Interactive Theorem Proving and Program Verification
Lecture 1

Pablo Buiras and Karl Palmskog

Academic Year 2019/20, Period 3–4

Based on slides by Thomas Tuerk

Part I

Introduction

Motivation

- complex systems almost certainly contain bugs
- modern society increasingly depends on complex systems.
- critical systems (e. g. , avionics) need to meet very high standards.
- 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 million lost in 45 min on stock exchange)
- ...

Interesting reads

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

Proof

- proof can demonstrate the absence of errors
- but proofs (usually) talk about a **design**, not a **real system**
- $\Rightarrow$ testing and proving can 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, convinces other mathematicians
- varying degrees of rigor
- checked by community of domain experts ("elders")
- subtle errors are hard to find
- often provide some new insight about our world
- often short, but may require creativity and brilliant ideas

Formal Proof

- encoded in a logical formalism
- sequence of low-level steps
- checkable by *stupid* machines
- trustworthy
- usually contain no new ideas or amazing insights
- often long and tedious, but have simple structure

We are interested in formal proofs in this course.

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 successes in certain domains (e. g. , SAT solving)
- 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 manual and automated proofs
  - mainly check existing proofs (e. g. HOL Zero)
  - user mainly provides lemma statements, computer searches for proofs using previous lemmas and a few hints (e. g. ACL2)
  - 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 the proof
  - keeps track of all used assumptions
  - provides automation to grind through lengthy, but trivial proof steps

Typical Interactive Proving 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 analysis
    - probability theory
  - specifications and implementations of real-world artefacts like
    - processors
    - programming languages
    - network protocols
    - compilers
  - reasoning tools

There is a strong connection to programming.
Lessons from software engineering apply.

Different Interactive Theorem Provers

- there are many different interactive provers, e. g. ,
  - Isabelle/HOL
  - Coq
  - PVS
  - HOL4
  - HOL Light
  - ACL2
  - Lean
  - ...
- important differences
  - the formalism used (set theory, simple types, dependent types, ...)
  - level of trustworthiness
  - level of automation
  - libraries
  - languages for proof automation
  - languages for specification
  - user interfaces
  - ...

The Best Interactive Theorem Prover

- there is no best interactive theorem prover
- better question: which one is the best for a certain purpose?
- important points to consider
  - existing libraries
  - formalism
  - required level of automation
  - user interfaces and build tools
  - importance of development speed vs. trustworthiness
  - familiarity
  - access to experts for help
  - your personal preferences
  - ...

In this course we use the HOL4 theorem prover.

Interactive Theorem Prover Success Stories

- CompCert verified compiler for C (Coq)
- CakeML verified compiler for ML (HOL4)
- Kepler conjecture (HOL Light)
- seL4 verified operating system kernel (Isabelle/HOL)
- Feit-Thompson theorem (Coq)
- Four color theorem (Coq)

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 HOL4
- Learn how to build a formal model
- Learn how to express and prove important properties of such a model
- 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

- Course takes place in Periods 3/4 of the academic year 2019/2020
- Location: room 4523, except for 2020-01-29, when it is in room 1537 instead.
- Wednesdays every week, 10:00 – 11:45 (lectures)

Exercises

- After each lecture an exercise sheet is handed out/uploaded
- Work on these exercises alone, except if stated otherwise explicitly
- First exercise sheet is due by 23:59, Tuesday January 29, 2020
- Main purpose: understanding ITP and learn how to use HOL4
  - no detailed grading, just pass/fail
  - retries possible till pass
  - if stuck, ask us
  - we will set up office hours
- Exercises are to be handed-in electronically (email for now)

Passing the ITP Course

- There is only a pass/fail mark
- To pass the course you need get a pass on all the exercises and a final project

Communication

- Information and material posted on course website: `https://kth-step.github.io/itppv-course/`
- Contact us directly via email:
  - Pablo Buiras, `buiras@kth.se`
  - Karl Palmskog, `palmskog@kth.se`

