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:

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

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 $\mathbb{N} \to \mathbb{N}$).

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

Judgements

Judgements are justifiable statements about expressions. We will look at two kinds of judgements today:

- A *set judgement* has the form

  $$S \ \textbf{set}$$

  stating that the expression S is a legitimate set.
- A *typing judgement* has the form

  $$\Gamma \ \vdash \ t : S$$

  where Γ is a list $x_0 : S_0$, ..., $x_{n-1} : S_{n-1}$ of type assignments to variables x_0, ..., x_{n-1}, which can appear in t and S. This states that, under the typing assumptions in Γ, the expression t has type S (i.e., t is a legitimate element of the set S).
  Γ can be empty, in which case we simply write $\vdash \ t : S$.

Derivations

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

A deduction rule has the form

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

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.

Note that 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 Γ and is the rightmost type assignment to x.

Set specification

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

- *formation rule* — what makes up the name of the set,
- *introduction rule(s)* — how to construct elements of the set, and
- *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:

$$\frac{A\ \mathbf{set} \qquad B\ \mathbf{set}}{A \times B\ \mathbf{set}}\ (\times\mathsf{F})$$

- 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. (Assuming A **set** and B **set**) give a derivation of an element of $B \times A$ under the assumption $p : A \times B$.

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

Exercise. Give a derivation of an element of $A \times (B \times C)$ under the assumption $p : (A \times B) \times C$.

Function types (implication)

- Formation:

$$\frac{A \ \mathbf{set} \qquad B \ \mathbf{set}}{A \to B \ \mathbf{set}} \ (\to\mathsf{F})$$

- 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. Show that $(A \to B \to C) \to B \to A \to C$ is true.

Coproduct types (disjunction)

- Formation:

$$\frac{A\ \mathbf{set} \qquad B\ \mathbf{set}}{A + B\ \mathbf{set}}\ (+\mathsf{F})$$

- 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:

$$\frac{\Gamma\ \vdash\ q : A + B \quad \Gamma, x : A\ \vdash\ c_l : C \quad \Gamma, y : B\ \vdash\ c_r : C}{\Gamma\ \vdash\ \mathtt{case}\,(q\,;\, x.\ c_l\,;\, y.\ c_r) : C}\ (+\mathsf{E})$$

Exercise. Show that $A + B \to B + A$ is true.

Unit type (truth)

- Formation:

$$\frac{}{1\ \mathbf{set}}\ (1\mathsf{F})$$

- Introduction:

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

- Elimination: none

Empty type (falsity)

- Formation:

$$\frac{}{0\ \ \mathbf{set}}\ (0\mathsf{F})$$

- Introduction: none
- Elimination:

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

  This formalises the "principle of explosion".

We define the *negation* of a proposition A to be $A \to 0$, 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 '$\to$', '$\times$', '$+$', 1, and 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*.

Simple type theory

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.

Definition. In simple type theory, a proposition P is called a *theorem* exactly when we have a derivation of $\vdash\ p : P$ for some term p.

Consistency

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

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.

Theorems and non-theorems

For arbitrary sets P and Q:

Theorems	Non-theorems
$\neg\neg(P + \neg P)$	$P + \neg P$ (*law of excluded middle*)
$P \to \neg\neg P$	$\neg\neg P \to P$ (*principle of indirect proof*)
$\neg P + \neg Q \to \neg(P \times Q)$	$\neg(P \times Q) \to \neg P + \neg Q$
$(P \to Q) \to (\neg Q \to \neg P)$	$(\neg Q \to \neg P) \to (P \to Q)$

Intuitionism

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

Intuitionism

Per Martin-Löf: "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 compute**er** science and comput**ing** science),

then it no longer seems possible to distinguish the discipline of programming from constructive mathematics."