

Interactive Theorem Proving (ITP) Course

Thomas Tuerk (tuerk@kth.se)



Academic Year 2016/17, Period 4

version 60d0b38 of Mon May 29 01:06:27 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.

5 / 247

6 / 247

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 / 247

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 / 247

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

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
 - ▶ ...

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.**

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



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



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)

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

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**

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



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

17 / 247



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

19 / 247



18 / 247



20 / 247

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)

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

22 / 247





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(Γ ⊢ a ⇒ b)(Δ ⊢ a) = (Γ ∪ Δ ⊢ 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 / 247

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 / 247



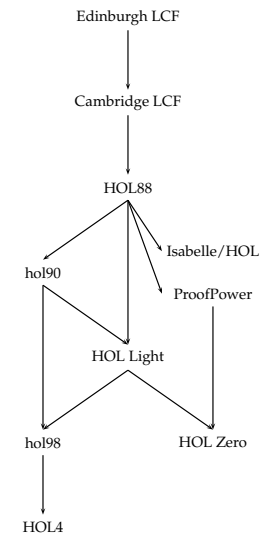
There are now many interactive theorem provers out there that use an approach similar to that of Edinburgh LCF.

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

26 / 247

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 / 247

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.

30 / 247

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 / 247

Terms



- SML datatype for terms
 - ▶ **Variables** (x, y, \dots)
 - ▶ **Constants** (c, \dots)
 - ▶ **Function Application** ($f a$)
 - ▶ **Lambda Abstraction** ($\backslash x. f x$ 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 / 247



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

- 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

33 / 247

34 / 247



- 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**

- 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

35 / 247

36 / 247

HOL Light Inferences I

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

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

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

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

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

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



HOL Light Inferences II

$$\frac{\Gamma \vdash p \Leftrightarrow q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q} \text{EQ-MP}$$

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p \Leftrightarrow q} \text{DEDUCT_ANTISYML_RULE}$$

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

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



37 / 247

38 / 247

HOL Light Axioms and Definition Principles

- 3 axioms needed

$$\begin{array}{l} \text{ETA_AX} \quad | - (\lambda x. t x) = t \\ \text{SELECT_AX} \quad | - P x \implies P((@)P) \\ \text{INFINITY_AX} \quad \text{predefined type } \text{ind} \text{ is infinite} \end{array}$$

- 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.

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



39 / 247

40 / 247

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



41 / 247

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

42 / 247



44 / 247

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

45 / 247

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

47 / 247

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

46 / 247

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
 - ▶ ...

48 / 247

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
- disadvantages 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)



49 / 247

50 / 247

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



52 / 247

Common Terms and Types

	Unicode	ASCII
type vars	α, β, \dots	'a, 'b, ...
type annotated term	term: type	term: type
true	T	T
false	F	F
negation	$\neg b$	$\sim b$
conjunction	$b1 \wedge b2$	$b1 \ \& \ b2$
disjunction	$b1 \vee b2$	$b1 \ \vee \ b2$
implication	$b1 \implies b2$	$b1 \ ==> \ b2$
equivalence	$b1 \iff b2$	$b1 \ <=> \ b2$
disequation	$v1 \neq v2$	$v1 \ <> \ v2$
all-quantification	$\forall x. P \ x$	$!x. P \ x$
existential quantification	$\exists x. P \ x$	$?x. P \ x$
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. $\forall x. P \ x$ means $(\forall) (\lambda x. P \ x)$
- infix and quantifier notation functions can be 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: $\forall x \ y. P \ x \ y$ is short for $\forall x. \forall y. P \ x \ y$
- 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.

53 / 247

54 / 247

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

```
``:bool``
``T``
``~b``
```

Syntax Funs

```
mk_type ("bool", []) or bool
mk_const ("T", bool) or T
mk_neg (
  mk_var ("b", bool))
mk_conj (... , ...)
mk_disj (... , ...)
mk_imp (... , ...)
mk_eq (... , ...)
mk_eq (... , ...)
mk_neg (mk_eq (... , ...))
```

type of Booleans
 term true
 negation of Boolean var b
 conjunction
 disjunction
 implication
 equation
 equivalence
 negated equation



55 / 247

56 / 247

Inference Rules for Equality



$$\begin{array}{c}
 \frac{}{\vdash t = t} \text{REFL} \\
 \\
 \frac{\Gamma \vdash s = t}{\Gamma \vdash \lambda x. s = \lambda x. t} \text{ABS} \\
 \\
 \frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)} \text{MK_COMB} \\
 \text{types fit} \\
 \\
 \frac{\Gamma \vdash s = t}{\Gamma \vdash t = s} \text{GSYM} \\
 \\
 \frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u} \text{TRANS} \\
 \\
 \frac{\Gamma \vdash p \Leftrightarrow q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q} \text{EQ_MP} \\
 \\
 \frac{}{\vdash (\lambda x. t)x = t} \text{BETA}
 \end{array}$$

Inference Rules for free Variables



$$\begin{array}{c}
 \frac{\Gamma[x_1, \dots, x_n] \vdash p[x_1, \dots, x_n]}{\Gamma[t_1, \dots, t_n] \vdash p[t_1, \dots, t_n]} \text{INST} \\
 \\
 \frac{\Gamma[\alpha_1, \dots, \alpha_n] \vdash p[\alpha_1, \dots, \alpha_n]}{\Gamma[\gamma_1, \dots, \gamma_n] \vdash p[\gamma_1, \dots, \gamma_n]} \text{INST_TYPE}
 \end{array}$$

57 / 247

58 / 247

Inference Rules for Implication



$$\begin{array}{c}
 \frac{\Gamma \vdash p \Rightarrow q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q} \text{MP, MATCH_MP} \\
 \\
 \frac{\Gamma \vdash p = q}{\Gamma \vdash p \Rightarrow q} \text{EQ_IMP_RULE} \\
 \Gamma \vdash q \Rightarrow p \\
 \\
 \frac{\Gamma \vdash p \Rightarrow q \quad \Delta \vdash q \Rightarrow p}{\Gamma \cup \Delta \vdash p = q} \text{IMP_ANTISYM_RULE} \\
 \\
 \frac{\Gamma \vdash p \Rightarrow q \quad \Delta \vdash q \Rightarrow r}{\Gamma \cup \Delta \vdash p \Rightarrow r} \text{IMP_TRANS} \\
 \\
 \frac{\Gamma \vdash p}{\Gamma - \{q\} \vdash q \Rightarrow p} \text{DISCH} \\
 \\
 \frac{\Gamma \vdash q \Rightarrow p}{\Gamma \cup \{q\} \vdash p} \text{UNDISCH} \\
 \\
 \frac{\Gamma \vdash p \Rightarrow F}{\Gamma \vdash \sim p} \text{NOT_INTRO} \\
 \\
 \frac{\Gamma \vdash \sim p}{\Gamma \vdash p \Rightarrow F} \text{NOT_ELIM}
 \end{array}$$

Inference Rules for Conjunction / Disjunction



$$\begin{array}{c}
 \frac{\Gamma \vdash p \quad \Delta \vdash q}{\Gamma \cup \Delta \vdash p \wedge q} \text{CONJ} \\
 \\
 \frac{\Gamma \vdash p \wedge q}{\Gamma \vdash p} \text{CONJUNCT1} \\
 \\
 \frac{\Gamma \vdash p \wedge q}{\Gamma \vdash q} \text{CONJUNCT2} \\
 \\
 \frac{\Gamma \vdash p}{\Gamma \vdash p \vee q} \text{DISJ1} \\
 \\
 \frac{\Gamma \vdash q}{\Gamma \vdash p \vee q} \text{DISJ2} \\
 \\
 \frac{\Gamma \vdash p \vee q \quad \Delta_1 \cup \{p\} \vdash r \quad \Delta_2 \cup \{q\} \vdash r}{\Gamma \cup \Delta_1 \cup \Delta_2 \vdash r} \text{DISJ_CASES}
 \end{array}$$

59 / 247

60 / 247



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

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

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

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

- 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

61 / 247

62 / 247

Forward Proofs — Example I



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

```
val IMP_REFL_THM = let
  val tm1 = 'p:bool';
  val thm1 = ASSUME tm1;
  val thm2 = DISCH tm1 thm1;
in
  GEN tm1 thm2
end

fun IMP_REFL t =
  SPEC t IMP_REFL_THM;
```

```
> val tm1 = 'p': term
> val thm1 = [p] |- p: thm
> val thm2 = |- p ==> p: thm

> val IMP_REFL_THM =
  |- !p. p ==> p: thm

> val IMP_REFL =
  fn: term -> thm
```

63 / 247

Forward Proofs — Example II



Let's prove $\forall P v. (\exists x. (x = v) \wedge P x) \iff P v$.

```
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 thm1b =
    CONJ (REFL tm_v) thm1a;
  val thm1c =
    EXISTS (tm_lhs, tm_v) thm1b
in
  DISCH tm_rhs thm1c
end

> val thm1a = [P v] |- P v: thm
> val thm1b =
  [P v] |- (v = v) /\ P v: thm
> val thm1c =
  [P v] |- ?x. (x = v) /\ P x

> val thm1 = [] |-
  P v ==> ?x. (x = v) /\ P x: thm
```

64 / 247

Forward Proofs — Example II cont.



```

val thm2 = let
  val thm2a =
    ASSUME (('u:'a = v) /\ P u) |
  val thm2b = AP_TERM tm_P
    (CONJUNCT1 thm2a);
  val thm2c = EQ_MP thm2b
    (CONJUNCT2 thm2a);
  val thm2d =
    CHOOSE (('u:'a',
      ASSUME tm_lhs) thm2c
in
  DISCH tm_lhs thm2d
end

val thm3 = IMP_ANTISYM_RULE thm2 thm1
val thm4 = GENL [tm_P, tm_v] thm3

```

```

> val thm2a = [(u = v) /\ P u] |
  (u = v) /\ P u: thm
> val thm2b = [(u = v) /\ P u] |
  P u <=> P v
> val thm2c = [(u = v) /\ P u] |
  P v
> val thm2d = [?x. (x = v) /\ P x] |
  P v

> val thm2 = [] |
  ?x. (x = v) /\ P x ==> P v

> val thm3 = [] |
  ?x. (x = v) /\ P x <=> P v
> val thm4 = [] |
  ?x. (x = v) /\ P x <=> P v

```

Part VII

Backward Proofs



65 / 247

Motivation I

- let's prove $\neg A \vee B. A \wedge B \Leftrightarrow B \wedge A$

```

(* 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**

67 / 247

Motivation II - thinking backwards



- we want to prove
 - $\neg A \vee B. A \wedge B \Leftrightarrow B \wedge A$
- all-quantifiers can easily be added later, so let's get rid of them
 - $A \wedge B \Leftrightarrow B \wedge A$
- now we have an equivalence, let's show 2 implications
 - $A \wedge B \Rightarrow B \wedge A$
 - $B \wedge A \Rightarrow A \wedge B$
- we have an implication, so we can use the precondition as an assumption
 - using $A \wedge B$ show $B \wedge A$
 - $A \wedge B \Rightarrow B \wedge A$

68 / 247

Motivation III - thinking backwards



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

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

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

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
<code>([‘‘A‘‘, ‘‘B‘‘], ‘‘A \wedge B‘‘)</code>	<code>{A, B} - A \wedge B</code>
<code>([‘‘B‘‘, ‘‘A‘‘], ‘‘A \wedge B‘‘)</code>	<code>{A, B} - A \wedge B</code>
<code>([‘‘B \wedge A‘‘], ‘‘A \wedge B‘‘)</code>	<code>{B \wedge A} - A \wedge B</code>
<code>([], ‘‘(B \wedge A) \implies (A \wedge B)‘‘)</code>	<code> - (B \wedge A) \implies (A \wedge B)</code>



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



$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{\Gamma \cup \Delta \vdash p \wedge q} \text{CONJ} \qquad \frac{t \equiv \text{conj1} \wedge \text{conj2} \quad \text{asl} \vdash \text{conj1} \quad \text{asl} \vdash \text{conj2}}{\text{asl} \vdash t}$$

