Theorem Prover HOL, overview

Wishnu Prasetya

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

Assumed background

- Functional programming

- Predicate logic, you know how to read this:

$$(\forall x.\ \text{foo}\ x = x) \Rightarrow (\exists x.\ \text{foo}\ x = x)$$

and know how to prove it.

HOL Proof General: *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)
```

Features

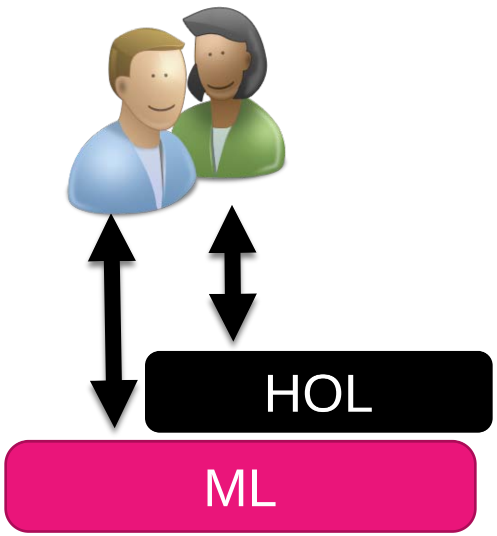
- “*higher order*” → highly expressive! (you can model lots of things in HOL)
- A *huge* collection of theories and proof utilities. Well documented.
- *Safe* !
  - computer-checked proofs.
  - Simple underlying logic, you can trust it.
  - Unless you hack it, you cannot infer falsity.
- Embedded in ML (yay… an EDSL!)

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.

Embedding in ML



- ML is a mature functional programming language.
- You can access ML from HOL.

- It gives you *powerful meta programming* of HOL! E.g. for:
  - scripting your proofs
  - manipulating your terms
  - translating back and forth to other representations

ML → programming level

- E.g. these functions in ML:

```
val  zero  =  0 ;

fun  identity x  =  x ;

fun  after f g   =   (fn x=> f (g x)) ;
```

- These are ordinary programs, you can execute them. E.g. evaluating:

  after identity identity zero

will produce 0.

What if I what to prove properties about my functions?

- For example:

$$\boxed{(\forall x.\ \text{after identity identity}\ \ x\ =\ x)}$$

We can’t prove this in plain ML itself, since it has no built-in theorem proving ability.

(most programming language has no built-in TP)

- Model this in HOL, then verify.

HOL level

- We model them in HOL as follows:

```
val zero_def      =  Define  `zero = 0` ;

val identity_def  =  Define  `identity x = x` ;

val after_def     =  Define  `after f g = (\x. f (g x))` ;
```

- The property we want to prove:

```
--`  !x. after  identity  identity  x   =   x  `--
```

The proof in HOL

```
val my_first_theorem = prove (

    --`!x. after identity identity x = x`--,

    REWRITE_TAC [after_def]
    THEN BETA_TAC
    THEN REWRITE_TAC [identity_def]

