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/`)
- Karl Palmskog (`https://setoid.com`)

# Interactive Theorem Proving and Program Verification
# Lecture 8

Pablo Buiras and Karl Palmskog

Academic Year 2019/20, Period 3–4

Based on slides by Thomas Tuerk

# Part XV

## 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 lemmas change (e. g. , become more general)
  - used libraries change
  - HOL4 changes
    - automation becomes more powerful
    - rewrite rules in certain simpsets change
    - definition packages produce slightly different theorems
    - autogenerated variable names change
    - ...
- even if HOL4 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 cope 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

# 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 semicolons 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 overly long lines
  - ...
- don't use `open` in the middle of files
- lecturers' opinion: avoid using 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 lemmas

# 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 lemmas

# 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 obin_NONE_TAC =
  Cases_on `o1` >> Cases_on `o2` >>
  SIMP_TAC std_ss [oadd_def, osub_def, omul_def];

val oadd_NONE = prove (
  ``!o1 o2. (oadd o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  obin_NONE_TAC);
val osub_NONE = prove (
  ``!o1 o2. (osub o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  obin_NONE_TAC);
val omul_NONE = prove (
  ``!o1 o2. (omul o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  obin_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_NONE = 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_NONE = prove (
  ``!o1 o2. (oadd o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  REWRITE_TAC[oadd_def, obin_NONE]);
val osub_NONE = prove (
  ``!o1 o2. (osub o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  REWRITE_TAC[osub_def, obin_NONE]);
val omul_NONE = prove (
  ``!o1 o2. (omul o1 o2 = NONE) <=> (o1 = NONE) \/ (o2 = NONE)``,
  REWRITE_TAC[omul_def, obin_NONE]);
```

# Use many subgoals and lemmas

- 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 enough subgoals should become lemmas
  - this improves reusability
  - proof script for main lemma becomes shorter
  - proofs are disentangled

# Subgoal Example

- the following example is taken from exercise 5
- we try to prove `!l. IS_WEAK_SUBLIST_FILTER l l`
- given are following definitions and lemmas

```
val FILTER_BY_BOOLS_def = Define `
  FILTER_BY_BOOLS bl l = MAP SND (FILTER FST (ZIP (bl, l)))`;

val IS_WEAK_SUBLIST_FILTER_def = Define `IS_WEAK_SUBLIST_FILTER l1 l2 =
  ?(bl : bool list). (LENGTH bl = LENGTH l1) /\ (l2 = FILTER_BY_BOOLS bl l1)`;

val FILTER_BY_BOOLS_REWRITES = store_thm ("FILTER_BY_BOOLS_REWRITES",
  ``(FILTER_BY_BOOLS [] [] = []) /\
    (!b bl x xs. (FILTER_BY_BOOLS (b::bl) (x::xs) =
     if b then x::(FILTER_BY_BOOLS bl xs) else FILTER_BY_BOOLS bl xs))``,
REWRITE_TAC [FILTER_BY_BOOLS_def, ZIP, MAP, FILTER] >>
Cases_on `b` >> REWRITE_TAC [MAP]);
```

# Subgoal Example II

## 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 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 III

- 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 IV

- 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 HOL4 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
- HOL4 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'
```

# Part XVI

## ITP Support Tools

# ITP Support Tools

- there is a large tool ecosystem around ITPs, e. g. , for
  - proof automation
  - maintenance
  - processing and generation of definitions
  - searching large libraries
- using the right tools can be crucial for productivity
  - avoid spending hours reproving known facts
  - generate boilerplate automatically
  - highlight flaws in definitions early

# Example Tool: Ott

- tool for writing calculi in ASCII syntax that can be exported to HOL4, Coq, Isabelle (and LaTeX)
- `https://github.com/ott-lang/ott`
- helpful for doing deep embeddings of languages
- generates boilerplate for **abstract syntax** and **relations**

# Untyped Lambda Calculus Syntax in Ott

```
metavar var, x ::=
  {{ isa string }} {{ coq nat }} {{ coq-equality }} {{ hol string }}
  {{ tex \mathit{[[termvar]]} }} {{ com term variable }}

grammar
term, t :: 't_' ::= {{ com term }}
| x             ::   :: var                      {{ com variable }}
| \ x . t       ::   :: lam   (+ bind x in t +) {{ com abstraction }}
| t t'          ::   :: app {{ com application }}
| ( t )         :: S :: paren {{ ichl [[t]] }}
| { t / x } t'  :: M :: tsub {{ ichl (tsubst_t [[t]] [[x]] [[t']] ) }}
val, v :: 'v_' ::= {{ com value   }}
  | \ x . t       ::   :: lam                      {{ com lambda  }}

subrules
  val <:: term

substitutions
  single term var :: tsubst
```

