Guide to HOL4 interaction and basic proofs (Using Emacs)

February 28, 2025

1 Introduction

This document gives readers, with no experience in using HOL4, the most minimum knowledge needed to start using HOL4. The aim is to give a concise description of the basics in a format usable as a beginners' reference manual.

Section 2:	Interaction with HOL4 (via emacs)
Section 3:	Searching for theorems and theories
Section 4:	Common proof tactics
Section 5:	Further reading and general advice

The text assumes that the reader has HOL4 installed. You can download and install HOL4 following the instructions on `https://hol-theorem-prover.org`.

2 Interaction with HOL4 (via emacs)

HOL comes with emacs modes that make script files look prettier, and help when interacting with HOL sessions. To install the scripts, add the following lines to your emacs initialisation file (`.emacs` or `.emacs.d/init.el`) with `<path>` replaced with the full path to your HOL4 installation:

```
(load "<path>/HOL/tools/editor-modes/emacs/hol-mode")
(load "<path>/HOL/tools/editor-modes/emacs/hol-unicode")
```

If your version of emacs does not highlight the active region by default, also add the following line to your initialisation file:

```
(transient-mark-mode 1)
```

Restart emacs to make these changes take effect.

2.1 Starting a HOL4 session

1. Start emacs.
2. Press `C-x C-f` to open a file (its name should have suffix `Script.sml`)
3. Press `M-h H`, then press `RET` or down arrow and then `RET`.

The HOL window should look something like this:

```
---------------------------------------------------------------------
       HOL-4 [Kananaskis 13 (stdknl, built Tue Feb 18 15:39:00 2020)]

       For introductory HOL help, type: help "hol";
       To exit type <Control>-D
---------------------------------------------------------------------
> > > > >
```

2.2 Copying input into HOL4 (Opening a theory)

First, make sure you know how to select text in emacs. Either:

- Move the cursor while holding the shift key; or
- Hit `C-space`, and then move the cursor normally; or
- Use the mouse (hold the primary button and drag)

To copy and paste the selected region into the HOL session press `M-h M-r`. For example, selecting the following line, and then pressing `M-h M-r`

```
open arithmeticTheory listTheory;
```

makes HOL4 open the library theories for arithmetic (over natural numbers) and lists. This should not produce any significant output.

2.3 Starting a goal-oriented proof

Most HOL4 proofs are constructed using an interactive *goal stack* and then put together using tactic combinators (Section 2.6, 2.7). To start the goal stack:

1. Write the outline of a theorem, in this case called `less_add_1`. One can type ∀ by pressing the `!` key. More key shortcuts will be explained below.

    ```
    Theorem less_add_1:
      ∀n. n < n + 1
    Proof

    QED
    ```

2. Move the cursor between the `Theorem`-line and `Proof`-line.
3. Press `M-h g` to push the goal onto the goal stack.

`M-h H`	–	start HOL	`M-h g`	–	push goal onto goal stack
`M-h M-r`	–	copy region into HOL	`M-h e`	–	apply tactic to goal
`M-h C-t`	–	display types on/off	`M-h b`	–	move back in proof
`M-h C-c`	–	interrupt HOL	`M-h p`	–	print current goal
‘	–	writes ‘’	`M-h d`	–	drop current goal
‘‘	–	writes “”	`M-h M-s`	–	start subgoal proof for `by`

Table 1: Most important key bindings in the emacs HOL4 mode. Note that all of these actions are also available in the HOL menu within Emacs.

The HOL4 window should look something like this:

```
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        ∀n. n < n + 1
```

Typing Special Symbols in Emacs As above, we can write ∀ as `!` in HOL4. Indeed, with the special HOL-input map turned on in Emacs (which happens by default, and is indicated by the presence of a Π on the left of the modeline), typing `!` will produce a ∀ character automatically. A number of other connectives are handled in similar ways. There are also many abbreviations for even more common Unicode characters on `Control-Shift` modifiers (i.e., keys that are pressed at the same time as the Control and Shift modifier keys are held down). For example, `C-S-l` gives a λ character. Finally, there are yet more options when a leading backslash is typed. For example `\a` will generate an α character (as will typing out `\alpha` in full). In some situations, the machinery is even more sophisticated. If one types `\l`, this will generate a listing of many different left arrow options in the mini-buffer. If a space character is entered, the currently highlighted option will be selected. Alternatively, pick a number or move the cursor with left and right arrow keys to choose that option (which will become the default option for next time). See Table 2 for more information.

