Tutorial HOL

Program Verification Course 11/12

It is important that you do *all* the exercies here. A solution is provided to help you; but try not to peek into it too quickly.

1 Learning Basic Tactics

Do not use HOL's power tactics in the exercises below.

1. Prove this: $p \wedge (p \Rightarrow q) \Rightarrow q$.

   Hints. Check out your HOL documentation. You have it in the `help` directory of your HOL install, or you can also check the online version.

   Basic tactics are provided by the 'Structure' `Tactic`. Check the documentation of the tactics `STRIP_TAC` and `RES_TAC`.

2. Doing a case split.

   Prove this: $p \wedge (p \Rightarrow q \Rightarrow r) \Rightarrow (\neg q \vee r)$

   Hints. This time you have a disjunction on your 'hypothesis': $\neg q \vee r$. To prove it you may want to do a case split, e.g. on the cases that q is true and false. Check out the documentation of the tactic `ASM_CASES_TAC`.

   The structure `Rewrite` provides a whole bunch of basic rewriting utilities, including rewriting tactics. Check the doc of the tactic `ASM_REWRITE_TAC`.

3. Composing your proof to a 'proof script'.

   So far you writing your proof interactively. This works fine, but it is not the best form to document and to share the proof. Gather your proof fragments from No. 2 and compose them into a single tactic. Write your proof in this style:

   ```
   val mytheorem1 = prove(
       --`p /\ (p ==> q ==> r) ==> (~q \/ r)`-- ,
         tactic1
         THEN tactic2
         THEN ... etc
       ) ;
   ```

   Check the doc of `THEN` and `THENL` in the structure `Tactical`.

4. Defining new concepts.

   Let define the function identity. Let me give you two definitions:

   ```
   fun identity x = x ;
   val identity_def = Define `identity x = x` ;
   ```

   What is the important difference between the two definitions?

5. Querying the type of something.

   You can query the type of an ML-value simply by typing the value in the ML interpreter, as in:

   ```
   - identity;
   > val 'a it = fn : 'a -> 'a
   ```

   Querying the type of a HOL-value is a bit more involed. See below; and see if you understand why you get those different answers.

   ```
   - identity_def ;
   > val it = |- !x. identity x = x : thm

   - `identity` ;
   > val 'a it = [QUOTE " (*#loc 152 2*)identity"] : 'a frag/1 list

   - (--`identity`--) ;
   <<HOL message: inventing new type variable names: 'a>>
   > val it = ``identity`` : term

   - Term `identity` ;
   <<HOL message: inventing new type variable names: 'a>>
   > val it = ``identity`` : term

   - type_of (--`identity`--) ;
   <<HOL message: inventing new type variable names: 'a>>
   > val it = ``:'a -> 'a`` : hol_type

   - type_of (Term `identity`) ;
   <<HOL message: inventing new type variable names: 'a>>
   > val it = ``:'a -> 'a`` : hol_type
   ```

   Note: to ask the type of an ML infix operator:

   ```
   - + ;
   ! Ill-formed infix expression

   - op+ ;
   > val it = fn : int * int -> int
   ```

   To ask the type of a HOL infix operator:

   ```
   - type_of (--`+`--) ;
   ! HOL_ERR

   - type_of (--`(+)`--) ;
   <<HOL message: more than one resolution of overloading was possible>>
   > val it = ``:int -> int -> int`` : hol_type
   ```

6. Let's prove some properties about functions. Prove this:

   $$(\forall x.\ identity\ (identity\ x) = x)$$

   Hint. Check the doc of `REWRITE_TAC`. It is in the structure `Rewrite`.

7. Sometimes you get stuck. Prove this:

   $$identity \circ identity \ = \ identity$$

   In HOL function composition is represented by the infix `o`. To prove the property above, you will also need to use the definition of `o`. This is provided in the theory `combinTheory`. This theory has been pre-loaded by HOL, but not 'opened'.

