Interactive Theorem Proving (ITP) Course

Thomas Tuerk (tuerk@kth.se)

KTH

Academic Year 2016/17, Period 4

version 5056611 of Wed May 3 09:55:18 2017

Part I

Introduction

Motivation

- Complex systems almost certainly contain bugs.
- Critical systems (e. g. avionics) need to meet very high standards.
- It is infeasible in practice to achieve such high standards just by testing.
- Debugging via testing suffers from diminishing returns.

"Program testing can be used to show the presence of bugs, but never to show their absence!"
— Edsger W. Dijkstra

Famous Bugs

- Pentium FDIV bug (1994)
  (missing entry in lookup table, $475 million damage)
- Ariane V explosion (1996)
  (integer overflow, $1 billion prototype destroyed)
- Mars Climate Orbiter (1999)
  (destroyed in Mars orbit, mixup of units pound-force and newtons)
- Knight Capital Group Error in Ultra Short Time Trading (2012)
  (faulty deployment, repurposing of critical flag, $440 lost in 45 min on stock exchange)
- ...

Fun to read

`http://www.cs.tau.ac.il/~nachumd/verify/horror.html`
`https://en.wikipedia.org/wiki/List_of_software_bugs`

Proof

- proof can show absence of errors in design
- but proofs talk about a **design**, not a **real system**
- $\Rightarrow$ testing and proving complement each other

**"As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality."
— Albert Einstein**

Mathematical vs. Formal Proof

Mathematical Proof

- informal, convince other mathematicians
- checked by community of domain experts
- subtle errors are hard to find
- often provide some new insight about our world
- often short, but require creativity and a brilliant idea

Formal Proof

- formal, rigorously use a logical formalism
- checkable by *stupid* machines
- very reliable
- often contain no new ideas and no amazing insights
- often long, very tedious, but largely trivial

We are interested in formal proofs in this lecture.

Detail Level of Formal Proof

In **Principia Mathematica** it takes 300 pages to prove 1+1=2.

This is nicely illustrated in **Logicomix - An Epic Search for Truth**.

Automated vs Manual (Formal) Proof

Fully Manual Proof

- very tedious one has to grind through many trivial but detailed proofs
- easy to make mistakes
- hard to keep track of all assumptions and preconditions
- hard to maintain, if something changes (see Ariane V)

Automated Proof

- amazing success in certain areas
- but still often infeasible for interesting problems
- hard to get insights in case a proof attempt fails
- even if it works, it is often not that automated
  - run automated tool for a few days
  - abort, change command line arguments to use different heuristics
  - run again and iterate till you find a set of heuristics that prove it fully automatically in a few seconds

Interactive Proofs

- combine strengths of manual and automated proofs
- many different options to combine automated and manual proofs
  - mainly check existing proofs (e. g. HOL Zero)
  - user mainly provides lemmata statements, computer searches proofs using previous lemmata and very few hints (e. g. ACL 2)
  - most systems are somewhere in the middle
- typically the human user
  - provides insights into the problem
  - structures the proof
  - provides main arguments
- typically the computer
  - checks proof
  - keeps track of all use assumptions
  - provides automation to grind through lengthy, but trivial proofs

Typical Interactive Proof Activities

- provide precise definitions of concepts
- state properties of these concepts
- prove these properties
  - human provides insight and structure
  - computer does book-keeping and automates simple proofs
- build and use libraries of formal definitions and proofs
  - formalisations of mathematical theories like
    - lists, sets, bags, ...
    - real numbers
    - probability theory
  - specifications of real-world artefacts like
    - processors
    - programming languages
    - network protocols
  - reasoning tools

There is a strong connection with programming. Lessons learned in Software Engineering apply.

Different Interactive Provers

- there are many different interactive provers, e. g.
  - Isabelle/HOL
  - Coq
  - PVS
  - HOL family of provers
  - ACL2
  - ...
- important differences
  - the formalism used
  - level of trustworthiness
  - level of automation
  - libraries
  - languages for writing proofs
  - user interface
  - ...

Which theorem prover is the best one? :-)

- there is no **best** theorem prover
- better question: Which is the **best one for a certain purpose**?
- important points to consider
  - existing libraries
  - used logic
  - level of automation
  - user interface
  - importance development speed versus trustworthiness
  - How familiar are you with the different provers?
  - Which prover do people in your vicinity use?
  - your personal preferences
  - ...

In this course we use the HOL theorem prover, because it is used by the TCS group.

Part II

Organisational Matters

Aims of this Course

Aims

- introduction to interactive theorem proving (ITP)
- being able to evaluate whether a problem can benefit from ITP
- hands-on experience with HOL
- learn how to build a formal model
- learn how to express and prove important properties of such a model
- learn about basic conformance testing
- use a theorem prover on a small project

Required Prerequisites

- some experience with functional programming
- knowing Standard ML syntax
- basic knowledge about logic (e. g. First Order Logic)

Dates

- Interactive Theorem Proving Course takes place in Period 4 of the academic year 2016/2017
- always in room 4523 or 4532
- each week

Mondays	10:15 - 11:45	lecture
Wednesdays	10:00 - 12:00	practical session
Fridays	13:00 - 15:00	practical session

- no lecture on Monday, 1st of May, instead on Wednesday, 3rd May
- last lecture: 12th of June
- last practical session: 21st of June
- 9 lectures, 17 practical sessions

Exercises

- after each lecture an exercise sheet is handed out
- work on these exercises alone, except if stated otherwise explicitly
- exercise sheet contains due date
  - usually 10 days time to work on it
  - hand in during practical sessions
  - lecture Monday $\longrightarrow$ hand in at latest in next week's Friday session
- main purpose: understanding ITP and learn how to use HOL
  - no detailed grading, just pass/fail
  - retries possible till pass
  - if stuck, ask me or one another
  - practical sessions intend to provide this opportunity

Practical Sessions

- very informal
- main purpose: work on exercises
  - I have a look and provide feedback
  - you can ask questions
  - I might sometimes explain things not covered in the lectures
  - I might provide some concrete tips and tricks
  - you can also discuss with each other
- attendance not required, but highly recommended
  - exception: session on 21st April
- only requirement: turn up long enough to hand in exercises
- **you need to bring your own computer**

Handing-in Exercises

- exercises are intended to be handed-in during practical sessions
- attend at least one practical session each week
- leave reasonable time to discuss exercises
  - don't try to hand your solution in Friday 14:55
- retries possible, but reasonable attempt before deadline required
- handing-in outside practical sessions
  - only if you have a good reason
  - decided on a case-by-case basis
- electronic hand-ins
  - only to get detailed feedback
  - does not replace personal hand-in
  - exceptions on a case-by-case basis if there is a good reason
  - I recommend using a KTH GitHub repo

Passing the ITP Course

- there is only a pass/fail mark
- to pass you need to
  - attend at least 7 of the 9 lectures
  - pass 8 of the 9 exercises

Communication

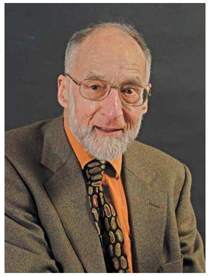
- we have the advantage of being a small group
- therefore we are flexible
- so please ask questions, even during lectures
- there are many shy people, therefore
  - anonymous checklist after each lecture
  - anonymous background questionnaire in first practical session
- further information is posted on **Interactive Theorem Proving Course** group on Group Web
- contact me (Thomas Tuerk) directly, e. g. via email `thomas@kth.se`

Part III

HOL 4 History and Architecture

LCF - Logic of Computable Functions

- **Standford LCF** 1971-72 by Milner et al.
- formalism devised by Dana Scott in 1969
- intended to reason about recursively defined functions
- intended for computer science applications
- strengths
  - powerful simplification mechanism
  - support for backward proof
- limitations
  - proofs need a lot of memory
  - fixed, hard-coded set of proof commands



