Interactive Theorem Proving (ITP) Course
Parts V, VI

Thomas Tuerk (tuerk@kth.se)

KTH

Academic Year 2016/17, Period 4

version 1b43e90 of Sun Apr 23 20:08:16 2017

Part V

Basic HOL Usage

HOL Technical Usage Issues

- practical issues are discussed in practical sessions
  - how to install HOL
  - which key-combinations to use in emacs-mode
  - detailed signature of libraries and theories
  - all parameters and options of certain tools
  - ...
- exercise sheets sometimes
  - ask to read some documentation
  - provide examples
  - list references where to get additional information
- if you have problems, ask me outside lecture (tuerk@kth.se)
- covered only very briefly in lectures

Installing HOL

- webpage: `https://hol-theorem-prover.org`
- HOL supports two SML implementations
  - Moscow ML (`http://mosml.org`)
  - PolyML (`http://www.polyml.org`)
- I recommend using PolyML
- please use emacs with
  - hol-mode
  - sml-mode
  - hol-unicode, if you want to type Unicode
- please install recent revision from git repo or Kananaskis 11 release
- documentation found on HOL webpage and with sources

General Architecture

- HOL is a collection of SML modules
- starting HOL starts a SML Read-Eval-Print-Loop (REPL) with
  - some HOL modules loaded
  - some default modules opened
  - an input wrapper to help parsing terms called `unquote`
- `unquote` provides special quotes for terms and types
  - implemented as input filter
  - ``` ``my-term`` ``` becomes `Parse.Term [QUOTE "my-term"]`
  - ``` ``:my-type`` ``` becomes `Parse.Type [QUOTE ":my-type"]`
- main interfaces
  - **emacs** (used in the course)
  - vim
  - bare shell

Filenames

- `*Script.sml` — HOL proof script file
  - script files contain definitions and proof scripts
  - executing them results in HOL searching and checking proofs
  - this might take very long
  - resulting theorems are stored in `*Theory.{sml|sig}` files
  - `*Theory.sml` files load quickly, because they don't search/check proofs
- `*Theory.{sml|sig}` — HOL theory
  - auto-generated by corresponding script file
  - do not edit
- `*Syntax.{sml|sig}` — syntax libraries
  - contain syntax related functions
  - i. e. functions to construct and destruct terms and types
- `*Lib.{sml|sig}` — general libraries
- `*Simps.{sml|sig}` — simplifications
- `selftest.sml` — selftest for current directory

Directory Structure

- `bin` — HOL binaries
- `src` — HOL sources
- `examples` — HOL examples
  - interesting projects by various people
  - examples owned by their developer
  - coding style and level of maintenance differ a lot
- `help` — sources for reference manual
  - after compilation home of reference HTML page
- `Manual` — HOL manuals
  - Tutorial
  - Description
  - Reference (PDF version)
  - Interaction
  - Quick (sheet pages)
  - Style-guide
  - ...

Unicode

- HOL supports both Unicode and pure ASCII input and output
- advantages of Unicode compared to ASCII
  - easier to read (good fonts provided)
  - no need to learn special ASCII syntax
- disadvanges of Unicode compared to ASCII
  - harder to type (even with `hol-unicode.el`)
  - less portable between systems
- whether you like Unicode is highly a matter of personal taste
- HOL's policy
  - no Unicode in HOL's source directory `src`
  - Unicode in examples directory `examples` is fine
- I recommend turning Unicode output off initially
  - this simplifies learning the ASCII syntax
  - no need for special fonts
  - it is easier to copy and paste terms from HOL's output

Where to find help?