```
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" ""
```

73 / 247

74 / 247

Tactic Example — EQ_TAC



proofManagerLib / goalStack



$$\frac{\Gamma \vdash p \implies q \quad \Delta \vdash q \implies p}{\Gamma \cup \Delta \vdash p = q} \text{IMP_ANTISYM_RULE} \qquad \frac{t \equiv \text{lhs} = \text{rhs} \quad \text{asl} \vdash \text{lhs} \implies \text{rhs} \quad \text{asl} \vdash \text{rhs} \implies \text{lhs}}{\text{asl} \vdash t}$$

```
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" ""
```

- 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

75 / 247

76 / 247

Tactic Proof Example I



Previous Goalstack

-

User Action

g '! $A \wedge B \Leftrightarrow B \wedge A$ ';

New Goalstack

Initial goal:

$!A \wedge B \Leftrightarrow B \wedge A$

: proof

77 / 247

Tactic Proof Example III



Previous Goalstack

$A \wedge B \Leftrightarrow B \wedge A$

: proof

User Action

e EQ_TAC;

New Goalstack

$B \wedge A \Rightarrow A \wedge B$

$A \wedge B \Rightarrow B \wedge A$

: proof

79 / 247

Tactic Proof Example II



Previous Goalstack

Initial goal:

$!A \wedge B \Leftrightarrow B \wedge A$

: proof

User Action

e GEN_TAC;

e GEN_TAC;

New Goalstack

$A \wedge B \Leftrightarrow B \wedge A$

: proof

78 / 247

Tactic Proof Example IV



Previous Goalstack

$B \wedge A \Rightarrow A \wedge B$

$A \wedge B \Rightarrow B \wedge A$: proof

User Action

e STRIP_TAC;

New Goalstack

$B \wedge A$

0. A

1. B

80 / 247

Tactic Proof Example V



Previous Goalstack

B \wedge A

-
- 0. A
 - 1. B

User Action

e CONJ_TAC;

New Goalstack

A

-
- 0. A
 - 1. B

B

-
- 0. A
 - 1. B

81 / 247

Tactic Proof Example VII



Previous Goalstack

B \wedge A \implies A \wedge B

: proof

User Action

e STRIP_TAC;
e (ASM_REWRITE_TAC[]);

New Goalstack

Initial goal proved.
|- !A B. A \wedge B \iff B \wedge A:
proof

83 / 247

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 \wedge A \implies A \wedge B

: proof

82 / 247

Tactic Proof Example VIII



Previous Goalstack

Initial goal proved.
|- !A B. A \wedge B \iff B \wedge A:
proof

User Action

val thm = top_thm();

Result

val thm =
|- !A B. A \wedge B \iff B \wedge A:
thm

84 / 247

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
```

85 / 247



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
 - ▶ ...

87 / 247

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
```

86 / 247

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

89 / 247

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

91 / 247

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	fail

90 / 247

92 / 247

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 $\vdash P\ x = Q\ x$ 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 $\vdash X \iff (X \wedge T)$

`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

93 / 247

Assumption Tactics



`POP_ASSUM` thm-tac use and remove first assumption
 common usage `POP_ASSUM MP_TAC`

`PAT_ASSUM` term thm-tac use (and remove) first
 also `PAT_X_ASSUM` term thm-tac assumption matching pattern

`WEAKEN_TAC` term-pred removes first assumption
 satisfying predicate

95 / 247

Case-Split and Induction Tactics



<code>Induct_on</code> 'term'	induct on term
<code>Induct</code>	induct on all-quantor
<code>Cases_on</code> 'term'	case-split on term
<code>Cases</code>	case-split on all-quantor
<code>MATCH_MP_TAC</code> thm	apply rule
<code>IRULE_TAC</code> thm	generalised apply rule

94 / 247

Decision Procedure Tactics



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

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

96 / 247

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

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)



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

97 / 247



Tactical Proof - Example I - Slide 1

98 / 247



- we want to prove !1. LENGTH (APPEND 1 1) = 2 * LENGTH 1
- first step: set up goal on goalStack
- at same time start writing proof script

Proof Script

```
val LENGTH_APPEND_SAME = prove (  
  “!1. LENGTH (APPEND 1 1) = 2 * LENGTH 1“ ,
```

Actions

- run g “!1. LENGTH (APPEND 1 1) = 2 * LENGTH 1“
- this is done by hol-mode
- move cursor inside term and press M-h g
(menu-entry HOL - Goalstack - New goal)

99 / 247

100 / 247

Tactical Proof - Example I - Slide 2



Current Goal

```
!1. LENGTH (1 ++ 1) = 2 * LENGTH 1
```

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

Proof Script

```
val LENGTH_APPEND_SAME = prove (  
  “!1. LENGTH (1 ++ 1) = 2 * LENGTH 1“,  
  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)

101 / 247

Tactical Proof - Example I - Slide 4



Current Goal

```
LENGTH (1 ++ 1) = 2 * LENGTH 1
```

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

Proof Script

```
val LENGTH_APPEND_SAME = prove (  
  “!1. LENGTH (APPEND 1 1) = 2 * LENGTH 1“,  
  GEN_TAC >>  
  REWRITE_TAC[listTheory.LENGTH_APPEND]
```

Actions

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

103 / 247

Tactical Proof - Example I - Slide 3



Current Goal

```
LENGTH (1 ++ 1) = 2 * LENGTH 1
```

- 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`
`|- !11 12. LENGTH (11 ++ 12) = LENGTH 11 + LENGTH 12`

102 / 247

Tactical Proof - Example I - Slide 5



Current Goal

```
LENGTH 1 + LENGTH 1 = 2 * LENGTH 1
```

- let's search a theorem for simplifying `2 * LENGTH 1`
- prepare for extending the previous rewrite tactic

Proof Script

```
val LENGTH_APPEND_SAME = prove (  
  “!1. LENGTH (APPEND 1 1) = 2 * LENGTH 1“,  
  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)

104 / 247

Tactical Proof - Example I - Slide 6



Current Goal

```
LENGTH (1 ++ 1) = 2 * LENGTH 1
```

- extend the previous rewrite tactic
- finish proof

Proof Script

```
val LENGTH_APPEND_SAME = prove (  
  ‘!1. LENGTH (APPEND 1 1) = 2 * LENGTH 1‘,  
  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

105 / 247

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 11 12 13.  
  (MEM x1 11 /\ MEM x2 12 /\ MEM x3 13) /\  
  ((x1 <= x2) /\ (x2 <= x3) /\ x3 <= SUC x1) ==>  
  ~(ALL_DISTINCT (11 ++ 12 ++ 13))‘,
```

107 / 247

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 (  
  ‘!1. LENGTH (APPEND 1 1) = 2 * LENGTH 1‘,  
  GEN_TAC >>  
  REWRITE_TAC[listTheory.LENGTH_APPEND, arithmeticTheory.TIMES2]);
```

Polished Proof Script

```
val LENGTH_APPEND_SAME = prove (  
  ‘!1. LENGTH (APPEND 1 1) = 2 * LENGTH 1‘,  
  REWRITE_TAC[listTheory.LENGTH_APPEND, arithmeticTheory.TIMES2]);
```

106 / 247

Tactical Proof - Example II - Slide 2



Current Goal

```
!x1 x2 x3 11 12 13.  
  (MEM x1 11 /\ MEM x2 12 /\ MEM x3 13) /\  
  x1 <= x2 /\ x2 <= x3 /\ x3 <= SUC x1 ==>  
  ~ALL_DISTINCT (11 ++ 12 ++ 13)
```

- let's strip the goal

Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (‘!x1 x2 x3 11 12 13.  
  (MEM x1 11 /\ MEM x2 12 /\ MEM x3 13) /\  
  ((x1 <= x2) /\ (x2 <= x3) /\ x3 <= SUC x1) ==>  
  ~(ALL_DISTINCT (11 ++ 12 ++ 13))‘,  
  REPEAT STRIP_TAC
```

108 / 247

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

109 / 247

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])
```

111 / 247

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

110 / 247

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 $x2 = x1 \vee x2 = x3$
- 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
```

112 / 247



Current Goals — 2 subgoals, one for each disjunct

```
~((ALL_DISTINCT 11 /\ ALL_DISTINCT 12 /\ !e. MEM e 11 ==> ~MEM e 12) /\
  ALL_DISTINCT 13 /\ !e. MEM e 11 \/ MEM e 12 ==> ~MEM e 13)
```

- | | |
|--------------|-----------------|
| 0. MEM x1 11 | 4. x2 <= x3 |
| 1. MEM x2 12 | 5. x3 <= SUC x1 |
| 2. MEM x3 13 | 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[]
  ));
```

Part IX

Induction Proofs



Finished Proof Script

```
val NOT_ALL_DISTINCT_LEMMA = prove (
  '!x1 x2 x3 11 12 13.
  (MEM x1 11 /\ MEM x2 12 /\ MEM x3 13) /\
  ((x1 <= x2) /\ (x2 <= x3) /\ x3 <= SUC x1) ==>
  ~(ALL_DISTINCT (11 ++ 12 ++ 13))',
  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

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.

$$\vdash !P. P\ 0 \ /\ (\!n. P\ n \ ==> P\ (SUC\ n)) \ ==> \!n. P\ n$$
- 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

$$\vdash !P. (\!n. (\!m. m < n \ ==> P\ m) \ ==> P\ n) \ ==> \!n. P\ n$$

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: $\vdash \forall P. P [] \wedge (!t. P t \implies !h. P (h::t)) \implies !l. P l$
- structural induction is used very frequently in HOL
- for each algebraic datatype, there is an induction theorem

117 / 247

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

119 / 247

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
```

118 / 247

Induction Proof - Example I - Slide 1



- let's prove via induction
 - $!l1\ l2. \text{REVERSE } (l1 ++ l2) = \text{REVERSE } l2 ++ \text{REVERSE } l1$
- 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
```

120 / 247

Induction Proof - Example I - Slide 2



- the induction tactic produced two cases
- base case:
 $!12. \text{REVERSE } ([] ++ 12) = \text{REVERSE } 12 ++ \text{REVERSE } []$
- induction step:
 $!h 12. \text{REVERSE } (h::11 ++ 12) = \text{REVERSE } 12 ++ \text{REVERSE } (h::11)$

 $!12. \text{REVERSE } (11 ++ 12) = \text{REVERSE } 12 ++ \text{REVERSE } 11$
- both goals can be easily proved by rewriting

Proof Script

```
val REVERSE_APPEND = prove (''  
!11 12. REVERSE (11 ++ 12) = REVERSE 12 ++ REVERSE 11'',  
Induct >| [  
  REWRITE_TAC[REVERSE_DEF, APPEND, APPEND_NIL],  
  ASM_REWRITE_TAC[REVERSE_DEF, APPEND, APPEND_ASSOC]  
]);
```

121 / 247

Induction Proof - Example II - Slide 2



- the induction tactic produced two cases
- base case:
 $\text{REVERSE } (\text{REVERSE } []) = []$
- induction step:
 $!h. \text{REVERSE } (\text{REVERSE } (h::11)) = h::11$

 $\text{REVERSE } (\text{REVERSE } 1) = 1$
- again both goals can be easily proved by rewriting

Proof Script

```
val REVERSE_REVERSE = prove (''  
!1. REVERSE (REVERSE 1) = 1'',  
Induct >| [  
  REWRITE_TAC[REVERSE_DEF],  
  ASM_REWRITE_TAC[REVERSE_DEF, REVERSE_APPEND, APPEND]  
]);
```

123 / 247

Induction Proof - Example II - Slide 2



- let's prove via induction
 $!1. \text{REVERSE } (\text{REVERSE } 1) = 1$
- we set up the goal and start and induction proof on 1

Proof Script

