This document is based on material from the "Interactive Theorem Proving Course" by Thomas Tuerk (https://www.thomas-tuerk.de):
https://github.com/thtuerk/ITP-course

This document includes additions by:

- Pablo Buiras (https://people.kth.se/~buiras/)
- Arve Gengelbach (https://people.kth.se/~arveg/)
- Karl Palmskog (https://setoid.com)

ITPPV Homework 3

due 23:59 CET, Tuesday February 11, 2020

1 Self-Study

1.1 Tactics and Tacticals

Read background information on the tactics mentioned in lecture 3. Specifically, for each of the tactics and tacticals mentioned in the slides, read the entry in the description manual.

1.2 hol-mode

Carefully study the `Goalstack` submenu of hol-mode. Learn the keycodes to set goals, expand tactics, undo the last tactic expansion, restart the current proof and drop the current goal.

1.3 The Number and List Theories

We will use the natural number theory `numTheory` as well as the list theories `listTheory` and `rich_listTheory` a lot. Please familiarise yourself with these HOL4 theories (e. g. by looking at their signature in the HTML version of the HOL4 Reference). I also recommend reading up on other common theories like `optionTheory`, `oneTheory` and `pairTheory`. These won't be needed for this weeks exercises, though.

2 Backward Proofs

Part of this homework is about searching for useful existing theorems (e.g., useful rewrite rules). Another part is understanding the effect of different rewrite rules and their combinations. Even if you have experience with using HOL4, please refrain from using automated rewrite tools that have not been covered in the lectures yet (which negate the purpose of the homework). Please don't use HOL4's simplifier, and in particular, don't use stateful simp-sets. Similarly, please don't use the compute lib via e.g., `EVAL_TAC`.

For these tasks, it might be beneficial to open the modules `listTheory` and `rich_listTheory` via: `open listTheory rich_listTheory;`

You will notice that when opening them that a lot of definitions are printed. This can consume quite some time when opening many large theories. Play around with the hol-mode commands `Send region to HOL - hide non-errors` and `Quite - hide output except errors` to avoid this printout and the associated waiting time.

2.1 Replay Proofs from Lecture

If you have never done a tactical proof in HOL4 before, we recommend following the example interactive proofs from Part VIII of the slides from lecture 3. Type them in your own HOL4 session, making the same mistakes as in the lecture. Use hol-mode to control the goalStack via commands like expand, back-up, set goal, and drop goal. Get a feeling for how to interactively develop a tactical proof. This task is optional; if you already feel confident with tactical proving, feel free to skip it.

2.2 Formalise Induction Proofs from Exercise 1

For homework 1, some simple properties of lists were proved with pen and paper via structural induction. Let's now prove them formally using HOL4. Prove `!l. l ++ [] = l` by induction, using the definition of `APPEND` (`++`). Similarly, prove the associativity of `APPEND`, i.e., prove

```
!l1 l2 l3. l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3
```

2.3 Reverse

SML's `revAppend` is called `REV` in HOL4. Via any useful theorems you can find, prove the theorem

```
!l1 l2. LENGTH (REV l1 l2) = (LENGTH l1 + LENGTH l2)
```

Next, let's as an exercise reprove the existing theorems `REVERSE_REV` and `REV_REVERSE_LEM`. This means, first prove

```
!l1 l2. REV l1 l2 = REVERSE l1 ++ l2
```

and then, prove `!l. REVERSE l = REV l []` using this theorem. You should not use the theorems `REVERSE_REV` or `REV_REVERSE_LEM` in these proofs.

2.4 Length of Drop

Prove the theorem

```
!l1 l2. LENGTH (DROP (LENGTH l2) (l1 ++ l2)) = LENGTH l1
```

directly using induction, i.e., without using lemmas like `LENGTH_DROP`. Do one proof with `Induct_on` and a very similar proof with `Induct`. This is a bit tricky. Please play around with the proof for some time. If you can't figure it out, look at the hints at the end of this homework.

2.5 More Efficient List Operations

Often lists are used to encode sets, and the ordering of the list items is irrelevant. More efficient list operations can avoid reversing the list. In this exercise you show first that `map_rev` corresponds to `MAP` despite the ordering of elements, and then prove the same for a list function of your choice.

Using the following definitions of `map_rev`, defined in terms of `map_rev_acc`.

```
Definition map_rev_acc_def:
  map_rev_acc acc f [] = acc
  /\ map_rev_acc acc f (h::ls) = map_rev_acc (f h :: acc) f ls
End

Definition map_rev_def:
  map_rev f = map_rev_acc [] f
End
```

Prove correctness of `map_rev`:

```
map_rev f ls = REVERSE (MAP f ls)
```

Next, choose one of the list functions `FILTER`, `TAKE`, or `FLAT` and implement a version `*_rev` that avoids reversing the argument. Prove correctness of the implementation, similar to the above.

2.6 Making Change

In homework 1, you were asked to implement a function `make_change` in SML. Let's now define it in HOL4 and prove some properties about it. Define the function `MAKE_CHANGE` in HOL4 using

```
val MAKE_CHANGE_def = Define `
  (MAKE_CHANGE [] a = if (a = 0) then [[]] else []) /\
  (MAKE_CHANGE (c::cs) a = (
     (if (c <= a /\ 0 < a) then
       (MAP (\l. c::l) (MAKE_CHANGE cs (a - c)))
     else []) ++ (MAKE_CHANGE cs a)))`;
```

Then, prove the theorems

```
!cs. MAKE_CHANGE cs 0 = [[]]
```

and

```
!cs a l. MEM l (MAKE_CHANGE cs a) ==> (SUM l = a)
```

3 Hints

3.1 More Efficient List Operations

As one intermediate step in the correctness proof for the above accumulator function `map_rev_acc` you need to show the following invariant:

```
!f ls acc. map_acc acc f ls = (map_acc [] f ls) ++ acc
```

3.2 Length of Drop

For proving `!l1 l2. LENGTH (DROP (LENGTH l2) (l1 ++ l2)) = LENGTH l1`, induction on the structure of `l2` is a good strategy. However, one needs to be careful that `l1` stays universally quantified. Expanding naively with `GEN_TAC >> Induct` will remove the needed universal quantification of `l1`.

To solve this, you can either use `Induct_on `l2`` or get rid of both universal quantifiers and then introduce them in a different order again. This is achieved by `REPEAT GEN_TAC >> SPEC_TAC (``l1:'a list``, ``l1:'a list``) >> SPEC_TAC (``l2:'a list``, ``l2:'a list``)`.