1-st subgoal
- e (RW_TAC list_ss []) ;
solving the first subgoal
[HOL says] Goal proved: |- MEM x [] ==> MEM (f x) (MAP f [])
[HOL says] Remaining subgoals:
!h. MEM x (h::s) ==> MEM (f x) (MAP f (h::s))
MEM x s ==> MEM (f x) (MAP f s)
Raw--**-XEmacs: *shell* (Shell:run)----All
start EN C:\apps\hol\hol.kan.... Microsoft PowerPoint ... HOL Proof General: ... Adobe Reader 10:33 AM

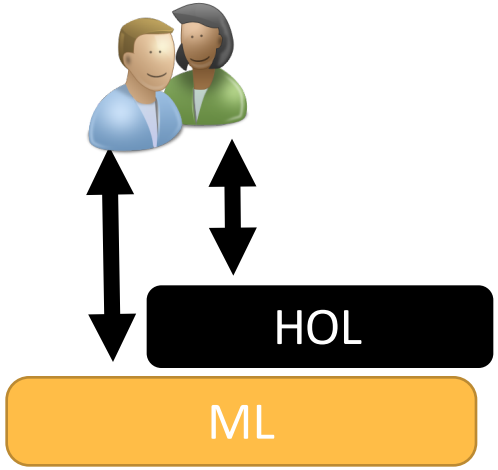
FEATURES

- "*higher order*", as opposed to a first order prover as Z3 → highly expressive! You can model lots of things in HOL.

- A *huge* collection of theories and proof utilities. Well documented.

- *Safe*
  - computer-checked proofs.
  - Simple underlying logic, you can trust it.
  - Unless you hack it, you cannot infer falsity.

HOL IS A DSL EMBEDDED IN ML



- ML is a mature functional programming language.
- You can access both HOL and ML
- It gives you *powerful meta programming* of HOL! E.g. for:
  - scripting your proofs
  - manipulating your terms
  - translating back and forth to other representations
- Interesting options to embed your own DSL/logic

NON-FEATURES

- The name “theorem prover” is a bit misleading. Higher order logic is *undecidable*.
  - Don’t expect HOL to do all the work for you!
- It doesn’t have a good incremental learning material.

  But once you know your way… it’s really a powerful thing.

ML → PROGRAMMING LEVEL

- E.g. these functions in ML:

```
val zero = 0;
fun identity x = x;
```

- These are ordinary programs, you can execute them. E.g. evaluating *identity zero* will produce 0.
- Unfortunately, there is no direct way to prove properties of these ML functions. E.g. that identity . identity = identity

HOL LEVEL

- We can model them in HOL as follows:

```
val zero_def     =  Define   `zero = 0` ;
val identity_def =  Define   `identity x = x` ;
```

- The property we want to prove:

```
--`identity o identity  =  identity`--
```

THE PROOF (SCRIPTED) IN HOL

```
val my_first_theorem = prove (

   --`identity o identity = identity`--,

    CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC [o_DEF]
    THEN BETA_TAC
    THEN REWRITE_TAC [identity_def]

) ;
```

But usually you prefer to prove your formula first interactively, and later on collect your proof steps to make a neat proof script as above.

MODEL AND THE REAL THING

- Keep in mind that *what we have proven is a theorem about the models* !

- E.g. in the real "identity" in ML :

  identity (3/0) = 3/0 → exception!

- We didn't capture this behavior in HOL.
  - HOL will say it is true (aside from the fact that x/0 is undefined in HOL).
  - There is no built-in notion of exception in HOL, though we can model it if we choose so.

CORE SYNTAX OF HOL

- Core syntax is that of simply typed λ-calculus:

$$
\begin{aligned}
term \;::=\; & var \mid const \\
& \mid term \;\; term \\
& \mid \backslash var.\;\; term \\
& \mid term : type
\end{aligned}
$$

- Terms are typed.
- We usually interact on its extended syntax, but all extensions are actually built on this core syntax.

TYPES

- Primitive: bool, num, int, etc.
- Product, e.g.: (1,2) is a term of type **num#num**
- List, e.g.: [1;2;3] is a term of type **num list**
- Function, e.g.: (\x. x+1) is a term of type **num->num**
- Type variables, e.g.:

  (\x. x) is a term of type **'a -> 'a**

- You can define new types.

