Interactive Theorem Proving (ITP) Course
Part XV

Thomas Tuerk (tuerk@kth.se)

Academic Year 2016/17, Period 4

version d88c12b of Wed Jun 7 10:08:28 2017

Part XIV

Maintainable Proofs

Motivation

- proofs are hopefully still used in a few weeks, months or even years
- often the environment changes slightly during the lifetime of a proof
  - your definitions change slightly
  - your own lemmata change (e. g. become more general)
  - used libraries change
  - HOL changed
    - automation became more powerful
    - rewrite rules in certain simpsets changed
    - definition packages produce slightly different theorems
    - autogenerated variable-names change
    - ...
- even if HOL and used libraries are stable, proofs often go through several iterations
- often they are adapted by someone else than the original author
- therefore it is important that proofs are easily maintainable

Nice Properties of Proofs

- maintainability is closely linked to other desirable properties of proofs
- proofs should be
  - easily understandable
  - well-structured
  - robust
    - they should be able to scope with minor changes to environment
    - if they fail they should do so at sensible points
  - reusable
- How can one write proofs with such properties?
- as usual, there are no easy answers but plenty of good advice
- I recommend following the advice of **ProofStyle** manual
- parts of this advice as well as a few extra points are discussed in the following

Formatting

- format your proof such that it easily understandable
- make the structure of the proof very clear
- show clearly where subgoals start and stop
- use indentation to mark proofs of subgoals
- use empty lines to separate large proofs of subgoals
- use comments where appropriate

Formatting Example I

Bad Example Term Formatting

```
prove (``!l1 l2. l1 <> [] ==> LENGTH l2 <
LENGTH (l1 ++ l2)``,
...)
```

Good Example Term Formatting

```
prove (``!l1 l2. l1 <> [] ==>
                 (LENGTH l2 < LENGTH (l1 ++ l2))``,
...)
```

Formatting Example II

Bad Example Subgoals

```
prove (``!l1 l2. l1 <> [] ==> (LENGTH l2 < LENGTH (l1 ++ l2))``,
Cases >>
REWRITE_TAC[] >>
REWRITE_TAC[listTheory.LENGTH, listTheory.LENGTH_APPEND] >>
REPEAT STRIP_TAC >>
DECIDE_TAC)
```

Improved Example Subgoals

At least show when a subgoal starts and ends

```
prove (``!l1 l2. l1 <> [] ==> (LENGTH l2 < LENGTH (l1 ++ l2))``,
Cases >> (
  REWRITE_TAC[]
) >>
REWRITE_TAC[listTheory.LENGTH, listTheory.LENGTH_APPEND] >>
REPEAT STRIP_TAC >>
DECIDE_TAC)
```

Formatting Example II 2

Good Example Subgoals

Make sure REWRITE_TAC is only applied to first subgoal and proof fails, if it does not solve this subgoal.

```
prove (``!l1 l2. l1 <> [] ==> (LENGTH l2 < LENGTH (l1 ++ l2))``,
Cases >- (
  REWRITE_TAC[] >>
)
REWRITE_TAC[listTheory.LENGTH, listTheory.LENGTH_APPEND] >>
REPEAT STRIP_TAC >>
DECIDE_TAC)
```

Formatting Example II 3

Alternative Good Example Subgoals

Alternative good formatting using THENL

```
prove (``!l1 l2. l1 <> [] ==> (LENGTH l2 < LENGTH (l1 ++ l2))``,
Cases >| [
  REWRITE_TAC[],

  REWRITE_TAC[listTheory.LENGTH, listTheory.LENGTH_APPEND] >>
  REPEAT STRIP_TAC >>
  DECIDE_TAC
])
```

Another Bad Example Subgoals

Bad formatting using THENL

```
prove (``!l1 l2. l1 <> [] ==> (LENGTH l2 < LENGTH (l1 ++ l2))``,
Cases >| [REWRITE_TAC[],
REWRITE_TAC[listTheory.LENGTH, listTheory.LENGTH_APPEND] >>
REPEAT STRIP_TAC >> DECIDE_TAC])
```

Some basic advice

- use semicoli after each declaration
  - if exception is raised during interactive processing (e. g. by a failing proof), previous successful declarations are kept
  - it sometimes leads to better error messages in case of parsing errors
- use plenty of parentheses to make structure very clear
- don't ignore parser warnings
  - especially warnings about multiple possible parse trees are likely to lead to unstable proofs
  - understand why such warnings occur and make sure there is no problem
- format your development well
  - use indentation
  - use linebreaks at sensible points
  - don't use overlong lines
  - ...
- don't use `open` in middle of files
- personal opinion: avoid unicode in source files