Robin Milner
(1934 - 2010)

LCF - Logic of Computable Functions II

- Milner worked on improving LCF in Edinburgh
- research assistants
  - Lockwood Morris
  - Malcolm Newey
  - Chris Wadsworth
  - Mike Gordon
- **Edinburgh LCF** 1979
- introduction of **Meta Language** (ML)
- ML was invented to write proof procedures
- ML become an influential functional programming language
- using ML allowed implementing the **LCF approach**

LCF Approach

- implement an abstract datatype **thm** to represent theorems
- semantics of ML ensure that values of type thm can only be created using its interface
- interface is very small
  - predefined theorems are axioms
  - function with result type theorem are inferences
- $\Longrightarrow$ However you create a theorem, it is valid.
- together with similar abstract datatypes for types and terms, this forms the **kernel**

LCF Approach II

Modus Ponens Example

Inference Rule

$$\frac{\Gamma \vdash a \Rightarrow b \qquad \Delta \vdash a}{\Gamma \cup \Delta \vdash b}$$

SML function

```
val MP : thm -> thm -> thm
```

$\mathtt{MP}(\Gamma \vdash a \Rightarrow b)(\Delta \vdash a) = (\Gamma \cup \Delta \vdash b)$

- very trustworthy — only the small kernel needs to be trusted
- efficient — no need to store proofs

Easy to extend and automate

However complicated and potentially buggy your code is, if a value of type theorem is produced, it has been created through the small trusted interface. Therefore the statement really holds.

LCF Style Systems

There are now many interactive theorem provers out there that use an approach similar to that of Edinburgh LCF.

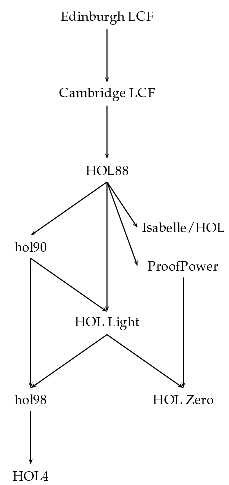
- HOL family
  - HOL theorem prover
  - HOL Light
  - HOL Zero
  - Proof Power
  - ...
- Isabelle
- Nuprl
- Coq
- ...

History of HOL

- 1979 Edinburgh LCF by Milner, Gordon, et al.
- 1981 Mike Gordon becomes lecturer in Cambridge
- 1985 Cambridge LCF
  - Larry Paulson and Gèrard Huet
  - implementation of ML compiler
  - powerful simplifier
  - various improvements and extensions
- 1988 HOL
  - Mike Gordon and Keith Hanna
  - adaption of Cambridge LCF to classical higher order logic
  - intention: hardware verification
- 1990 HOL90
  reimplementation in SML by Konrad Slind at University of Calgary
- 1998 HOL98
  implementation in Moscow ML and new library and theory mechanism
- since then HOL Kananaskis releases, called informally **HOL 4**

Family of HOL

- **ProofPower**
  commercial version of HOL88 by Roger Jones, Rob Arthan et al.
- **HOL Light**
  lean CAML / OCaml port by John Harrison
- **HOL Zero**
  trustworthy proof checker by Mark Adams
- **Isabelle**
  - 1990 by Larry Paulson
  - meta-theorem prover that supports multiple logics
  - however, mainly HOL used, ZF a little
  - nowadays probably the most widely used HOL system
  - originally designed for software verification



Part IV

HOL's Logic

HOL Logic

- the HOL theorem prover uses a version of classical **h**igher **o**rder **l**ogic:
  classical higher order predicate calculus with
  terms from the typed lambda calculus (i. e. simple type theory)
- this sounds complicated, but is intuitive for SML programmers
- (S)ML and HOL logic designed to fit each other
- if you understand SML, you understand HOL logic

HOL = functional programming + logic

Ambiguity Warning

The acronym *HOL* refers to both the *HOL interactive theorem prover* and the *HOL logic* used by it. It's also a common abbreviation for *higher order logic* in general.

Types

- SML datatype for types
  - **Type Variables** (`'a`, α, `'b`, β, ...)
    Type variables are implicitly universally quantified. Theorems containing type variables hold for all instantiations of these. Proofs using type variables can be seen as proof schemata.
  - **Atomic Types** (`c`)
    Atomic types denote fixed types. Examples: `num`, `bool`, `unit`
  - **Compound Types** ($(\sigma_1, \ldots, \sigma_n) op$)
    op is a **type operator** of arity *n* and $\sigma_1, \ldots, \sigma_n$ **argument types**.
    Type operators denote operations for constructing types.
    Examples: `num list` or `'a # 'b`.
  - **Function Types** ($\sigma_1 \to \sigma_2$)
    $\sigma_1 \to \sigma_2$ is the type of **total** functions from σ_1 to σ_2.
- types are never empty in HOL, i. e.
  for each type at least one value exists
- all HOL functions are total

Terms

- SML datatype for terms
  - **Variables** ($\mathtt{x}, \mathtt{y}, \ldots$)
  - **Constants** ($\mathtt{c}, \ldots$)
  - **Function Application** (`f a`)
  - **Lambda Abstraction** (`\x. f x` or $\lambda x.\ fx$)
    Lambda abstraction represents anonymous function definition.
    The corresponding SML syntax is `fn x => f x`.
- terms have to be well-typed
- same typing rules and same type-inference as in SML take place
- terms very similar to SML expressions
- notice: predicates are functions with return type `bool`, i. e. no distinction between functions and predicates, terms and formulae

Terms II

HOL term	**SML expression**	**type HOL / SML**
`0`	`0`	`num` / `int`
`x:'a`	`x:'a`	variable of type `'a`
`x:bool`	`x:bool`	variable of type `bool`
`x + 5`	`x + 5`	applying function `+` to `x` and `5`
`\x. x + 5`	`fn x => x + 5`	anonymous (a. k. a. inline) function of type `num -> num`
`(5, T)`	`(5, true)`	`num # bool` / `int * bool`
`[5;3;2]++[6]`	`[5,3,2]@[6]`	`num list` / `int list`

Free and Bound Variables / Alpha Equivalence

- in SML, the names of function arguments does not matter (much)
- similarly in HOL, the names of variables used by lambda-abstractions does not matter (much)
- the lambda-expression $\lambda x.\ t$ is said to **bind** the variables x in term t
- variables that are guarded by a lambda expression are called **bound**
- all other variables are **free**
- Example: x is free and y is bound in $(x = 5) \wedge (\lambda y.\ (y < x))\ 3$
- the names of bound variables are unimportant semantically
- two terms are called **alpha-equivalent** iff they differ only in the names of bound variables
- Example: $\lambda x.\ x$ and $\lambda y.\ y$ are alpha-equivalent
- Example: x and y are not alpha-equivalent

Theorems

- theorems are of the form $\Gamma \vdash p$ where
  - Γ is a set of hypothesis
  - p is the conclusion of the theorem
  - all elements of Γ and p are formulae, i. e. terms of type `bool`
- $\Gamma \vdash p$ records that using Γ the statement p **has been** proved
- notice difference to logic: there it means **can be** proved
- the proof itself is not recorded
- theorems can only be created through a small interface in the **kernel**

HOL Light Kernel

- the HOL kernel is hard to explain
  - for historic reasons some concepts are represented rather complicated
  - for speed reasons some derivable concepts have been added
- instead consider the HOL Light kernel, which is a cleaned-up version
- there are two predefined constants
  - `= : 'a -> 'a -> bool`
  - `@ : ('a -> bool) -> 'a`
- there are two predefined types
  - `bool`
  - `ind`
- the meaning of these types and constants is given by inference rules and axioms

