



# Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

Jakob von Raumer, Sebastian Ullrich | SS 2021



THEOREM PROVER

[TODO: Organisatorisches]

# Outline

- What is interactive theorem proving?
- Terms and types
- The Curry-Howard correspondence
- The structure of a Lean file
- Some basic types in Lean
- Propositional logic in Lean

# What is Interactive Theorem Proving (ITP)?

- Language and IDE supporting the user in writing *formal proofs* about certain structures which are then *verified* by the computer.
- ITP is *not* automated theorem proving (ATP)! User input is usually required to guide the system.
- Formal foundation of Lean: Type theory, inductive types (Calculus of Inductive Constructions, CIC).

# Interactive Theorem Proving – How is it Useful?

- Verify proofs independently of an academic review system.
- Write programs and verify them – in the same language!

# A Cherry-Picked List of Theorem Proving Projects

2005 *Gonthier & Werner*: four color theorem [Coq]

2014 *Hales et al*: Kepler conjecture [HOL Light/Isabelle]

2014+ *Leroy et al*: CompCert, a verified C compiler [Coq]

2020 *Han & van Doorn*: independence of the continuum hypothesis [Lean 3]

# On Lean Versions

- This course is about *Lean 4*, a rewritten (in Lean!) and greatly extended implementation of Lean
- Lean 4 is **not** backwards-compatible with Lean 3, the latest stable release
- Most information you can find online will be about Lean 3
- Lean's sizable *mathlib* is not available for Lean 4 yet, but we won't need it
- If you find bugs or room for improvement, do tell us!

# Terms and Types

- Martin-Löf type theory (MLTT) is a very strong, static type system.
- It is based on *type judgments*:

$$t : \alpha$$

means “ $t$  is an instance of  $A$ ” for a term  $t$  and a type  $\alpha$ .

- Everything has a fixed type, even types themselves: Universes are types that contain types:

$$\begin{array}{lcl}
 t & : & \alpha \\
 \alpha & : & \text{Sort}_u \\
 \text{Sort}_u & : & \text{Sort}_{u+1} \\
 \text{Sort}_{u+1} & : & \dots
 \end{array}$$



# The Curry-Howard Correspondence



- Named after Haskell Curry and William Alvin Howard, 1969
- Paraphrasing:
  - “**Proving** in intuitionistic (constructive) logic  
is the same as  
**programming** in typed lambda calculus.”
- We prove statements by constructing instances of certain types.

# The Curry-Howard Correspondence

	Proving	Programming
$\alpha : \text{Sort}_u$	$\alpha$ is a proposition	$\alpha$ is a data type
$t : \alpha$	$t$ is a proof of $\alpha$	$t$ is an element of $\alpha$
$\alpha \rightarrow \beta$	$\alpha$ implies $\beta$	type of functions from $\alpha$ to $\beta$
$\alpha \times \beta$	$\alpha$ and $\beta$	cartesian product of $\alpha$ and $\beta$
$\alpha + \beta$	$\alpha$ or $\beta$	disjoint sum of $\alpha$ and $\beta$

# The Difference Between Proof and Data

- Some provers do not distinguish between proof and data
- The biggest difference in Lean is:

We can treat instances  $h : p$  and  $h' : p$  for a proposition  $p$  as equal,

but

we need to distinguish instances  $x : \alpha$  and  $y : \alpha$  of a data type.

- Lean keeps propositions and data types in different universes, has aliases

$$\begin{aligned}\text{Prop} &::= \text{Sort}_0 \\ \text{Type} &::= \text{Type}_0 ::= \text{Sort}_1 \\ \text{Type}_1 &::= \text{Sort}_2 \\ &\dots \quad \dots\end{aligned}$$

- Definitions of data are (conventionally) preceded by the keyword **def**, propositional definitions by **theorem**

# The Structure of a Lean File

```
-- single-line comments start with "--"
import MyProject.Utils -- import file `MyProject/Utils.lean` from current project/dependency

def add_three_mul (x y : Nat) : Nat := -- define a function with two arguments x and y
  x + 3 * y

theorem add_three_mul_gt {x y : Nat} : add_three_mul x y >= x :=
  ...

example {y : Nat} : 5 + 3 * y >= 5 := -- an example is a theorem without a name
  add_three_mul_gt

#check add_three_mul_gt -- check the type of our theorem
```

# Propositional Logic in Lean: True and False

The canonical way to prove the proposition `True` is called `True.intro` :

```
example : True := True.intro
```

There is no way of proving false (hopefully!) but we can use the principle of “ex falso quodlibet” by using `nomatch` :

```
example (p : Prop) (fa : False) : p := nomatch fa
```

# Propositional Logic in Lean: Implication

Implication is modelled by function types. So, applying an implication is function application:

```
section
variable (p q r : Prop) -- automatically become arguments when referenced in the section

theorem modus_ponens (hpq : p → q) (hp : p) : q :=
  hpq hp -- function application is written with a space instead of parentheses!

example (hpqr : p → q → r) (hp : p) (hq : q) : r := -- → has implicit parentheses on the right
  hpqr hp hq -- function application has implicit parentheses on the left

example (hnp : True → p) : p :=
  hnp True.intro
end
```

We can input  $\rightarrow$  writing `\to`.

