



# Interactive Theorem Proving using Isabelle/HOL

## Session 1

René Thiemann

Department of Computer Science

- Organization
- Motivation and Introduction
- Higher-Order Logic
- First Steps with Isabelle/HOL

RT (DCS @ UIBK)

session 1

2/32

## Organization

### Course Info (VU 3)

- IV-Number: 703315
- instructor: René Thiemann
- VU: attendance mandatory, shared lecture and proseminar
- website: <http://cl-informatik.uibk.ac.at/teaching/ss26/itpIsa> (slides and Isabelle files are available online)
- consultation hours: Tuesday 10:15 – 11:15 in 3M09 (ICT building)

### Grading

- weekly exercises (50 %)
- project (50 %)
  - finished projects must be submitted through OLAT
  - deadline: August 1

Organization



RT (DCS @ UIBK)

session 1

4/32

## The Exercises

- weakly exercise will be handed out each Tuesday
- mark and upload solved exercises in [OLAT](#) by Monday, 6am
- solutions will be discussed at start of each VU

## The Project

- list of potential formalization projects will be made available
- projects will be assigned on April 20
- work alone or in small groups (depending on specific project)
- projects have to be finished before August 1
- be able to answer project related questions

## Course Information

- courses involving **interactive theorem proving** provided by CL
- VU3 Interactive Theorem Proving (2024, Cezary Kaliszyk)
  - different proof assistants based on different logics
  - covers foundations of interactive theorem provers
- VU3 Semantics of Programming Languages (2025, Manuel Eberl)
  - focus is on programming languages
  - verification part is performed using proof assistant Isabelle/HOL
- VU3 Interactive Theorem Proving using Isabelle/HOL (2024, 2026, this course)
  - single proof assistant (Isabelle), one logic (HOL: higher-order logic)
  - how to use Isabelle/HOL for a variety of verification tasks
  - practical course to obtain hands-on experience

↔ each course contains novel techniques that are not covered by the other courses

## Literature

- Isabelle documentation  
(<https://isabelle.in.tum.de>)
  - Tobias Nipkow: Programming and Proving in Isabelle/HOL
  - ...
- Tobias Nipkow and Gerwin Klein: Concrete Semantics with Isabelle/HOL  
(<http://www.concrete-semantics.org>)

## Motivation and Introduction

## Motivation

- bugs in unverified software and hardware may have severe consequences
- these can be costly (crash of Ariane, Pentium bug, ...)
- or fatal (control software of aircrafts, medical devices, ...)

## One Solution: Formal Verification

Proving program correctness with respect to given **formal specification**

## State of the Art in Formal Verification

- verified operating system kernel **seL4**, ACM Software System Award 2022 (no arithmetic exceptions, deadlocks, buffer overflows, ...)
- verified **Nitro Isolation Engine** of Amazon Web Services, 2025 (workloads are isolated from each other and AWS operators)
- verification of Kepler conjecture: optimal density of packing spheres is  $\pi/\sqrt{18}$
- 99 % of a **top 100 mathematical theorems** list has been verified

<https://www.cs.ru.nl/~freek/100/>

## Formal Verification via Theorem Proving

- various logics to write formal specifications
  - propositional logic, SMT, first-order logic
  - higher-order logic (HOL), calculus of inductive constructions (CIC)
- logics differ in expressivity and automation
  - automated theorem proving (ATP)
    - push button verification (SAT solver, SMT solver, first-order resolution prover, ...)
    - limited expressivity
  - **interactive theorem proving** (ITP)
    - proofs are developed manually (within a **proof assistant**)
    - less automation
    - high expressivity (mathematical theorems, program verification, ...)
- **Isabelle** is a popular proof assistant (besides Rocq, Lean, PVS, ...) that supports HOL
- **HOL** is sweet spot between expressivity and automation

## What is a Proof Assistant?

- combination of automated theorem prover (ATP) and proof checker
- structure of proofs is designed manually, some subproofs are found automatically
- all proofs are **checked** rigorously, e.g., in an **LCF-style** proof assistant such as Isabelle

## Examples

- automatic methods
  - logical reasoning (e.g., linear arithmetic, first-order reasoning)
  - equational reasoning
  - ...
- manual steps
  - provide intermediate statements or auxiliary lemmas
  - perform induction or case analysis
  - ...
- proof checking
  - check that all cases have been covered, that inference rules are applied correctly, ...