HOL Light Inferences I

$$\frac{}{\vdash t = t}\ \text{REFL}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \text{TRANS}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash u = v \quad \textit{types fit}}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \text{COMB}$$

$$\frac{\Gamma \vdash s = t \quad x \textit{ not free in } \Gamma}{\Gamma \vdash \lambda x.\ s = \lambda x.\ t}\ \text{ABS}$$

$$\frac{}{\vdash (\lambda x.\ t)x = t}\ \text{BETA}$$

$$\frac{}{\{p\} \vdash p}\ \text{ASSUME}$$

HOL Light Inferences II

$$\frac{\Gamma \vdash p \Leftrightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \text{EQ\_MP}$$

$$\frac{\Gamma \vdash p \qquad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p \Leftrightarrow q}\ \text{DEDUCT\_ANTISYM\_RULE}$$

$$\frac{\Gamma[x_1, \dots, x_n] \vdash p[x_1, \dots, x_n]}{\Gamma[t_1, \dots, t_n] \vdash p[t_1, \dots, t_n]}\ \text{INST}$$

$$\frac{\Gamma[\alpha_1, \dots, \alpha_n] \vdash p[\alpha_1, \dots, \alpha_n]}{\Gamma[\gamma_1, \dots, \gamma_n] \vdash p[\gamma_1, \dots, \gamma_n]}\ \text{INST\_TYPE}$$

HOL Light Axioms and Definition Principles

- 3 axioms needed

ETA_AX	$\mid - (\lambda x.\ t\ x) = t$
SELECT_AX	$\mid - P\ x \Longrightarrow P((@)P))$
INFINITY_AX	predefined type `ind` is infinite

- definition principle for constants
  - constants can be introduced as abbreviations
  - constraint: no free vars and no new type vars
- definition principle for types
  - new types can be defined as non-empty subtypes of existing types
- both principles
  - lead to conservative extensions
  - preserve consistency

HOL Light derived concepts

Everything else is derived from this small kernel.

$$\begin{array}{lll}
T & =_{def} & (\lambda p.\ p) = (\lambda p.\ p) \\
\wedge & =_{def} & \lambda p\, q.\ (\lambda f.\ f\ p\ q) = (\lambda f.\ f\ T\ T) \\
\Longrightarrow & =_{def} & \lambda p\, q.\ (p \wedge q \Leftrightarrow p) \\
\forall & =_{def} & \lambda P.\ (P = \lambda x.\ T) \\
\exists & =_{def} & \lambda P.\ (\forall q.\ (\forall x.\ P(x) \Longrightarrow q) \Longrightarrow q) \\
\dots & &
\end{array}$$

Multiple Kernels

- Kernel defines abstract datatypes for types, terms and theorems
- one does not need to look at the internal implementation
- therefore, easy to exchange
- there are at least 3 different kernels for HOL
  - standard kernel (de Bruijn indices)
  - experimental kernel (name / type pairs)
  - OpenTheory kernel (for proof recording)

HOL Logic Summary

- HOL theorem prover uses classical higher order logic
- HOL logic is very similar to SML
  - syntax
  - type system
  - type inference
- HOL theorem prover very trustworthy because of LCF approach
  - there is a small kernel
  - proofs are not stored explicitly
- you don't need to know the details of the kernel
- usually one works at a much higher level of abstraction

Part V

Basic HOL Usage

HOL Technical Usage Issues

- practical issues are discussed in practical sessions
  - how to install HOL
  - which key-combinations to use in emacs-mode
  - detailed signature of libraries and theories
  - all parameters and options of certain tools
  - ...
- exercise sheets sometimes
  - ask to read some documentation
  - provide examples
  - list references where to get additional information
- if you have problems, ask me outside lecture (tuerk@kth.se)
- covered only very briefly in lectures

Installing HOL

- webpage: `https://hol-theorem-prover.org`
- HOL supports two SML implementations
  - Moscow ML (`http://mosml.org`)
  - **PolyML** (`http://www.polyml.org`)
- I recommend using PolyML
- please use emacs with
  - hol-mode
  - sml-mode
  - hol-unicode, if you want to type Unicode
- please install recent revision from git repo or Kananaskis 11 release
- documentation found on HOL webpage and with sources

General Architecture

- HOL is a collection of SML modules
- starting HOL starts a SML Read-Eval-Print-Loop (REPL) with
  - some HOL modules loaded
  - some default modules opened
  - an input wrapper to help parsing terms called `unquote`
- `unquote` provides special quotes for terms and types
  - implemented as input filter
  - ``` ``my-term`` ``` becomes `Parse.Term [QUOTE "my-term"]`
  - ``` ``:my-type`` ``` becomes `Parse.Type [QUOTE ":my-type"]`
- main interfaces
  - **emacs** (used in the course)
  - vim
  - bare shell

Filenames

- `*Script.sml` — HOL proof script file
  - script files contain definitions and proof scripts
  - executing them results in HOL searching and checking proofs
  - this might take very long
  - resulting theorems are stored in `*Theory.{sml|sig}` files
- `*Theory.{sml|sig}` — HOL theory
  - auto-generated by corresponding script file
  - load quickly, because they don't search/check proofs
  - do not edit theory files
- `*Syntax.{sml|sig}` — syntax libraries
  - contain syntax related functions
  - i. e. functions to construct and destruct terms and types
- `*Lib.{sml|sig}` — general libraries
- `*Simps.{sml|sig}` — simplifications
- `selftest.sml` — selftest for current directory

Directory Structure

- `bin` — HOL binaries
- `src` — HOL sources
- `examples` — HOL examples
  - interesting projects by various people
  - examples owned by their developer
  - coding style and level of maintenance differ a lot
- `help` — sources for reference manual
  - after compilation home of reference HTML page
- `Manual` — HOL manuals
  - Tutorial
  - Description
  - Reference (PDF version)
  - Interaction
  - Quick (cheat pages)
  - Style-guide
  - ...

Unicode

- HOL supports both Unicode and pure ASCII input and output
- advantages of Unicode compared to ASCII
  - easier to read (good fonts provided)
  - no need to learn special ASCII syntax
- disadvanges of Unicode compared to ASCII
  - harder to type (even with `hol-unicode.el`)
  - less portable between systems
- whether you like Unicode is highly a matter of personal taste
- HOL's policy
  - no Unicode in HOL's source directory `src`
  - Unicode in examples directory `examples` is fine
- I recommend turning Unicode output off initially
  - this simplifies learning the ASCII syntax
  - no need for special fonts
  - it is easier to copy and paste terms from HOL's output

Where to find help?

- reference manual
  - available as HTML pages, single PDF file and in-system help
- description manual
- Style-guide (still under development)
- HOL webpage (`https://hol-theorem-prover.org`)
- mailing-list `hol-info`
- `DB.match` and `DB.find`
- `*Theory.sig` and `selftest.sml` files
- ask someone, e. g. me :-) (tuerk@kth.se)

Part VI

Forward Proofs

Kernel too detailed

- we already discussed the HOL Logic
- the kernel itself does not even contain basic logic operators
- usually one uses a much higher level of abstraction
  - many operations and datatypes are defined
  - high-level derived inference rules are used
- let's now look at this more common abstraction level

Common Terms and Types

	Unicode	ASCII
type vars	α, β, ...	'a, 'b, ...
type annotated term	`term:type`	`term:type`
true	`T`	`T`
false	`F`	`F`
negation	`¬b`	`~b`
conjunction	`b1 ∧ b2`	`b1 /\ b2`
disjunction	`b1 ∨ b2`	`b1 \/ b2`
implication	`b1 ⟹ b2`	`b1 ==> b2`
equivalence	`b1 ⟺ b2`	`b1 <=> b2`
disequation	`v1 ≠ v2`	`v1 <> v2`
all-quantification	`∀x. P x`	`!x. P x`
existential quantification	`∃x. P x`	`?x. P x`
Hilbert's choice operator	`@x. P x`	`@x. P x`

There are similar restrictions to constant and variable names as in SML.
HOL specific: don't start variable names with an underscore

Syntax conventions

- common function syntax
  - prefix notation, e. g. `SUC x`
  - infix notation, e. g. `x + y`
  - quantifier notation, e. g. `∀x. P x` means `(∀) (λx. P x)`
- infix and quantifier notation functions can turned into prefix notation
  Example: `(+) x y` and `$+ x y` are the same as `x + y`
- quantifiers of the same type don't need to be repeated
  Example: `∀x y. P x y` is short for `∀x. ∀y. P x y`
- there is special syntax for some functions
  Example: `if c then v1 else v2` is nice syntax for `COND c v1 v2`
- associative infix operators are usually right-associative
  Example: `b1 /\ b2 /\ b3` is parsed as `b1 /\ (b2 /\ b3)`

Operator Precedence

It is easy to misjudge the binding strength of certain operators. Therefore use plenty of parenthesis.

