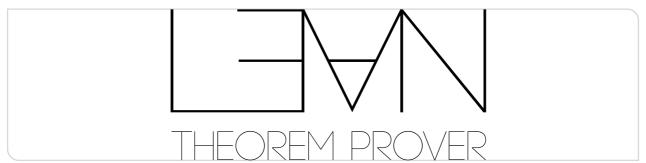




#### Theorembeweiserpraktikum

Anwendungen in der Sprachtechnologie

Jakob von Raumer, Sebastian Ullrich | SS 2021



[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





- 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 [Cog]
- 2014 Hales et al: Kepler conjecture [HOL Light/Isabelle]
- 2014+ Leroy et al: CompCert, a verified C compiler [Cog]
  - 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  $\alpha$ " for a term t and a type  $\alpha$ .

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

 $t : \alpha$  $\alpha$  : Sort,,  $\mathsf{Sort}_u$  :  $\mathsf{Sort}_{u+1}$   $\mathsf{Sort}_{u+1}$  : . . .

#### 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
$\alpha : Sort_{u}$	$\alpha$ is a proposition	$\alpha$ is a data type
$t: \alpha$	$t$ is a proof of $\alpha$	t is an element of $lpha$
$\alpha \to \beta$	lpha implies $eta$	type of functions from $lpha$ to $eta$
$\alpha \times \beta$	lpha and $eta$	cartesian product of $lpha$ and $eta$
$\alpha + \beta$	lpha or $eta$	disjoint sum of $lpha$ and $eta$





- 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

 $\mathsf{Prop} \coloneqq \mathsf{Sort}_0$   $\mathsf{Type} \coloneqq \mathsf{Type}_0 \coloneqq \mathsf{Sort}_1$   $\mathsf{Type}_1 \coloneqq \mathsf{Sort}_2$ 

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



#### The Structure of a Lean File



#### 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 \rightarrow q) (hp : p) : q := hpq hp -- function application is written with a space instead of parentheses!

example (hpqr : p \rightarrow q \rightarrow r) (hp : p) (hq : q) : r := -- \rightarrow has implicit parentheses on the right hpqr hp hq -- function application has implicit parentheses on the left example (htp : True \rightarrow p) : p := htp True.intro end
```

We can input  $\rightarrow$  writing \to.

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



#### **Propositional Logic in Lean: Implication**

Proving an implication is done by lambda abstraction:

The expression fun a  $\Rightarrow$  b is what Mathematicians write as  $a \mapsto b!$ 



#### **Arguments vs. Implications**

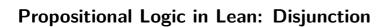
The following two Lean snippets are the same:

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

#### and

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

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





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

```
example (p q : Prop) (hp : p) : p ∨ q := 0r.inl hp

example (p q r s : Prop) (hp : p) (hs : s) : (p ∨ q) ∧ (r ∨ s) :=
  And.intro (0r.inl hp) (0r.inr hs)

example (p q r : Prop) (hpq : p ∨ q) (hpr : p → r) (hqr : q → r) : r :=
  match hpq with
  | 0r.inl hp => hpr hp
  | 0r.inr hq => hqr hq
```

Write v as \or.



## **Propositional Logic in Lean: Conjunction**

- The conjunction of two propositions p and q is written  $p \land q$ .
- A proof of p ∧ 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  $\Lambda$  using \and.



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  $\mid$  is a *constructor* to 0r 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
```



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.



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

```
infixr:35 " \ \ " => And infixr:30 " \ \ " => 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 0r (And a (And b c)) d
```





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 => 3
  | Weekday.thursday => 4
  | Weekday.friday => 5
  | Weekday.saturday => 6
  | Weekday.sunday => 7
```

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





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

```
example (f : \beta \to \beta \to \gamma) (g : \alpha \to \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.