```
val REVERSE_REVERSE = prove (''  
!1. REVERSE (REVERSE 1) = 1'',  
Induct
```

122 / 247

Part X

Basic Definitions



Definitional Extensions



- there are **conservative definition principles** for types and constants
- conservative means that all theorems that can be proved in extended theory can also be proved in original one
- however, such extensions make the theory more comfortable
- definitions introduce **no new inconsistencies**
- the HOL community has a very strong tradition of a purely definitional approach

125 / 247

Oracles

- **oracles** are families of axioms
- however, they are used differently than axioms
- they are used to enable usage of external tools and knowledge
- you might want to use an external automated prover
- this external tool acts as an oracle
 - ▶ it provides answers
 - ▶ it does not explain or justify these answers
- you don't know, whether this external tool might be buggy
- all theorems proved via it are tagged with a special oracle-tag
- tags are propagated
- this allows keeping track of everything depending on the correctness of this tool

127 / 247

Axiomatic Extensions



- **axioms** are a different approach
- they allow postulating arbitrary properties, i. e. extending the logic with arbitrary theorems
- this approach might introduce new inconsistencies
- in HOL axioms are very rarely needed
- using definitions is often considered more elegant
- it is hard to keep track of axioms
- use axioms only if you really know what you are doing

126 / 247

Oracles II



- Common oracle-tags
 - ▶ `DISK_THM` — theorem was written to disk and read again
 - ▶ `HolSatLib` — proved by MiniSat
 - ▶ `HolSmtLib` — proved by external SMT solver
 - ▶ `fast_proof` — proof was skipped to compile a theory rapidly
 - ▶ `cheat` — we cheated :-)
- **cheating** via e. g. the `cheat` tactic means skipping proofs
- it can be helpful during proof development
 - ▶ test whether some lemmata allow you finishing the proof
 - ▶ skip lengthy but boring cases and focus on critical parts first
 - ▶ experiment with exact form of invariants
 - ▶ ...
- cheats should be removed reasonable quickly
- HOL warns about cheats and skipped proofs

128 / 247

Pitfalls of Definitional Approach

- definitions can't introduce new inconsistencies
- they force you to state all assumed properties at one location
- however, you still need to be careful
- Is your definition really expressing what you had in mind ?
- Does your formalisation correspond to the real world artefact ?
- How can you convince others that this is the case ?
- we will discuss methods to deal with this later in this course
 - ▶ formal sanity
 - ▶ conformance testing
 - ▶ code review
 - ▶ comments, good names, clear coding style
 - ▶ ...
- this is highly complex and needs a lot of effort in general



Specifications

- HOL allows to introduce new constants with certain properties, provided the existence of such constants has been shown

Specification of EVEN and ODD

```
> EVEN_ODD_EXISTS
val it = |- ?even odd. even 0 /\ ~odd 0 /\ (!n. even (SUC n) <=> odd n) /\
        (!n. odd (SUC n) <=> even n)

> val EO_SPEC = new_specification ("EO_SPEC", ["EVEN", "ODD"], EVEN_ODD_EXISTS);
val EO_SPEC = |- EVEN 0 /\ ~ODD 0 /\ (!n. EVEN (SUC n) <=> ODD n) /\
              (!n. ODD (SUC n) <=> EVEN n)
```

- `new_specification` is a convenience wrapper
 - ▶ it uses existential quantification instead of Hilbert's choice
 - ▶ deals with pair syntax
 - ▶ stores resulting definitions in theory
- `new_specification` captures the underlying principle nicely



129 / 247

130 / 247

Definitions

- special case: new constant defined by equality

Specification with Equality

```
> double_EXISTS
val it =
|- ?double. (!n. double n = (n + n))

> val double_def = new_specification ("double_def", ["double"], double_EXISTS);
val double_def =
|- !n. double n = n + n
```

- there is a specialised methods for such non-recursive definitions

Non Recursive Definitions

```
> val DOUBLE_DEF = new_definition ("DOUBLE_DEF", 'DOUBLE n = n + n')
val DOUBLE_DEF =
|- !n. DOUBLE n = n + n
```



Restrictions for Definitions

- all variables occurring on right-hand-side (rhs) need to be arguments
 - ▶ e.g. `new_definition (... , 'F n = n + m')` fails
 - ▶ `m` is free on rhs
- all type variables occurring on rhs need to occur on lhs
 - ▶ e.g. `new_definition ("IS_FIN_TY", 'IS_FIN_TY = FINITE (UNIV : 'a set)')` fails
 - ▶ `IS_FIN_TY` would lead to inconsistency
 - ▶ `|- FINITE (UNIV : bool set)`
 - ▶ `|- ~FINITE (UNIV : num set)`
 - ▶ `T <=> FINITE (UNIV:bool set) <=> IS_FIN_TY <=> FINITE (UNIV:num set) <=> F`
 - ▶ therefore, such definitions can't be allowed



131 / 247

132 / 247

Underspecified Functions



- function specification do not need to define the function precisely
- multiple different functions satisfying one spec are possible
- functions resulting from such specs are called **underspecified**
- underspecified functions are still total, one just lacks knowledge
- one common application: modelling **partial functions**

- ▶ functions like e. g. HD and TL are total
- ▶ they are defined for empty lists
- ▶ however, it is not specified, which value they have for empty lists
- ▶ only known: HD [] = HD [] and TL [] = TL []

```
val MY_HD_EXISTS = prove (''hd. !x xs. (hd (x::xs) = x)'', ...);
val MY_HD_SPEC =
  new_specification ("MY_HD_SPEC", ["MY_HD"], MY_HD_EXISTS)
```

133 / 247

Primitive Type Definitions - Example 1



- lets try to define a type dlist of lists containing no duplicates
- predicate ALL_DISTINCT : 'a list -> bool is used to define it
- easy to prove theorem dlist_exists: |- ?l. ALL_DISTINCT l
- val dlist_TY_DEF = new_type_definitions("dlist", dlist_exists) defines a new type 'a dlist and returns a theorem

```
|- ?(rep : 'a dlist -> 'a list).
  TYPE_DEFINITION ALL_DISTINCT rep
```

- rep is a function taking a 'a dlist to the list representing it
 - ▶ rep is injective
 - ▶ a list satisfies ALL_DISTINCT iff there is a corresponding dlist

135 / 247

Primitive Type Definitions



- HOL allows introducing non-empty subtypes of existing types
- a predicate $P : ty \rightarrow bool$ describes a subset of an existing type ty
- ty may contain type variables
- only **non-empty** types are allowed
- therefore a non-emptiness proof ex-thm of form $?e. P e$ is needed
- new_type_definition (op-name, ex-thm) then introduces a new type op-name specified by P

134 / 247

Primitive Type Definitions - Example 2



- define_new_type_bijections can be used to define bijections between old and new type

```
> define_new_type_bijections {name="dlist_tybij", ABS="abs_dlist",
  REP="rep_dlist", tyax=dlist_TY_DEF}
```

```
val it =
  |- (!a. abs_dlist (rep_dlist a) = a) /\
    (!r. ALL_DISTINCT r <=> (rep_dlist (abs_dlist r) = r))
```

- other useful theorems can be automatically proved by
 - ▶ prove_abs_fn_one_one
 - ▶ prove_abs_fn_onto
 - ▶ prove_rep_fn_one_one
 - ▶ prove_rep_fn_onto

136 / 247



- primitive definition principles are easily explained
- they lead to conservative extensions
- however, they are cumbersome to use
- LCF approach allows implementing more convenient definition tools
 - ▶ Datatype package
 - ▶ TFL (Total Functional Language) package
 - ▶ IndDef (Inductive Definition) package
 - ▶ quotientLib Quotient Types Library
 - ▶ ...

137 / 247

Functional Programming Example

```
> Datatype 'mylist = E | L 'a mylist'
val it = (): unit

> Define '(mylen E = 0) /\ (mylen (L x xs) = SUC (mylen xs))'
Definition has been stored under "mylen_def"
val it =
  |- (mylen E = 0) /\ !x xs. mylen (L x xs) = SUC (mylen xs):
  thm

> EVAL ''mylen (L 2 (L 3 (L 1 E)))''
val it =
  |- mylen (L 2 (L 3 (L 1 E))) = 3:
  thm
```

139 / 247



- the Datatype package allows to define datatypes conveniently
- the TFL package allows to define (mutually recursive) functions
- the EVAL conversion allows evaluating those definitions
- this gives many HOL developments the feeling of a functional program
- there is really a close connection between functional programming a definitions in HOL
 - ▶ functional programming design principles apply
 - ▶ EVAL is a great way to test quickly, whether your definitions are working as intended

138 / 247

Datatype Package



- the Datatype package allows to define SML style datatypes easily
- there is support for
 - ▶ algebraic datatypes
 - ▶ record types
 - ▶ mutually recursive types
 - ▶ ...
- many constants are automatically introduced
 - ▶ constructors
 - ▶ case-split constant
 - ▶ size function
 - ▶ field-update and accessor functions for records
 - ▶ ...
- many theorems are derived and stored in current theory
 - ▶ injectivity and distinctness of constructors
 - ▶ nchotomy and structural induction theorems
 - ▶ rewrites for case-split, size and record update functions
 - ▶ ...

140 / 247

Datatype Package - Example I



Tree Datatype in SML

```
datatype ('a,'b) btree = Leaf of 'a
                       | Node of ('a,'b) btree * 'b * ('a,'b) btree
```

Tree Datatype in HOL

```
Datatype 'btree = Leaf 'a
              | Node btree 'b btree'
```

Tree Datatype in HOL — Deprecated Syntax

```
Hol_datatype 'btree = Leaf of 'a
              | Node of btree => 'b => btree'
```

141 / 247

Datatype Package - Example I - Derived Theorems 2



btree_size_def

```
|- (!f f1 a. btree_size f f1 (Leaf a) = 1 + f a) /\
   (!f f1 a0 a1 a2.
    btree_size f f1 (Node a0 a1 a2) =
    1 + (btree_size f f1 a0 + (f1 a1 + btree_size f f1 a2)))
```

bbtree_case_def

```
|- (!a f f1. btree_CASE (Leaf a) f f1 = f a) /\
   (!a0 a1 a2 f f1. btree_CASE (Node a0 a1 a2) f f1 = f1 a0 a1 a2)
```

btree_case_cong

```
|- !M M' f f1.
   (M = M') /\ (!a. (M' = Leaf a) ==> (f a = f' a)) /\
   (!a0 a1 a2.
    (M' = Node a0 a1 a2) ==> (f1 a0 a1 a2 = f1' a0 a1 a2)) ==>
   (btree_CASE M f f1 = btree_CASE M' f' f1')
```

143 / 247

Datatype Package - Example I - Derived Theorems 1



btree_distinct

```
|- !a2 a1 a0 a. Leaf a <> Node a0 a1 a2
```

btree_11

```
|- (!a a'. (Leaf a = Leaf a') <=> (a = a')) /\
   (!a0 a1 a2 a0' a1' a2'.
    (Node a0 a1 a2 = Node a0' a1' a2') <=>
    (a0 = a0') /\ (a1 = a1') /\ (a2 = a2'))
```

btree_nchotomy

```
|- !bb. (?a. bb = Leaf a) \ / (?b b1 b0. bb = Node b b1 b0)
```

btree_induction

```
|- !P. (!a. P (Leaf a)) /\
   (!b b0. P b /\ P b0 ==> !b1. P (Node b b1 b0)) ==>
   !b. P b
```

142 / 247

Datatype Package - Example II



Enumeration type in SML

```
datatype my_enum = E1 | E2 | E3
```

Enumeration type in HOL

```
Datatype 'my_enum = E1 | E2 | E3'
```

144 / 247

Datatype Package - Example II - Derived Theorems



my_enum_nchotomy

```
|- !P. P E1 /\ P E2 /\ P E3 ==> !a. P a
```

my_enum_distinct

```
|- E1 <> E2 /\ E1 <> E3 /\ E2 <> E3
```

my_enum2num_thm

```
|- (my_enum2num E1 = 0) /\ (my_enum2num E2 = 1) /\ (my_enum2num E3 = 2)
```

my_enum2num_num2my_enum