Creating Terms

Term Parser

Use special quotation provided by `unquote`.

Use Syntax Functions

Terms are just SML values of type `term`. You can use syntax functions (usually defined in `*Syntax.sml` files) to create them.

Creating Terms II

Parser	Syntax Funs	
`` ``:bool`` ```	`mk_type ("bool", [])` or `bool`	type of Booleans
`` ``T`` ```	`mk_const ("T", bool)` or `T`	term true
`` ``~b`` ```	`mk_neg (mk_var ("b", bool))`	negation of Boolean var b
`` ``... /\ ...`` ```	`mk_conj (..., ...)`	conjunction
`` ``... \/ ...`` ```	`mk_disj (..., ...)`	disjunction
`` ``... ==> ...`` ```	`mk_imp (..., ...)`	implication
`` ``... = ...`` ```	`mk_eq (..., ...)`	equation
`` ``... <=> ...`` ```	`mk_eq (..., ...)`	equivalence
`` ``... <> ...`` ```	`mk_neg (mk_eq (..., ...))`	negated equation

Inference Rules for Equality

$$\frac{}{\vdash t = t}\ \text{REFL}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash t = s}\ \text{GSYM}$$

$$\frac{\begin{array}{c}\Gamma \vdash s = t \\ x \textit{ not free in } \Gamma\end{array}}{\Gamma \vdash \lambda x.\ s = \lambda x.t}\ \text{ABS}$$

$$\frac{\begin{array}{c}\Gamma \vdash s = t \\ \Delta \vdash t = u\end{array}}{\Gamma \cup \Delta \vdash s = u}\ \text{TRANS}$$

$$\frac{\Gamma \vdash p \Leftrightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \text{EQ\_MP}$$

$$\frac{\begin{array}{c}\Gamma \vdash s = t \\ \Delta \vdash u = v \\ \textit{types fit}\end{array}}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \text{MK\_COMB}$$

$$\frac{}{\vdash (\lambda x.\ t)x = t}\ \text{BETA}$$

Inference Rules for free Variables

$$\frac{\Gamma[x_1, \ldots, x_n] \vdash p[x_1, \ldots, x_n]}{\Gamma[t_1, \ldots, t_n] \vdash p[t_1, \ldots, t_n]}\ \text{INST}$$

$$\frac{\Gamma[\alpha_1, \ldots, \alpha_n] \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma[\gamma_1, \ldots, \gamma_n] \vdash p[\gamma_1, \ldots, \gamma_n]}\ \text{INST\_TYPE}$$

Inference Rules for Implication

$$\frac{\begin{array}{c}\Gamma \vdash p \Longrightarrow q \\ \Delta \vdash p\end{array}}{\Gamma \cup \Delta \vdash q}\ \text{MP, MATCH\_MP}$$

$$\frac{\Gamma \vdash p = q}{\begin{array}{c}\Gamma \vdash p \Longrightarrow q \\ \Gamma \vdash q \Longrightarrow p\end{array}}\ \text{EQ\_IMP\_RULE}$$

$$\frac{\begin{array}{c}\Gamma \vdash p \Longrightarrow q \\ \Delta \vdash q \Longrightarrow p\end{array}}{\Gamma \cup \Delta \vdash p = q}\ \text{IMP\_ANTISYM\_RULE}$$

$$\frac{\begin{array}{c}\Gamma \vdash p \Longrightarrow q \\ \Delta \vdash q \Longrightarrow r\end{array}}{\Gamma \cup \Delta \vdash p \Longrightarrow r}\ \text{IMP\_TRANS}$$

$$\frac{\Gamma \vdash p}{\Gamma - \{q\} \vdash q \Longrightarrow p}\ \text{DISCH}$$

$$\frac{\Gamma \vdash q \Longrightarrow p}{\Gamma \cup \{q\} \vdash p}\ \text{UNDISCH}$$

$$\frac{\Gamma \vdash p \Longrightarrow \mathsf{F}}{\Gamma \vdash {\sim}p}\ \text{NOT\_INTRO}$$

$$\frac{\Gamma \vdash {\sim}p}{\Gamma \vdash p \Longrightarrow \mathsf{F}}\ \text{NOT\_ELIM}$$

Inference Rules for Conjunction / Disjunction

$$\frac{\Gamma \vdash p \qquad \Delta \vdash q}{\Gamma \cup \Delta \vdash p\ \wedge\ q}\ \text{CONJ}$$

$$\frac{\Gamma \vdash p\ \wedge\ q}{\Gamma \vdash p}\ \text{CONJUNCT1}$$

$$\frac{\Gamma \vdash p\ \wedge\ q}{\Gamma \vdash q}\ \text{CONJUNCT2}$$

$$\frac{\Gamma \vdash p}{\Gamma \vdash p\ \vee\ q}\ \text{DISJ1}$$

$$\frac{\Gamma \vdash q}{\Gamma \vdash p\ \vee\ q}\ \text{DISJ2}$$

$$\frac{\begin{array}{c}\Gamma \vdash p \vee q \\ \Delta_1 \cup \{p\} \vdash r \\ \Delta_2 \cup \{q\} \vdash r\end{array}}{\Gamma \cup \Delta_1 \cup \Delta_2 \vdash r}\ \text{DISJ\_CASES}$$

Inference Rules for Quantifiers

$$\frac{\Gamma \vdash p \qquad x \text{ not free in } \Gamma}{\Gamma \vdash \forall x.\ p}\ \mathrm{GEN}$$

$$\frac{\Gamma \vdash \forall x.\ p}{\Gamma \vdash p[u/x]}\ \mathrm{SPEC}$$

$$\frac{\Gamma \vdash p[u/x]}{\Gamma \vdash \exists x.\ p}\ \mathrm{EXISTS}$$

$$\frac{\begin{array}{c}\Gamma \vdash \exists x.\ p \\ \Delta \cup \{p[u/x]\} \vdash r \\ u \text{ not free in } \Gamma, \Delta, p \text{ and } r\end{array}}{\Gamma \cup \Delta \vdash r}\ \mathrm{CHOOSE}$$

Forward Proofs

- axioms and inference rules are used to derive theorems
- this method is called **forward proof**
  - one starts with basic building blocks
  - one moves step by step forward
  - finally the theorem one is interested in is derived
- one can also implement own proof tools

Forward Proofs — Example I

Let's prove $\forall p.\ p \Longrightarrow p$.

```
val IMP_REFL_THM = let
  val tm1 = ``p:bool``;          > val tm1 = ``p``: term
  val thm1 = ASSUME tm1;         > val thm1 = [p] |- p: thm
  val thm2 = DISCH tm1 thm1;     > val thm2 = |- p ==> p: thm
in
  GEN tm1 thm2                   > val IMP_REFL_THM =
                                     |- !p. p ==> p: thm
end

fun IMP_REFL t =                 > val IMP_REFL =
  SPEC t IMP_REFL_THM;               fn: term -> thm
```