# Generated HOL4 Embedding

```
val _ = type_abbrev("var", ``:string``); (* term variable *)
term =  (* term *)
   t_var of var (* variable *)
 | t_lam of var => term (* lambda *)
 | t_app of term => term (* app *)
`;

(** subrules *)
val _ = ottDefine "is_val_of_term" `
    ( is_val_of_term (t_var x) = F)
/\  ( is_val_of_term (t_lam x t) = (T))
/\  ( is_val_of_term (t_app t t') = F)
`;

(** substitutions *)
val _ = ottDefine "tsubst_term" `
(tsubst_term t5 x5 (t_var x) = (if x=x5 then t5 else (t_var x)))
/\ (tsubst_term t5 x5 (t_lam x t) =
 t_lam x (if MEM x5 [x] then t else (tsubst_term t5 x5 t)))
/\ (tsubst_term t5 x5 (t_app t t') =
 t_app (tsubst_term t5 x5 t) (tsubst_term t5 x5 t'))
`;
```

# Untyped Lambda Calculus Semantics in Ott

```
defn
 t1 --> t2 :: :: reduce :: ''
  {{ com $[[t1]]$ reduces to $[[t2]]$ }} by

   --------------------------  :: ax_app
   (\x.t1) v2 -->  {v2/x}t1

   t1 --> t1'
   -------------- :: ctx_app_fun
   t1 t --> t1' t

   t1 --> t1'
   -------------- :: ctx_app_arg
   v t1 --> v t1'
```

For the whole Ott definition, see:

https://github.com/ott-lang/ott/blob/master/tests/test10.ott

# Generated HOL4 Relation

```
val (Jop_rules, Jop_ind, Jop_cases) = Hol_reln`
(* defn reduce *)

(! (x:var) (t1:term) (v2:term) . (clause_name "ax_app") /\
((is_val_of_term v2))
 ==> (* ax_app *)
((reduce (t_app  (t_lam x t1)  v2) (tsubst_term v2 x t1))))

/\ (! (t1:term) (t:term) (t1':term) . (clause_name "ctx_app_fun") /\
(( ( reduce t1 t1' )))
 ==> (* ctx_app_fun *)
((reduce (t_app t1 t) (t_app t1' t))))

/\ (!(v:term) (t1:term) (t1':term) . (clause_name "ctx_app_arg") /\
((is_val_of_term v) /\
( ( reduce t1 t1' )))
 ==> (* ctx_app_arg *)
((reduce (t_app v t1) (t_app v t1'))))
`;
```

For the complete generated HOL4 definition, see:

https://github.com/kth-step/itppv-course/blob/master/hol4-examples/untyped-lambda/lambdaScript.sml

# Example Tool: Lem

- general tool for generating semantic definitions in ITPs
- `https://github.com/rems-project/lem`
- Ott can export Lem definitions
- used in the CakeML verified compiler project
- has library with many standard semantic concepts

# Leveraging External Automatic Solvers

- built-in automatic solvers don't need to be trusted (more than HOL4 itself)
- external solvers can still be useful to try conjectures
- external solver results can be oracle-tagged and integrated into HOL4 developments
- common external solver types: SAT, SMT, FOL
- example external solvers: MiniSAT, Z3, Yices, CVC4, Vampire
- HOL(y)hammer (see HOL4 examples) tries to get benefits of both automatic solvers and HOL4 trust by reconstructing solver proofs inside HOL4

# Testing Properties (QuickCheck)

- when properties to be proven are decidable when instantiated, they can be **tested**
- in Isabelle and Coq, there are **frameworks** that can test properties on many instances and find counterexamples
- in HOL4, this is possible manually through the EmitML module
  1. extract all necessary code to executable language
  2. generate lots of instances of datatypes
  3. check desired property for all generated instances, report successes/failures

More about EmitML can be found in the default course project description:
`https://kth-step.github.io/itppv-course/homeworks/project.pdf`