FLOLAC'14

# Type theory and logic

## Lecture I: simple type theory

1 July 2014

柯向上

Department of Computer Science
University of Oxford

Hsiang-Shang.Ko@cs.ox.ac.uk

# Explaining typing

Consider the Haskell program:

$$\begin{array}{lcl} \mathit{swap} & :: & (a\,,\,b) \to (b\,,\,a) \\ \mathit{swap} & = & \lambda p \to (\mathit{snd}\ p\,,\,\mathit{fst}\ p) \end{array}$$

How do we explain that the program is type-correct?

The function *swap* is from $(a\,,\,b)$ to $(b\,,\,a)$. Assume that we have an input $p$ of type $(a\,,\,b)$; we need to construct a term of type $(b\,,\,a)$. To do so, we need to construct a term of type $b$ and another term of type $a$, and pair them together. We can use $\mathit{snd}\ p$ as the first term, since $p$ has type $(a\,,\,b)$ and the type of $\mathit{snd}\ p$ is the type of the second component. Symmetrically, $\mathit{fst}\ p$ can be used as the second term.

# Typing derivation

The reasoning can be formalised as the following *typing derivation*:

$$\dfrac{\dfrac{\dfrac{}{p :: (a, b) \vdash p :: (a, b)}\,(\text{var})}{p :: (a, b) \vdash \mathit{snd}\ p :: b}\,(\text{snd}) \qquad \dfrac{\dfrac{}{p :: (a, b) \vdash p :: (a, b)}\,(\text{var})}{p :: (a, b) \vdash \mathit{fst}\ p :: a}\,(\text{fst})}{\dfrac{p :: (a, b) \vdash (\mathit{snd}\ p, \mathit{fst}\ p) :: (b, a)}{\vdash \lambda p \to (\mathit{snd}\ p, \mathit{fst}\ p) :: (a, b) \to (b, a)}\,(\text{abs})}\,(\text{pair})$$

Why formalise?

- Conciseness. (A *domain-specific language* for explaining typing, if you like.)
- Mechanisation (e.g., for implementing a typechecker).

# Logical derivation

We can also read it as a logical derivation of the proposition "$a$ and $b$ implies $b$ and $a$":

$$\dfrac{\dfrac{\dfrac{}{p :: (a, b) \vdash p :: (a, b)}\,(\text{assum})}{p :: (a, b) \vdash \mathit{snd}\ p :: b}\,(\wedge\text{ER}) \qquad \dfrac{\dfrac{}{p :: (a, b) \vdash p :: (a, b)}\,(\text{assum})}{p :: (a, b) \vdash \mathit{fst}\ p :: a}\,(\wedge\text{EL})}{p :: (a, b) \vdash (\mathit{snd}\ p, \mathit{fst}\ p) :: (b, a)}\,(\wedge\text{I})$$
$$\dfrac{}{\vdash \lambda p \to (\mathit{snd}\ p, \mathit{fst}\ p) :: (a, b) \to (b, a)}\,(\to\text{I})$$

This is Gentzen's *natural deduction* system, in which only the "type part" is present.

What about the "program part"?

# Constructive logic

In *constructive logic*, the meaning of a proposition is a *set of valid proofs* that we admit as proving the proposition, and the proposition is said to be true exactly when we can construct a proof in the set.

For example,

- proofs of "$A$ and $B$" should be pairs of proofs, one of $A$ and the other of $B$;
- proofs of "$A$ implies $B$" should be procedures transforming a proof of $A$ to a proof of $B$.

... But these are just programs having pair or function types!

# The propositions-as-types principle

Slogan:

*Propositions are types.*

*Proofs are programs.*

That is, logical reasoning is simply functional programming.

For example, if we want to show that "$a$ and $b$ implies $b$ and $a$", it suffices to construct a functional program of type $(a, b) \to (b, a)$.

Not every functional programming language will do, however.

# Intuitionistic type theory

Per Martin-Löf's *intuitionistic type theory* was designed in the '70s to serve as a foundation for *intuitionistic mathematics*. It is simultaneously

- a computationally meaningful higher-order logic system and
- a very expressively typed functional programming language.

The dependently typed programming language Agda is theoretically based on MLTT.

# Sets

Activities in type theory consist of construction of elements of various *sets* (which we regard as synonymous with "types").

- Note that element construction includes proving logical propositions (when we regard sets as propositions) and carrying out general mathematical constructions (e.g., constructing functions of type $\mathbb{N} \to \mathbb{N}$).
- In these lectures we will mainly focus on sets that have a logical interpretation.

Specification of sets is thus the central part of type theory.

