Seminar on Computer-assisted mathematics

Judith Ludwig and Florent Schaffhauser

Heidelberg University - Summer semester 2025

Session 2 - April 24, 2025

Fundamental concepts from Lecture 1

- Curried functions.

```
Nat.add 1 2 = 1 + 2
```

- Inductive types (*e.g.* `Nat` or `Bool`).

```
inductive Prod (X : Type) (Y : Type) : Type where
| mk (x : X) (y : Y) : Prod X Y
```

- Pattern-matching on constructors to construct functions that go *out* of an inductive type.

```
def add_u : Nat × Nat → Nat :=
fun (t : Nat × Nat) ↦ match t with
| Prod.mk m n => m + n
```

Propositions as types and proofs as programs

- Propositions are a special kind of type, in which *proof irrelevance* holds (if `P : Prop` and `p q : P`, then `p = q`).
- Propositions can be defined inductively, in which case one can pattern match on constructors.

```
inductive False : Prop where

def False.elim (P : Prop) : False → P :=
fun (t : False) ↦ nomatch t
```

- A well-formed proposition is not necessarily inhabited.

```
def Fallacy : Prop := 2 + 2 = 5
#check Fallacy  -- Fallacy : Prop

def proof : Fallacy := sorry  -- declaration uses 'sorry'
#check proof  -- proof : Fallacy
```

Sum types

- The sum of two types `X` and `Y` is constructed as an inductive type with two constructors.

```
inductive Sum (X : Type) (Y : Type) : Type where
| inl (x : X) : Sum X Y   -- "injection from the left"
| inr (y : Y) : Sum X Y   -- "injection from the right"
```

- `Sum X Y` can also be denoted by `X ⊕ Y`. In set theory, the analogous notion is that of *disjoint union* of two sets.
- To define functions out of a sum, we can pattern-match on the constructors.

```
def charac_left {X : Type} {Y : Type} : X ⊕ Y → Bool :=
fun (t : X ⊕ Y) ↦  match t with
| Sum.inl (x : X) => Bool.true
| Sum.inr (y : Y) => Bool.false
```

Disjunctions

Let $\mathtt{P}$ and $\mathtt{Q}$ be propositions.

- Then the types $\mathtt{P} \to \mathtt{Q}$ and $\mathtt{P} \times \mathtt{Q}$ are propositions. But the type $\mathtt{P} \oplus \mathtt{Q}$ is *not* necessarily a proposition (for instance, the type $\mathtt{True} \oplus \mathtt{True}$ does not satisfy proof irrelevance).
- We can construct a proposition $\mathtt{P} \vee \mathtt{Q}$ by identifying any two terms in $\mathtt{P} \oplus \mathtt{Q}$. This can be viewed as taking a quotient of $\mathtt{P} \oplus \mathtt{Q}$ by the trivial equivalence relation (just one equivalence class).
- In particular, the canonical projection π from $\mathtt{P} \oplus \mathtt{Q}$ to $\mathtt{P} \vee \mathtt{Q}$ satisfies the *universal property* of a quotient: for all $f : P \oplus Q \to R$ such that, for all t, t' in $P \oplus Q$, $f(t) = f(t')$, there is a unique map $\hat{f} : P \vee Q \to R$ such that $\overline{f} \circ \pi = f$.

$$\begin{tikzcd}
P \oplus Q \arrow[d, "\pi"] \arrow[dr, "f"] & \\
P \vee Q \arrow[r, dashed, "\exists!\ \hat{f}"] & R
\end{tikzcd}$$

Fallunterscheidung

- Note that the compatibility condition for $f : P \oplus Q \to R$ is necessarily satisfied if R is a proposition. So, to prove an implication of the form `P ∨ Q → R` (where `R` is a proposition), it suffices to construct a function `f : P ⊕ Q → R`.
- This is done via pattern-matching, which in this case can also be viewed as a universal property.

$$
\begin{tikzcd}
P \arrow[dr, "\mathrm{Sum.inl}"'] \arrow[drr, bend left, "f_1"] & & \\
 & P \oplus Q \arrow[r, dashed, "\exists!"] & R \\
Q \arrow[ur, "\mathrm{Sum.inr}"] \arrow[urr, bend right, "f_2"'] & &
\end{tikzcd}
$$

- So the typing system is telling us to prove an implication of the form `P ∨ Q → R` by case analysis: first assume `P` and deduce `R`, then assume `Q` and deduce `R`.

Falsity and negation

- It was a seminal insight of N. de Bruijn's (the creator of Automath[1]) that, when viewed as a type, a proposition is to be deemed proved if, and only if, the corresponding type is inhabited.
- From that point of view, the *negation* $\neg P$ of a proposition P should be defined without any reference to whether P has a proof or not. And indeed we have:

$$\neg P := (P \rightarrow \mathsf{False})$$

- So, by definition, proving $\neg P$ means proving that, given a proof of P, we can construct a proof of False, which is considered an absurdity.
- With this definition, we can prove a number of tautologies (propositions depending on other propositions and which are inhabited regardless of whether the ones they depend on are inhabited). For instance $\neg P \wedge P \rightarrow \mathrm{False}$ (special case of *modus ponens*) or $P \rightarrow \neg\neg P$ (exercise!).

[1] The first programming language equipped with a type-checking algorithm, implemented in 1967. It was followed by Mizar, due to A. Trybulec, in 1973.