2.4 Applying a tactic

Make progress in a proof using *proof tactics.*

1. Write the name of a tactic, *e.g.* `decide_tac`, see Section 4 for more tactics
2. Select the text of the tactic
3. Press `M-h e` to apply the tactic.

A tactic makes HOL4 update the current goal. The HOL4 window will either display the new goal(s) or print:

Symbol	Control-Shift keybinding	“Quail” binding(s)
$\forall$	`C-S-!`	`!` `\all`
$\exists$	`C-S-?`	`?` `\exists`
$\wedge$	`C-S-&`	`/\` `\and`
$\vee$	`C-S-|`	`\/` `\or`
$\neg$	`C-S-~`	`\neg`
$\iff$	`C-S-=`	`\lr=` `\iff`†
$\neq$	`C-M-S-\ =`	`<>` `\neq`
$\Rightarrow$	`C-M-S->`	`==>` `\r=`†
α	`C-S-a`	`\a` `\alpha`
β	`C-S-b`	`\b` `\beta`
γ	`C-S-g`	`\g` `\gamma`
Γ	`C-M-S-g`	`\GG` `\Gamma`
δ	`C-S-d`	`\d` `\delta`
Δ	`C-M-S-d`	`\GD` `\Delta`
λ	`C-S-l`	`\lambda`
Λ	`C-M-S-l`	`\GL` `\Lambda`
	...	
$\cup$	`C-S-u`	`\cup` `\union`†
$\cap$	`C-S-i`	`\cap` `\i`†
$\in$	`C-S-:`	`\in`
$\notin$	`C-M-S-\ :`	`\notin`
$\emptyset$	`C-M-S-\ 0`	`\empty`
$\subseteq$	`C-S-c`	`\subseteq`
⦇	`C-M-S-( C-M-S-|`	
⦈	`C-M-S-) C-M-S-|`	
$\llbracket$	`C-S-[`	`\(`†
$\rrbracket$	`C-S-]`	`\)`†
$\langle$	`C-M-S-( C-S-<`	`\langle` `\(`†
$\rangle$	`C-M-S-) C-S->`	`\rangle` `\)`†

Table 2: Illustrative Input methods for various Unicode symbols. Multiple keys (separated by spaces) in the Control-Shift column indicate that a sequence of keys must be struck to achieve the desired character. (For example, the `C-M-S-\` prefix is used to open up a set of bindings for characters with various forms of slash through them; following that with a colon puts a slash through the $\in$ symbol.) Quail bindings are enabled if there is a Π character visible on the left of the current buffer’s mode-line. Quail can be toggled on and off with the `C-\` keybinding. Full details of the “quail” input map can be seen interactively by typing the command `M-x hol-input-show-translations`. Quail bindings marked with daggers (†) are those that lead to multi-way selections that need to be made with numbers or left-right arrow keys. Finally note that it is always possible to enter characters without special processing by first typing a `C-q`.

```
    Initial goal proved.
    |- ∀n. n < n + 1 : goalstack
```

You can undo the effect of the applied tactic by pressing `M-h b`. Press `M-h p` to view the current goal. To go all the way back to the start of the proof (to restart), press `M-h R`.

2.5 Ending a goal-oriented proof

One can pop goals off the goal stack by pressing `M-h d`, which gives:

```
OK..
val it = There are currently no proofs.: proofs
```

2.6 Saving the resulting theorem

The tactic should be written between the `Proof`-line and the `QED`-line.

```
Theorem less_add_1:
  ∀n. n < n + 1
Proof
  decide_tac
QED
```

When the above lines are copied into HOL4 (using text-selection then `M-h M-r`, as described in Section 2.2), HOL4 responds with:

```
val less_add_1 = ⊢ ∀n. n < n + 1: thm
```

2.7 Saving proofs based on multiple tactics

Suppose we have proved the goal `∀n. n <= n * n` with the following tactics. Note that `‘n’` (with the "pretty" single quotation marks) can be produced by typing `` `n` ``.

```
Induct_on ‘n’                   (* comment: induction on n  *)

  decide_tac                    (* comment: solve base case *)

  asm_simp_tac bool_ss [MULT]   (* comment: simplify goal   *)
  decide_tac                    (* comment: solve step case *)
```

Tactics can be composed together for `Theorem` using `>>` and `>-`. The `>>` operator is an infix that composes two tactics into one. The `>-` is used to prove subgoals: `>-` *tactic* proves the first subgoal using *tactic*.