EXTENDED SYNTAX

- Boolean operators: ~p , p /\ q , p \/ q , p ==> q
- Quantifications: // ! = ∀, ? = ∃

  (**!**x. f x = x) , (**?**x. f x = x)

- Conditional: **if** ... **then** ... **else** ... // alternatively g -> e1 | e2
- Tuples and lists → you have seen.
- Sets, e.g. {1,2,3} → encoded as int->bool.
- You can define your own constants, operators, quantifiers etc.

EXAMPLES OF MODELING IN HOL

- Define `double x = x + x` ;

- Define `(sum [ ] = 0)
  /\
  (sum (x::s) = x + sum s)` ;

- Define `(map f [ ] = [ ])
  /\
  (map f (x::s) = f x :: map f s)` ;

MODELING PROGRAMS

- **Deep embedding**: using data type + denotational semantic.
- **Shallow embedding**: we **represent** statement constructs with function names. We **model what they do** in terms of their denotational semantics. For example:
  - Define `skip state = state` ;
  - Define `SEQ $S_1$ $S_2$ state = $S_2$ ($S_1$ state)` ;
- Notice the use of higher order functions
- So, how do we define GCL's box ?

MODELING PROPERTIES

- Suppose that you want to prove that skip is “reflexive”, and “;” is “transitive”. First you need to define what such concepts mean. For example:
  - Define `isReflexive R = (!x. R x x)` ;
    (so what is the type of isReflexive?)
  - Define `isTransitive R
    =
    (!x y z. R x y /\ R y z ==> R x y)` ;
  - How to define the reflexive and transitive *closure* of R ?

THEOREMS AND PROOFS

THEOREM

- HOL terms: --`0`-- --`x = x`--
- Theorem : a bool-typed HOL term wrapped in a special type called "*thm*", meant to represent a valid fact.

$$\vdash \quad x = x$$

- The type *thm* is a protected data type, in such a way that you can only produce an instance of it via a set of ML functions encoding HOL axioms and primitive inference rules (HOL primitive logic).
  - So, if this primitive logic is sound, there is no way a user can produce an invalid theorem.
  - This primitive logic is very simple; so you can easily convince yourself of its soundness.

THEOREM IN HOL

- More precisely, a theorem is internally a pair (term list * term), which is pretty printed e.g. like:

$$[a_1,\ a_2,\ \ldots]\ \ |-\ \ c$$

Intended to mean that $a_1 /\backslash a_2 /\backslash \ \ldots$ implies c.

- Terminology: *assumptions*, *conclusion*.

- |- c abbreviates [ ] |- c.

INFERENCE RULE

- An (inference) rule is essentially a function that produces a theorem. E.g. this (primitive) inf. rule :

$$\frac{A \vdash t_1 \Rightarrow t_2 \quad , \quad B \vdash t_1}{A \,@\, B \vdash t_2}\ \textit{Modus Ponens}$$

is implemented by a rule called MP : *thm→thm→thm*

- Rules can be composed:

```
fun  myMP  t1 t2  =  GEN_ALL (MP t1 t2)
```

BACKWARD PROVING

- Since a "rule" is a function of type (essentially) *thm→thm,* it implies that to get a theorem you have to "compose" theorems.

  → forward proof; you have to work from axioms

- For human it is usually easier to work a proof backwardly.
- HOL has support for backward proving. Concepts :
  - **Goal** → terms representing what you want to prove
  - **Tactic** → a function that reduce a goal to new goals

