Functional Programming

Lecture 2

Komi Golova (she/her)
komi.golov@jetbrains.com

Constructor University Bremen

Announcements

- GitHub Classroom bug seems fixed

Announcements

- GitHub Classroom bug seems fixed
- Please do your research before asking a question
  - README files in `student-materials` and in the homework repo
  - Lecture slides and recommended reading material
  - If your question is answered there, I will not answer it separately

Announcements

- GitHub Classroom bug seems fixed
- Please do your research before asking a question
  - README files in `student-materials` and in the homework repo
  - Lecture slides and recommended reading material
  - If your question is answered there, I will not answer it separately
- Best practices when asking a question:
  - Send **one** message containing your question
  - Make sure your code is on GitHub
  - Link to the GitHub Actions run that shows your issue

Announcements

- GitHub Classroom bug seems fixed
- Please do your research before asking a question
  - README files in `student-materials` and in the homework repo
  - Lecture slides and recommended reading material
  - If your question is answered there, I will not answer it separately
- Best practices when asking a question:
  - Send **one** message containing your question
  - Make sure your code is on GitHub
  - Link to the GitHub Actions run that shows your issue
- If you have not officially registered for this course, do so

Announcements

- GitHub Classroom bug seems fixed
- Please do your research before asking a question
  - README files in `student-materials` and in the homework repo
  - Lecture slides and recommended reading material
  - If your question is answered there, I will not answer it separately
- Best practices when asking a question:
  - Send **one** message containing your question
  - Make sure your code is on GitHub
  - Link to the GitHub Actions run that shows your issue
- If you have not officially registered for this course, do so
- Expect 1-2 homework questions to be hard

Typing judgements

When working in Lean, you will often see things like this:

$$x : A, y : B\ x \vdash f\ x\ y : C\ x\ y$$

How do we read this?

Typing judgements

When working in Lean, you will often see things like this:

$$x : A, y : B\ x \vdash f\ x\ y : C\ x\ y$$

How do we read this?

- The following is given:
  - x has type A
  - y has type $B\ x$

Typing judgements

When working in Lean, you will often see things like this:

$$x : A, y : B\ x \vdash f\ x\ y : C\ x\ y$$

How do we read this?

- The following is given:
  - x has type A
  - y has type $B\ x$
- From which we can conclude:
  - $f\ x\ y$ has type $C\ x\ y$

Typing judgements (example)

For example:

$$n : \mathbb{N}, m : \mathbb{N}, h : n = m \vdash h.\mathrm{symm} : m = n$$

Typing judgements (example)

For example:

$$n : \mathbb{N}, m : \mathbb{N}, h : n = m \vdash h.\mathrm{symm} : m = n$$

Here we are given:

Typing judgements (example)

For example:

$$n : \mathbb{N}, m : \mathbb{N}, h : n = m \vdash h.\mathrm{symm} : m = n$$

Here we are given:

- n has type $\mathbb{N}$
- m has type $\mathbb{N}$
- h has type $n = m$

Typing judgements (example)

For example:

$$n : \mathbb{N}, m : \mathbb{N}, h : n = m \vdash h.\text{symm} : m = n$$

Here we are given:

- n has type $\mathbb{N}$
- m has type $\mathbb{N}$
- h has type $n = m$

And we conclude:

- $h.\text{symm}$ has type $m = n$

Propositions as types

Note: we say that $n = m$ is a type.

Propositions as types

Note: we say that $n = m$ is a type. In general:

Propositions are types.

Proofs are terms.

Propositions as types

Note: we say that $n = m$ is a type. In general:

Propositions are types.
Proofs are terms.

There is a rich theory here, but our focus is practical.

Propositions as types

Note: we say that $n = m$ is a type. In general:

Propositions are types.

Proofs are terms.

There is a rich theory here, but our focus is practical. The rules:

- True is a type with one term.
- False is a type with no terms.

Propositions as types

Note: we say that $n = m$ is a type. In general:

Propositions are types.

Proofs are terms.

There is a rich theory here, but our focus is practical. The rules:

- True is a type with one term.
- False is a type with no terms.
- A term of type $\varphi \wedge \psi$ is a pair of terms of type φ and ψ.
- A term of type $\varphi \vee \psi$ is a term of type φ or of type ψ.

Propositions as types

Note: we say that $n = m$ is a type. In general:

Propositions are types.

Proofs are terms.

There is a rich theory here, but our focus is practical. The rules:

- True is a type with one term.
- False is a type with no terms.
- A term of type $\varphi \wedge \psi$ is a pair of terms of type φ and ψ.
- A term of type $\varphi \vee \psi$ is a term of type φ or of type ψ.
- A term of type $\varphi \to \psi$ is a function term sending φ to ψ.

Propositions as types

Note: we say that $n = m$ is a type. In general:

Propositions are types.

Proofs are terms.

There is a rich theory here, but our focus is practical. The rules:

- True is a type with one term.
- False is a type with no terms.
- A term of type $\varphi \wedge \psi$ is a pair of terms of type φ and ψ.
- A term of type $\varphi \vee \psi$ is a term of type φ or of type ψ.
- A term of type $\varphi \to \psi$ is a function term sending φ to ψ.