```
|- !r. r < 3 <=> (my_enum2num (num2my_enum r) = r)
```

145 / 247

Datatype Package - Example III - Derived Theorems



rgb_component_equality

```
|- !r1 r2. (r1 = r2) <=>
  (r1.r = r2.r) /\ (r1.g = r2.g) /\ (r1.b = r2.b)
```

rgb_nchotomy

```
|- !rr. ?n n0 n1. rr = rgb n n0 n1
```

rgb_r_fupd

```
|- !f n n0 n1. rgb n n0 n1 with r updated_by f = rgb (f n) n0 n1
```

rgb_updates_eq_literal

```
|- !r n1 n0 n.
  r with <|r := n1; g := n0; b := n|> = <|r := n1; g := n0; b := n|>
```

147 / 247

Datatype Package - Example III



Record type in SML

```
type rgb = { r : int, g : int, b : int }
```

Record type in HOL

```
Datatype 'rgb = <| r : num; g : num; b : num |>'
```

146 / 247

Datatype Package - Example IV



- nested record types are not allowed
- however, mutual recursive types can mitigate this restriction

Filesystem Datatype in SML

```
datatype file = Text of string
  | Dir of {owner : string ,
           files : (string * file) list}
```

Not Supported Nested Record Type Example in HOL

```
Datatype 'file = Text string
  | Dir <| owner : string ;
           files : (string # file) list |>'
```

Filesystem Datatype - Mutual Recursion in HOL

```
Datatype 'file = Text string
  | Dir directory
  ;
  directory = <| owner : string ;
              files : (string # file) list |>'
```

148 / 247

Datatype Package - No support for Co-Algebraic Types



- there is no support for co-algebraic types
- the Datatype package could be extended to do so
- other systems like Isabelle/HOL provide high-level methods for defining such types

Co-algebraic Type Example in SML — Lazy Lists

```
datatype 'a lazylist = Nil
                    | Cons of ('a * (unit -> 'a lazylist))
```

149 / 247

Total Functional Language (TFL) package



- TFL package implements support for terminating functional definitions
- `Define` defines functions from high-level descriptions
- there is support for pattern matching
- look and feel is like function definitions in SML
- based on **well-founded recursion** principle
- `Define` is the most common way for definitions in HOL

151 / 247

Datatype Package - Discussion



- Datatype package allows to define many useful datatypes
- however, there are many limitations
 - ▶ some types cannot be defined in HOL, e.g. empty types
 - ▶ some types are not supported, e.g. co-algebraic types
 - ▶ there are bugs (currently e.g. some trouble with certain mutually recursive definitions)
- biggest restrictions in practice (in my opinion and my line of work)
 - ▶ no support for co-algebraic datatypes
 - ▶ no nested record datatypes
- depending on datatype, different sets of useful lemmata are derived
- most important ones are added to `TypeBase`
 - ▶ tools like `Induct_on`, `Cases_on` use them
 - ▶ there is support for pattern matching

150 / 247

Well-Founded Relations



- a relation $R : 'a \rightarrow 'a \rightarrow \text{bool}$ is called **well-founded**, iff there are no infinite descending chains
$$\text{wellfounded } R = \sim?f. !n. R (f (SUC n)) (f n)$$
- Example: $\$< : \text{num} \rightarrow \text{num} \rightarrow \text{bool}$ is well-founded
- if arguments of recursive calls are smaller according to well-founded relation, the recursion terminates
- this is the essence of termination proofs

152 / 247



- a well-founded relation R can be used to define recursive functions
- this recursion principle is called WFREC in HOL
- idea of WFREC
 - ▶ if arguments get smaller according to R, perform recursive call
 - ▶ otherwise abort and return ARB
- WFREC always defines a function
- if all recursive calls indeed decrease according to R, the original recursive equations can be derived from the WFREC representation
- TFL uses this internally
- however, this is well-hidden from the user

Define discussion

- Define feels like a function definition in HOL
- it can be used to define "terminating" recursive functions
- Define is implemented by a large, non-trivial piece of SML code
- it uses many heuristics
- outcome of Define sometimes hard to predict
- the input descriptions are only hints
 - ▶ the produced function and the definitional theorem might be different
 - ▶ in simple examples, quantifiers added
 - ▶ pattern compilation takes place
 - ▶ earlier "conjuncts" have precedence

153 / 247



Define - Initial Examples



Simple Definitions

```
> val DOUBLE_def = Define 'DOUBLE n = n + n'
val DOUBLE_def =
  |- !n. DOUBLE n = n + n :
  thm

> val MY_LENGTH_def = Define '(MY_LENGTH [] = 0) /\
  (MY_LENGTH (x::xs) = SUC (MY_LENGTH xs))'
val MY_LENGTH_def =
  |- (MY_LENGTH [] = 0) /\ !x xs. MY_LENGTH (x::xs) = SUC (MY_LENGTH xs) :
  thm

> val MY_APPEND_def = Define '(MY_APPEND [] ys = ys) /\
  (MY_APPEND (x::xs) ys = x :: (MY_APPEND xs ys))'
val MY_APPEND_def =
  |- (!ys. MY_APPEND [] ys = ys) /\
  (!x xs ys. MY_APPEND (x::xs) ys = x::MY_APPEND xs ys) :
  thm
```

154 / 247

Define - More Examples



```
> val MY_HD_def = Define 'MY_HD (x :: xs) = x'
val MY_HD_def = |- !x xs. MY_HD (x::xs) = x : thm

> val IS_SORTED_def = Define '
  (IS_SORTED (x1 :: x2 :: xs) = ((x1 < x2) /\ (IS_SORTED (x2::xs)))) /\
  (IS_SORTED _ = T)'
val IS_SORTED_def =
  |- (!xs x2 x1. IS_SORTED (x1::x2::xs) <=> x1 < x2 /\ IS_SORTED (x2::xs)) /\
  (IS_SORTED [] <=> T) /\ (!v. IS_SORTED [v] <=> T)

> val EVEN_def = Define '(EVEN 0 = T) /\ (ODD 0 = F) /\
  (EVEN (SUC n) = ODD n) /\ (ODD (SUC n) = EVEN n)'
val EVEN_def =
  |- (EVEN 0 <=> T) /\ (ODD 0 <=> F) /\ (!n. EVEN (SUC n) <=> ODD n) /\
  (!n. ODD (SUC n) <=> EVEN n) : thm

> val ZIP_def = Define '(ZIP (x::xs) (y::ys) = (x,y)::(ZIP xs ys)) /\
  (ZIP _ _ = [])'
val ZIP_def =
  |- (!ys y xs x. ZIP (x::xs) (y::ys) = (x,y)::ZIP xs ys) /\
  (!v1. ZIP [] v1 = []) /\ (!v4 v3. ZIP (v3::v4) [] = []) : thm
```

155 / 247

156 / 247

Primitive Definitions



- Define introduces (if needed) the function using WFREC
- intended definition derived as a theorem
- the theorems are stored in current theory
- usually, one never needs to look at it

Examples

```
val IS_SORTED_primitive_def =
|- IS_SORTED =
  WFREC (@R. WF R /\ !x1 xs x2. R (x2::xs) (x1::x2::xs))
  (\IS_SORTED a.
    case a of
      [] => I T
    | [x1] => I T
    | x1::x2::xs => I (x1 < x2 /\ IS_SORTED (x2::xs)))

|- !R M. WF R ==> !x. WFREC R M x = M (RESTRICT (WFREC R M) R x) x
|- !f R x. RESTRICT f R x = (\y. if R y x then f y else ARB)
```

157 / 247

Define failing



- Define might fail for various reasons to define a function
 - ▶ such a function cannot be defined in HOL
 - ▶ such a function can be defined, but not via the methods used by TFL
 - ▶ TFL can define such a function, but its heuristics are too weak and user guidance is required
 - ▶ there is a bug :-)
- **termination** is an important concept for Define
- it is easy to misunderstand termination in the context of HOL
- we need to understand what is meant by termination

159 / 247

Induction Theorems



- Define automatically defines induction theorems
- these theorems are stored in current theory with suffix `ind`
- use `DB.fetch "-" "something_ind"` to retrieve them
- these induction theorems are useful to reason about corresponding recursive functions

Example

```
val IS_SORTED_ind = |- !P.
  ((!x1 x2 xs. P (x2::xs) ==> P (x1::x2::xs)) /\
   P [] /\
   (!v. P [v])) ==>
  !v. P v
```

158 / 247

Termination in HOL



- in SML it is natural to talk about termination of functions
- in the HOL logic there is no concept of execution
- thus, there is no concept of termination in HOL

3 characterisations of a function $f : \text{num} \rightarrow \text{num}$

- ▶ `|- !n. f n = 0`
- ▶ `|- (f 0 = 0) /\ !n. (f (SUC n) = f n)`
- ▶ `|- (f 0 = 0) /\ !n. (f n = f (SUC n))`

Is f terminating? All 3 theorems are equivalent.

160 / 247

Termination in HOL II



- it is useful to think in terms of termination
- the TFL package implements heuristics to define functions that would terminate in SML
- the TFL package uses well-founded recursion
- the required well-founded relation corresponds to a termination proof
- therefore, it is very natural to think of `Define` searching a termination proof
- important: this is the idea behind this function definition package, not a property of HOL

HOL is not limited to "terminating" functions

161 / 247

Manual Termination Proofs I



- TFL uses various heuristics to find a well-founded relation
- however, these heuristics may not be strong enough
- in such cases the user can provide a well-founded relation manually
- the most common well-founded relations are **measures**
- measures map values to natural numbers and use the less relation
 $\vdash \!(f : 'a \rightarrow \text{num}) \ x \ y. \ \text{measure } f \ x \ y \iff (f \ x < f \ y)$
- all measures are well-founded: $\vdash \!f. \ \text{WF } (\text{measure } f)$
- moreover, existing well-founded relations can be combined
 - ▶ lexicographic order `LEX`
 - ▶ list lexicographic order `LLEX`
 - ▶ ...

163 / 247

Termination in HOL III



- one can define "non-terminating" functions in HOL
- however, one cannot do so (easily) with `Define`

Definition of `WHILE` in HOL

```
 $\vdash \!P \ g \ x. \ \text{WHILE } P \ g \ x = \text{if } P \ x \ \text{then } \text{WHILE } P \ g \ (g \ x) \ \text{else } x$ 
```

Execution Order

There is no "execution order". One can easily define a complicated constant function:

```
 $(\text{myk} : \text{num} \rightarrow \text{num}) \ (n : \text{num}) = (\text{let } x = \text{myk } (n+1) \ \text{in } 0)$ 
```

Unsound Definitions

A function $f : \text{num} \rightarrow \text{num}$ with the following property cannot be defined in HOL unless HOL has an inconsistency:

```
 $\!n. \ f \ n = ((f \ n) + 1)$ 
```

Such a function would allow to prove $0 = 1$.

162 / 247

Manual Termination Proofs II



- if `Define` fails to find a termination proof, `Hol_defn` can be used
- `Hol_defn` defers termination proofs
- it derives termination conditions and sets up the function definitions
- all results are packaged as a value of type `defn`
- after calling `Hol_defn` the defined function(s) can be used
- however, the intended definition theorem has not been derived yet
- to derive it, one needs to
 - ▶ provide a well-founded relation
 - ▶ show that termination conditions respect that relation
- `Defn.tprove` and `Defn.tgoal` are intended for this
- proofs usually start by providing relation via tactic `WF_REL_TAC`

164 / 247

Manual Termination Proof Example 1

```
> val qsort_defn = Hol_defn "qsort" `
  (qsort ord [] = []) /\
  (qsort ord (x::rst) =
    (qsort ord (FILTER ($~ o ord x) rst)) ++
    [x] ++
    (qsort ord (FILTER (ord x) rst)))`

val qsort_defn = HOL function definition (recursive)

Equation(s) :
[... ] |- qsort ord [] = []
[... ] |- qsort ord (x::rst) =
  qsort ord (FILTER ($~ o ord x) rst) ++ [x] ++
  qsort ord (FILTER (ord x) rst)

Induction : ...