GOAL

- type goal = term list * term

  Pretty printed: $[\ a_1, a_2, \ldots\ ]\ \ ?-\ \ h$

  Represent our intention to prove

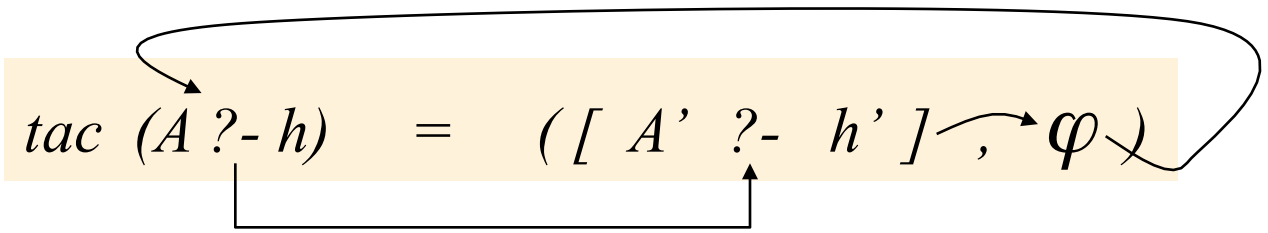
  $[\ a_1, a_2, \ldots\ ]\ \ |-\ h$

- Terminology : assumptions, hypothesis

- type tactic = goal → goal list * proof_func

PROOF FUNCTION

- type tactic = goal → goal list * proof_func
- So, suppose you have this definition of tac :



// so, just 1 new subgoal

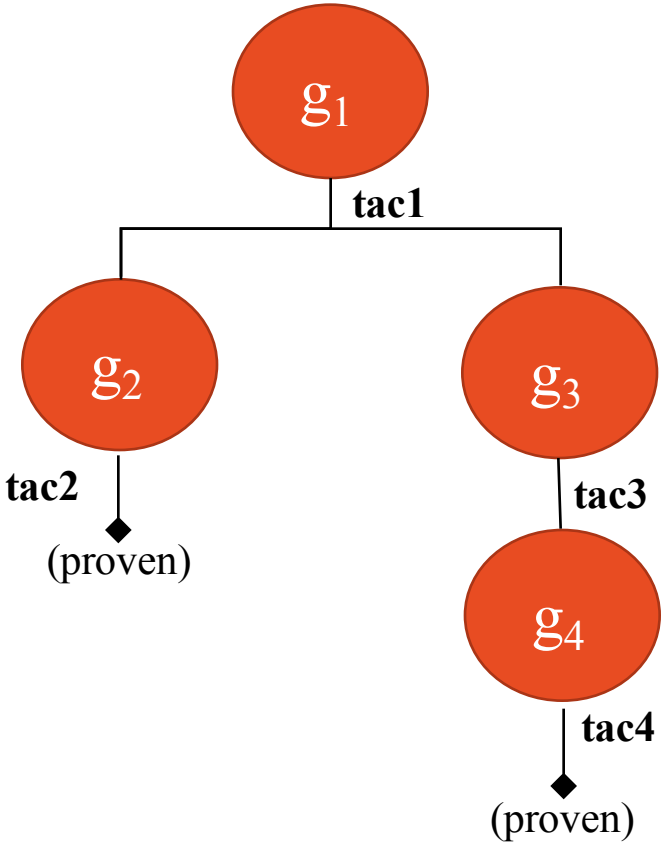
Then the φ has to be such that :

$$\varphi\ [A' \vdash h'] \quad = \quad A \vdash h$$

- So, φ is technically an inference rule, and *tac* is essentially the reverse of this rule.

PROOF TREE

A proof constructed by applying tactics has in principle a tree structure, where at every node we also keep the proof function to 'rebuild' the node from its children.



If all leaves are 'closed' (proven) we build the root-theorem by applying the proof functions in the bottom-up way.

In interactive-proof-mode, such a 'proof tree' is actually implemented as a '*proof stack*' (show example).

INTERACTIVE BACKWARD PROOF
(DESC 5.2)

- HOL maintains a global state variable of type *proofs* :
  - **proofs** : set of active/unfinished goalstacks
  - **goalstack** : implementation of proof tree as a stack

- A set of basic functions to work on these structures.
  - Setting up a new goalstack :

    g : term quotation → proofs

    set_goal : goal → proofs

  - Applying a tactic to the current goal in the current goalstack:

    e (expand) : tactic → goalstack

FOR WORKING ON PROOFS/GOALSTACK…

- Switching focus

  r (rotate) : int → goalstack

- Undo

  b : unit → goalstack
  restart : unit → goalstack
  drop : unit → proofs

SOME BASIC RULES AND TACTICS

SHIFTING FROM/TO ASM… (OLD DESC 10.3)

$$\frac{A \vdash v}{A \setminus \{u\} \vdash u \Longrightarrow v}\ \text{DISCH } u$$

$$\frac{A \;?\!\!-\; u \Longrightarrow v}{A + u \;?\!\!-\; v}\ \text{DISCH\_TAC}$$

$$\frac{A \vdash u \Longrightarrow v}{A + u \vdash v}\ \text{UNDISCH}$$

$$\frac{A \;?\!\!-\; v}{A / \{u\} \;?\!\!-\; u \Longrightarrow v}\ \text{UNDISCH\_TAC } u$$