# Judgements

*Judgements* are justifiable statements about expressions. Today we will exclusive use *typing judgements*.

A typing judgement has the form

$$\Gamma \vdash t : S$$

where the *context* $\Gamma$ is a finite list "$x : A$, $y : B$, ..." of type assignments to distinct variables, which can appear in $t$ and $S$. In $\Gamma$, a variable can also appear in the types to its right (e.g., $x$ can appear in $B$).

The judgement states that, under the typing assumptions in $\Gamma$, the expression $t$ has type $S$ (i.e., $t$ is a legitimate element of the set $S$).

In a typing judgement $\Gamma \vdash t : S$, the context $\Gamma$ can be empty, in which case we simply write $\vdash t : S$.

# Set of sets

We assume that there is a set of sets named $\mathcal{U}$, so when we write down, for example, $\Gamma \vdash A : \mathcal{U}$, this states that $A$ is a set under the assumptions in $\Gamma$.

Whenever we write down a typing judgement, we require that the expressions appearing on the right of all the colons in the judgement are already judged to be sets.

**Remark.** This rough treatment is actually paradoxical; we will resolve the paradox tomorrow.

# Derivations

Judgements are justified by *derivations*, which are constructed using a predetermined collection of *deduction rules*.

A deduction rule has the form

$$\frac{J_0 \qquad \ldots \qquad J_{n-1}}{J}\ (\textit{rule name})$$

which says that the judgement $J$, called the *conclusion* of the rule, can be established if the judgements $J_0$, ..., $J_{n-1}$, called the *premises* of the rule, can be established.

# Assumption rule

A rule can have zero premises, meaning that its conclusion is self-evident.

For example, there is an *assumption rule*

$$\frac{}{\Gamma \vdash x : S}\ (\text{assum})$$

which has a *side condition* that $x : S$ appears in $\Gamma$.

# Set specification

Today, we give three kinds of rules for specifying each set:

- *Formation rule* — what constitute the name of the set.
- *Introduction rule(s)* — how to construct *canonical* elements of the set.
- *Elimination rule(s)* — how to deconstruct elements of the set and transform them to elements of some other sets.

(More to come tomorrow.)

# Cartesian product types (conjunction)

- Formation:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash B : \mathcal{U}}{\Gamma \vdash A \times B : \mathcal{U}} \,(\times\mathsf{F})$$

- Introduction:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a\,,\, b) : A \times B} \,(\times\mathsf{I})$$

- Elimination:

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathtt{fst}\; p : A} \,(\times\mathsf{EL}) \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathtt{snd}\; p : B} \,(\times\mathsf{ER})$$

# Cartesian product types (conjunction)

**Exercise.** Let $\Gamma := A : \mathcal{U}, B : \mathcal{U}, p : A \times B$. Give a derivation of $\Gamma \vdash \_ : B \times A$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash p : A \times B}\,(\text{assum})}{\Gamma \vdash \mathtt{snd}\ p : B}\,(\times\text{ER}) \qquad \dfrac{\dfrac{}{\Gamma \vdash p : A \times B}\,(\text{assum})}{\Gamma \vdash \mathtt{fst}\ p : A}\,(\times\text{EL})}{\Gamma \vdash (\mathtt{snd}\ p, \mathtt{fst}\ p) : B \times A}\,(\times\text{I})$$

**Exercise.** Derive

$$A : \mathcal{U}, B : \mathcal{U}, C : \mathcal{U}, p : (A \times B) \times C \vdash \_ : A \times (B \times C)$$

# Function types (implication)

- Formation:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash B : \mathcal{U}}{\Gamma \vdash A \to B : \mathcal{U}} \; (\to\mathsf{F})$$

- Introduction:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \; (\to\mathsf{I})$$

- Elimination:

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B} \; (\to\mathsf{E})$$

  This formalises the "modus ponens" rule in logic.

**Exercise.** Derive

$$A : \mathcal{U}, B : \mathcal{U}, C : \mathcal{U} \vdash \_ : (A \to B \to C) \to B \to A \to C$$

# Coproduct types (disjunction)

- Formation:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash B : \mathcal{U}}{\Gamma \vdash A + B : \mathcal{U}}\ (+\mathsf{F})$$

- Introduction:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathtt{left}\ a : A + B}\ (+\mathsf{IL}) \qquad \frac{\Gamma \vdash b : B}{\Gamma \vdash \mathtt{right}\ b : A + B}\ (+\mathsf{IR})$$

- Elimination:

$$\frac{\Gamma \vdash q : A + B \qquad \Gamma \vdash f : A \to C \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash \mathtt{case}\ q\ f\ g : C}\ (+\mathsf{E})$$