Forward Proofs — Example II

Let's prove $\forall P\ v.\ (\exists x.\ (x = v) \land P\ x) \Longleftrightarrow P\ v$.

```
val tm_v = ``v:'a``;
val tm_P = ``P:'a -> bool``;
val tm_lhs = ``?x. (x = v) /\ P x``
val tm_rhs = mk_comb (tm_P, tm_v);

val thm1 = let
  val thm1a = ASSUME tm_rhs;             > val thm1a = [P v] |- P v: thm
  val thm1b =                            > val thm1b =
    CONJ (REFL tm_v) thm1a;                  [P v] |- (v = v) /\ P v: thm
  val thm1c =                            > val thm1c =
    EXISTS (tm_lhs, tm_v) thm1b              [P v] |- ?x. (x = v) /\ P x
in
  DISCH tm_rhs thm1c                     > val thm1 = [] |-
end                                          P v ==> ?x. (x = v) /\ P x: thm
```

Forward Proofs — Example II cont.

```
val thm2 = let
  val thm2a =                          > val thm2a = [(u = v) /\ P u] |-
    ASSUME ``(u:'a = v) /\ P u``           (u = v) /\ P u: thm
  val thm2b = AP_TERM tm_P             > val thm2b = [(u = v) /\ P u] |-
    (CONJUNCT1 thm2a);                     P u <=> P v
  val thm2c = EQ_MP thm2b              > val thm2c = [(u = v) /\ P u] |-
    (CONJUNCT2 thm2a);                     P v
  val thm2d =                          > val thm2d = [?x. (x = v) /\ P x] |-
    CHOOSE (``u:'a``,                      P v
      ASSUME tm_lhs) thm2c
in
  DISCH tm_lhs thm2d                   > val thm2 = [] |-
end                                        ?x. (x = v) /\ P x ==> P v


val thm3 = IMP_ANTISYM_RULE thm2 thm1  > val thm3 = [] |-
                                           ?x. (x = v) /\ P x <=> P v
val thm4 = GENL [tm_P, tm_v] thm3      > val thm4 = [] |- !P v.
                                           ?x. (x = v) /\ P x <=> P v
```


Part VII

Backward Proofs

Motivation I

- let's prove `!A B. A /\ B <=> B /\ A`

```
(* Show |- A /\ B ==> B /\ A *)
val thm1a = ASSUME ``A /\ B``;
val thm1b = CONJ (CONJUNCT2 thm1a) (CONJUNCT1 thm1a);
val thm1  = DISCH ``A /\ B`` thm1b

(* Show |- B /\ A ==> A /\ B *)
val thm2a = ASSUME ``B /\ A``;
val thm2b = CONJ (CONJUNCT2 thm2a) (CONJUNCT1 thm2a);
val thm2  = DISCH ``B /\ A`` thm2b

(* Combine to get |- A /\ B <=> B /\ A *)
val thm3  = IMP_ANTISYM_RULE thm1 thm2

(* Add quantifiers *)
val thm4  = GENL [``A:bool``, ``B:bool``] thm3
```

- this is how you write down a proof
- for finding a proof it is however often useful to think **backwards**

Motivation II - thinking backwards

- we want to prove
  - `!A B. A /\ B <=> B /\ A`
- all-quantifiers can easily be added later, so let's get rid of them
  - `A /\ B <=> B /\ A`
- now we have an equivalence, let's show 2 implications
  - `A /\ B ==> B /\ A`
  - `B /\ A ==> A /\ B`
- we have an implication, so we can use the precondition as an assumption
  - using `A /\ B` show `B /\ A`
  - `A /\ B ==> B /\ A`

Motivation III - thinking backwards

- we have a conjunction as assumption, let's split it
  - using `A` and `B` show `B /\ A`
  - `A /\ B ==> B /\ A`
- we have to show a conjunction, so let's show both parts
  - using `A` and `B` show `B`
  - using `A` and `B` show `A`
  - `A /\ B ==> B /\ A`
- the first two proof obligations are trivial
  - `A /\ B ==> B /\ A`
- ...
- we are done

Motivation IV

- common practise
  - think backwards to find proof
  - write found proof down in forward style
- often switch between backward and forward style within a proof
  Example: induction proof
  - backward step: induct on ...
  - forward steps: prove base case and induction case
- whether to use forward or backward proofs depend on
  - support by the interactive theorem prover you use
    - HOL 4 and close family: emphasis on backward proof
    - Isabelle/HOL: emphasis on forward proof
    - Coq : emphasis on backward proof
  - your way of thinking
  - the theorem you try to prove

HOL Implementation of Backward Proofs

- in HOL
  - proof tactics / backward proofs used for most user-level proofs
  - forward proofs used usually for writing automation
- backward proofs are implemented by **tactics** in HOL
  - decomposition into subgoals implemented in SML
  - SML datastructures used to keep track of all open subgoals
  - forward proof used to construct theorems
- to understand backward proofs in HOL we need to look at
  - `goal` — SML datatype for proof obligations
  - `goalStack` — library for keeping track of goals
  - `tactic` — SML type for functions performing backward proofs

Goals

- goals represent proof obligations, i. e. theorems we need/want to prove
- the SML type `goal` is an abbreviation for `term list * term`
- the goal `([asm_1, ..., asm_n], c)` records that we need/want to prove the theorem `{asm_1, ..., asm_n} |- c`

Example Goals

Goal	**Theorem**
`([``A``, ``B``], ``A /\ B``)`	`{A, B} |- A /\ B`
`([``B``, ``A``], ``A /\ B``)`	`{A, B} |- A /\ B`
`([``B /\ A``], ``A /\ B``)`	`{B /\ A} |- A /\ B`
`([], ``(B /\ A) ==> (A /\ B)``)`	`|- (B /\ A) ==> (A /\ B)`

Tactics

- the SML type `tactic` is an abbreviation for the type `goal -> goal list * validation`
- `validation` is an abbreviation for `thm list -> thm`
- given a goal, a tactic
  - decides into which subgoals to decompose the goal
  - returns this list of subgoals
  - returns a validation that
    - given a list of theorems for the computed subgoals
    - produces a theorem for the original goal
- special case: empty list of subgoals
  - the validation (given `[]`) needs to produce a theorem for the goal
- notice: a tactic might be invalid

Tactic Example — `CONJ_TAC`

$$\frac{\Gamma \vdash p \qquad \Delta \vdash q}{\Gamma \cup \Delta \vdash p \ \wedge \ q}\,\mathrm{CONJ} \qquad\qquad \frac{\begin{array}{c}\texttt{t} \equiv \texttt{conj1 /\textbackslash\ conj2}\\ \texttt{asl} \vdash \texttt{conj1} \qquad \texttt{asl} \vdash \texttt{conj2}\end{array}}{\texttt{asl} \vdash \texttt{t}}$$

```
val CONJ_TAC: tactic = fn (asl, t) =>
  let
    val (conj1, conj2) = dest_conj t
  in
    ([(asl, conj1), (asl, conj2)],
     fn [th1, th2] => CONJ th1 th2 | _ => raise Match)
  end
  handle HOL_ERR _ => raise ERR "CONJ_TAC" ""
```


Tactic Example — `EQ_TAC`

$$\frac{\begin{array}{c}\Gamma \vdash p \Longrightarrow q\\ \Delta \vdash q \Longrightarrow p\end{array}}{\Gamma \cup \Delta \vdash p = q}\,\mathrm{IMP\_ANTISYM\_RULE} \qquad\qquad \frac{\begin{array}{c}\texttt{t} \equiv \texttt{lhs = rhs}\\ \texttt{asl} \vdash \texttt{lhs ==> rhs}\\ \texttt{asl} \vdash \texttt{rhs ==> lhs}\end{array}}{\texttt{asl} \vdash \texttt{t}}$$

```
val EQ_TAC: tactic = fn (asl, t) =>
  let
    val (lhs, rhs) = dest_eq t
  in
    ([(asl, mk_imp (lhs, rhs)), (asl, mk_imp (rhs, lhs))],
     fn [th1, th2] => IMP_ANTISYM_RULE th1 th2
      | _          => raise Match)
  end
  handle HOL_ERR _ => raise ERR "EQ_TAC" ""
```


proofManagerLib / goalStack

- the `proofManagerLib` keeps track of open goals
- it uses `goalStack` internally
- important commands
  - **g** — set up new goal
  - **e** — expand a tactic
  - **p** — print the current status
  - **top_thm** — get the proved thm at the end