## What is LCF-Style?

- theorems are represented by abstract data type (**thm**)
- set of (basic) logical inferences provided as interface (**trusted kernel**)
- no other ways to create theorem (value of type **thm**) due to abstraction barrier and strong type system

## Example

- kernel provides functions `assume : cterm -> thm` and `implies_elim : thm -> thm -> thm`
- implements inference rules

$$\frac{}{A \vdash A} \quad \frac{\Gamma \vdash A \implies B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}$$

- if desired, inspect implementation of kernel functions to increase trust

## History of Isabelle

- 1986: creation of Isabelle, a proof assistant for various logics (University of Cambridge, Technische Universität München)
- 1993: support for higher-order logic: Isabelle/HOL
- 1996: human-readable proof language: Isabelle/Isar
- 2011: prover IDE: Isabelle/jEdit
- since 2004: archive of formal proofs (a library of formalized proofs with currently 578 authors and 312 100 lemmas)



Tobias Nipkow



Lawrence Paulson



Makarius Wenzel

## Higher-Order Logic

## Higher-Order Logic

- we assume knowledge of first-order logic
- higher-order logic has two main differences to first-order logic
  - terms are **typed**
  - **quantification for each type**, including function-types
- higher-order logic will be used both to specify functional programs as well as logical specifications

HOL = Functional Programming + Logic

## Types in HOL

- very similar to Haskell types
  - basic types for booleans, natural numbers, integers, ...
  - type variables
  - function types
  - algebraic data types: lists, trees, pairs, tuples, ...
- in Isabelle
  - function types have form  $input\_type \Rightarrow output\_type$
  - $ty_1 \Rightarrow ty_2 \Rightarrow ty_3$  is the same as  $ty_1 \Rightarrow (ty_2 \Rightarrow ty_3)$
  - type variables are written with a leading prime: 'a, 'b, ...
  - most type constructors are written postfix: 'a list, nat list list, ...
  - tuples are encoded as nested pairs: 'a × 'b × 'c is the same as 'a × ('b × 'c)
  - new algebraic data types can be created via `datatype` as in
 

```
datatype ('a, 'b)tree = Leaf 'a | Node "('a, 'b)tree" 'b "('a, 'b)tree"
```
  - type synonyms (abbreviations) can be created via `type_synonym` as in
 

```
type_synonym ('a)special_tree = "(nat × 'a, 'a list)tree"
type_synonym string = "char list"
```

## Inner and Outer Syntax

- Isabelle contains various languages
  - implementation languages Scala and ML
  - language to write Isabelle theories: **outer syntax**
    - add a function definition
    - add a type definition
    - state a lemma
    - perform a proof step
    - ...
  - language to specify terms and types: **inner syntax**
    - provide defining equations of a function
    - provide definition of type
    - provide a formula that describes the lemma
    - instantiate some inference rule, e.g., provide a term as existential witness
    - ...
- important
  - content of inner syntax needs to be surrounded by double-quotes
  - exception: if content is atomic, then double-quotes can be dropped

## Example for Outer and Inner Syntax: Data Type Definitions

- general definition is specified by outer syntax:
 

```
datatype ('a1, ..., 'an) ty =
  C1 ty11 ... ty1k1 | ... | Cm tym1 ... tymkm
```
- each  $ty_{ij}$  is a type, i.e., something that is specified by inner syntax
- consider concrete data type definition from previous slide
 

```
datatype ('a, 'b)tree = Leaf 'a | Node "('a, 'b)tree" 'b "('a, 'b)tree"
```

  - the first argument of Node is `"('a, 'b)tree"` – double-quotes required
  - the second argument of Node is `'b` – double-quotes not required
  - further examples
    - both `nat` and `"nat"` are okay
    - `"nat ⇒ bool"`
    - `"nat list"`
    - `"(nat × 'a) list"`
  - once we are inside inner syntax, no further double-quotes are allowed:
 `"(nat × 'a) list"` is not permitted

## Difference Between Types in Haskell and in HOL

- although HOL types look similar to Haskell types there are two important differences
- data type definitions in Isabelle/HOL do not include infinite applications of constructors
  - consider `datatype 'a list = Nil | Cons 'a "'a list"`
  - in Haskell, lists can be infinite, e.g., `ones = Cons 1 ones`
  - in Isabelle/HOL, only finite lists are covered by type `'a list`
- **all types in HOL must be inhabited**
  - for each `datatype` invocation, Isabelle internally checks that at least one term of the new type can be created, and if not, the new type is not accepted
  - example: `datatype foo = Bar foo` is refused