KISS and Premature Optimisation

- follow standard design principles
  - **KISS** principle
  - "**premature optimization is the root of all evil**" (Donald Knuth)
- don't try to be overly clever
- simple proofs are preferable
- proof-checking-speed mostly unimportant
- conciseness not a value in itself but desirable if it helps
  - readability
  - maintainability
- abstraction is often desirable, but also has a price
  - don't use too complex, artificial definitions and lemmata

Too much abstraction

Too much abstraction Example

```
val TOO_ABSTRACT_LEMMA = prove (``
!(size :'a -> num) (P : 'a -> bool) (combine : 'a -> 'a -> 'a).
  (!x. P x ==> (0 < size x)) /\
  (!x1 x2. size x1 + size x2 <= size (combine x1 x2)) ==>

  (!x1 x2. P x1 ==> (size x2 < size (combine x1 x2)))``,
...)


prove (``!l1 l2. l1 <> [] ==> (LENGTH l2 < LENGTH (l1 ++ l2))``,
  some proof using ABSTRACT_LEMMA
)
```

Too clever tactics

- a common mistake is to use too clever tactics
  - intended to work on many (sub)goals
  - using `TRY` and other fancy trial and error mechanisms
  - intended to replace multiple simple, clear tactics
- typical case: a tactic containing `TRY` applied to many subgoals
- it is often hard to see why such tactics work
- if something goes wrong, they are hard to debug
- general advice: don't factor with tactics, instead use definitions and lemmata

Too Clever Tactics Example I

Bad Example Subgoals

```
prove (``!l1 l2. l1 <> [] ==> (LENGTH l2 < LENGTH (l1 ++ l2))``,
Cases >> (
  REWRITE_TAC[listTheory.LENGTH, listTheory.LENGTH_APPEND] >>
  REPEAT STRIP_TAC >>
  DECIDE_TAC
))
```

Alternative Good Example Subgoals II

```
prove (``!l1 l2. l1 <> [] ==> (LENGTH l2 < LENGTH (l1 ++ l2))``,
Cases >> SIMP_TAC list_ss [])

prove (``!l1 l2. l1 <> [] ==> (LENGTH l2 < LENGTH (l1 ++ l2))``,
Cases >| [
  REWRITE_TAC[],

  REWRITE_TAC[listTheory.LENGTH, listTheory.LENGTH_APPEND] >>
  REPEAT STRIP_TAC >>
  DECIDE_TAC
])
```

Too Clever Tactics Example II

Bad Example

```
val oadd_def = Define `(oadd (SOME n1) (SOME n2) = (SOME (n1 + n2))) /\
                       (oadd _         _         = NONE)`;
val osub_def = Define `(osub (SOME n1) (SOME n2) = (SOME (n1 - n2))) /\
                       (osub _         _         = NONE)`;
val omul_def = Define `(omul (SOME n1) (SOME n2) = (SOME (n1 * n2))) /\
                       (omul _         _         = NONE)`;

val onum_NONE_TAC =
  Cases_on `o1` >> Cases_on `o2` >>
  SIMP_TAC std_ss [oadd_def, osub_def, omul_def];

val oadd_NULL = prove (
  ``!o1 o2. (oadd o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  onum_NONE_TAC);
val osub_NULL = prove (
  ``!o1 o2. (osub o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  onum_NONE_TAC);
val omul_NULL = prove (
  ``!o1 o2. (omul o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  onum_NONE_TAC);
```

Too Clever Tactics Example II

Good Example

```
val obin_def = Define `(obin op (SOME n1) (SOME n2) = (SOME (op n1 n2))) /\
                       (obin _  _         _         = NONE)`;
val oadd_def = Define `oadd = obin $+`;
val osub_def = Define `osub = obin $-`;
val omul_def = Define `omul = obin $*`;

val obin_NULL = prove (
  ``!op o1 o2. (obin op o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  Cases_on `o1` >> Cases_on `o2` >> SIMP_TAC std_ss [obin_def]);

val oadd_NULL = prove (
  ``!o1 o2. (oadd o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  REWRITE_TAC[oadd_def, obin_NULL]);
val osub_NULL = prove (
  ``!o1 o2. (osub o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  REWRITE_TAC[osub_def, obin_NULL]);
val omul_NULL = prove (
  ``!o1 o2. (omul o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  REWRITE_TAC[omul_def, obin_NULL]);
```