Termination conditions :
0. !rst x ord. R (ord,FILTER (ord x) rst) (ord,x::rst)
1. !rst x ord. R (ord,FILTER ($~ o ord x) rst) (ord,x::rst)
2. WF R
```

Manual Termination Proof Example 3

```
> val (qsort_def, qsort_ind) =
  Defn.tprove (qsort_defn,
    WF_REL_TAC 'measure (\(., 1). LENGTH 1)' >> ...)

val qsort_def =
|- (qsort ord [] = []) /\
  (qsort ord (x::rst) =
    qsort ord (FILTER ($~ o ord x) rst) ++ [x] ++
    qsort ord (FILTER (ord x) rst))

val qsort_ind =
|- !P. (!ord. P ord []) /\
  (!ord x rst.
    P ord (FILTER (ord x) rst) /\
    P ord (FILTER ($~ o ord x) rst) ==>
    P ord (x::rst)) ==>
!v v1. P v v1
```



165 / 247



167 / 247

Manual Termination Proof Example 2

```
> Defn.tgoal qsort_defn

Initial goal:

?R.
  WF R /\
  (!rst x ord. R (ord,FILTER (ord x) rst) (ord,x::rst)) /\
  (!rst x ord. R (ord,FILTER ($~ o ord x) rst) (ord,x::rst))

> e (WF_REL_TAC 'measure (\(., 1). LENGTH 1)')

1 subgoal :

(!rst x ord. LENGTH (FILTER (ord x) rst) < LENGTH (x::rst)) /\
(!rst x ord. LENGTH (FILTER (\x'. ~ord x x') rst) < LENGTH (x::rst))

> ...
```



166 / 247

Part XI

Good Definitions



Importance of Good Definitions



- using *good* definitions is very important
 - ▶ good definitions are vital for **clarity**
 - ▶ **proofs** depend a lot on the form of definitions
- unluckily, it is hard to state what a good definition is
- even harder to come up with good definitions
- let's look at it a bit closer anyhow

169 / 247

Importance of Good Definitions — Clarity II



Guarded by HOL

- proofs checked
- internal, technical definitions
- technical lemmata
- proof tools

Manual review needed for

- meaning of main theorems
- meaning of definitions used by main theorems
- meaning of types used by main theorems

171 / 247

Importance of Good Definitions — Clarity I



- HOL guarantees that theorems do indeed hold
- However, does the theorem mean what you think it does?
- you can separate your development in
 - ▶ main theorems you care for
 - ▶ auxiliary stuff used to derive your main theorems
- it is essential to understand your main theorems

170 / 247

Importance of Good Definitions — Clarity III



- it is essential to understand your main theorems
 - ▶ you need to understand all the definitions directly used
 - ▶ you need to understand the indirectly used ones as well
 - ▶ you need to convince others that you express the intended statement
 - ▶ therefore, it is vital to **use very simple, clear definitions**
- defining concepts is often the main development task
- checking resulting model against real artefact is vital
 - ▶ testing via e. g. EVAL
 - ▶ formal sanity
 - ▶ conformance testing
- wrong models are main source of error when using HOL
- proofs, auxiliary lemmata and auxiliary definitions
 - ▶ can be as technical and complicated as you like
 - ▶ correctness is guaranteed by HOL
 - ▶ reviewers don't need to care

172 / 247

Importance of Good Definitions — Proofs

- good definitions can shorten proofs significantly
- they improve maintainability
- they can improve automation drastically
- unluckily for proofs definitions often need to be technical
- this contradicts clarity aims



How to come up with good definitions

- unluckily, it is hard to state what a good definition is
- it is even harder to come up with them
 - ▶ there are often many competing interests
 - ▶ a lot of experience and detailed tool knowledge is needed
 - ▶ much depends on personal style and taste
- general advice: use more than one definition
 - ▶ in HOL you can derive equivalent definitions as theorems
 - ▶ define a concept as clearly and easily as possible
 - ▶ derive equivalent definitions for various purposes
 - ★ one very close to your favourite textbook
 - ★ one nice for certain types of proofs
 - ★ another one good for evaluation
 - ★ ...
- lessons from functional programming apply



173 / 247

Good Definitions in Functional Programming

Objectives

- clarity (readability, maintainability)
- performance (runtime speed, memory usage, ...)

General Advice

- use the powerful type-system
- use many small function definitions
- encode invariants in types and function signatures



Good Definitions – no number encodings

- many programmers familiar with C encode everything as a number
- enumeration types are very cheap in SML and HOL
- use them instead

Example Enumeration Types

In C the result of an order comparison is an integer with 3 equivalence classes: 0, negative and positive integers. In SML and HOL, it is better to use a variant type.

```
val _ = Datatype 'ordering = LESS | EQUAL | GREATER';

val compare_def = Define '
  (compare LESS    lt eq gt = lt)
  /\ (compare EQUAL lt eq gt = eq)
  /\ (compare GREATER lt eq gt = gt) ' ;

val list_compare_def = Define '
  (list_compare cmp [] [] = EQUAL) /\ (list_compare cmp [] l2 = LESS)
  /\ (list_compare cmp l1 [] = GREATER)
  /\ (list_compare cmp (x::l1) (y::l2) = compare (cmp (x:'a) y)
      (* x<y *) LESS
      (* x=y *) (list_compare cmp l1 l2)
      (* x>y *) GREATER) ' ;
```

174 / 247



175 / 247

176 / 247

Good Definitions — Isomorphic Types



- the type-checker is your friend
 - ▶ it helps you find errors
 - ▶ code becomes more robust
 - ▶ using good types is a great way of writing self-documenting code
- therefore, use many types
- even use types isomorphic to existing ones

Virtual and Physical Memory Addresses

Virtual and physical addresses might in a development both be numbers. It is still nice to use separate types to avoid mixing them up.

```
val _ = Datatype 'vaddr = VAddr num';
val _ = Datatype 'paddr = PAddr num';

val virt_to_phys_addr_def = Define '
  virt_to_phys_addr (VAddr a) = PAddr( translation of a );
```

177 / 247

Good Definitions — Record Types II



- using records
 - ▶ introduces field names
 - ▶ provides automatically defined accessor and update functions
 - ▶ leads to better type-checking error messages
- records improve readability
 - ▶ accessors and update functions lead to shorter code
 - ▶ field names act as documentation
- records improve maintainability
 - ▶ improved error messages
 - ▶ much easier to add extra fields

179 / 247

Good Definitions — Record Types I



- often people use tuples where records would be more appropriate
- using large tuples quickly becomes awkward
 - ▶ it is easy to mix up order of tuple entries
 - ★ often types coincide, so type-checker does not help
 - ▶ no good error messages for tuples
 - ★ hard to decipher type mismatch messages for long product types
 - ★ hard to figure out which entry is missing at which position
 - ★ non-local error messages
 - ★ variable in last entry can hide missing entries
- records sometimes require slightly more proof effort
- however, records have many benefits

178 / 247

Good Definitions — Encoding Invariants



- try to encode as many invariants as possible in the types
- this allows the type-checker to ensure them for you
- you don't have to check them manually any more
- your code becomes more robust and clearer

Network Connections (Example by Yaron Minsky from Jane Street)

Consider the following datatype for network connections. It has many implicit invariants.

```
datatype connection_state = Connected | Disconnected | Connecting;

type connection_info = {
  state           : connection_state,
  server          : inet_address,
  last_ping_time  : time option,
  last_ping_id    : int option,
  session_id      : string option,
  when_initiated  : time option,
  when_disconnected : time option
}
```

180 / 247

Good Definitions — Encoding Invariants II



Network Connections (Example by Yaron Minsky from Jane Street) II

The following definition of `connection_info` makes the invariants explicit:

```
type connected = { last_ping      : (time * int) option,
                  session_id     : string };
type disconnected = { when_disconnected : time };
type connecting = { when_initiated  : time };

datatype connection_state =
  Connected of connected
| Disconnected of disconnected
| Connecting of connecting;

type connection_info = {
  state : connection_state,
  server : inet_address
}
```

181 / 247

Good Definitions in HOL II



Technical Issues

- write definition such that they work well with HOL's tools
- this requires you to know HOL well
- a lot of experience is required
- general advice
 - ▶ avoid explicit case-expressions
 - ▶ prefer curried functions

Example

```
val ZIP_GOOD_def = Define '(ZIP (x::xs) (y::ys) = (x,y)::(ZIP xs ys)) /\
  (ZIP _ _ = [])'

val ZIP_BAD1_def = Define 'ZIP xs ys = case (xs, ys) of
  (x::xs, y::ys) => (x,y)::(ZIP xs ys)
| (_, _) => []'

val ZIP_BAD2_def = Define '(ZIP (x::xs, y::ys) = (x,y)::(ZIP (xs, ys))) /\
  (ZIP _ = [])'
```

183 / 247

Good Definitions in HOL



Objectives

- clarity (readability)
- good for proofs
- performance (good for automation, easily evaluable, ...)

General Advice

- same advice as for functional programming applies
- use even smaller definitions
 - ▶ introduce auxiliary definitions for important function parts
 - ▶ use extra definitions for important constants
 - ▶ ...
- tiny definitions
 - ▶ allow keeping proof state small by unfolding only needed ones
 - ▶ allow many small lemmata
 - ▶ improve maintainability

182 / 247

Good Definitions in HOL III



Multiple Equivalent Definitions

- satisfy competing requirements by having multiple equivalent definitions
- derive them as theorems
- initial definition should be as clear as possible
 - ▶ clarity allows simpler reviews
 - ▶ simplicity reduces the likelihood of errors

Example - ALL_DISTINCT

```
|- (ALL_DISTINCT [] <=> T) /\
  (!h t. ALL_DISTINCT (h::t) <=> ~MEM h t /\ ALL_DISTINCT t)

|- !l. ALL_DISTINCT l <=>
  (!x. MEM x l ==> (FILTER ($= x) l = [x]))

|- !ls. ALL_DISTINCT ls <=> (CARD (set ls) = LENGTH ls):
```

184 / 247



Formal Sanity

- to ensure correctness test your definitions via e. g. EVAL
- in HOL testing means symbolic evaluation, i. e. proving lemmata
- formally proving sanity check lemmata** is very beneficial
 - they should express core properties of your definition
 - thereby they check your intuition against your actual definitions
 - these lemmata are often useful for following proofs
 - using them improves robustness and maintainability of your development
- I highly recommend using formal sanity checks

```
> val ALL_DISTINCT = Define '
  (ALL_DISTINCT [] = T) /\
  (ALL_DISTINCT (h::t) = ~MEM h t /\ ALL_DISTINCT t)';
```

Example Sanity Check Lemmata

```
|- ALL_DISTINCT []
|- !x xs. ALL_DISTINCT (x::xs) <=> ~MEM x xs /\ ALL_DISTINCT xs
|- !x. ALL_DISTINCT [x]
|- !x xs. ~(ALL_DISTINCT (x::x::xs))
|- !l. ALL_DISTINCT (REVERSE l) <=> ALL_DISTINCT l
|- !x l. ALL_DISTINCT (SNOC x l) <=> ~MEM x l /\ ALL_DISTINCT l
|- !l1 l2. ALL_DISTINCT (l1 ++ l2) <=>
  ALL_DISTINCT l1 /\ ALL_DISTINCT l2 /\ !e. MEM e l1 ==> ~MEM e l2
```

Formal Sanity Example II 1



```
> val ZIP_def = Define '
  (ZIP [] ys = []) /\ (ZIP xs [] = []) /\
  (ZIP (x::xs) (y::ys) = (x, y)::(ZIP xs ys))'
```

```
val ZIP_def =
|- (!ys. ZIP [] ys = []) /\ (!v3 v2. ZIP (v2::v3) [] = []) /\
  (!ys y xs x. ZIP (x::xs) (y::ys) = (x,y)::ZIP xs ys)
```

- above definition of ZIP looks straightforward
- small changes cause heuristics to produce different theorems
- use formal sanity lemmata to compensate