) ;
```

But usually you prefer to prove your formula first <u>interactively</u>, 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 “after” and “identity” in ML :

  after identity identity (3/0) = 3/0 $\rightarrow$ 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 simple typed λ-calculus:

$$
\begin{array}{lcl}
\textit{term} & ::= & \textit{var} \mid \textit{const} \\
 & \mid & \textit{term}\ \ \textit{term} \\
 & \mid & \backslash\, \textit{var}.\ \textit{term} \\
 & \mid & \textit{term} : \textit{type}
\end{array}
$$

- 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

(Old Desc ch. 7)

- 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 `skip state = state` ; // higher order at work!

  (so … what is the type of skip?).
- Define `assign x val
  =
  (\state. (\v. <u>if</u> v=x <u>then</u> val <u>else</u> state v))` ;

  (type of assign?)

Modelling list functions

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

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

Modeling properties

- How to express that a relation is reflexive and transitive?

- 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)` ;

- Your turn. Define the reflexive and transitive *closure* of a given R.

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

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.

  |- $x = x$

- 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:

$$[a_1,\ a_2,\ \ldots]\ \ \vdash\ \ c$$

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

- Terminology: *assumptions*, *conclusion*.

- |- c abbreviates [] |- c.

Inference rule

- An (inference) rule is essentially just a function of type:

  thm → *thm*

- E.g. this (primitive) inf. rule :

$$
\frac{A \mid\!- \; t_1 \Rightarrow t_2 \quad , \quad B \mid\!- \; t_1}{A \,@\, B \mid\!- \; t_2} \; \textit{Modus Ponens}
$$

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

- You can compose your own:

fun myMP t_1 t_2 = GEN_ALL (MP t_1 t_2)

Backward proving

- Since a "rule" is a function of type (essentially) $thm \rightarrow thm$,

  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: $[\, a_1, a_2, \ldots \,] \;\; ?\text{-} \;\; h$

  Represent our intention to prove

  $$[\, a_1 , a_2 , \ldots \,] \;\; \vdash \;\; h$$

- Terminology : assumptions, hypothesis

- type tactic = goal $\rightarrow$ goal list * proof_func

Proof Function

- <u>type</u> tactic = goal → goal list * proof_func

- So, suppose you have this definition of tac :

$$tac\ \ (A\ ?\text{-}\ h) \quad = \quad (\ [\ A'\ \ ?\text{-}\ \ h'\ ]\ ,\ \varphi\ )$$

// so, just 1 new subgoal

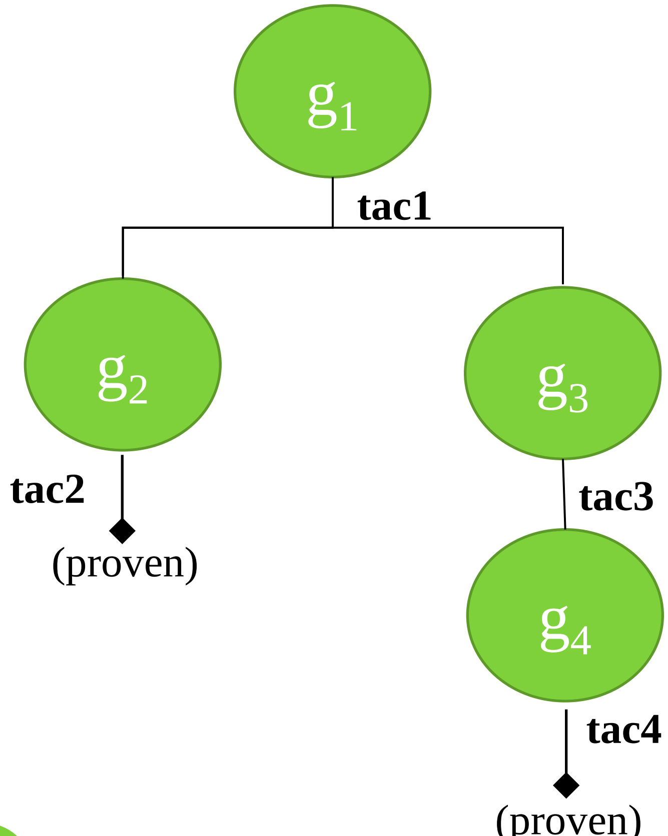
Then the φ has to be such that :

$$\varphi\ [\ A'\ |\text{-}\ \ h'\ ] \quad = \quad A\ |\text{-}\ h$$

- So, a pf is 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)

```
  A |- v
------------------------ DISCH u
 A / {u}  |- u ==> v
```

```
  A ?-   u ==> v
------------------------DISCH_TAC
  A + u ?-    v
```

```
  A |- u ==> v
---------------------- UNDISCH
  A + u |- v
```

```
   A ?- v
------------------------ UNDISCH_TAC u
  A / {u}  ?- u ==> v
```

Some basic rules and tactics

Modus Ponens (Old Desc 10.3)

$$
\begin{array}{ll}
A_1 \ \vdash \ t & \\
A_2 \ \vdash \ t \Rightarrow u & \\
\text{------------------------} & \text{MP} \\
A_1 \cup A_2 \ \vdash \ u &
\end{array}
$$

$$
\begin{array}{ll}
A_1 \ \vdash \ t_o & \\
A_2 \ \vdash \ !x.\ t_x \Rightarrow u_x & \\
\text{----------------------------} & \text{MATCH\_MP} \\
A_1 \cup A_2 \ \vdash \ u_o &
\end{array}
$$

$$
\begin{array}{ll}
A \ \ ?\text{-} \ \ u & \\
A' \ \vdash \ t & \\
\text{-------------------} & \text{MP\_TAC} \\
A \ \ ?\text{-} \ \ t \Rightarrow u &
\end{array}
$$

$$
\begin{array}{ll}
A \ \ ?\text{-} \ \ u_o & \\
A' \ \vdash \ !x.\ t_x \Rightarrow u_x & \\
\text{----------------------------} & \text{MATCH\_MP} \\
A \ \ ?\text{-} \ \ t_o &
\end{array}
$$

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
```

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

provided x is not free in A

x' is chosen so that it is not free 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* $A' \subseteq A$
- *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) ;

Some common proof techniques

(Desc 5.3 – 5)

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

Power Tactics: Simplifier

- Power rewriter, usually to simplify goal :

  SIMP_TAC: simpset$\rightarrow$ thm list $\rightarrow$ 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.
- You hope that its behavior is stable over future versions.

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 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 )
```

Case split

- ASM_CASES_TAC : term → tactic

```
        A  ?-   u
----------------------------  ASM_CASES_TAC   t
    (1)    t + A    ?-  u
    (2)  ~t + A   ?-  u
```

- Split on data constructors, Cases / Cases_on

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

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

Adding “lemma”

- by : (quotation * tactic) $\rightarrow$ tactic   // infix

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

If tac proves the lemma

$$\frac{A \ \ ?\text{-} \ \ t}{\begin{array}{l}(1)\ \textit{lemma} + A \ \ ?\text{-} \ \ t \\ (2)\ A \ \ ?\text{-} \ \ z\end{array}} \quad \textit{lemma} \ \textbf{by} \ \text{tac}$$

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
```