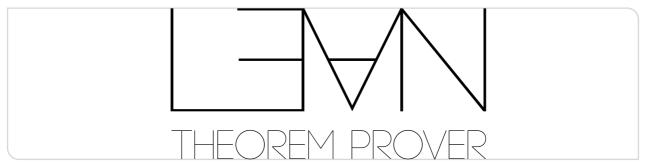




Theorembeweiserpraktikum

Introduction & Propositional Logic

Jakob von Raumer, Sebastian Ullrich | SS 2022



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.
 - Puts the burden of proof verification on the author.
 - Less prone to human error.
- Write programs and verify them in the same language!
- Alternative: Embed a different language and equip it with a formal semantics.



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* has mostly not been ported to Lean 4 yet
- 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 α " for a term t and a type α .

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

 $42 : \mathbb{N}$ \mathbb{N} : Sort₁ $Sort_1 : Sort_2$ Sort₂ : . . .

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
$\alpha : Sort_{u}$	α is a proposition	α is a data type
$t: \alpha$	t is a proof of α	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

```
-- 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 : add_three_mul x y >= x := -- x and y are implicitly universally quantified

...

example : 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 False.elim:

```
example (p : Prop) (fa : False) : p := False.elim 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.

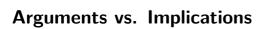
Convention: We write *curried* functions: $\alpha \to \beta \to \gamma$ instead of $\alpha \times \beta \to \gamma$ or $\alpha \wedge \beta \to \gamma$!



Propositional Logic in Lean: Implication

Proving an implication is done by lambda abstraction:

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





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)!



Propositional Logic in Lean: Disjunction

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

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

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

Write V as \or.





- The conjunction of two propositions p and q is written $p \wedge q$.
- 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 \( \lambda \) ; p :=
  match hpq with
  | And.intro hp hq => hp

example (p q : Prop) (hp : p) (hq : q) : p \( \lambda \) q := And.intro hp hq

example (p q r s : Prop) (hp : p) (hs : s) : (p \( \nabla \) q) \( \lambda \) (r \( \nabla \) s) :=
  And.intro (Or.inl hp) (Or.inr hs)
```

We can input \(\text{using \and.} \)



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!

The term **nomatch** lets us use the fact that False does not have any constructors, and False.elim is defined by using **nomatch**:

```
def False.elim (h : False) : α := nomatch h
```



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 Or (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 => 2
  | Weekday.wednesday => 3
  | 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 \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 have:

```
example (f : \beta \to \beta \to \gamma) (g : \alpha \to \beta) (a : \alpha) : \gamma := have b : \beta := g a f b bha
```

For longer proofs, you will find that using have is a way to make them more readable and concise!

Demo

Summary



type	introduction	elimination
р	⊦ p	h : p, ⊦
	"how do I prove this"	"how do I use this"
implication/function type	abstraction	application
$q\rightarrowr$	fun hq => (_ : r)	h hq
inductive type	constructor app.	matching
qVr	Or.inl hq , Or.inr hr	match h with Or.inl hq => _ Or.inr hr => _
qΛr	And.intro hq hr	match h with And.intro hq hr => _
True	True.intro	_
False ————	_	False.elim h \neq nomatch h $=$