```
> val ZIP_def = Define '
  (ZIP xs [] = []) /\ (ZIP [] ys = []) /\
  (ZIP (x::xs) (y::ys) = (x, y)::(ZIP xs ys))'
```

```
val ZIP_def =
|- (!xs. ZIP xs [] = []) /\ (!v3 v2. ZIP [] (v2::v3) = []) /\
  (!ys y xs x. ZIP (x::xs) (y::ys) = (x,y)::ZIP xs ys)
```

Formal Sanity Example II 2

```
val ZIP_def =
|- (!ys. ZIP [] ys = []) /\ (!v3 v2. ZIP (v2::v3) [] = []) /\
  (!ys y xs x. ZIP (x::xs) (y::ys) = (x,y)::ZIP xs ys)
```

Example Formal Sanity Lemmata

```
|- (!xs. ZIP xs [] = []) /\ (!ys. ZIP [] ys = []) /\
  (!y ys x xs. ZIP (x::xs) (y::ys) = (x,y)::ZIP xs ys)
|- !xs ys. LENGTH (ZIP xs ys) = MIN (LENGTH xs) (LENGTH ys)
|- !x y xs ys. MEM (x, y) (ZIP xs ys) ==> (MEM x xs /\ MEM y ys)
|- !xs1 xs2 ys1 ys2. LENGTH xs1 = LENGTH ys1 ==>
  (ZIP (xs1++xs2) (ys1++ys2) = (ZIP xs1 ys1 ++ ZIP xs2 ys2))
...

```

- in your proofs use sanity lemmata, not original definition
- this makes your development robust against
 - small changes to the definition required later
 - changes to Define and its heuristics
 - bugs in function definition package



Part XII

Deep and Shallow Embeddings



- often one models some kind of formal language
- important design decision: use **deep** or **shallow** embedding
- in a nutshell:
 - ▶ shallow embeddings just model semantics
 - ▶ deep embeddings model syntax as well
- a shallow embedding directly uses the HOL logic
- a deep embedding
 - ▶ defines a datatype for the syntax of the language
 - ▶ provides a function to map this syntax to a semantic

Example: Embedding of Propositional Logic I



- propositional logic is a subset of HOL
- a shallow embedding is therefore trivial

```
val sh_true_def      = Define 'sh_true = T';
val sh_var_def      = Define 'sh_var (v:bool) = v';
val sh_not_def      = Define 'sh_not b = ~b';
val sh_and_def      = Define 'sh_and b1 b2 = (b1 /\ b2)';
val sh_or_def       = Define 'sh_or b1 b2 = (b1 \/ b2)';
val sh_implies_def  = Define 'sh_implies b1 b2 = (b1 ==> b2)';
```

Example: Embedding of Propositional Logic II



- we can also define a datatype for propositional logic
- this leads to a deep embedding

```
val _ = Datatype 'bvar = BVar num'
val _ = Datatype 'prop = d_true | d_var bvar | d_not prop
                | d_and prop prop | d_or prop prop
                | d_implies prop prop';
```

```
val _ = Datatype 'var_assignment = BAssign (bvar -> bool)'
val VAR_VALUE_def = Define 'VAR_VALUE (BAssign a) v = (a v)'
```

```
val PROP_SEM_def = Define '
(PROP_SEM a d_true = T) /\
(PROP_SEM a (d_var v) = VAR_VALUE a v) /\
(PROP_SEM a (d_not p) = ~(PROP_SEM a p)) /\
(PROP_SEM a (d_and p1 p2) = (PROP_SEM a p1 /\ PROP_SEM a p2)) /\
(PROP_SEM a (d_or p1 p2) = (PROP_SEM a p1 \/ PROP_SEM a p2)) /\
(PROP_SEM a (d_implies p1 p2) = (PROP_SEM a p1 ==> PROP_SEM a p2))';
```


Shallow vs. Deep Embeddings



Shallow

- quick and easy to build
- extensions are simple

Deep

- can reason about syntax
- allows verified implementations
- sometimes tricky to define
 - ▶ e. g. bound variables

Important Questions for Deciding

- Do I need to reason about syntax?
- Do I have hard to define syntax like bound variables?
- How much time do I have?

193 / 247

Verified vs. Verifying Program



Verified Programs

- are formalised in HOL
- their properties have been proven once and for all
- all runs have proven properties
- are usually less sophisticated, since they need verification
- is what one wants ideally
- often require deep embedding

Verifying Programs

- are written in meta-language
- they produce a separate proof for each run
- only certain that current run has properties
- allow more flexibility, e. g. fancy heuristics
- good pragmatic solution
- shallow embedding fine

195 / 247

Example: Embedding of Propositional Logic III



- with deep embedding one can easily formalise syntactic properties like
 - ▶ Which variables does a propositional formula contain?
 - ▶ Is a formula in negation-normal-form (NNF)?
- with shallow embeddings
 - ▶ syntactic concepts can't be defined in HOL
 - ▶ however, they can be defined in SML
 - ▶ no proofs about them possible

```
val _ = Define `
  (IS_NNF (d_not d_true) = T) /\ (IS_NNF (d_not (d_var v)) = T) /\
  (IS_NNF (d_not _) = F) /\

  (IS_NNF d_true = T) /\ (IS_NNF (d_var v) = T) /\
  (IS_NNF (d_and p1 p2) = (IS_NNF p1 /\ IS_NNF p2)) /\
  (IS_NNF (d_or p1 p2) = (IS_NNF p1 /\ IS_NNF p2)) /\
  (IS_NNF (d_implies p1 p2) = (IS_NNF p1 /\ IS_NNF p2))`
```

194 / 247

Summary Deep vs. Shallow Embeddings



- deep embeddings require more work
- they however allow reasoning about syntax
 - ▶ induction and case-splits possible
 - ▶ a semantic subset can be carved out syntactically
- syntax sometimes hard to define for deep embeddings
- combinations of deep and shallow embeddings common
 - ▶ certain parts are deeply embedded
 - ▶ others are embedded shallowly

196 / 247

Part XIII

Rewriting



Rewriting in HOL

- simplification via rewriting was already a strength of Edinburgh LCF
- it was further improved for Cambridge LCF
- HOL inherited this powerful rewriter
- equational reasoning is still the main workhorse
- there are many different equational reasoning tools in HOL
 - ▶ Rewrite library
 - inherited from Cambridge LCF
 - you have seen it in the form of REWRITE_TAC
 - ▶ computeLib — fast evaluation
 - build for speed, optimised for ground terms
 - seen in the form of EVAL
 - ▶ simpLib — Simplification
 - sophisticated rewrite engine, HOL's main workhorse
 - not discussed in this lecture, yet
 - ▶ ...

198 / 247

Semantic Foundations



- we have seen primitive inference rules for equality before

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

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

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

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

- these rules allow us to replace any subterm with an equal one
- this is the core of rewriting

Conversions



- in HOL, equality reasoning is implemented by **conversions**
- a conversion is a SML function of type `term -> thm`
- given a term `t`, a conversion
 - ▶ produces a theorem of the form `|- t = t'`
 - ▶ raises an UNCHANGED exception or
 - ▶ fails, i.e. raises an HOL_ERR exception

Example

```
> BETA_CONV '(λx. SUC x) y'
val it = |- (λx. SUC x) y = SUC y

> BETA_CONV 'SUC y'
Exception-HOL_ERR ... raised

> REPEATC BETA_CONV 'SUC y'
Exception- UNCHANGED raised
```



- similar to tactics and tacticals there are **conversionals** for conversions
- conversionals allow building conversions from simpler ones
- there are many of them
 - ▶ THENC
 - ▶ ORELSEC
 - ▶ REPEATC
 - ▶ TRY_CONV
 - ▶ RAND_CONV
 - ▶ RATOR_CONV
 - ▶ ABS_CONV
 - ▶ ...

REWR_CONV

- it remains to rewrite terms at top-level
- this is achieved by REWR_CONV
- given a term t and a theorem $\vdash t_1 = t_2$, REWR_CONV t thm
 - ▶ searches an instantiation of term and type variables such that t_1 becomes α -equivalent to t
 - ▶ fails, if no instantiation is found
 - ▶ otherwise, instantiate the theorem and get $\vdash t_1' = t_2'$
 - ▶ return theorem $\vdash t = t_2'$

Example

```
term LENGTH [1;2;3], theorem |- LENGTH ((x:'a)::xs) = SUC (LENGTH xs)
found type instantiation: [':a' |-> ':num']
found term instantiation: ['x:num' |-> '1'; 'xs' |-> '[2;3]']
returned theorem: |- LENGTH [1;2;3] = SUC (LENGTH [2;3])
```

- the tricky part is finding the instantiation
- this problem is called the (term) **matching** problem



- for rewriting depth-conversionals are important
- a depth-conversional applies a conversion to all subterms
- there are many different ones
 - ▶ ONCE_DEPTH_CONV c — top down, applies c once at highest possible positions in distinct subterms
 - ▶ TOP_SWEEP_CONV c — top down, like ONCE_DEPTH_CONV, but continues processing rewritten terms
 - ▶ TOP_DEPTH_CONV c — top down, like TOP_SWEEP_CONV, but try top-level again after change
 - ▶ DEPTH_CONV c — bottom up, recurse over subterms, then apply c repeatedly at top-level
 - ▶ REDEPTH_CONV c — bottom up, like DEPTH_CONV, but revisits subterms

Term Matching



- given term t_org and a term t_goal try to find
 - ▶ type substitution ty_s
 - ▶ term substitution tm_s
- such that $subst\ tm_s\ (inst\ ty_s\ t_org) \equiv^{\alpha} t_goal$
- this can be easily implemented by a recursive search

t_org	t_goal	action
$t1_org\ t2_org$	$t1_goal\ t2_goal$	recurse
$t1_org\ t2_org$	otherwise	fail
$\backslash x. t_org\ x$	$\backslash y. t_goal\ y$	match types of x, y and recurse
$\backslash x. t_org\ x$	otherwise	fail
const	same const	match types
const	otherwise	fail
var	anything	try to bind var, take care of existing bindings

Examples Term Matching



t_org	t_goal	subst
LENGTH ((x:'a)::xs)	LENGTH [1;2;3]	'a → num, x → 1, xs → [2;3]
[]:'a list	[]:'b list	'a → 'b
0	0	empty substitution
b /\ T	(P (x:'a) ==> Q) /\ T	b → P x ==> Q
b /\ b	P x /\ P x	b → P x
b /\ b	P x /\ P y	fail
!x:num. P x /\ Q x	!y:num. P' y /\ Q' y	P → P', Q → Q'
!x:num. P x /\ Q x	!y. (2 = y) /\ Q' y	P → (\$= 2), Q → Q'
!x:num. P x /\ Q x	!y. (y = 2) /\ Q' y	fail

- it is often very annoying that the last match fails
- it prevents us for example rewriting $!y. (2 = y) /\ Q y$ to $(!y. (2=y)) /\ (!y. Q y)$
- Can we do better? Yes, with higher order (term) matching.

205 / 247

Examples Higher Order Term Matching



t_org	t_goal	subst
!x:num. P x /\ Q x	!y. (y = 2) /\ Q' y	P → (\y. y = 2), Q → Q'
!x. P x /\ Q x	!x. P x /\ Q x /\ Z x	Q → \x. Q x /\ Z x
!x. P x /\ Q	!x. P x /\ Q x	fails
!x. P (x, x)	!x. Q x	fails
!x. P (x, x)	!x. FST (x,x) = SND (x,x)	P → \xx. FST xx = SND xx

Don't worry, it might look complicated, but in practice it is easy to get a feeling for higher order matching.

207 / 247

Higher Order Term Matching



- term matching searches for substitutions such that `t_org` becomes α -equivalent to `t_goal`
- **higher order term matching** searches for substitutions such that `t_org` becomes `t_subst` such that the $\beta\eta$ -normalform of `t_subst` is α -equivalent equivalent to $\beta\eta$ -normalform of `t_goal`, i. e. **higher order term matching is aware of the semantics of λ**

β -reduction $(\lambda x. f) y = f[y/x]$

η -conversion $(\lambda x. f x) = f$ where x is not free in f

- the HOL implementation expects `t_org` to be a **higher-order pattern**
 - ▶ `t_org` is β -reduced
 - ▶ if X is a variable that should be instantiated, then all arguments should be distinct variables
- for other forms of `t_org`, HOL's implementation might fail
- higher order matching is used by `HO_REWR_CONV`

206 / 247

Rewrite Library



- the rewrite library combines `REWR_CONV` with depth conversions
- there are many different conversions, rules and tactics
- at they core, they all work very similarly
 - ▶ given a list of theorems, a set of rewrite theorems is derived
 - ★ split conjunctions
 - ★ remove outermost universal quantification
 - ★ introduce equations by adding $= T$ (or $= F$) if needed
 - ▶ `REWR_CONV` is applied to all the resulting rewrite theorems
 - ▶ a depth-conversion is used with resulting conversion
- for performance reasons an efficient indexing structure is used
- by default implicit rewrites are added

208 / 247



- REWRITE_CONV
- REWRITE_RULE
- REWRITE_TAC
- ASM_REWRITE_TAC
- ONCE_REWRITE_TAC
- PURE_REWRITE_TAC
- PURE_ONCE_REWRITE_TAC
- ...

Examples Rewrite and Ho_Rewrite Library