**Exercise.** Derive

$$A : \mathcal{U},\ B : \mathcal{U} \vdash \_ : A + B \to B + A$$

# Unit type (truth)

- Formation:

$$\frac{}{\Gamma \vdash \mathbf{1} : \mathcal{U}}\ (\mathbf{1}\mathsf{F})$$

- Introduction:

$$\frac{}{\Gamma \vdash \mathtt{unit} : \mathbf{1}}\ (\mathbf{1}\mathsf{I})$$

- Elimination: none

# Empty type (falsity)

- Formation:

$$\frac{}{\Gamma \vdash \mathbf{0} : \mathcal{U}}\ (\mathbf{0}\mathsf{F})$$

- Introduction: none
- Elimination:

$$\frac{\Gamma \vdash b : \mathbf{0}}{\Gamma \vdash \mathtt{absurd}\, b : A}\ (\mathbf{0}\mathsf{E})$$

  This formalises the "principle of explosion".

We define the *negation* of a proposition $A$ to be $A \to \mathbf{0}$, which we abbreviate as $\neg A$. Note that $\neg A$ has a proof if and only if $A$ has no proof.

**Exercise.** Show that $A \to \neg\neg A$ is true.

# Simple type theory

We have specified the set formers '$\to$', '$\times$', '$+$', $\mathbf{1}$, and $\mathbf{0}$, which are respectively interpreted logically as implication, conjunction, disjunction, truth, and falsity.

The fragment of type theory consisting of these sets is called *simple type theory*; the type part (with, e.g., the natural deduction system) is traditionally called *propositional logic*.

# Propositional connectives

We study simple type theory (in isolation) because we are interested in understanding the role of propositional set formers (connectives) when they are used to combine propositions into more complex ones.

For an extreme example, the truth of the following proposition is determined by the way we use the connectives alone.

**if** *herba viridi* **and** *area est infectum*, **then** *area est infectum*

The actual meanings/structures of the two propositions "*herba viridi*" and "*area est infectum*" do not matter.

# Consistency

As a logic system, simple type theory is *consistent*, meaning that not all propositions are provable.

Consistency is a basic requirement of any (traditional) mathematical logic: if a logic is *inconsistent*, meaning that every proposition is provable, then we might as well throw the logic away and simply declare everything to be true.

The type system of Haskell is inconsistent, and hence inadequate as a (traditional) mathematical logic system.

# Non-provable propositions

Assuming $A : \mathcal{U}$ and $B : \mathcal{U}$:

| We can prove | but not |
|---|---|
| $\neg\neg(A + \neg A)$ | $A + \neg A$ |
| | (*law of excluded middle*) |
| $A \to \neg\neg A$ | $\neg\neg A \to A$ |
| | (*principle of indirect proof*) |
| $\neg A + \neg B \to \neg(A \times B)$ | $\neg(A \times B) \to \neg A + \neg B$ |
| $(A \to B) \to (\neg B \to \neg A)$ | $(\neg B \to \neg A) \to (A \to B)$ |

# Intuitionism

What's "wrong" with the type-theoretic logic?

- Nothing's wrong; the logic just reflects a different way of viewing mathematics.

*Intuitionism* was founded by L.E.J. Brouwer (1881–1966), which holds the position that mathematical objects are *mental constructions*, rather than existing in an ideal world, independent from human mind. The latter is the position of *classical mathematics*.

# Computability

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. 1937.

- [...] the justification [of the definitions] lies in the fact that the human memory is necessarily limited.
- We may compare a man in the process of computing a real number to a machine [...]
- [On number of symbols...] We cannot tell at a glance whether 9999999999999999 and 9999999999999999 are the same.
- [On number of states...] If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused.

# Unification of mathematics and programming

Per Martin-Löf. Constructive mathematics and computer programming. 1984.

> If programming is understood
>
> - not as the writing of instructions for this or that computing machine
> - but as the design of methods of computation that it is the computer's duty to execute
>   - (a difference that Dijkstra has referred to as the difference between comput**er** science and comput**ing** science),
>
> then it no longer seems possible to distinguish the discipline of programming from constructive mathematics.

# Some more exercises

Assuming $A : \mathcal{U}$, $B : \mathcal{U}$, and $C : \mathcal{U}$, prove

- $(A \to B \to C) \to (A \to B) \to A \to C$
- $(A + B) + C \to A + (B + C)$
- $\neg(A + B) \leftrightarrow \neg A \times \neg B$
- $A + (B \times C) \leftrightarrow (A + B) \times (A + C)$