## Terms in HOL

- terms in Isabelle/HOL are similar to Haskell terms, they include
  - literals: `0`, `5`, `"hello"`, `CHR 'c'`, ...
  - variables: free `x`, `y`, `xs`, ... or bound `x`, `y`, `xs`, ...
  - constants: `True`, `False`, `Nil`, `Cons`, `(V)`, `(Λ)`, `(¬)`, `(⟶)`, `(=)`, `(<)`, `(+)`, `map`, ...
  - application:  $t_1 t_2$  – multiple arguments  $t_1 t_2 t_3$  are encoded as  $(t_1 t_2) t_3$
  - $\lambda$ -abstractions:  $\lambda x. t$  – multiple arguments  $\lambda x y. t$  are encoded as  $\lambda x. (\lambda y. t)$
  - type constraints:  $t :: ty$
  - Isabelle/HOL provides further syntactic conveniences like if-then-else, let, case, infix-syntax, special syntax for lists and quantifiers, ...
- terms are typed, Isabelle performs type inference and type checking
- **HOL-formulas are just terms of type `bool`**
- example terms
  - `map (λ x :: nat. x + 1) [1, 3]` is a term with type `nat list`
  - `map f (Cons x xs) = Cons (f x) (map f xs)` might be a defining equation of `map`
  - `(x :: nat) + (y + z) = (x + y) + z ∧ x + y = y + x :: bool` states that addition of natural numbers is associative and commutative

## Quantors and Equality in HOL

- unlike in Haskell, **equality is available for all types**
- two consequences
  - equality is not necessarily executable
  - quantors are not primitive in HOL, but can be encoded
- example
  - define universal quantification as a function `All :: ('a ⇒ bool) ⇒ bool` via `definition "All P = (P = (λ x. True))"`
  - $\forall$ -quantifier is nothing else than syntactic sugar, e.g. `∀ x. P x y` is syntax for `All (λ x. P x y)`
  - properties of universal quantifiers (introduction and elimination rules) can be derived  $\hookrightarrow$  we will work with these derived properties and ignore the internal definition
- facts
  - Isabelle/HOL contains only very few axiomatized types and constants (bool and some infinite type,  $(\longrightarrow)$ ,  $(=)$  and `The`, `Eps :: ('a ⇒ bool) ⇒ 'a`)
  - all other types and constants are defined on top of these
  - we won't cover the details of these foundations in this course

## Examples Beyond First Order Logic

```
(* well-foundedness of a binary relation can be expressed *)
type_synonym 'a rel = "'a ⇒ 'a ⇒ bool"
definition "well_founded (R :: 'a rel)
  = (¬ (∃ f :: nat ⇒ 'a . ∀ n :: nat. R (f n) (f (n + 1))))"

(* the transitive closure of a relation can be expressed *)
definition "trans_cl (R :: 'a rel) a b
  = (∃ (f :: nat ⇒ 'a) (n :: nat).
    f 0 = a ∧ f n = b ∧ n ≠ 0 ∧
    (∀ i. i < n → R (f i) (f (i + 1))))"

lemma "well_founded (trans_cl R) = well_founded R" oops

(* induction on natural numbers is sound *)
lemma "∀ P :: nat ⇒ bool.
  P 0 → (∀ n. P n → P (n + 1)) → (∀ n. P n)" oops
```

## Color-Codes of Isabelle

- **keyword** a keyword of outer syntax
- **command** a command of outer syntax
- **const** a constant that has been defined before
- **free** a free variable
- **bound** a bound variable (of  $\lambda$  or quantifier)
- **fixed** a fixed variable (e.g., after  $\forall$ -introduction in a proof)
- colors help to identify mistakes, e.g. in `definition "select_first fst _ = fst"` the black color of `fst` indicates that `fst` is an already defined constant (and not a bound variable `fst`), so that a name clash needs to be resolved
- at the time of a definition, the used name is free (**name**), only afterwards it turns to black (**name**)
- **free** variables in lemmas are implicitly universally quantified (and can be instantiated after the lemma has been proven)

## Functional Programming in HOL

- functional programs can be written similarly to Haskell
- already seen: type definitions
- new: function definitions
- **non-recursive** function (or constant) definitions
  - outer syntax: `definition name :: ty where eqn` or just `definition eqn`
  - `eqn` is a boolean term of the shape `name x1 ... xn = t`
  - important: often `t` needs to be put in parenthesis `definition "sorted_triple x y z = (x ≤ y ∧ y ≤ z)"`
- **recursive** function definitions
  - outer syntax: `fun name :: ty where eqn1 | ... | eqnm`
  - each `eqni` is a boolean term of the shape `name pat1 ... patn = t`
  - example `fun append :: "'a list ⇒ 'a list ⇒ 'a list" where "append Nil ys = ys" | "append (Cons x xs) ys = Cons x (append xs ys)"`