SOME BASIC RULES AND TACTICS

MODUS PONENS (OLD DESC 10.3)

$$\frac{A_1 \vdash t \Rightarrow u \qquad A_2 \vdash t}{A_1 \cup A_2 \vdash u}\ \text{MP}$$

$$\frac{A \ ?\text{-}\ u \qquad A' \vdash t}{A \ ?\text{-}\ t \Rightarrow u}\ \text{MP\_TAC}$$

A' should be a subset of A

$$\frac{A_1 \vdash !x.\ t_x \Rightarrow u_x \qquad A_2 \vdash t_o}{A_1 \cup A_2 \vdash u_o}\ \text{MATCH\_MP}$$

$$\frac{A \ ?\text{-}\ u_o \qquad A' \vdash !x.\ t_x \Rightarrow u_x}{A \ ?\text{-}\ t_o}\ \text{MATCH\_MP\_TAC}$$

A' should be a subset of A

SOME BASIC RULES AND TACTICS

STRIPPING AND INTRODUCING ∀ (OLD DESC 10.3)

```
 A |-  !x. P
--------------- SPEC u
 A |-  P[u/x]
```

```
A   ?-   P
---------------------   SPEC_TAC(u,x)
 A  ?-   !x. P[x/u]
```

```
 A |-  P
--------------- GEN x
 A |-  !x. P
```

provided x is does not appear as a free variable in A

```
A   ?-   !x. P x
--------------------   GEN_TAC
 A  ?-   P[x'/x]
```

x' is chosen so that it does not appear as a free variable in A

SOME BASIC RULES AND TACTICS
INTRO/STRIPING ∃ (OLD DESC 10.3)

```
  A |- P
---------------------  EXISTS (?x. P[x/u], u)
  A |- ?x. P[x/u]
```

```
  A  ?-  ?x. P
-------------------------  EXISTS_TAC u
  A  ?-   P[u/x]
```

REWRITING (OLD DESC 10.3)

```
 A ?-  t
------------------  SUBST_TAC [A' |- u=v]
 A ?-  t[v/u]
```

- *provides A'⊆ A*
- *you can supply more equalities…*

```
 A ?-  t
------------------  REWRITE_TAC [A' |- u=v]
 A ?-  t[v/u]
```

- *also performs matching e.g. |- f x = x will also match "…f (x+1)"*
- *recursive*
- *may not terminate!*

TACTICS COMBINATORS (TACTICALS)
(OLD DESC 10.4)

- The unit and zero ☺
  - **ALL_TAC** // a 'skip' ☺
  - **NO_TAC** // always fail

- Sequencing :
  - t1 **THEN** t2 → apply t1, then t2 on all subgoals generated by t1
  - t **THENL** [t1,t2,…] → apply t, then t_i on i-th subgoal generated by t
  - **REPEAT** t → repeatedly apply t until it fails (!)

EXAMPLES

- DISCH_TAC **ORELSE** GEN_TAC

- **REPEAT** DISCH_TAC
  THEN EXISTS_TAC “foo”
  THEN ASM_REWRITE_TAC [ ]

- **fun** UD1 (asms,h)
  =
  (**if** null asms **then** NO_TAC
  else UNDISCH_TAC (hd asms)) (asms,h) ;

PRACTICAL THING: QUOTING HOL TERMS
(DESC 5.1.3)

- Remember that HOL is embedded in ML, so you have to quote HOL terms; else ML thinks it is a plain ML expression.
- 'Quotation' in Moscow ML is essentially just a string:

`x y z` → is just "x y z"

Notice the backquotes!

- But it is represented a bit differently to support antiquotation:

```
val  aap = 101

` a b c  ^aap  d e f `

→  [QUOTE "a b c", ANTIQUOTE 101, QUOTE "d e f"] : int frag list
```

QUOTING HOL TERMS

- The ML functions Term and Type parse a quotation to ML "term" and "hol_type"; these are ML datatypes representing HOL term and HOL type.