Use many subgoals and lemmata

- often it is beneficial to use subgoals
  - they structure long proofs well
  - they help keeping the proof state clean
  - they mark clearly what one tries to proof
  - they provide points where proofs can break sensibly
- general subgoals should often become lemmata
  - this improves reusability
  - proof scripts become shorter
  - proofs are disentangled

Subgoal Example

First Version

```
val IS_WEAK_SUBLIST_FILTER_REFL = store_thm ("IS_WEAK_SUBLIST_FILTER_REFL",
  ``!l. IS_WEAK_SUBLIST_FILTER l l``,
REWRITE_TAC[IS_WEAK_SUBLIST_FILTER_def] >>
Induct_on `l` >- (
  Q.EXISTS_TAC `[]` >>
  SIMP_TAC list_ss [FILTER_BY_BOOLS_REWRITES]
) >>
FULL_SIMP_TAC std_ss [] >>
GEN_TAC >>
Q.EXISTS_TAC `T::bl` >>
ASM_SIMP_TAC list_ss [FILTER_BY_BOOLS_REWRITES])
```

- the example above is taken from exercise 5
- the proof mixes properties of IS_WEAK_SUBLIST_FILTER and properties of FILTER_BY_BOOLS
- it is hard to see what the main idea is

Subgoal Example II

- the following proof separates the property of FILTER_BY_BOOLS as a subgoal
- the main idea becomes clearer

Subgoal Version

```
val IS_WEAK_SUBLIST_FILTER_REFL = store_thm ("IS_WEAK_SUBLIST_FILTER_REFL",
  ``!l. IS_WEAK_SUBLIST_FILTER l l``,
GEN_TAC >>
REWRITE_TAC[IS_WEAK_SUBLIST_FILTER_def] >>
`FILTER_BY_BOOLS (REPLICATE (LENGTH l) T) l = l` suffices_by (
  METIS_TAC[LENGTH_REPLICATE]
) >>
Induct_on `l` >> (
  ASM_SIMP_TAC list_ss [FILTER_BY_BOOLS_REWRITES, REPLICATE]
))
```

Subgoal Example II

- the subgoal is general enough to justify a lemma
- the structure becomes even cleaner
- this improves reusability

Lemma Version

```
val FILTER_BY_BOOLS_REPL_T = store_thm ("FILTER_BY_BOOLS_REPL_T",
  ``!l. FILTER_BY_BOOLS (REPLICATE (LENGTH l) T) l = l``,
Induct >> ASM_REWRITE_TAC [REPLICATE, FILTER_BY_BOOLS_REWRITES, LENGTH]);

val IS_WEAK_SUBLIST_FILTER_REFL = store_thm ("IS_WEAK_SUBLIST_FILTER_REFL",
  ``!l. IS_WEAK_SUBLIST_FILTER l l``,
GEN_TAC >>
REWRITE_TAC[IS_WEAK_SUBLIST_FILTER_def] >>
Q.EXISTS_TAC `REPLICATE (LENGTH l) T` >>
SIMP_TAC list_ss [FILTER_BY_BOOLS_REPL_T, LENGTH_REPLICATE])
```

Avoid Autogenerated Names

- many HOL-tactics introduce new variable names
  - `Induct`
  - `Cases`
  - ...
- the new names are often very artificial
- even worse, generated names might change in future
- proof scripts using autogenerated names are therefore
  - hard to read
  - potentially fragile
- therefore rename variables after they have been introduced
- HOL has multiple tactics supporting renaming
- most useful is `rename1 ‘pat‘`, it searches for pattern and renames vars accordingly

Autogenerated Names Example

Bad Example

```
prove (``!l. 1 < LENGTH l ==> (?x1 x2 l'. l = x1::x2::l')``,
GEN_TAC >>
Cases_on `l` >> SIMP_TAC list_ss [] >>
Cases_on `t` >> SIMP_TAC list_ss [])
```

Good Example

```
prove (``!l. 1 < LENGTH l ==> (?x1 x2 l'. l = x1::x2::l')``,
GEN_TAC >>
Cases_on `l` >> SIMP_TAC list_ss [] >>
rename1 `LENGTH l2` >>
Cases_on `l2` >> SIMP_TAC list_ss [])
```

Proof State before rename1

```
1 < SUC (LENGTH t) ==> ?x2 l'. t = x2::l'
```

Proof State after rename1

```
1 < SUC (LENGTH l2) ==> ?x2 l'. l2 = x2::l'
```