# Dependent Types and Theorem Proving: Proving is programming in disguise

Wojciech Kołowski

May 2021

# Plan of lectures

- Lecture 1: Programming with dependent types.
- **Lecture 2: Proving theorems with dependent types.**
- Lecture 3: Differences between programming and proving.
- Lecture 4: Examples of bigger programs and longer proofs.
- Lecture 5: A deeper dive into F*.

# Boolean "logic"

- Being a programmer, you are good friends with the booleans, aren't you?
- There are two booleans, `true` and `false`.
- We can combine booleans `b` and `c` with the usual boolean functions:
- `not b` – "not b"
- `b && c` – "b and c"
- `b || c` – "b or c"

# What is a logic

- Boolean logic is not an example of what logicians call a "logic", in the sense that it is not a "logical system", but merely a type with some unary and binary functions on it.
- A logic usually consists of:
- A definition of what **propositions** we're dealing with.
- A **semantics**, which tells us what these propositions mean.
- A **proof system**, which tells us which propositions can be proven and disproven.
- A **soundness theorem** which states that propositions proven true using the proof system are semantically true.
- Optionally, there may also be a **completeness theorem** which states that all semantically true propositions can be proven.

# Propositions

- A proposition asserts that something is the case, irrespectively of whether this really is the case or not.
- Math example: "4 is a prime number."
- Software example: "For each input string $x$, if $x$ is not malformed, my program produces as output an array of length at most 10."
- Hardware example: "This circuit implements addition of 16 bit integers."
- Real world example: "It's raining or I like trains."
- **Beware! Formal logic is not very good for reasoning about the real world!**

# Propositional constants and connectives

- Propositions (usual letters: $P, Q, R$) are defined as follows:
- $\top$ – the true proposition.
- $\bot$ – the false proposition.
- $P, Q, R, \ldots$ – propositional variables.
- $\neg P$ – negation, read "not $P$".
- $P \vee Q$ – disjunction, read "$P$ or $Q$".
- $P \wedge Q$ – conjunction, read "$P$ and $Q$".
- $P \implies Q$ – implication, read "$P$ implies $Q$" or "if $P$ then $Q$".
- $P \iff Q$ – logical equivalence, read "$P$ if and only if $Q$".

## Classical logic

- Classical logic is the most widely known/taught/used logical system in the world.
- In classical logic, **we think of propositions as being either true or false.**
- Therefore, classical logic is the logic in which truth values are the booleans.
- The truth value of a propositional variable is determined by a **valuation** $v : \mathrm{Var} \to \mathrm{Bool}$.
- If $v(P) = \texttt{true}$, then $P$ is considered to be true.
- Otherwise it's considered false.

# Semantics of classical logic 1/2

- Given a valuation $v : \mathrm{Var} \to \mathrm{Bool}$, the truth value of a proposition can be determined with a recursive function $[\![-]\!] : \mathrm{Prop} \to \mathrm{Bool}$.
- $[\![\top]\!] = \texttt{true}$
- $[\![\bot]\!] = \texttt{false}$
- $[\![P]\!] = v(P)$, where $P$ is a variable.
- $[\![\neg P]\!] = \texttt{not}\ [\![P]\!]$
- $[\![P \vee Q]\!] = [\![P]\!]\ \texttt{||}\ [\![Q]\!]$
- $[\![P \wedge Q]\!] = [\![P]\!]\ \texttt{\&\&}\ [\![Q]\!]$
- $[\![P \implies Q]\!] = (\texttt{not}\ [\![P]\!])\ \texttt{||}\ [\![Q]\!]$
- $[\![P \iff Q]\!] = [\![P]\!]\ \texttt{==}\ [\![Q]\!]$

## Semantics of classical logic 2/2

- $P$ is satisfiable when $[\![P]\!] = \texttt{true}$ for some valuation.
- $P$ is falsifiable when $[\![P]\!] = \texttt{false}$ for some valuation.
- $P$ is a tautology when $[\![P]\!] = \texttt{true}$ for all valuations.

## Example

- Example: the proposition $P \implies Q$ is satisfiable (for $v(P) = \mathtt{true}, v(Q) = \mathtt{true}$).
- It is also falsifiable (for $v(P) = \mathtt{true}, v(Q) = \mathtt{false}$).
- Therefore, it is not a tautology.
- Example: the proposition $P \wedge Q \implies Q \wedge P$ is a tautology.
- We have $[\![P \wedge Q \implies Q \wedge P]\!] =$ $(\mathtt{not}\ (v(P)\ \mathtt{\&\&}\ v(Q)))\ \mathtt{||}\ (v(P)\ \mathtt{\&\&}\ v(Q))$.
- For any values of $v(P)$ and $v(Q)$ we always get $\mathtt{true}$.