- reference manual
  - available as HTML pages, single PDF file and in-system help
- description manual
- Style-guide (still under development)
- HOL webpage (`https://hol-theorem-prover.org`)
- mailing-list `hol-info`
- `DB.match` and `DB.find`
- `*Theory.sig` and `selftest.sml` files
- ask someone, e. g. me :-) (tuerk@kth.se)

Part VI

Forward Proofs

Kernel too detailed

- we already discussed the HOL Logic
- the kernel itself does not even contain basic logic operators
- usually one uses a much higher level of abstraction
  - many operations and datatypes are defined
  - high-level derived inference rules are used
- let's now look at this more common abstraction level

Common Terms and Types

	Unicode	**ASCII**
type vars	α, β, ...	'a, 'b, ...
type annotated term	`term:type`	`term:type`
true	`T`	`T`
false	`F`	`F`
negation	`¬b`	`~b`
conjunction	`b1 ∧ b2`	`b1 /\ b2`
disjunction	`b1 ∨ b2`	`b1 \/ b2`
implication	`b1 ⟹ b2`	`b1 ==> b2`
equivalence	`b1 ⟺ b2`	`b1 <=> b2`
disequation	`v1 ≠ v2`	`v1 <> v2`
all-quantification	`∀x. P x`	`!x. P x`
existential quantification	`∃x. P x`	`?x. P x`
Hilbert's choice operator	`@x. P x`	`@x. P x`

There are similar restrictions to constant and variable names as in SML.
HOL specific: don't start variable names with an underscore

Syntax conventions

- common function syntax
  - prefix notation, e. g. `SUC x`
  - infix notation, e. g. `x + y`
  - quantifier notation, e. g. `∀x. P x` means `(∀) (λx. P x)`
- infix and quantifier notation functions can turned into prefix notation
  Example: `(+) x y` and `$+ x y` are the same as `x + y`
- quantifiers of the same type don't need to be repeated
  Example: `∀x y. P x y` is short for `∀x. ∀y. P x y`
- there is special syntax for some functions
  Example: `if c then v1 else v2` is nice syntax for `COND c v1 v2`
- associative infix operators are usually right-associative
  Example: `b1 /\ b2 /\ b3` is parsed as `b1 /\ (b2 /\ b3)`

Operator Precedence

It is easy to misjudge the binding strength of certain operators. Therefore use plenty of parenthesis.

Creating Terms

Term Parser

Use special quotation provided by `unwind`.

Use Syntax Functions

Terms are just SML value of type `term`. You can use syntax functions (usually defined in `*Syntax.sml` files) to create them.

Creating Terms II

Parser	Syntax Funs	
‘‘:bool‘‘	mk_type ("bool", []) or bool	type of Booleans
‘‘T‘‘	mk_const ("T", bool) or T	term true
‘‘ b‘‘	mk_neg (mk_var ("b", bool))	negation of Boolean var b
‘‘... /\ ...‘‘	mk_conj (..., ...)	conjunction
‘‘... \/ ...‘‘	mk_disj (..., ...)	disjunction
‘‘... ==> ...‘‘	mk_imp (..., ...)	implication
‘‘... = ...‘‘	mk_eq (..., ...)	equation
‘‘... <=> ...‘‘	mk_eq (..., ...)	equivalence
‘‘... <> ...‘‘	mk_neg (mk_eq (..., ...))	negated equation

Inference Rules for Equality

$$\frac{}{\vdash t = t}\ \text{REFL}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash t = s}\ \text{GSYM}$$

$$\frac{\Gamma \vdash s = t \qquad x \textit{ not free in } \Gamma}{\Gamma \vdash \lambda x.\ s = \lambda x.t}\ \text{ABS}$$

$$\frac{\Gamma \vdash s = t \qquad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \text{TRANS}$$

$$\frac{\Gamma \vdash p \Leftrightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \text{EQ\_MP}$$

$$\frac{\Gamma \vdash s = t \qquad \Delta \vdash u = v \qquad \textit{types fit}}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \text{MK\_COMB}$$

$$\frac{}{\vdash (\lambda x.\ t)x = t}\ \text{BETA}$$

Inference Rules for free Variables

$$\frac{\Gamma[x_1, \ldots, x_n] \vdash p[x_1, \ldots, x_n]}{\Gamma[t_1, \ldots, t_n] \vdash p[t_1, \ldots, t_n]} \text{ INST}$$

$$\frac{\Gamma[\alpha_1, \ldots, \alpha_n] \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma[\gamma_1, \ldots, \gamma_n] \vdash p[\gamma_1, \ldots, \gamma_n]} \text{ INST\_TYPE}$$