Tactic Proof Example I

Previous Goalstack

```
-
```

User Action

```
g `!A B. A /\ B <=> B /\ A`;
```

New Goalstack

```
Initial goal:

!A B. A /\ B <=> B /\ A

: proof
```


Tactic Proof Example II

Previous Goalstack

```
Initial goal:

!A B. A /\ B <=> B /\ A

: proof
```

User Action

```
e GEN_TAC;
e GEN_TAC;
```

New Goalstack

```
A /\ B <=> B /\ A

: proof
```


Tactic Proof Example III

Previous Goalstack

```
A /\ B <=> B /\ A

: proof
```

User Action

```
e EQ_TAC;
```

New Goalstack

```
B /\ A ==> A /\ B

A /\ B ==> B /\ A

: proof
```


Tactic Proof Example IV

Previous Goalstack

```
B /\ A ==> A /\ B

A /\ B ==> B /\ A : proof
```

User Action

```
e STRIP_TAC;
```

New Goalstack

```
B /\ A
------------------------------------
  0.  A
  1.  B
```

Tactic Proof Example V

Previous Goalstack

```
B /\ A
------------------------------------
  0.  A
  1.  B
```

User Action

```
e CONJ_TAC;
```

New Goalstack

```
A
------------------------------------
  0.  A
  1.  B

B
------------------------------------
  0.  A
  1.  B
```


Tactic Proof Example VI

Previous Goalstack

```
A
------------------------------------
  0.  A
  1.  B

B
------------------------------------
  0.  A
  1.  B
```

User Action

```
e (ACCEPT_TAC (ASSUME ``B:bool``));
e (ACCEPT_TAC (ASSUME ``A:bool``));
```

New Goalstack

```
B /\ A ==> A /\ B

: proof
```


Tactic Proof Example VII

Previous Goalstack

```
B /\ A ==> A /\ B

: proof
```

User Action

```
e STRIP_TAC;
e (ASM_REWRITE_TAC[]);
```

New Goalstack

```
Initial goal proved.
|- !A B. A /\ B <=> B /\ A:
   proof
```


Tactic Proof Example VIII

Previous Goalstack

```
Initial goal proved.
|- !A B. A /\ B <=> B /\ A:
   proof
```

User Action

```
val thm = top_thm();
```

Result

```
val thm =
   |- !A B. A /\ B <=> B /\ A:
   thm
```

Tactic Proof Example IX

Combined Tactic

```
val thm = prove (``!A B. A /\ B <=> B /\ A``,
  GEN_TAC >> GEN_TAC >>
  EQ_TAC >| [
    STRIP_TAC >>
    STRIP_TAC >| [
      ACCEPT_TAC (ASSUME ``B:bool``),
      ACCEPT_TAC (ASSUME ``A:bool``)
    ],

    STRIP_TAC >>
    ASM_REWRITE_TAC[]
  ]);
```

Result

```
val thm =
   |- !A B. A /\ B <=> B /\ A:
   thm
```


Tactic Proof Example X

Cleaned-up Tactic

```
val thm = prove (``!A B. A /\ B <=> B /\ A``,
  REPEAT GEN_TAC >>
  EQ_TAC >> (
    REPEAT STRIP_TAC >>
    ASM_REWRITE_TAC []
  ));
```

Result

```
val thm =
   |- !A B. A /\ B <=> B /\ A:
   thm
```


Summary Backward Proofs

- in HOL most user-level proofs are tactic-based
  - automation often written in forward style
  - low-level, basic proofs written in forward style
  - nearly everything else is written in backward (tactic) style
- there are **many** different tactics
- in the lecture only the most basic ones will be discussed
- **you need to learn about tactics on your own**
  - good starting point: `Quick` manual
  - learning finer points takes a lot of time
  - exercises require you to read up on tactics
- often there are many ways to prove a statement, which tactics to use depends on
  - personal way of thinking
  - personal style and preferences
  - maintainability, clarity, elegance, robustness
  - ...

Part VIII

Basic Tactics

Syntax of Tactics in HOL

- originally tactics were written all in capital letters with underscores
  Example: `ALL_TAC`
- since 2010 more and more tactics have overloaded lower-case syntax
  Example: `all_tac`
- sometimes, the lower-case version is shortened
  Example: `REPEAT`, `rpt`
- sometimes, there is special syntax
  Example: `THEN`, `\\`, `>>`
- which one to use is mostly a matter of personal taste
  - all-capital names are hard to read and type
  - however, not for all tactics there are lower-case versions
  - mixed lower- and upper-case tactics are even harder to read
  - often shortened lower-case name is not *speaking*

In the lecture we will use mostly the old-style names.

Some Basic Tactics

`GEN_TAC`	remove outermost all-quantifier
`DISCH_TAC`	move antecedent of goal into assumptions
`CONJ_TAC`	splits conjunctive goal
`STRIP_TAC`	splits on outermost connective (combination of `GEN_TAC`, `CONJ_TAC`, `DISCH_TAC`, ...)
`DISJ1_TAC`	selects left disjunct
`DISJ2_TAC`	selects right disjunct
`EQ_TAC`	reduce Boolean equality to implications
`ASSUME_TAC` thm	add theorem to list of assumptions
`EXISTS_TAC` term	provide witness for existential goal

Tacticals

- tacticals are SML functions that combine tactics to form new tactics
- common workflow
  - develop large tactic interactively
  - using `goalStack` and editor support to execute tactics one by one
  - combine tactics manually with tacticals to create larger tactics
  - finally end up with one large tactic that solves your goal
  - use `prove` or `store_thm` instead of `goalStack`
- make sure to **clearly mark proof structure** by e. g.
  - use indentation
  - use parentheses
  - use appropriate connectives
  - ...
- goalStack commands like e or g should not appear in your final proof

Some Basic Tacticals

tac1 >> tac2	`THEN`, `\\`	applies tactics in sequence
tac >| tacL	`THENL`	applies list of tactics to subgoals
tac1 >- tac2	`THEN1`	applies tac2 to the first subgoal of tac1
`REPEAT` tac	`rpt`	repeats tac until it fails
`NTAC` n tac		apply tac n times
`REVERSE` tac	`reverse`	reverses the order of subgoals
tac1 `ORELSE` tac2		applies tac1 only if tac2 fails
`TRY` tac		do nothing if tac fails
`ALL_TAC`	`all_tac`	do nothing
`NO_TAC`		fail

Basic Rewrite Tactics

- (equational) rewriting is at the core of HOL's automation
- we will discuss it in detail later
- details complex, but basic usage is straightforward
  - given a theorem `rewr_thm` of form `|- P x = Q x` and a term `t`
  - rewriting `t` with `rewr_thm` means
  - replacing each occurrence of a term `P c` for some `c` with `Q c` in `t`
- **warning:** rewriting may loop
  Example: rewriting with theorem `|- X <=> (X /\ T)`

`REWRITE_TAC` thms	rewrite goal using equations found in given list of theorems
`ASM_REWRITE_TAC` thms	in addition use assumptions
`ONCE_REWRITE_TAC` thms	rewrite once in goal using equations
`ONCE_ASM_REWRITE_TAC` thms	rewrite once using assumptions

Case-Split and Induction Tactics

`Induct_on` 'term'	induct on `term`
`Induct`	induct on all-quantor
`Cases_on` 'term'	case-split on `term`
`Cases`	case-split on all-quantor
`MATCH_MP_TAC` thm	apply rule
`IRULE_TAC` thm	generalised apply rule