Convention: We write *curried* functions:  $\alpha \rightarrow \beta \rightarrow \gamma$  instead of  $\alpha \times \beta \rightarrow \gamma$  or  $\alpha \wedge \beta \rightarrow \gamma$ !

# Propositional Logic in Lean: Implication

Proving an implication is done by lambda abstraction:

```
section
variable (p q r : Prop)

example : p → p → p :=
  fun hp hp' => hp

example (hq : q) : p → q :=
  fun hp => hq

example (hpq : p → q) (hqr : q → r) : p → r :=
  fun hp => hqr (hpq a)
end
```

The expression `fun a => b` is what Mathematicians write as  $a \mapsto b$ !

# Arguments vs. Implications

The following two Lean snippets are the same:

```
example (p q r : Prop) (hpq : p → q) (hqr : q → r) : p → r :=  
  fun hp => hqr (hpq hp)
```

and

```
example (p q r : Prop) : (p → q) → (q → r) → p → r :=  
  fun hpq hqr hp => hqr (hpq hp)
```

Arguments of a definition are just shorthand for iterated implications (functions)!



# Propositional Logic in Lean: Disjunction

- The disjunction of two propositions is  $p$  and  $q$  is written  $p \vee q$ .
- A case distinction on  $p \vee q$  can be done by *matching* on the proof.

```
example (p q : Prop) (hp : p) : p ∨ q := Or.inl hp

example (p q r s : Prop) (hp : p) (hs : s) : (p ∨ q) ∧ (r ∨ s) :=
  And.intro (Or.inl hp) (Or.inr hs)

example (p q r : Prop) (hpq : p ∨ q) (hpr : p → r) (hqr : q → r) : r :=
  match hpq with
  | Or.inl hp => hpr hp
  | Or.inr hq => hqr hq
```

Write  $\vee$  as `\or`.

# Propositional Logic in Lean: Conjunction

- The conjunction of two propositions  $p$  and  $q$  is written  $p \wedge q$ .
- A proof of  $p \wedge q$  is created using `And.intro`.
- A proof of  $p$  can be recovered using matching:

```
example (p q : Prop) (hpq : p ∧ q) : p :=  
  match hpq with  
  | And.intro hp hq => hp  
  
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := And.intro hp hq
```

We can input  $\wedge$  using `\and`.

# Algebraic Data Types and Propositions

Where does the strange `match` syntax come from? `Or` is defined as an algebraic data type:

```
inductive Or (p q : Prop) : Prop where
| inl (hp : p) : Or p q
| inr (hq : q) : Or p q
```

Each line starting with `|` is a *constructor* to `Or p q`. It states one way to construct an instance of the proposition.

`And` can be defined by the same mechanism, but only has *one* constructor with *multiple* parameters:

```
inductive And (p q : Prop) : Prop where
| intro (hp : p) (hq : q) : And p q
```

# Algebraic Data Types and Propositions

True and False are algebraic data types as well, with one and zero constructors, respectively!

```
inductive True : Prop where
  | intro : True

inductive False : Prop where
```

The term 'nomatch' lets us use the fact that 'False' does not have any constructors.

# Algebraic Data Types and Propositions

And and Or are given more convenient *infix notations* as we've seen.

```
infixr:35 " ^ " => And  
infixr:30 " v " => Or
```

Both notations associate to the right, with And binding more tightly, i.e.

$$a \wedge b \wedge c \vee d \equiv (a \wedge (b \wedge c)) \vee d \equiv \text{Or } (\text{And } a \ (\text{And } b \ c)) \ d$$

# Algebraic Data Types and Propositions

Data types can be defined the same way, like this type of days of the week:

```
inductive Weekday : Type where
| monday : Weekday
| tuesday : Weekday
| wednesday : Weekday
| thursday : Weekday
| friday : Weekday
| saturday : Weekday
| sunday : Weekday
```

This definition then enables us to *match* on variables of this data type:

```
def dayNumber (w : Weekday) : Nat :=
  match w with
  | Weekday.monday => 0
  | Weekday.tuesday => 1
  | Weekday.wednesday => 2
  | Weekday.thursday => 3
  | Weekday.friday => 4
  | Weekday.saturday => 5
  | Weekday.sunday => 6
```

Lean will throw an error whenever we forget one of the constructors!

# Placeholders

Often, you will not write a proof in one go, but instead compose the proof term bit by bit. In this situation we can use the underscore `_` as a placeholder. Lean will try to infer its type and display it when hovering over the underscore as well as in the *Infoview*:

```
example (f :  $\alpha \rightarrow \beta$ ) (a :  $\alpha$ ) :  $\beta$  :=  
  f _ -- shows that we need to provide a term of type  $\alpha$ 
```

# Let-Expressions

To create a local variable which abbreviates another expressions, we can use **let** :

```
example (f :  $\beta \rightarrow \beta \rightarrow \gamma$ ) (g :  $\alpha \rightarrow \beta$ ) (a :  $\alpha$ ) :  $\gamma$  :=  
  let b := g a  
  f b b
```

For longer proofs, you will find that using **let** is a way to make them more readable and concise!  
In some cases it might be beneficial to first fix the type of a let-expression: The corresponding line above could be also annotated as **let** b :  $\beta$  := g a .