Inference Rules for Implication

$$\frac{\Gamma \vdash p \Longrightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q} \text{ MP, MATCH\_MP}$$

$$\frac{\Gamma \vdash p = q}{\Gamma \vdash p \Longrightarrow q \qquad \Gamma \vdash q \Longrightarrow p} \text{ EQ\_IMP\_RULE}$$

$$\frac{\Gamma \vdash p \Longrightarrow q \qquad \Delta \vdash q \Longrightarrow p}{\Gamma \cup \Delta \vdash p = q} \text{ IMP\_ANTISYM\_RULE}$$

$$\frac{\Gamma \vdash p \Longrightarrow q \qquad \Delta \vdash q \Longrightarrow r}{\Gamma \cup \Delta \vdash p \Longrightarrow r} \text{ IMP\_TRANS}$$

$$\frac{\Gamma \vdash p}{\Gamma - \{q\} \vdash q \Longrightarrow p} \text{ DISCH}$$

$$\frac{\Gamma \vdash q \Longrightarrow p}{\Gamma \cup \{q\} \vdash p} \text{ UNDISCH}$$

$$\frac{\Gamma \vdash p \Longrightarrow \mathsf{F}}{\Gamma \vdash {\sim} p} \text{ NOT\_INTRO}$$

$$\frac{\Gamma \vdash {\sim} p}{\Gamma \vdash p \Longrightarrow \mathsf{F}} \text{ NOT\_ELIM}$$

Inference Rules for Conjunction / Disjunction

$$\frac{\Gamma \vdash p \qquad \Delta \vdash q}{\Gamma \cup \Delta \vdash p \ \wedge \ q}\ \text{CONJ}$$

$$\frac{\Gamma \vdash p \ \wedge \ q}{\Delta \vdash p}\ \text{CONJUNCT1}$$

$$\frac{\Gamma \vdash p \ \wedge \ q}{\Delta \vdash q}\ \text{CONJUNCT2}$$

$$\frac{\Gamma \vdash p}{\Delta \vdash p \ \vee \ q}\ \text{DISJ1}$$

$$\frac{\Gamma \vdash q}{\Delta \vdash p \ \vee \ q}\ \text{DISJ2}$$

$$\frac{\begin{array}{c}\Gamma \vdash p \vee q \\ \Delta_1 \cup \{p\} \vdash r \\ \Delta_2 \cup \{q\} \vdash r\end{array}}{\Gamma \cup \Delta_1 \cup \Delta_2 \vdash r}\ \text{DISJ\_CASES}$$

Inference Rules for Quantifiers

$$\frac{\Gamma \vdash p \qquad x \text{ not free in } \Gamma}{\Gamma \vdash \forall x.\ p}\ \text{GEN}$$

$$\frac{\Gamma \vdash p[u/x]}{\Gamma \vdash \exists x.\ p}\ \text{EXISTS}$$

$$\frac{\Gamma \vdash \forall x.\ p}{\Gamma \vdash p[u/x]}\ \text{SPEC}$$

$$\frac{\begin{array}{c}\Gamma \vdash \exists x.\ p \\ \Delta \cup \{p[v/x]\} \vdash r \\ v \text{ not free in } \Gamma, \Delta, p \text{ and } r\end{array}}{\Gamma \cup \Delta \vdash r}\ \text{CHOOSE}$$