Here is the entire proof when composed using `>>` and `>-`.

```
Theorem less_eq_mult:
  ∀n:num. n <= n * n
Proof
  Induct_on ‘n’
  >- decide_tac
  >- (asm_simp_tac bool_ss [MULT]
      >> decide_tac)
QED
```

Copy the above into HOL4 using text-selection, and then `M-h M-r`, as in Section 2.2.

2.8 Displaying types in HOL4

HOL4 does not by default display types. Press `M-h C-t` to switch printing of type information on or off.

2.9 Interrupting HOL4

Press `M-h C-c` to interrupt HOL4 — useful when a tactic fails to terminate (*e.g.* `metis_tac` often fails to terminate when unsuccessfully applied).

2.10 Making a definition

Functions are defined using `Definition ... End`, *e.g.* a function that squares a natural number is defined as follows.

```
Definition SQUARE_def:
  SQUARE n = n * n
End
```

Data-types are defined using `Datatype ... End`, *e.g.* a binary tree which holds values of type `'a` (a type variable) at the leaves:

```
Datatype:
  TREE = LEAF 'a | BRANCH TREE TREE
End
```

A valid tree is *e.g.* `BRANCH (LEAF 5) (BRANCH (LEAF 1) (LEAF 7))` with type `num TREE`, where `num` is the type name for a natural number. We can define recursive functions, *e.g.*

```
Definition MAP_TREE_def:
 (MAP_TREE f (LEAF n) = LEAF (f n)) ∧
 (MAP_TREE f (BRANCH u v) = BRANCH (MAP_TREE f u) (MAP_TREE f v))
End
```

`SQUARE_def` and `MAP_TREE_def` are theorems containing the above definitions. Theorems describing `TREE` can be retrieved by copying the following into HOL4 by pressing `C-space` then `M-h M-r`, as described in Section 2.2.

```
val TREE_11 = fetch "-" "TREE_11";
val TREE_distinct = fetch "-" "TREE_distinct";
```

2.11 Making a theory

Proofs and definitions are stored in files called scripts, *e.g.* we can store the definitions from above in a file called `less_lemmaScript.sml`, which should begin with the lines

```
open HolKernel boolLib bossLib Parse
val _ = new_theory "less_lemma";
```

and end with the line

```
val _ = export_theory();
```

Thus, the entire file can be:

```
open HolKernel boolLib bossLib Parse
val _ = new_theory "less_lemma";

Theorem less_add_1:
  ∀n. n < n + 1
Proof
  decide_tac
QED

val _ = export_theory();
```

The theory file `less_lemmaTheory` is created by executing `Holmake` in the directory where `less_lemmaScript.sml` is stored. A human readable version of the compiled theory is stored under `less_lemmaTheory.sig`.

3 Searching for theorems and theories

HOL4 has a large collection of library theories. The most commonly used are:

`arithmeticTheory`	– natural numbers, *e.g.* `0, 1, 2, SUC 0, SUC 6`
`listTheory`	– lists, *e.g.* `[1;2;3] = 1::2::3::[], HD xs`
`pred_setTheory`	– simple sets, *e.g.* `{1;2;3}, x IN s UNION t`
`pairTheory`	– pairs/tuples, *e.g.* `(1,x), (2,3,4,5), FST (x,y)`
`wordsTheory`	– n-bit words, *e.g.* `0w:word32, 1w:'a word, x !! 1w`

Other standard theories include:

```
bagTheory  boolTheory  combinTheory  fcpTheory  finite_mapTheory
fixedPointTheory  floatTheory  integerTheory  limTheory
optionTheory  probTheory  ratTheory  realTheory
relationTheory  rich_listTheory  ringTheory  seqTheory
sortingTheory  state_transformerTheory  stringTheory  sumTheory
topologyTheory  transcTheory  whileTheory
```

The library theories are conveniently browsed using the following HTML reference page (created when HOL4 is compiled). Replace `<path>` with the path to your HOL4 installation.

```
<path>/HOL/help/HOLindex.html
```