Assumption Tactics

`POP_ASSUM` thm-tac	use and remove first assumption common usage `POP_ASSUM MP_TAC`
`PAT_ASSUM` term thm-tac also `PAT_X_ASSUM` term thm-tac	use (and remove) first assumption matching pattern
`WEAKEN_TAC` term-pred	removes first assumption satisfying predicate

Decision Procedure Tactics

- decision procedures try to solve the current goal completely
- they either succeed of fail
- no partial progress
- decision procedures vital for automation

`TAUT_TAC`	propositional logic tautology checker
`DECIDE_TAC`	linear arithmetic for `num`
`METIS_TAC` thms	first order prover
`numLib.ARITH_TAC`	Presburger arithmetic
`intLib.ARITH_TAC`	uses Omega test

Subgoal Tactics

- it is vital to structure your proofs well
  - improved maintainability
  - improved readability
  - improved reusability
  - saves time in medium-run
- therefore, use many small lemmata
- also, use many explicit subgoals

‘term-frag‘ by tac	show term with tac and add it to assumptions
‘term-frag‘ `sufficies_by` tac	show it sufficies to prove term

Term Fragments / Term Quotations

- notice that `by` and `sufficies_by` take **term fragments**
- term fragments are also called **term quotations**
- they represent (partially) unparsed terms
- parsing takes time place during execution of tactic in context of goal
- this helps to avoid type annotations
- however, this means syntax errors show late as well
- the library **Q** defines many tactics using term fragments

Importance of Exercises

- here many tactics are presented in a very short amount of time
- there are many, many more important tactics out there
- few people can learn a programming language just by reading manuals
- similar few people can learn HOL just by reading and listening
- you should write your own proofs and play around with these tactics
- solving the exercises is highly recommended
  (and actually required if you want credits for this course)

Tactical Proof - Example I - Slide 1

- we want to prove `!l. LENGTH (APPEND l l) = 2 * LENGTH l`
- first step: set up goal on `goalStack`
- at same time start writing proof script

Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
```

Actions

- run `g ``!l. LENGTH (APPEND l l) = 2 * LENGTH l```
- this is done by hol-mode
- move cursor inside term and press `M-h g`
  (menu-entry `HOL - Goalstack - New goal`)

Tactical Proof - Example I - Slide 2

Current Goal

`!l. LENGTH (l ++ l) = 2 * LENGTH l`

- the outermost connective is an all-quantor
- let's get rid of it via `GEN_TAC`

Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (l ++ l) = 2 * LENGTH l``,
GEN_TAC
```

Actions

- run `e GEN_TAC`
- this is done by hol-mode
- mark line with `GEN_TAC` and press `M-h e`
  (menu-entry `HOL - Goalstack - Apply tactic`)

Tactical Proof - Example I - Slide 3

Current Goal

`LENGTH (l ++ l) = 2 * LENGTH l`

- `LENGTH` of `APPEND` can be simplified
- let's search an appropriate lemma with `DB.match`

Actions

- run `DB.print_match [] ``LENGTH (_ ++ _)``
- this is done via hol-mode
- press `M-h m` and enter term pattern
  (menu-entry `HOL - Misc - DB match`)
- this finds the theorem `listTheory.LENGTH_APPEND`
  `|- !l1 l2.  LENGTH (l1 ++ l2) = LENGTH l1 + LENGTH l2`

Tactical Proof - Example I - Slide 4

Current Goal

`LENGTH (l ++ l) = 2 * LENGTH l`

- let's rewrite with found theorem `listTheory.LENGTH_APPEND`

Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
GEN_TAC >>
REWRITE_TAC[listTheory.LENGTH_APPEND]
```

Actions

- connect the new tactic with tactical `>>` (`THEN`)
- use hol-mode to expand the new tactic

Tactical Proof - Example I - Slide 5

Current Goal

`LENGTH l + LENGTH l = 2 * LENGTH l`

- let's search a theorem for simplifying `2 * LENGTH l`
- prepare for extending the previous rewrite tactic

Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
GEN_TAC >>
REWRITE_TAC[listTheory.LENGTH_APPEND]
```

Actions

- `DB.match` finds theorem `arithmeticTheory.TIMES2`
- press `M-h b` and undo last tactic expansion
  (menu-entry `HOL - Goalstack - Back up`)

Tactical Proof - Example I - Slide 6

Current Goal

```
LENGTH (l ++ l) = 2 * LENGTH l
```

- extend the previous rewrite tactic
- finish proof

Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
GEN_TAC >>
REWRITE_TAC[listTheory.LENGTH_APPEND, arithmeticTheory.TIMES2]);
```

Actions

- add TIMES2 to the list of theorems used by rewrite tactic
- use hol-mode to expand the extended rewrite tactic
- goal is solved, so let's add closing parenthesis and semicolon

Tactical Proof - Example I - Slide 7

- we have a finished tactic proving our goal
- notice that `GEN_TAC` is not needed
- let's polish the proof script

Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
GEN_TAC >>
REWRITE_TAC[listTheory.LENGTH_APPEND, arithmeticTheory.TIMES2]);
```

Polished Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
REWRITE_TAC[listTheory.LENGTH_APPEND, arithmeticTheory.TIMES2]);
```


Tactical Proof - Example II - Slide 1

- let's prove something slightly more complicated
- drop old goal by pressing `M-h d` (menu-entry `HOL - Goalstack - Drop goal`)
- set up goal on `goalStack` (`M-h g`)
- at same time start writing proof script

Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (``!x1 x2 x3 l1 l2 l3.
  (MEM x1 l1 /\ MEM x2 l2 /\ MEM x3 l3) /\
  ((x1 <= x2) /\ (x2 <= x3) /\ x3 <= SUC x1) ==>
  ~(ALL_DISTINCT (l1 ++ l2 ++ l3))``,
```


Tactical Proof - Example II - Slide 2

Current Goal

```
!x1 x2 x3 l1 l2 l3.
  (MEM x1 l1 /\ MEM x2 l2 /\ MEM x3 l3) /\
  x1 <= x2 /\ x2 <= x3 /\ x3 <= SUC x1 ==>
  ~ALL_DISTINCT (l1 ++ l2 ++ l3)
```

- let's strip the goal

Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (``!x1 x2 x3 l1 l2 l3.
  (MEM x1 l1 /\ MEM x2 l2 /\ MEM x3 l3) /\
  ((x1 <= x2) /\ (x2 <= x3) /\ x3 <= SUC x1) ==>
  ~(ALL_DISTINCT (l1 ++ l2 ++ l3))``,
REPEAT STRIP_TAC
```

Tactical Proof - Example II - Slide 2

Current Goal

```
!x1 x2 x3 l1 l2 l3.
  (MEM x1 l1 /\ MEM x2 l2 /\ MEM x3 l3) /\
  x1 <= x2 /\ x2 <= x3 /\ x3 <= SUC x1 ==>
  ~ALL_DISTINCT (l1 ++ l2 ++ l3)
```

- let's strip the goal

Proof Script

```
val LENGTH_APPEND_SAME = prove (
  ``!l. LENGTH (APPEND l l) = 2 * LENGTH l``,
REPEAT STRIP_TAC
```

Actions

- add `REPEAT STRIP_TAC` to proof script
- expand this tactic using hol-mode

Tactical Proof - Example II - Slide 3

Current Goal

```
F
------------------------------------
  0.  MEM x1 l1      4.  x2 <= x3
  1.  MEM x2 l2      5.  x3 <= SUC x1
  2.  MEM x3 l3      6.  ALL_DISTINCT (l1 ++ l2 ++ l3)
  3.  x1 <= x2
```

- oops, we did too much, we would like to keep `ALL_DISTINCT` in goal

Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (``...``,
REPEAT GEN_TAC >> STRIP_TAC
```