How do we define these types?

Function types

Function types let us move things from the left of $\vdash$ to the right.

Function types

Function types let us move things from the left of $\vdash$ to the right.

$$\Gamma, x : A \vdash e : B\ x$$

becomes

$$\Gamma \vdash (\text{fun } x \Rightarrow e) : (x : A) \to B\ x$$

The `intro` tactic lets us turn the latter into the former.

Function types

Function types let us move things from the left of $\vdash$ to the right.

$$\Gamma, x : A \vdash e : B\ x$$

becomes

$$\Gamma \vdash (\text{fun } x \Rightarrow e) : (x : A) \to B\ x$$

The `intro` tactic lets us turn the latter into the former.

Given a function object, we can get rid of it by applying it.

Rules for types

To understand a type, we need to know two things:

- What can we do with it?
- What is the result?

Rules for types

To understand a type, we need to know two things:

- What can we do with it?
- What is the result?

These split up into two categories each.

What can we do with it?

- How do we build it?
- How do we use it?

What is the result?

- Of building then using?
- Of using then building?

Example: product types

How do we make a product (pair)?

Example: product types

How do we make a product (pair)? $\mathrm{Prod.mk} : X \to Y \to X \times Y$

Example: product types

How do we make a product (pair)? Prod.mk : $X \to Y \to X \times Y$

How do we use a pair?

Example: product types

How do we make a product (pair)? $\mathrm{Prod.mk} : X \to Y \to X \times Y$

How do we use a pair? $\mathrm{Prod.rec}\ \{\alpha\} : (X \to Y \to \alpha) \to X \times Y \to \alpha$

Example: product types

How do we make a product (pair)? $\mathrm{Prod.mk} : X \to Y \to X \times Y$

How do we use a pair? $\mathrm{Prod.rec}\ \{\alpha\} : (X \to Y \to \alpha) \to X \times Y \to \alpha$

What is the result of building then using?

Example: product types

How do we make a product (pair)? $\mathrm{Prod.mk} : X \to Y \to X \times Y$

How do we use a pair? $\mathrm{Prod.rec}\ \{\alpha\} : (X \to Y \to \alpha) \to X \times Y \to \alpha$

What is the result of building then using?
$\mathrm{Prod.rec}\ f\ (\mathrm{Prod.mk}\ x\ y) = f\ x\ y$

Example: product types

How do we make a product (pair)? $\mathrm{Prod.mk} : X \to Y \to X \times Y$

How do we use a pair? $\mathrm{Prod.rec}\ \{\alpha\} : (X \to Y \to \alpha) \to X \times Y \to \alpha$

What is the result of building then using?
$\mathrm{Prod.rec}\ f\ (\mathrm{Prod.mk}\ x\ y) = f\ x\ y$

What is the result of using then building?

Example: product types

How do we make a product (pair)? $\text{Prod.mk} : X \to Y \to X \times Y$

How do we use a pair? $\text{Prod.rec}\ \{\alpha\} : (X \to Y \to \alpha) \to X \times Y \to \alpha$

What is the result of building then using?
$\text{Prod.rec}\ f\ (\text{Prod.mk}\ x\ y) = f\ x\ y$

What is the result of using then building?
$\text{Prod.rec Prod.mk}\ p = p$

Example: product types

How do we make a product (pair)? $\mathrm{Prod.mk} : X \to Y \to X \times Y$

How do we use a pair? $\mathrm{Prod.rec}\ \{\alpha\} : (X \to Y \to \alpha) \to X \times Y \to \alpha$

What is the result of building then using?
$\mathrm{Prod.rec}\ f\ (\mathrm{Prod.mk}\ x\ y) = f\ x\ y$

What is the result of using then building?
$\mathrm{Prod.rec}\ \mathrm{Prod.mk}\ p = p$

Note: in Lean, α can depend on $x : X$ and $y : Y$.

Inductive types

Notice that the rules are very predictable.

If we specify how to build something, the rest follows automatically.

Inductive types

Notice that the rules are very predictable.
If we specify how to build something, the rest follows automatically.

An inductive type in Lean is a type defined by its introduction rules.
For example:

```
inductive BinTree α where
  | node : BinTree α → BinTree α → BinTree α
  | leaf : α → BinTree α
```

This creates introduction functions `BinTree.node` and `BinTree.leaf`, and an elimination function `BinTree.rec`.

Soundness

Lean allows us to state propositions as types.

Constructing a term means proving the proposition.

Soundness

Lean allows us to state propositions as types.

Constructing a term means proving the proposition.

Is there a proof of false? i.e. is there a term of type `Empty`?

Soundness

Lean allows us to state propositions as types.

Constructing a term means proving the proposition.

Is there a proof of false? i.e. is there a term of type `Empty`?

If there is, that's a bug! (It has happened.)

Some things we have to do to avoid it:

- Functions have to be total.
- Inductive types have to be restricted.
  - For example, we cannot have `A -> Empty ~= A`
  - So we cannot allow `inductive A where mk : (A -> Empty) -> A`

Recommended reading

- Theorem Proving in Lean 4, chapters 7-8
- Functional Programming in Lean, sections 1.3, 1.5-1.7