```
> REWRITE_CONV [LENGTH] ``LENGTH [1;2]``
val it = |- LENGTH [1; 2] = SUC (SUC 0)

> ONCE_REWRITE_CONV [LENGTH] ``LENGTH [1;2]``
val it = |- LENGTH [1; 2] = SUC (LENGTH [2])

> REWRITE_CONV [] ``A /\ A /\ ~A``
Exception- UNCHANGED raised

> PURE_REWRITE_CONV [NOT_AND] ``A /\ A /\ ~A``
val it = |- A /\ A /\ ~A <=> A /\ F

> REWRITE_CONV [NOT_AND] ``A /\ A /\ ~A``
val it = |- A /\ A /\ ~A <=> F

> REWRITE_CONV [FORALL_AND_THM] ``!x. P x /\ Q x /\ R x``
Exception- UNCHANGED raised

> Ho_Rewrite.REWRITE_CONV [FORALL_AND_THM] ``!x. P x /\ Q x /\ R x``
val it = |- !x. P x /\ Q x /\ R x <=> (!x. P x) /\ (!x. Q x) /\ (!x. R x)
```

209 / 247



211 / 247



- similar to Rewrite lib, but uses higher order matching
- internally uses HO_REWR_CONV
- similar conversions, rules and tactics as Rewrite lib
 - ▶ Ho_Rewrite.REWRITE_CONV
 - ▶ Ho_Rewrite.REWRITE_RULE
 - ▶ Ho_Rewrite.REWRITE_TAC
 - ▶ Ho_Rewrite.ASM_REWRITE_TAC
 - ▶ Ho_Rewrite.ONCE_REWRITE_TAC
 - ▶ Ho_Rewrite.PURE_REWRITE_TAC
 - ▶ Ho_Rewrite.PURE_ONCE_REWRITE_TAC
 - ▶ ...

Summary Rewrite and Ho_Rewrite Library

210 / 247



- the Rewrite and Ho_Rewrite library provide powerful infrastructure for term rewriting
- thanks to clever implementations they are reasonably efficient
- basics are easily explained
- however, efficient usage needs some experience

212 / 247

Term Rewriting Systems



- to use rewriting efficiently, one needs to understand about term rewriting systems
- this is a large topic
- one can easily give whole course just about term rewriting systems
- however, in practise you quickly get a feeling
- important points in practise
 - ▶ ensure termination of your rewrites
 - ▶ make sure they work nicely together

Termination — Subterm examples

- a proper subterm is always simpler
 - ▶ !l. APPEND [] l = l
 - ▶ !n. n + 0 = n
 - ▶ !l. REVERSE (REVERSE l) = l
 - ▶ !t1 t2. if T then t1 else t2 <=> t1
 - ▶ !n. n * 0 = 0
- the right hand side should not use extra vars, throwing parts away is usually simpler
 - ▶ !x xs. (SNOC x xs = []) = F
 - ▶ !x xs. LENGTH (x::xs) = SUC (LENGTH xs)
 - ▶ !n x xs. DROP (SUC n) (x::xs) = DROP n xs

213 / 247



215 / 247

Term Rewriting Systems — Termination



Theory

- choose well-founded order \prec
- for each rewrite theorem $| - t_1 = t_2$ ensure $t_2 \prec t_1$

Practice

- informally define for yourself what **simpler** means
- ensure each rewrite makes terms simpler
- good heuristics
 - ▶ subterms are simpler than whole term
 - ▶ use an order on functions

214 / 247



Termination — use simpler terms

- it is useful to consider some functions simple and other complicated
- replace complicated ones with simple ones
- never do it in the opposite direction
- clear examples
 - ▶ $| - !m n. \text{MEM } m (\text{COUNT_LIST } n) \Leftrightarrow (m < n)$
 - ▶ $| - !l s n. (\text{DROP } n \text{ } l s = []) \Leftrightarrow (n \geq \text{LENGTH } l s)$
- unclear example
 - ▶ $| - !L. \text{REVERSE } L = \text{REV } L []$

216 / 247

Termination — Normalforms



- some equations can be used in both directions
- one should decide on one direction
- this implicitly defined a **normalform** one wants terms to be in
- examples
 - ▶ `|- !f l. MAP f (REVERSE l) = REVERSE (MAP f l)`
 - ▶ `|- !l1 l2 l3. l1 ++ (l2 ++ l3) = l1 ++ l2 ++ l3`

Rewrites working together

- rewrite rules should not complete with each other
- if a term `ta` can be rewritten to `ta1` and `ta2` applying different rewrite rules, then the `ta1` and `ta2` should be further rewritten to a common `tb`
- this can often be achieved by adding extra rewrite rules

Example

Assume we have the rewrite rules `|- DOUBLE n = n + n` and `|- EVEN (DOUBLE n) = T`.

With these the term `EVEN (DOUBLE 2)` can be rewritten to

- `T` or
- `EVEN (2 + 2)`.

To avoid a hard to predict result, `EVEN (2+2)` should be rewritten to `T`. Adding an extra rewrite rule `|- EVEN (n + n) = T` achieves this.

217 / 247



219 / 247

Termination — Problematic rewrite rules



- some equations immediately lead to non-termination, e. g.
 - ▶ `|- !m n. m + n = n + m`
 - ▶ `|- !m. m = m + 0`
- slightly more subtle are rules like
 - ▶ `|- !n. fact n = if (n = 0) then 1 else n * fact(n-1)`
- often combination of multiple rules leads to non-termination this is especially problematic when adding to predefined set of rewrites
 - ▶ `|- !m n p. m + (n + p) = (m + n) + p` and
 - ▶ `|- !m n p. (m + n) + p = m + (n + p)`

218 / 247



Rewrites working together II

- to design rewrite systems that work well, normalforms are vital
- a term is in **normalform**, if it cannot be rewritten any further
- one should have a clear idea what the normalform of common terms looks like
- all rules should work together to establish this normalform
- the right-hand-side of each rule should be in normalform
- the left-hand-side should not be simplifiable by any other rule
- the order in which rules are applied should not influence the final result

220 / 247

computeLib



- computeLib is the library behind EVAL
- it is a rewriting library designed for evaluating ground terms (i. e. terms without variables) efficiently
- it uses a call-by-value strategy similar to SML's
- it uses first order term matching
- it performs β reduction in addition to rewrites

221 / 247

EVAL

- EVAL uses `the_compset`
- tools like the `Datatype` of TFL automatically extend `the_compset`
- this way, EVAL knows about (nearly) all types and functions
- one can extend `the_compset` manually as well
- rewrites exported by `Define` are good for ground terms but may lead to non-termination for non-ground terms
- `zDefine` prevents TFL from automatically extending `the_compset`

223 / 247

compset



- computeLib uses compsets to store its rewrites
- a compset stores
 - ▶ rewrite rules
 - ▶ extra conversions
- the extra conversions are guarded by a term pattern for efficiency
- users can define their own compsets
- however, computeLib maintains one special compset called `the_compset`
- `the_compset` is used by EVAL

222 / 247

simpLib



- simpLib is a sophisticated rewrite engine
- it is HOL's main workhorse
- it provides
 - ▶ higher order rewriting
 - ▶ usage of context information
 - ▶ conditional rewriting
 - ▶ arbitrary conversions
 - ▶ support for decision procedures
 - ▶ simple heuristics to avoid non-termination
 - ▶ fancier preprocessing of rewrite theorems
 - ▶ ...
- it is very powerful, but compared to `Rewrite` lib sometimes slow



224 / 247

Basic Usage I

- `simpLib` uses **simpsets**
- simpsets are special datatypes storing
 - ▶ rewrite rules
 - ▶ conversions
 - ▶ decision procedures
 - ▶ congruence rules
 - ▶ ...
- in addition there are simpset-fragments
- simpset-fragments contain similar information as simpsets
- fragments can be added to and removed from simpsets
- common usage: basic simpset combined with one or more simpset-fragments, e. g.
 - ▶ `list_ss ++ pairSimps.gen_beta_ss`
 - ▶ `std_ss ++ QI_ss`
 - ▶ ...

Basic Simplifier Examples

```
> SIMP_CONV bool_ss [LENGTH] ``LENGTH [1;2]``  
val it = |- LENGTH [1; 2] = SUC (SUC 0)  
  
> SIMP_CONV std_ss [LENGTH] ``LENGTH [1;2]``  
val it = |- LENGTH [1; 2] = 2  
  
> SIMP_CONV list_ss [] ``LENGTH [1;2]``  
val it = |- LENGTH [1; 2] = 2
```



Basic Usage II

- a call to the simplifier takes as arguments
 - ▶ a simpset
 - ▶ a list of rewrite theorems
- common high-level entry points are
 - ▶ `SIMP_CONV ss thmL` — conversion
 - ▶ `SIMP_RULE ss thmL` — rule
 - ▶ `SIMP_TAC ss thmL` — tactic without considering assumptions
 - ▶ `ASM_SIMP_TAC ss thmL` — tactic using assumptions to simplify goal
 - ▶ `FULL_SIMP_TAC ss thmL` — tactic simplifying assumptions with each other and goal with assumptions
 - ▶ `REV_FULL_SIMP_TAC ss thmL` — similar to `FULL_SIMP_TAC` but with reversed order of assumptions
- there are many derived tools not discussed here

225 / 247



Common simpsets

- `pure_ss` — empty simpset
- `bool_ss` — basic simpset
- `std_ss` — standard simpset
- `arith_ss` — arithmetic simpset
- `list_ss` — list simpset
- `real_ss` — real simpset

227 / 247



226 / 247



228 / 247

- many theories and libraries provide their own simpset-fragments
- `PRED_SET_ss` — simplify sets
- `STRING_ss` — simplify strings
- `QI_ss` — extra quantifier instantiations
- `gen_beta_ss` — β reduction for pairs
- `ETA_ss` — η conversion
- `EQUIV_EXTRACT_ss` — extract common part of equivalence
- `CONJ_ss` — use conjunctions for context
- ...

Examples I

```
> SIMP_CONV std_ss [] ‘‘(\x. x + 2) 5‘‘
val it = |- (\x. x + 2) 5 = 7

> SIMP_CONV std_ss [] ‘‘!x. Q x /\ (x = 7) ==> P x‘‘
val it = |- (!x. Q x /\ (x = 7) ==> P x) <=> (Q 7 ==> P 7)‘‘