Actions

- undo `REPEAT STRIP_TAC` (`M-h b`)
- expand more fine-tuned strip tactic

Tactical Proof - Example II - Slide 4

Current Goal

```
~ALL_DISTINCT (l1 ++ l2 ++ l3)
------------------------------------
  0.  MEM x1 l1       3.  x1 <= x2
  1.  MEM x2 l2       4.  x2 <= x3
  2.  MEM x3 l3       5.  x3 <= SUC x1
```

- now let's simplify `ALL_DISTINCT`
- search suitable theorems with `DB.match`
- use them with rewrite tactic

Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (``...``,
REPEAT GEN_TAC >> STRIP_TAC >>
REWRITE_TAC[listTheory.ALL_DISTINCT_APPEND, listTheory.MEM_APPEND]
```


Tactical Proof - Example II - Slide 5

Current Goal

```
~((ALL_DISTINCT l1 /\ ALL_DISTINCT l2 /\ !e. MEM e l1 ==> ~MEM e l2) /\
  ALL_DISTINCT l3 /\ !e. MEM e l1 \/ MEM e l2 ==> ~MEM e l3)
------------------------------------
  0.  MEM x1 l1       3.  x1 <= x2
  1.  MEM x2 l2       4.  x2 <= x3
  2.  MEM x3 l3       5.  x3 <= SUC x1
```

- from assumptions 3, 4 and 5 we know `x2 = x1 \/ x2 = x3`
- let's deduce this fact by `DECIDE_TAC`

Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (``...``,
REPEAT GEN_TAC >> STRIP_TAC >>
REWRITE_TAC[listTheory.ALL_DISTINCT_APPEND, listTheory.MEM_APPEND] >>
`(x2 = x1) \/ (x2 = x3)` by DECIDE_TAC
```

Tactical Proof - Example II - Slide 6

Current Goals — 2 subgoals, one for each disjunct

```
~((ALL_DISTINCT l1 /\ ALL_DISTINCT l2 /\ !e. MEM e l1 ==> ~MEM e l2) /\
  ALL_DISTINCT l3 /\ !e. MEM e l1 \/ MEM e l2 ==> ~MEM e l3)
------------------------------------
  0.  MEM x1 l1       4.  x2 <= x3
  1.  MEM x2 l2       5.  x3 <= SUC x1
  2.  MEM x3 l3       6a. x2 = x1
  3.  x1 <= x2        6b. x2 = x3
```

- both goals are easily solved by first-order reasoning
- let's use METIS_TAC[] for both subgoals

Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (``...``,
REPEAT GEN_TAC >> STRIP_TAC >>
REWRITE_TAC[listTheory.ALL_DISTINCT_APPEND, listTheory.MEM_APPEND] >>
`(x2 = x1) \/ (x2 = x3)` by DECIDE_TAC >> (
  METIS_TAC[]
));
```


Tactical Proof - Example II - Slide 7

Finished Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (
``!x1 x2 x3 l1 l2 l3.
  (MEM x1 l1 /\ MEM x2 l2 /\ MEM x3 l3) /\
  ((x1 <= x2) /\ (x2 <= x3) /\ x3 <= SUC x1) ==>
  ~(ALL_DISTINCT (l1 ++ l2 ++ l3))``,
REPEAT GEN_TAC >> STRIP_TAC >>
REWRITE_TAC[listTheory.ALL_DISTINCT_APPEND, listTheory.MEM_APPEND] >>
`(x2 = x1) \/ (x2 = x3)` by DECIDE_TAC >> (
  METIS_TAC[]
));
```

- notice that proof structure is explicit
- parentheses and indentation used to mark new subgoals

Part IX

Induction Proofs

Mathematical Induction

- mathematical (a. k. a. natural) induction principle:
  If a property P holds for 0 and $P(n)$ implies $P(n+1)$ for all n, then $P(n)$ holds for all n.
- HOL is expressive enough to encode this principle as a theorem.

  `|- !P. P 0 /\ (!n. P n ==> P (SUC n)) ==> !n. P n`

- Performing mathematical induction in HOL means applying this theorem (e. g. via HO_MATCH_MP_TAC)
- there are many similarish induction theorems in HOL
- Example: complete induction principle

  `|- !P. (!n. (!m. m < n ==> P m) ==> P n) ==> !n. P n`

Structural Induction Theorems

- **structural induction** theorems are an important special form of induction theorems
- they describe performing induction on the structure of a datatype
- Example: `|- !P. P [] /\ (!t. P t ==> !h. P (h::t)) ==> !l. P l`
- structural induction is used very frequently in HOL
- for each algabraic datatype, there is an induction theorem

Other Induction Theorems

- there are many induction theorems in HOL
  - datatype definitions lead to induction theorems
  - recursive function definitions produce corresponding induction theorems
  - recursive relation definitions give rise to induction theorems
  - many are manually defined
- Examples

```
|- !P. P [] /\ (!l. P l ==> !x. P (SNOC x l)) ==> !l. P l

|- !P. P FEMPTY /\
     (!f. P f ==> !x y. x NOTIN FDOM f ==> P (f |+ (x,y))) ==> !f. P f

|- !P. P {} /\
     (!s. FINITE s /\ P s ==> !e. e NOTIN s ==> P (e INSERT s)) ==>
     !s. FINITE s ==> P s

|- !R P. (!x y. R x y ==> P x y) /\ (!x y z. P x y /\ P y z ==> P x z) ==>
     !u v. R+ u v ==> P u v
```


Induction (and Case-Split) Tactics

- the tactic `Induct` (or `Induct_on`) usually used to start induction proofs
- it looks at the type of the quantifier (or its argument) and applies the default induction theorem for this type
- this is usually what one needs
- other (non default) induction theorems can be applied via `INDUCT_THEN` or `HO_MATCH_MP_TAC`
- similarish `Cases_on` picks and applies default case-split theorems

Induction Proof - Example I - Slide 1

- let's prove via induction
  `!l1 l2. REVERSE (l1 ++ l2) = REVERSE l2 ++ REVERSE l1`
- we set up the goal and start and induction proof on `l1`

Proof Script

```
val REVERSE_APPEND = prove (
``!l1 l2. REVERSE (l1 ++ l2) = REVERSE l2 ++ REVERSE l1``,
Induct
```

Induction Proof - Example I - Slide 2

- the induction tactic produced two cases
- base case:
  `!l2. REVERSE ([] ++ l2) = REVERSE l2 ++ REVERSE []`
- induction step:

```
!h l2. REVERSE (h::l1 ++ l2) = REVERSE l2 ++ REVERSE (h::l1)
-----------------------------------------------------------
!l2. REVERSE (l1 ++ l2) = REVERSE l2 ++ REVERSE l1
```

- both goals can be easily proved by rewriting

Proof Script

```
val REVERSE_APPEND = prove (``
!l1 l2. REVERSE (l1 ++ l2) = REVERSE l2 ++ REVERSE l1``,
Induct >| [
  REWRITE_TAC[REVERSE_DEF, APPEND, APPEND_NIL],
  ASM_REWRITE_TAC[REVERSE_DEF, APPEND, APPEND_ASSOC]
]);
```


Induction Proof - Example II - Slide 2

- let's prove via induction
  `!l. REVERSE (REVERSE l) = l`
- we set up the goal and start and induction proof on `l`

Proof Script

```
val REVERSE_REVERSE = prove (
``!l. REVERSE (REVERSE l) = l``,
Induct
```


Induction Proof - Example II - Slide 2

- the induction tactic produced two cases
- base case:
  `REVERSE (REVERSE []) = []`
- induction step:

```
!h. REVERSE (REVERSE (h::l1)) = h::l1
--------------------------------------------
REVERSE (REVERSE l) = l
```

- again both goals can be easily proved by rewriting

Proof Script

```
val REVERSE_REVERSE = prove (
``!l. REVERSE (REVERSE l) = l``,
Induct >| [
  REWRITE_TAC[REVERSE_DEF],
  ASM_REWRITE_TAC[REVERSE_DEF, REVERSE_APPEND, APPEND]
]);
```