You can access things exported by a loaded module in a fully qualified way, as in `BoolTheory.o_DEF`. If the module is opened, then you don't need the full qualification; so simply `o_DEF`.

To load and open a module named `foo`, you do:

```
load  "foo" ;
open foo ;
```

For `boolTheory`, we just need to open it.

Try now to rewrite your goal with the definition of `o` and *identity*. Unfortunately now you get stuck in this goal:

```
(\x. x) = identity
```

This is clearly a tautology. HOL fails to see it because we have defined *identity* in HOL as follows:

$$\mathit{identity}\ x \ = \ x$$

HOL refuses to auto-lift this definition to $\mathit{identity} = (\lambda x.x)$. Although in this case it is desired to do so, there are other cases where we actually do not want this auto-lifting.

Anyway, to make this work we can convert the functional equality to 'point equality' by applying this so-called extensionality theorem:

```
- FUN_EQ_THM;
> val it = |- !f g. (f = g) <=> !x. f x = g x : thm
```

Now you should be able to finish the proof. Oh, one more: use `BETA_TAC` to do β-reduction.

8. Prove this:

$$(x = 0) \quad \Rightarrow \quad (\exists f : \mathit{num} \to \mathit{num}.\ f\ x = 0)$$

The hypothesis is in the existential form. In general HOL cannot automatically prove such a formula. Check the doc of `EXISTS_TAC`.

Note that `EXISTS_TAC` fails if you try to eliminate an existentially bound variable x with a term t whose type is more general than x. Compare the effect of applying these two:

```
- e (EXISTS_TAC (--`identity`--))

- e (EXISTS_TAC (--`identity:num->num`--))
```

2 Proving formulas involving simple integer arithmetics

1. Let's define this function $\mathit{dbl} : \mathit{int} \to \mathit{int}$

$$\mathit{dbl}\ x \ = \ x + x$$

First you will need the type `int` and the library supporting it. This is not pre-loaded in HOL. Load and open `intLib`. So:

```
load "intLib" ;
open intLib ;
```

Now you can write the definition. Make sure that the type of *dbl* is indeed $: int \to int$.

Prove this: $(\forall x.\ x{>}1 \quad \Rightarrow \quad dbl\ x > x{+}1)$.

In general proving an arithmetic formula can be quite involved. However for simple things, HOL comes with some decision procedures. The tactic `ARITH_TAC` exported by `intLib` can solve simple integer arithmetics. But do note that in general the problem is undecidable.

2. Try another one: $(\exists f.\ (\forall x.\ f\ x > 2 * x))$.

   Hint. The syntax for λ-expression is e.g. $(\backslash\mathtt{x}.\ \mathtt{x} + \mathtt{1})$.

3 Modelling, a simple case

Imagine a program (let's call it *skido*) that operates on a single variable $x : int$. It either leaves x unchanged, or double its value. The choice is non-deterministic. Model this program in HOL.

Because it is non-deterministic, note that you can *not* model it by a function of type $int \to int$; so you have to come up with something else.

Define the concept of *even* integer.

Now prove that if x is initially even, then iterating *skido* any number of times will keep its value even. You will first need to express/model iteration in HOL.

Hints.

1. Check chapter 4 of the *HOL Description* (download it from PV or HOL website) on how to define a recursive function in HOL.

2. Define the (higher order) function `iter n f` that will iterate a program `f`, `n` number of times. Make sure that `n` is typed as `num` to allow you to do a natural number induction later.

   You can limit the type of `f` so that it only represents a program that operates on a single variable `x`.

3. Check chapter 5 of the HOL Description on high level proofs, in particular the use of power tactics such as `RW_TAC` and `PROVE_TAC`.

4. Finally, some handy self-defined tactcis that maybe helpful for you:

```
val UNDISCH_TOP_TAC = FIRST_ASSUM (UNDISCH_TAC o concl) ;
val UNDISCH_ALL_TAC = REPEAT  UNDISCH_TOP_TAC ;
```