

Interactive Theorem Proving (ITP) Course

Parts I - IV

Thomas Tuerk (tuerk@kth.se)

KTH

Academic Year 2016/17, Period 4

version 42672d2 of Mon Apr 24 08:54:04 2017

1 / 42

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

3 / 42

Part I

Introduction

2 / 42

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

4 / 42

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

5 / 42

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.

6 / 42

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**.



7 / 42

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

8 / 42

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

9 / 42

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.**

10 / 42

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
 - ▶ ...

11 / 42

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.**

12 / 42

Part II

Organisational Matters

13 / 42

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

15 / 42

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)

14 / 42

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 → 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

16 / 42

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**

17 / 42

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

18 / 42

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

19 / 42

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

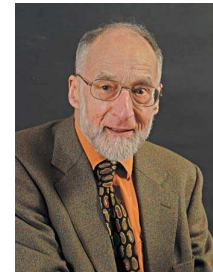
20 / 42

Part III

HOL 4 History and Architecture

LCF - Logic of Computable Functions

- **Stanford 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)

21 / 42

22 / 42

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**

23 / 42

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
- $\implies$ However you create a theorem, it is valid.
- together with similar abstract datatypes for types and terms, this forms the **kernel**

24 / 42

LCF Approach II

Modus Ponens Example

Inference Rule

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

SML function

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

25 / 42

History of HOL

- 1979 Edinburgh LCF by Milner, Gordon, et al.
- 1981 Mike Gordon becomes lecturer in Cambridge
- 1985 Cambridge LCF
 - ▶ Larry Paulson and Gérard 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**

27 / 42

LCF Style Systems

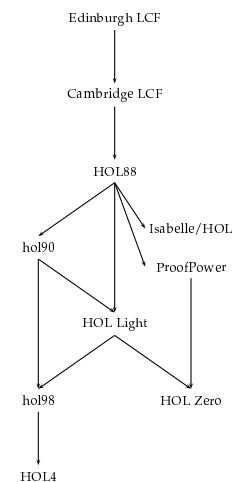
There are now many interactive theorem provers out there that use an approach similar to that of Edinburgh LCF.

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

26 / 42

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



28 / 42

Part IV

HOL's Logic

HOL Logic

- the HOL theorem prover uses a version of classical **higher order logic**: 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.

29 / 42

30 / 42

Types

- SML datatype for types
 - ▶ **Type Variables** ($'a, \alpha, 'b, \beta, \dots$)
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** $((\sigma_1, \dots, \sigma_n)op)$
 op is a **type operator** of arity n and $\sigma_1, \dots, \sigma_n$ **argument types**.
Type operators denote operations for constructing types.
Examples: `num list` or `'a # 'b`.
 - ▶ **Function Types** $(\sigma_1 \rightarrow \sigma_2)$
 $\sigma_1 \rightarrow \sigma_2$ 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

31 / 42

Terms

- SML datatype for terms
 - ▶ **Variables** ($x, y, \dots$)
 - ▶ **Constants** ($c, \dots$)
 - ▶ **Function Application** ($f\ a$)
 - ▶ **Lambda Abstraction** ($\lambda x. f\ x$ 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

32 / 42

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

33 / 42

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**

35 / 42

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

34 / 42

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
 - $= : 'a \rightarrow 'a \rightarrow \text{bool}$
 - $@ : ('a \rightarrow \text{bool}) \rightarrow 'a$
- there are two predefined types
 - bool
 - ind
- the meaning of these types and constants is given by inference rules and axioms

36 / 42

HOL Light Inferences I

$$\begin{array}{c}
 \frac{}{\vdash t = t} \text{REFL} \\
 \\
 \frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u} \text{TRANS} \\
 \\
 \frac{\Gamma \vdash s = t \quad \Delta \vdash u = v \quad \text{types fit}}{\Gamma \cup \Delta \vdash s(u) = t(v)} \text{COMB} \\
 \\
 \frac{\Gamma \vdash s = t \quad x \text{ not free in } \Gamma}{\Gamma \vdash \lambda x. s = \lambda x. t} \text{ABS} \\
 \\
 \frac{}{\vdash (\lambda x. t)_x = t} \text{BETA} \\
 \\
 \frac{}{\{p\} \vdash p} \text{ASSUME}
 \end{array}$$

37 / 42

HOL Light Inferences II

$$\begin{array}{c}
 \frac{\Gamma \vdash p \Leftrightarrow q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q} \text{EQ-MP} \\
 \\
 \frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p \Leftrightarrow q} \text{DEDUCT_ANTISYM_RULE} \\
 \\
 \frac{\Gamma[x_1, \dots, x_n] \vdash p[x_1, \dots, x_n]}{\Gamma[t_1, \dots, t_n] \vdash p[t_1, \dots, t_n]} \text{INST} \\
 \\
 \frac{\Gamma[\alpha_1, \dots, \alpha_n] \vdash p[\alpha_1, \dots, \alpha_n]}{\Gamma[\gamma_1, \dots, \gamma_n] \vdash p[\gamma_1, \dots, \gamma_n]} \text{INST_TYPE}
 \end{array}$$

38 / 42

HOL Light Axioms and Definition Principles

- 3 axioms needed
 - ETA_AX $(\lambda x. t \ x) = t$
 - SELECT_AX $P \ x \Longrightarrow P((\text{@})P)$
 - 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

39 / 42

HOL Light derived concepts

Everything else is derived from this small kernel.

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

40 / 42

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