Term `identity (x:int)` → returns a **term**

Type `:num->num` → returns a **hol_type**

Actually, we often just use this alternate notation, which has the same effect:

-- `identity (x:int)`--

A BIT INCONSISTENT STYLES

- Some functions in HOL expect a term, e.g. :

  prove : term -> tactic -> thm

- And some others expect a frag list / quotation ☹

  g : term frag list -> proofs

  Define : term frag list -> thm

SOME COMMON PROOF TECHNIQUES
(DESC 5.3 – 5)

- Power tactics
- Induction
- Proof by case split
- Proof by contradiction
- In-line lemma

POWER TACTICS: SIMPLIFIER

- Power rewriter, usually to simplify goal :

  SIMP_TAC: simpset→ thm list → tactic

  standard simpsets: std_ss, int_ss, list_ss

- Does not fail. May not terminate.
- Being a complex magic box, it is harder to predict what you get.

EXAMPLES

- Simplify goal with standard simpset:

  SIMP_TAC std_ss [ ]

  (what happens if we use list_ss instead?)

- And if you also want to use some definitions to simplify:

  SIMP_TAC std_ss [foo_def, fi_def , …]

  (what's the type of foo_def ?)

OTHER VARIATIONS OF SIMP_TAC

- ASM_SIMP_TAC
- FULL_SIMP_TAC
- RW_TAC is like SIMP_TAC but does a bit more :
  - case split on any if-then-else in the hypothesis
  - Reducing e.g. “SUC x = SUC y” to “x=y”
  - reduce let-terms in hypothesis

POWER TACTICS: AUTOMATED PROVERS

- 1-st order prover: PROVE_TAC : thm list -> tactic
- Integer arithmetic prover: ARITH_TAC, COOPER_TAC (from intLib)
- Natural numbers arith. prover: ARITH_CONV (from numLib)

- Undecidable.
- They may fail.
- Magic box.

EXAMPLES

- Simplify then use automated prover :

```
RW_TAC  std_ss  [ foo_def ]
THEN  PROVE_TAC  [ ]
```

- In which situations do you want to do these?

```
RW_TAC std_ss [ foo_def ]
THEN  TRY  (PROVE_TAC [ ])

RW_TAC std_ss [ foo_def ]
THEN  ( PROVE_TAC  [ ]  ORELSE  ARITH_TAC )
```

INDUCTION

- Induction over recursive data types: Induct/Induct_on

```
        ?-  ok s
--------------------------------   Induct_on `s`
    (1)    ?-  ok [ ]
    (2)    ok t   ?-  ok (x::t)
```

- Other types of induction:
  - Prove/get the corresponding induction theorem
  - Then apply MP

CASE SPLIT

- Cases_on, examples:

A ?- *u*

---------- Cases_on `p`

(1) *A*[T/*p*] ?- *u*[T/*p*]

(2) *A*[F/*p*] ?- *u*[F/*p*]

assuming p is a bool

?- **ok** *s*

---------- Cases_on `s`

(1) ?- **ok** [ ]

(2) ?- **ok** (*x*::*t*)

assuming s is a list

ADDING "LEMMA"

- **by** : (quotation * tactic) → tactic // infix

$$\frac{A \ ?\text{-} \ t}{lemma + A \ ?\text{-} \ t} \quad lemma \ \textbf{by} \ tac$$

If tac proves the lemma from A

$$\frac{A \ ?\text{-} \ t}{(1)\ lemma + A \ ?\text{-} \ t \qquad (2)\ A \ ?\text{-} \ z} \quad lemma \ \textbf{by} \ tac$$

If tac only reduces lemma to z

ADDING LEMMA

- But when you use it in an interactive proof perhaps you want to use it like this:

  `foo x > 0` **by** ALL_TAC

  What does this do ?

PROOF BY CONTRADICTION

- SPOSE_NOT_THEN : (thm→tactic)→tactic

  SPOSE_NOT_THEN f
  - assumes ¬hyp |- ¬hyp.
  - now you must prove False.
  - f (¬hyp |- ¬hyp) produces a tactic, this is then applied.

- Example:

```
      A  ?-  f x = x
---------------------------   SPOSE_NOT_THEN   ASSUME_TAC
~(f x = x) +  A   ?-  F
```