## Function Definitions in Isabelle/HOL

- syntactic differences to Haskell
  - let-expressions are of form `let x1 = t1; ... ; xn = tn in t`
  - case-expressions are of form `case t of pat1 ⇒ t1 | ... | patn ⇒ tn`
  - let, case, and if-then-else often have to be surrounded by parenthesis
  - let-expressions are sequential and non-recursive: `ti` may not refer to `xi`, ..., `xn`
  - no local recursive function definitions
  - no restriction to executable functions:
 

```
fun f where "f P = (if (∀ x :: nat. P x) then 0 else 1)"
```
- semantic difference to Haskell
  - functions defined by `fun` have to be terminating
  - if Isabelle is not able to prove termination, a function definition is not accepted

## First Steps with Isabelle/HOL

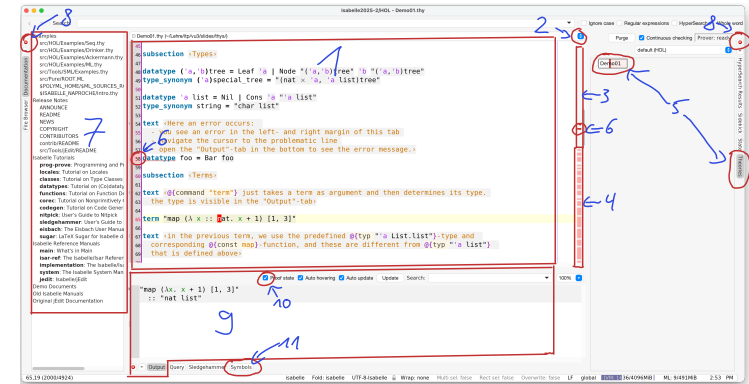
## Isabelle 2025-2 on Your Machine

- download and follow installation instructions available at <https://isabelle.in.tum.de>
- from now on prefix "\$" indicates bash prompt
- start Isabelle via `$ isabelle jedit (optional: File.thy)`
- in case `isabelle` is not found, add `$ISABELLE_HOME/bin/` to your `PATH` where `ISABELLE_HOME` is the installation directory of Isabelle 2025-2 (default depends on operating system)

## Demo

`$ isabelle jedit Demo01.thy`

## Isabelle/jEdit – Overview of User Interface



## Explanation of Previous Slide

1. main text area
2. switch between different theories
3. processed part of theory
4. unprocessed part of theory
5. progress indicator of several theories
6. indication of problem
7. documentation
8. click to close left or right panel
9. main output window
10. enable to view proof state in output (and not just errors)
11. symbol panel for information on special symbols

## Theory Files – General Structure

```
theory T
  imports T1 ... Tn
begin
(* definitions, theorems and proofs *)
...
end
```

### Notes

- store theory T in file T.thy
- definitions and theorems from theories T<sub>1</sub>, ..., T<sub>n</sub> available after `begin`
- new definitions, theorems and proofs go between `begin` and `end`
- qualify identifiers by theory name (like T.f) to disambiguate names
- theory Main is collection of basic definitions (like Haskell's `PreLude`) and should always be imported

## Entering Special Symbols

- aim: enter symbols like  $\forall$ ,  $\times$ ,  $\lambda$ , ...
- four methods
  - switch to Symbols-panel in Isabelle/jEdit, find and click on symbol;
  - important: hovering over symbol will reveal internal name and abbreviations
  - enter internal name prefixed by backslash and use auto-completion via `TAB`;  
example: `\f o r` will result in `\forall`, i.e.,  $\forall$
  - enter abbreviation followed by `TAB`, e.g.,  $\forall$  is also obtained via `! TAB`
  - some abbreviations have an auto completion where no `TAB` is required, e.g., `/ \` will immediately result in  $\wedge$

## Frequently Used Symbols

symbol	internal	auto completion	abbreviations
$\lambda$	<code>\&lt;lambda&gt;</code>		<code>%</code>
$\Rightarrow$	<code>\&lt;Rightarrow&gt;</code>	<code>= &gt;</code>	<code>. &gt;</code>
$\neg$	<code>\&lt;not&gt;</code>		<code>~</code>
$\wedge$	<code>\&lt;and&gt;</code>	<code>/ \</code>	<code>&amp;</code>
$\vee$	<code>\&lt;or&gt;</code>	<code>\ /</code>	<code> </code>
$\longrightarrow$	<code>\&lt;longrightarrow&gt;</code>	<code>- - &gt;</code>	<code>. &gt;</code>
$\longleftrightarrow$	<code>\&lt;longleftrightarrow&gt;</code>	<code>&lt; - - &gt;</code>	<code>&lt; &gt;</code>
$\forall$	<code>\&lt;forall&gt;</code>		<code>!</code> and <code>A L L</code>
$\exists$	<code>\&lt;exists&gt;</code>		<code>?</code> and <code>E X</code>
$\times$	<code>\&lt;times&gt;</code>	<code>&lt; * &gt;</code>	
$\leq$	<code>\&lt;le&gt;</code>	<code>&lt; =</code>	