Once theories have been opened (see Section 2.2), one can search for theorems in the current context using `print_match`. For example, with `arithmeticTheory` opened, doing `M-h M-r` with the following selected,

```
print_match [] “n DIV m <= k”
```

prints a list of theorems containing n `DIV` $m \leq k$ for some n, m, k:

```
arithmeticTheory.DIV_LE_MONOTONE (THEOREM)
------------------------------------------
⊢ ∀n x y. 0 < n ∧ x ≤ y ⇒ x DIV n ≤ y DIV n


arithmeticTheory.DIV_LE_X (THEOREM)
-----------------------------------
⊢ ∀x y z. 0 < z ⇒ (y DIV z ≤ x ⇔ y < (x + 1) * z)
[$(HOLDIR)/src/num/theories/arithmeticScript.sml:2717]


arithmeticTheory.DIV_LESS_EQ (THEOREM)
--------------------------------------
⊢ ∀n. 0 < n ⇒ ∀k. k DIV n ≤ k


dividesTheory.DIV_LE (THEOREM)
------------------------------
⊢ ∀x y z. 0 < y ∧ x ≤ y * z ⇒ x DIV y ≤ z


dividesTheory.DIV_LE_MONOTONE_REVERSE (THEOREM)
-----------------------------------------------
⊢ ∀x y. 0 < x ∧ 0 < y ∧ x ≤ y ⇒ ∀n. n DIV y ≤ n DIV x


dividesTheory.LE_MULT_LE_DIV (THEOREM)
--------------------------------------
⊢ ∀n. 0 < n ⇒ ∀k m. m MOD n = 0 ⇒ (m ≤ n * k ⇔ m DIV n ≤ k)


val it = (): unit
```

Try to write increasingly specific queries if the returned list is long, *e.g.* `print_match [] ``n DIV m``` returns a list of length 32. Note that `print_match [] ``DIV``` does not work since `DIV` is an infix operator, but `print_match [] ``$DIV``` works.

The key-binding `M-h m` (and the menu entry "DB match") will prompt for the term pattern to search for, and pass this query onto the HOL session (saving the need to type `print_match []` and the enclosing quotation marks).

It is also possible to search over theorem *names* using the function `DB.find`, or the key-binding `M-h f`. The string provided to this name is a regular expression that ignores case and scans all of the known theorems' names, searching for those that include a sub-string matching the regular expression. In addition to the standard operators (`|`, `*`, ...), a particularly useful addition is `~`, which is defined:

$$re_1\tilde{}\,re_2 = (.^*re_1.^*)\&(.^*re_2.^*)$$

where & is the regular expression intersection operator. Thus, if one writes `DB.find "foo~bar"`, one will get back a list of all theorems whose names include both the strings `"foo"` and `"bar"`, which is useful if one is not sure about the order in which those substrings occur in the theorem name.

4 Common proof tactics

Most HOL4 proofs are carried out by stating a goal and then applying *proof tactics* that reduce the goal. This section describes basic use of the most important proof tactics. Press `C-space` then `M-h e` to apply a tactic (Section 2.4).

4.1 Automatic provers

Simple goals can often be proved automatically by `metis_tac`, `decide_tac` or `EVAL_TAC`. Of these, `metis_tac` is first-order prover which is good at general problems, but requires the user to supply a list of relevant theorems, *e.g.* the following goal is proved by `metis_tac [MOD_TIMES2,MOD_MOD,MOD_PLUS]`.

```
∀k. 0 < k ==> ∀m p n. (m MOD k * p + n) MOD k = (m * p + n) MOD k
```

`decide_tac` handles linear arithmetic over natural numbers, *e.g.* `decide_tac` solves:

```
∀m n k. m < n ∧ n < m+k ∧ k <= 3 ∧ ~(n = m+1) ==> (n = m+2)
```

`EVAL_TAC` is good at fully instantiated goals, *e.g.* `EVAL_TAC` solves:

```
0 < 5 ∧ (HD [4;5;6;7] + 2**32 = 3500 DIV 7 + 4294966800)
```

4.2 Proof set-up

Goals that contain top-level universal quantifiers (`∀x.`), implication (`==>`) or conjunction (`∧`) are often taken apart using `rpt strip_tac` or just `strip_tac`, *e.g.* the goal `∀x. (∀z. x < h z) ==> ∃y. f x = y` becomes the following. (Assumptions are written under the line.)

```
∃y. f x = y
------------------------------------
  ∀z. x < h z
```

4.3 Existential quantifiers

Goals that have a top-level existential quantifier can be given a witness using `qexists_tac`, *e.g.* `qexists_tac ‘1’` applied to goal `∃n. ∀k. n * k = k` produces goal `∀k. 1 * k = k`.