# The rest

- We can check whether a proposition is a tautology by trying all possible valuations, but there are exponentially many of them.
- We can do better by defining a proof system with some axioms and inference rules, which would allow us to **prove** that a proposition is a tautology without trying all valuations.
- We won't do that because **classical logic is not the right logical system for proving programs correct**.
- We will use constructive logic instead.

# Constructive logic 1/2

- In constructive logic, **propositions ARE NOT either true or false**.
- In constructive logic we usually think about propositions **in terms of their proofs.**
- In everyday language and also in mathematics as it is usually practiced, a “proof” means an argument by which one human demonstrates the truth of a statement to another human.
- In constructive logic, a proof of $P$ is a certificate that $P$ holds, i.e. a formal object which **certifies that $P$ has been proven.**,
- Meaning of propositions is determined by how we can prove them and how we can use them to prove other propositions.

# Constructive logic 2/2

- We shouldn't think about propositions as being either "true" or "false", but it's a deeply ingrained and hard to avoid way of thinking, so a translation:
- If we have a proof of $P$, we may think that $P$ is "true".
- If we have a proof of $\neg P$, we may think that $P$ is "false".
- If we have neither proof, we don't know anything about $P$.

# Propositions vs types

- There's a strange parallel going on between propositions and types.
- Types are, obviously, not either true or false – they are inhabited by programs.
- A program $t$ of type $A$ is something that, after performing some computations, returns an element of type $A$.
- The meaning of a type $A$ is determined by how we can write programs of type $A$ and how we can use programs of type $A$ to write other programs.

# Propositions are types, proofs are programs

- This “strange parallel” is not a coincidence. There are no coincidences in mathematics!
- It is most often referred to as the Curry-Howard correspondence, after two out of many people who discovered it.
- But it is better presented as a set of slogans:
- **Propositions are types.**
- **Proofs are programs.**
- **Proving theorems is just writing programs.**
- ... and a few more, which we’ll see shortly.

# True is the unit type 1/2

- There's the unit type `unit`.
- It's sole element is `()`.
- We can't do anything useful with it.

## True is the unit type 2/2

- There's the true proposition $\top$.
- It's sole proof is ().
- We can't conclude anything useful from it.

# Conjunction is the product type 1/2

- If `a` and `b` are types, then `a * b` is also a type.
- Elements of `a * b` are pairs `(x, y)`, where `x : a` and `y : b`.
- If we have a pair `x : a * b`, then `fst x : a` and `snd x : b`.

# Conjunction is the product type 2/2

- If $P$ and $Q$ are propositions, then $P \wedge Q$ is also a proposition.
- To prove $P \wedge Q$, we have to prove $P$ and we have to prove $Q$, so...
- ...proofs of $P \wedge Q$ are of the form `(x, y)`, i.e. they are pairs where `x` is a proof of $P$ and `y` is a proof of $Q$.
- If $P \wedge Q$ holds, then we can conclude that $P$ holds and we can conclude that $Q$ holds, so...
- ...if `x` is a proof of $P \wedge Q$, then `fst x` is a proof of $P$ and `snd x` is a proof of $Q$.

# Implication is the function type 1/2

- If `a` and `b` are types, then `a -> b` is also a type.
- Elements of `a -> b` are of the form `fun (x : a) -> e` – they are functions which take an input `x` of type `a` and return `e` of type `b` as output.
- If we have a function `f : a -> b` and an `x : a`, then we we can apply `f` to `x`, written `f x`, to get an element of type `b`.

# Implication is the function type 2/2

- If $P$ and $Q$ are propositions, then $P \implies Q$ is also a proposition.
- To prove $P \implies Q$, we need to assume that $P$ holds and then provve $Q$ under this assumption, so...
- ... proofs of $P \implies Q$ are of the form `fun (p : P) -> q`, i.e. they are functions which take a proof of $P$ as input and return a proof of $Q$ as output.
- If $P \implies Q$ holds and $P$ holds, we can conclude that $Q$ holds, so...
- ... if `f` is a proof of $P \implies Q$ and `x` is a proof of $P$, then `f x` is a proof of $Q$.

# Disjunction is discriminated union

- If $P$ and $Q$ are propositions, then $P \vee Q$ is also a proposition.
- To prove $P \vee Q$, we need either to prove $P$ or to prove $Q$, so...
- ...proofs of $P \vee Q$ are of the form `inl p`, where `p` is a proof of $P$, or of the form `inr q`, where `q` is a proof of $Q$.
- If $P \vee Q$ holds and $P \implies R$ holds and $Q \implies R$ holds, we can conclude that $R$ holds, so...
- ...if `x` is a proof of $P \vee Q$, then we can match on `x` and retrieve the proofs of $P/Q$ and use them to prove $R$.

# NOT to be continued

- I would like to say otherwise, but this lecture series will NOT be continued.