Constructive vs. classical logic

- Let us now compare `(P → Q)` and `¬(P ∧ ¬Q)`. The following program provides a proof of the implication `(P → Q) → ¬(P ∧ ¬Q)`. The keyword `theorem` is used as a synonym of `def`, when the target type is a proposition.

```
theorem classical_imp {P Q : Prop} : (¬P ∨ Q) → (P → Q) :=
fun (t : ¬P ∨ Q) ↦ match t with
| Or.inl (f : P → False) => fun (p : P) ↦ False.elim (f p)
| Or.inr (q : Q)         => fun (p : P) ↦ q
```

- The reverse implication actually does not hold constructively. To prove it for all `P`, `Q`, you would need to use `¬P ∨ P`, which you get from the Law of Excluded Middle. Note that the constructive approach is more general (less axioms).
- In Lean, you can choose to work constructively or classically. In Mathlib, most proofs use classical logic in one form or another. As an exercise, you can show that the implication `(P → Q) → ¬(P ∧ ¬Q)` holds constructively but that its converse uses `¬Q ∨ Q` .

Logical equivalences

- The type of logical equivalences `P ↔ Q` is also defined inductively. Its terms are pairs $\langle$`f, g`$\rangle$ where `f : P → Q` and `g : Q → P`.

  ```
  inductive Iff (P Q : Prop) : Prop where
  | intro : (P → Q) → (Q → P) → Iff P Q
  ```

  Note that the target type of a constructor is always the inductive type that is being defined by that constructor.

- In Lean and other modern proof assistants, most (but not all) inductive types with only one constructor are passed as *structures*, which are not technically part of Martin-Löf's type theory but are useful for the implementation (they are *record types*, declared using the keyword 'structure').

  ```
  structure Iff (P Q : Prop) : Prop where
  intro :: (mp : P → Q) (mpr : Q → P)
  ```

De Morgan's laws

- A good way to manipulate these concepts is to prove De Morgan's laws, starting with the first one:

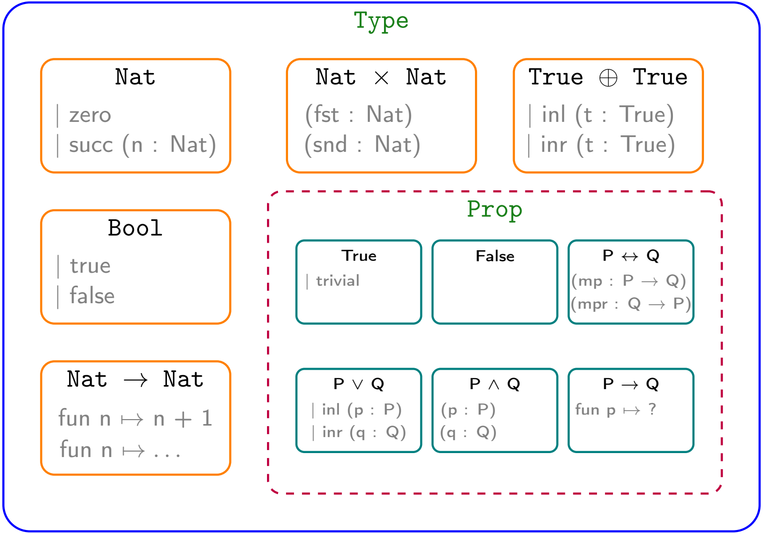
$$\neg(P \vee Q) \iff \neg P \wedge \neg Q$$

- In the second De Morgan rule, only one implication can be proved constructively, namely $\Leftarrow$.

$$\neg(P \wedge Q) \iff \neg P \vee \neg Q$$

- We will do that in a forthcoming practice file on Logic in Lean.

A universe of types!



Lean's tactic mode

- Lean's tactic mode can assist us in writing a program. To enter tactic mode, one simply puts the keyword `by` after the `:=` sign.
- This will be reflected in the infoview, which should display the *goal* of the program. This goal is what appears after the turnstile symbol $\vdash$. To see the goal in term mode, you can use the underscore symbol `_`.
- To close a goal in tactic mode, we need to use so-called *tactics*, like the `exact` tactic in the example below. Each new line must start with a tactic.

```
def a : Nat :=
by {                    -- ⊢ Nat
   exact 42             -- No goals
}
```

Note that the goal of a program is *always a type* (which may or may not be a proposition).

Basic tactics for deductive reasoning

The basic tactics we shall need are the following:

- `exact` and `apply`
- `intro` and `revert`
- `constructor`
- `cases` and `rcases`
- `left` and `right`
- `rfl`
- `exact?` and `apply?`
- `refine`

All of these are presented in our practice file on Basic Tactics.

Tactic proofs of the *modus ponens* rule

Let us use tactic mode to prove the *modus ponens* rule. The point is to see the proof state and the goal evolve after each use of a tactic, until the goal is closed.

```
theorem mp {P Q : Prop} : (P → Q) ∧ P → Q :=
by {
  intro t
  cases t with
  | intro f p =>
  exact f p
}
```

For comparison, the term mode proof would be of a similar length, but in term mode the infoview does not show anything when the goal is closed, except the absence of an error message.

```
theorem mp {P Q : Prop} : (P → Q) ∧ P → Q :=
fun t ↦ match t with | And.intro f p => f p
```

Recap and practical activity

- As we have seen in examples, a proof is a program. To prove a proposition, we have to construct a term of the relevant type. We can write a proof either in term mode or in tactic mode.
- In tactic mode, we get assistance from the kernel to help us write our proof: the infoview shows a *goal* (which is a type) and a *context* (which is a list of terms, of various types).
- Goal and context put together constitute the *proof state*. As we introduce tactics, our context and goal will change, until the goal is closed via *unification*, which occurs when a term is constructed, whose type coincides with the goal.

To manipulate the concepts seen in this lecture, you can try your hand at our Basic Tactics file!