4.4 Rewrites

Most HOL4 proofs are based on rewriting using equality theorems, *e.g.*

```
ADD_0:           |- ∀n. n + 0 = n
LESS_MOD:        |- ∀n k. k < n ==> (k MOD n = k)
```

`asm_simp_tac` and `full_simp_tac` are two commonly used rewriting tactics, *e.g.* suppose the goal is the following:

```
5 + 0 + m = (m MOD 10) + (5 MOD 8)
------------------------------------
  0.  p = 2 + 0 + (m MOD 10)
  1.  m < 10
```

`asm_simp_tac bool_ss [ADD_0,LESS_MOD]` rewrites the goal using the supplied theorems together with the current goal's assumptions and some boolean simplifications `bool_ss`:

```
5 + m = m + (5 MOD 8)
------------------------------------
  0.  p = 2 + 0 + (m MOD 10)
  1.  m < 10
```

`full_simp_tac bool_ss [ADD_0,LESS_MOD]` does the same except that it also applies the rewrites to the assumptions:

```
5 + m = m + (5 MOD 8)
------------------------------------
  0.  p = 2 + m
  1.  m < 10
```

`bool_ss` can be replaced by `std_ss`, which is a stronger simplification set that would infer `5 < 8` and hence simplify `5 MOD 8` as well. I recommend that the interested reader also reads about `AC`, `Once` and `srw_tac`.

4.5 Induction

Use the tactic `Induct_on ‘x’` to start an induction on `x`. Here `x` can be any variable with a recursively defined type, *e.g.* a natural number, a list or a `TREE` as defined in Section 2.10. One can start a complete (or strong) induction over the natural number `n` using `completeInduct_on ‘n’`. As with `Cases_on` one can also induct on terms (*e.g.*, `Induct_on ‘hi - lo’`), though these proofs can be harder to carry out.

4.6 Case splits

A goal can be split into cases using `Cases_on` ‘`x`’. The goal is split according to the constructors of the type of `x`, *e.g.* for the following goal

```
∀x. ~(x = []) ==> (x = HD x::TL x)
```

`Cases_on` ‘`x`’ splits the goal into two:

```
~(h::t = []) ==> (h::t = HD (h::t)::TL (h::t))

~([] = []) ==> ([] = HD []::TL [])
```

Case splits on boolean expressions are also useful, *e.g.* `Cases_on` ‘`n < 5`’.

4.7 Subproofs

It is often useful to start a mini-proof inside a larger proof, *e.g.* for the goal

```
foo n
------------------------------------
  0 < n
```

we might want to prove `h n = g n` assuming `0 < n`. We can start such a subproof by typing `sg` ‘`h n = g n`’.[1] The new goal stack:

```
foo n
------------------------------------
  0.  0 < n
  1.  h n = g n

h n = g n
------------------------------------
  0 < n
```

If `h n = g n` can be proved in one step, *e.g.* using `metis_tac [MY_LEMMA]`, then apply ‘`h n = g n`’ `by metis_tac [MY_LEMMA]` instead of `sg` ‘`h n = g n`’. If the subgoal requires multiple steps the tactic after the `by` will need to be parenthesised: ‘*`goal`*’ `by (`*tac*$_1$ `>>` *tac*$_2$ `...)`

4.8 Proof by contradiction

Use `CCONTR_TAC` to add the negation of the goal to the assumptions. The task is then to prove that one of the assumptions of the goal is false. One can *e.g.* add more assumptions using `` `...` by ...``, described above, until one assumption is the negation of another assumption (and then apply `metis_tac []`).

[1]You can also use the emacs binding `M-h M-s` with the cursor inside the sub-goal.

4.9 More tactics

An HTML reference of all tactics and proof tools is created when HOL4 is compiled. Replace `<path>` with the path to your HOL4 installation.

```
<path>/HOL4/help/src/htmlsigs/idIndex.html
```

The reference provides an easy way to access both the implementations of tactics as well as their documentation (where such exists). The interested reader may want to look up the following:

```
CONV_TAC  disj1_tac  disj2_tac  match_mp_tac  mp_tac  pat_assum  Q
```

5 Further reading and general advice

General advice on using HOL4:

1. State definitions carefully with the subsequent proofs in mind.
2. Make proofs reusable by splitting them into multiple small lemmas.
3. Strive to make the most of library theories and rewriting.

One can only learn HOL4 via examples, so try proving something. Example problems and solutions are presented in the *HOL Tutorial*, available under:

`https://hol-theorem-prover.org/#doc`

The same page also contains links to:

HOL Description – a description of the HOL4 system
HOL Reference – a detailed descriptions of proof tactics and other tools
HOL Logic – a presentation of the underlying logic

For day-to-day look-ups, I find `print_match` (illustrated in Section 3) and the HTML reference (mentioned in Section 4.9) most helpful.