Forward Proofs

- axioms and inference rules are used to derive theorems
- this method is called **forward proof**
  - one starts with basic building blocks
  - one moves step by step forwards
  - finally the theorem one is interested in is derived
- one can also implement own proof tools

Forward Proofs — Example I

Let's prove $\forall p.\ p \Longrightarrow p$.

```
val IMP_REFL_THM = let
  val tm1 = ``p:bool``;
  val thm1 = ASSUME tm1;
  val thm2 = DISCH tm1 thm1;
in
  GEN tm1 thm2

end

fun IMP_REFL t =
  SPEC t IMP_REFL_THM;
```

```
> val tm1 = ``p``: term
> val thm1 = [p] |- p: thm
> val thm2 = |- p ==> p: thm

> val IMP_REFL_THM =
    |- !p. p ==> p: thm

> val IMP_REFL =
    fn: term -> thm
```

Forward Proofs — Example II

Let's prove $\forall P\, v.\ (\exists x.\ (x = v) \wedge P\ x) \iff P\ v.$

```
val tm_v = ``v:'a``;
val tm_P = ``P:'a -> bool``;
val tm_lhs = ``?x. (x = v) / P x``
val tm_rhs = mk_comb (t_P, t_v);

val thm1 = let
  val thm1a = ASSUME tm_rhs;
  val thm1b =
    CONJ (REFL tm_v) thm1a;
  val thm1c =
    EXISTS (tm_lhs, tm_v) thm1b
in
  DISCH tm_rhs thm1c
end
```

```
> val thm1a = [P v] |- P v: thm
> val thm1b =
    [P v] |- (v = v) /\ P v: thm
> val thm1c =
    [P v] |- ?x. (x = v) /\ P x

> val thm1 = [] |-
    P v ==> ?x. (x = v) /\ P x: thm
```

Forward Proofs — Example II cont.

```
val thm2 = let
  val thm2a =
    ASSUME ``(u:'a = v) /\ P u``
  val thm2b = AP_TERM t_P
    (CONJUNCT1 thm2a);
  val thm2c = EQ_MP thm2b
    (CONJUNCT2 thm2a);
  val thm2d =
    CHOOSE (``u:'a``,
      ASSUME tm_lhs) thm2c
in
  DISCH tm_lhs thm2d
end
```

```
1
> val thm2a = [(u = v) /\ P u] |-
    (u = v) /\ P u: thm
> val thm2b = [(u = v) /\ P u] |-
    P u <=> P v
> val thm2c = [(u = v) /\ P u] |-
    P v
> val thm2d = [?x. (x = v) /\ P x] |-
    P v

> val thm2 = [] |-
    ?x. (x = v) /\ P x ==> P v
```

```
val thm3 = IMP_ANTISYM_RULE thm2 thm1
```

```
> val thm3 = [] |-
    ?x. (x = v) /\ P x <=> P v
```

```
val thm4 = GENL [t_P, t_v] thm3
```

```
> val thm4 = [] |- !P v.
    ?x. (x = v) /\ P x <=> P v
```

Derived Tools

- HOL lives from implementing reasoning tools in SML
- **rules** — use theorems to produce new theorems
  - SML-type `thm -> thm`
  - functions with similar type often called rule as well
- **conversions** — convert a term into an equal one
  - SML-type `term -> thm`
  - given term `t` produces theorem of form `[] |- t = t'`
  - may raise exceptions `HOL_ERR` or `UNCHANGED`
- ...

Conversions

- HOL has very good tool support for equality reasoning
- therefore **conversions** are important
- there is a lot of infrastructure for conversions
  - RAND_CONV, RATOR_CONV, ABS_CONV
  - DEPTH_CONV
  - THENC, TRY_CONV, FIRST_CONV
  - REPEAT_CONV
  - CHANGED_CONV, QCHANGED_CONV
  - NO_CONV, ALL_CONV
  - ...
- important conversions
  - REWR_CONV
  - REWRITE_CONV
  - ...