Part III

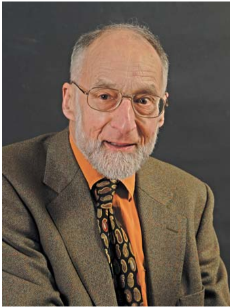
HOL4 History and Architecture

Interactive Theorem Proving Pre-history

- Aristotle, Prior Analytics (3rd century BCE)
- Leibniz, The Art of Discovery (1685)
- Frege, Grundgesetze der Arithmetik (1893)
- Russell & Whitehead, Principia Mathematica (1913)
- Gödel, Church, Turing (1930s)
- Davis & Putnam, Resolution proof system (1960)
- de Bruijn, Automath (1967)

LCF History

- **Stanford LCF** 1971-72 by Milner et al.
- LCF 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 became 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
- interface is carefully designed and checked
  - size of interface and implementation allow careful checking
  - one checks that the interface really implements only axioms and inferences that are valid in the used logic
- **However you create a theorem, there is a proof for it.**
- 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
```

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

- 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.

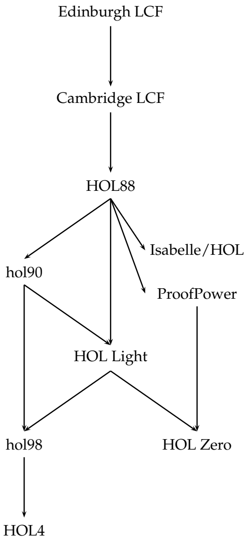
- HOL family
  - HOL theorem prover
  - HOL Light
  - HOL Zero
  - Proof Power
  - ...
- Isabelle
- Nuprl
- Coq
- Lean
- ...

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

Preliminaries and HOL's Logic

Standard ML refresher

- Standard ML (SML) is a statically-typed impure functional programming language with
  - Hindley-Milner type inference
  - parametric polymorphism
  - side-effects (mutable references, exceptions, IO, ...)
  - a module system
- Based on the simply-typed call-by-value λ-calculus
- Has a formal definition and semantics

Standard ML refresher

SML expressions

- Constants (i.e, literals): `42`, `true`, `[]`, `3.14`, `"hello"`, `...`
- Variables: `x`, `y`, `foo`, `...`
- Tuples: `(2, 4)`, `(true, 66)`, `...`
- Function application: `f A`, `f(A)`
- Function abstraction: `fn x => A`
- Sequencing: `A; B`
- Conditional expressions: `if A then B else C`
- **let**-expressions: `let <definitions> in A end`
- **case**-expressions: `case A of pattern => B | ...`
- ...

Standard ML refresher

SML definitions

- Variable binding: `val x = 42`
- Function definition: `fun add_two (x,y) = x + y`
  (equivalent to `val add_two = fn (x,y) => x + y`)
- Type definitions: `datatype colour = red | green | blue`
- ...

Example

```
fun fact 0 = 1
  | fact n = n * fact (n-1)
```

Standard ML refresher

Algebraic lists

- List expressions: `nil`, `[]`, `[1,2,3]`, `5 :: [2,4]`
- We read `[]` as "nil" and `::` as "cons"

Example

```
fun length [] = 0
  | length (x::xs) = 1 + length xs
```

Standard ML refresher

Types

- Base types: `int`, `bool`, ..., `1 : int`
- Function types: `T -> R`, `fact : int -> int`
- Tuple types: `T * R`, `(1,true) : int * bool`
- User-defined types: `colour`, `red : colour`
- ...

Algebraic type example

```
datatype 'a foo =
  constructor_1 of 'a * int
| constructor_2 of bool * ('a foo)
| constructor_3 of 'a * 'a
```

Standard ML refresher

- Constructors are values too
- `constructor_2 :  bool * ('a foo) -> 'a foo`
- What is the type of the other constructors?
- Values of type `A foo` can be deconstructed with a case expression (pattern matching):

```
case x of
  constructor_3 (x,y) => ... x and y are bound ...
| constructor_1 _ => ...
| constructor_2 (true, constructor_1 (x, 4)) => ...
| ...
```

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
- HOL is embedded in SML

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, $\ldots$)
    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
- 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

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

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

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

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

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

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

HOL Light Inferences II

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

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

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

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

HOL Light Axioms and Definition Principles

- 3 axioms needed

ETA_AX	$\vert - (\lambda x.\ t\ x) = t$
SELECT_AX	$\vert - P\ x \Longrightarrow P((@)P))$
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.

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

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