> SIMP_CONV std_ss [] ‘‘?x. Q x /\ (x = 7) /\ P x‘‘
val it = |- (?x. Q x /\ (x = 7) /\ P x) <=> (Q 7 /\ P 7)‘‘

> SIMP_CONV std_ss [] ‘‘x > 7 ==> x > 5‘‘
Exception- UNCHANGED raised

> SIMP_CONV arith_ss [] ‘‘x > 7 ==> x > 5‘‘
val it = |- (x > 7 ==> x > 5) <=> T
```



- in contrast to Rewrite lib the simplifier can run arbitrary conversions
- most useful is probably β reduction
- `std_ss` has support for basic arithmetic and numerals
- it also has simple, syntactic conversions for instantiating quantifiers
 - ▶ `!x. ... /\ (x = c) /\ ... ==> ...`
 - ▶ `!x. ... \/ ~(x = c) \/ ...`
 - ▶ `?x. ... /\ (x = c) /\ ...`
- besides very useful conversions, there are decision procedures as well
- the most frequently used one is probably the arithmetic decision procedure you already know from DECIDE

229 / 247



Higher Order Rewriting

- the simplifier supports higher order rewriting
- this is often very handy
- for example it allows moving quantifiers around easily

Examples

```
> SIMP_CONV std_ss [FORALL_AND_THM] ‘‘!x. P x /\ Q /\ R x‘‘
val it = |- (!x. P x /\ Q /\ R x) <=>
  (!x. P x) /\ Q /\ (!x. R x)

> SIMP_CONV std_ss [GSYM RIGHT_EXISTS_AND_THM, GSYM LEFT_FORALL_IMP_THM]
  ‘‘!y. (P y /\ (?x. y = SUC x)) ==> Q y‘‘
val it = |- (!y. P y /\ (?x. y = SUC x) ==> Q y) <=>
  !x. P (SUC x) ==> Q (SUC x)
```

231 / 247

230 / 247



232 / 247

Context



- a great feature of the simplifier is that it can use context information
- by default simple context information is used like
 - ▶ the precondition of an implication
 - ▶ the condition of `if-then-else`
- one can configure which context to use via congruence rules
 - ▶ by using `CONJ_ss` one can easily use context of conjunctions
 - ▶ warning: using `CONJ_ss` can be slow
 - ▶ using other contexts is outside the scope of this lecture
- using context often simplifies proofs drastically
 - ▶ using `Rewrite` lib, often a goal needs to be split and a precondition moved to the assumptions
 - ▶ then `ASM_REWRITE_TAC` can be used
 - ▶ with `SIMP_TAC` there is no need to split the goal

233 / 247

Conditional Rewriting I



- perhaps the most powerful feature of the simplifier is that it supports conditional rewriting
- this means it allows **conditional** rewrite theorems of the form $\text{|- cond} \implies (t1 = t2)$
- if the simplifier finds a term `t1'` it can rewrite via `t1 = t2` to `t2'`, it tries to discharge the assumption `cond'`
- for this, it calls itself recursively on `cond'`
 - ▶ all the decision procedures and all context information is used
 - ▶ conditional rewriting can be used
 - ▶ to prevent divergence, there is a limit on recursion depth
- if `cond' = T` can be shown, `t1'` is rewritten to `t2'`
- otherwise `t1'` is not modified

235 / 247

Context Examples



```
> SIMP_CONV std_ss [] ``((1 = []) ==> P 1) /\ Q 1``
val it = |- ((1 = []) ==> P 1) /\ Q 1 <=>
          ((1 = []) ==> P []) /\ Q 1

> SIMP_CONV arith_ss [] ``if (c /\ x < 5) then (P c /\ x < 6) else Q c``
val it = |- (if c /\ x < 5 then P c /\ x < 6 else Q c) <=>
          if c /\ x < 5 then P T else Q c:

> SIMP_CONV std_ss [] ``P x /\ (Q x /\ P x ==> Z x)``
Exception- UNCHANGED raised

> SIMP_CONV (std_ss++boolSimps.CONJ_ss) [] ``P x /\ (Q x /\ P x ==> Z x)``
val it = |- P x /\ (Q x /\ P x ==> Z x) <=> P x /\ (Q x ==> Z x)
```

234 / 247

Conditional Rewriting Example



- consider the conditional rewrite theorem $!1 n. \text{LENGTH } l \leq n \implies (\text{DROP } n l = [])$
- let's assume we want to prove $(\text{DROP } 7 [1;2;3;4]) ++ [5;6;7] = [5;6;7]$
- we can without conditional rewriting
 - ▶ show $\text{|- LENGTH } [1;2;3;4] \leq 7$
 - ▶ use this to discharge the precondition of the rewrite theorem
 - ▶ use the resulting theorem to rewrite the goal
- with conditional rewriting, this is all automated

```
> SIMP_CONV list_ss [DROP_LENGTH_TOO_LONG]
  `` (DROP 7 [1;2;3;4]) ++ [5;6;7] ``
val it = |- DROP 7 [1; 2; 3; 4] ++ [5; 6; 7] = [5; 6; 7]
```
- conditional rewriting often shortens proofs considerably

236 / 247

Conditional Rewriting Example II



Proof with Rewrite

```
prove ('(DROP 7 [1;2;3;4]) ++ [5;6;7] = [5;6;7] ',
'DROP 7 [1;2;3;4] = []' by (
  MATCH_MP_TAC DROP_LENGTH_TOO_LONG >>
  REWRITE_TAC[LENGTH] >>
  DECIDE_TAC
) >>
ASM_REWRITE_TAC[APPEND])
```

Proof with Simplifier

```
prove ('(DROP 7 [1;2;3;4]) ++ [5;6;7] = [5;6;7] ',
ASM_SIMP_TAC list_ss [])
```

237 / 247

Conditional Rewriting Pitfalls I



- if the pattern is too general, the simplifier becomes very slow
- consider the following, trivial but hopefully useful example

Looping example

```
> val my_thm = prove ('~P ==> (P = F)', PROVE_TAC[])
> time (SIMP_CONV std_ss [my_thm]) 'P1 /\ P2 /\ P3 /\ ... /\ P10'
runtime: 0.84000s, gctime: 0.02400s, systime: 0.02400s.
Exception- UNCHANGED raised

> time (SIMP_CONV std_ss []) 'P1 /\ P2 /\ P3 /\ ... /\ P10'
runtime: 0.00000s, gctime: 0.00000s, systime: 0.00000s.
Exception- UNCHANGED raised
```

- ▶ notice that the rewrite is applied at plenty of places (quadratic in number of conjuncts)
- ▶ notice that each backchaining triggers many more backchainings
- ▶ each has to be aborted to prevent diverging
- ▶ as a result, the simplifier becomes very slow
- ▶ incidentally, the conditional rewrite is useless

239 / 247

Conditional Rewriting II



- conditional rewriting is a very powerful technique
- decision procedures and sophisticated rewrites can be used to discharge preconditions without cluttering proof state
- it provides a powerful search for theorems that apply
- however, if used naively, it can be slow
- moreover, to work well, rewrite theorems need to of a special form

238 / 247

Conditional Rewriting Pitfalls II



- good conditional rewrites $|-\ c \implies (l = r)$ should mention only variables in c that appear in l
- if c contains extra variables $x_1 \dots x_n$, the conditional rewrite engine has to search instantiations for them
- this mean that conditional rewriting is trying discharge the precondition $?x_1 \dots x_n. c$
- the simplifier is usually not able to find such instances

Transitivity

```
> val P_def = Define 'P x y = x < y';
> val my_thm = prove ('!x y z. P x y ==> P y z ==> P x z', ...)
> SIMP_CONV arith_ss [my_thm] 'P 2 3 /\ P 3 4 ==> P 2 4'
Exception- UNCHANGED raised

(* However transitivity of < build in via decision procedure *)
> SIMP_CONV arith_ss [P_def] 'P 2 3 /\ P 3 4 ==> P 2 4'
val it = |- P 2 3 /\ P 3 4 ==> P 2 4 <=> T:
```

240 / 247

Conditional vs. Unconditional Rewrite Rules



- conditional rewrite rules are often much more powerful
- however, Rewrite lib does not support them
- for this reason there are often two versions of rewrite theorems

drop example

- DROP_LENGTH_NIL is a useful rewrite rule:
`|- !l. DROP (LENGTH l) l = []`
- in proofs, one needs to be careful though to preserve exactly this form
 - ▶ one should not (partly) evaluate LENGTH l or modify l somehow
- with the conditional rewrite rule DROP_LENGTH_TOO_LONG one does not need to be as careful
`|- !l n. LENGTH l <= n ==> (DROP n l = [])`
 - ▶ the simplifier can use simplify the precondition using information about LENGTH and even arithmetic decision procedures

241 / 247

Example Once



```
> SIMP_CONV pure_ss [Once ADD_COMM] ‘‘a + b = c + d’’
val it = |- (a + b = c + d) <=> (b + a = c + d)

> SIMP_CONV pure_ss [Ntimes ADD_COMM 2] ‘‘a + b = c + d’’
val it = |- (a + b = c + d) <=> (a + b = c + d)

> SIMP_CONV pure_ss [ADD_COMM] ‘‘a + b = c + d’’
Exception- UNCHANGED raised

> ONCE_REWRITE_CONV [ADD_COMM] ‘‘a + b = c + d’’
val it = |- (a + b = c + d) <=> (b + a = d + c)

> REWRITE_CONV [ADD_COMM] ‘‘a + b = c + d’’
... diverges ...
```

243 / 247

Special Rewrite Forms



- some theorems given in the list of rewrites to the simplifier are used for special purposes
- there are marked functions that mark these theorems
 - ▶ Once : thm -> thm use given theorem at most once
 - ▶ Ntimes : thm -> int -> thm use given theorem at most the given number of times
 - ▶ AC : thm -> thm -> thm use given associativity and commutativity theorems for AC rewriting
 - ▶ Cong : thm -> thm use given theorem as a congruence rule
- these special forms are easy ways to add this information to a simpset
- it can be directly set in a simpset as well

242 / 247

Stateful Simpset



- the simpset `srw_ss()` is maintained by the system
 - ▶ it is automatically extended by new type-definitions
 - ▶ theories can extend it via `export_rewrites`
 - ▶ libs can augment it via `augment_srw_ss`
- the stateful simpset contains many rewrites
- it is very powerful and easy to use

Example

```
> SIMP_CONV (srw_ss()) [] ‘‘case [] of [] => (2 + 4)’’
val it = |- (case [] of [] => 2 + 4 | v::v1 => ARB) = 6
```

244 / 247

Adding Own Conversions



- it is complicated to add arbitrary decision procedures to a simpset
- however, adding simple conversions is straightforward
- a conversion is described by a `stdconvdata` record

```
type stdconvdata = {
  name: string, (* name for debugging *)
  pats: term list, (* list of patterns, when to try conv *)
  conv: conv (* the conversion *)
}
```
- use `std_conv_ss` to create simpset-fragment

Example

```
val WORD_ADD_ss =
  simpLib.std_conv_ss
  {conv = CHANGED_CONV WORD_ADD_CANON_CONV,
   name = "WORD_ADD_CANON_CONV",
   pats = ['words$word_add (w:'a word) y'']}
```

245 / 247

Summary Simplifier



- the simplifier is HOL's main workhorse for automation
- it is very powerful
- conditional rewriting very powerful
 - ▶ here only simple examples were presented
 - ▶ experiment with it to get a feeling
- many advanced features not discussed here at all
 - ▶ using congruence rules
 - ▶ writing own decision procedures
 - ▶ rewriting with respect to arbitrary congruence relations

247 / 247

Discussion on Stateful Simpset



- the stateful simpset is very powerful and easy to use
- however, results are hard to predict
- proofs using it unwisely are hard to maintain
- the stateful simpset can expand too much
 - ▶ bigger, harder to read proof states
 - ▶ high level arguments become hard to see
- whether to use the stateful simpset depends on personal proof style
- **I advise at the beginning to not use `srw_ss`**
- once you got a good intuition on how the simplifier works, make your own choice

246 / 247