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*:

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

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":

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

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* Γ is a finite list "$x : A,\ y : B,\ \ldots$" of type assignments to distinct variables, which can appear in t and S. In Γ, 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 Γ, 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 Γ 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 $\mathcal{U}$ (for "universe"), so when we write down, for example, $\Gamma \vdash A : \mathcal{U}$, this states that A is a set under the assumptions in Γ.

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

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

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 a *variable rule*

$$\frac{}{\Gamma \vdash x : S}\ (\mathrm{V})$$

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

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.

(One more kind of rules to come tomorrow.)

Cartesian product types (conjunction)

- Formation:

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

- Introduction:

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

- Elimination:

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

Cartesian product types (conjunction)

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

$$\cfrac{\cfrac{\cfrac{}{\Gamma \vdash p : A \times B}\ (\mathrm{V})}{\Gamma \vdash \mathtt{snd}\ p : B}\ (\times\mathrm{ER}) \qquad \cfrac{\cfrac{}{\Gamma \vdash p : A \times B}\ (\mathrm{V})}{\Gamma \vdash \mathtt{fst}\ p : A}\ (\times\mathrm{EL})}{\Gamma \vdash (\mathtt{snd}\ p\,,\, \mathtt{fst}\ p) : B \times A}\ (\times\mathrm{I})$$

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:

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

- Introduction:

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

- Elimination:

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

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:

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

- Introduction:

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

- Elimination:

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

Exercise. Derive

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

Unit type (truth)

- Formation:

$$\frac{}{\Gamma \vdash \top : \mathcal{U}}\ (\top\mathrm{F})$$

- Introduction:

$$\frac{}{\Gamma \vdash \texttt{unit} : \top}\ (\top\mathrm{I})$$

- Elimination: none

Empty type (falsity)

- Formation:

$$\frac{}{\Gamma \vdash \bot : \mathcal{U}}\ (\bot\mathrm{F})$$

- Introduction: none
- Elimination:

$$\frac{\Gamma \vdash b : \bot}{\Gamma \vdash \mathsf{absurd}\ b : A}\ (\bot\mathrm{E})$$

  This formalises the “principle of explosion”.

We define the *negation* of a proposition A to be $A \to \bot$, 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 ‘$\to$’, ‘$\times$’, ‘$+$’, $\top$, and $\bot$, 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 999999999999999 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

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

FLOLAC'14

Type theory and logic

Lecture II: dependent type theory

2 July 2014

柯向上

Department of Computer Science
University of Oxford

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

Indexed families of sets (predicates)

Common mathematical statements involve predicates and universal/existential quantification.

For example: "For every $x : \mathbb{N}$, if x is not zero, then there exists $y : \mathbb{N}$ such that x is equal to $1 + y$."

In type theory, a predicate on A has type $A \to \mathcal{U}$ — a *family of sets* indexed by the domain A. For example:

$$\vdash \lambda x.\ \text{"if } x \text{ is zero then } \bot \text{ else } \top\text{"} : \mathbb{N} \to \mathcal{U}$$

Remark. The above treatment is in fact unfounded in our current theory. Why?

Dependent product types (universal quantification)

- Formation:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash B : A \to \mathcal{U}}{\Gamma \vdash \Pi\, A\, B : \mathcal{U}}\ (\Pi\text{F})$$

- Introduction:

$$\frac{\Gamma, x : A \vdash t : B\, x}{\Gamma \vdash \lambda x.\, t : \Pi\, A\, B}\ (\Pi\text{I})$$

- Elimination:

$$\frac{\Gamma \vdash f : \Pi\, A\, B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B\, a}\ (\Pi\text{E})$$

Notation. We usually write $\Pi[x:A]\ B\,x$ for $\Pi\, A\, B$, regarding '$\Pi[x:A]$' as a quantifier.

Exercise. Let $\Gamma := A : \mathcal{U}, B : \mathcal{U}, C : A \to B \to \mathcal{U}$. Derive

$$\Gamma \vdash \_ : (\Pi[x:A]\ \Pi[y:B]\ C\,x\,y) \to \Pi[y:B]\ \Pi[x:A]\ C\,x\,y$$

Dependent sum types (existential quantification)

- Formation:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash B : A \to \mathcal{U}}{\Gamma \vdash \Sigma\, A\, B : \mathcal{U}}\ (\Sigma\text{F})$$

- Introduction:

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

- Elimination:

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

Notation. We usually write $\Sigma[x:A]\ B\,x$ for $\Sigma\, A\, B$.

Exercise. Let $\Gamma := A : \mathcal{U}, B : \mathcal{U}, C : A \to B \to \mathcal{U}$. Derive

$$\Gamma \vdash \_ : (\Sigma[p : A \times B]\ C\,(\mathtt{fst}\ p)\,(\mathtt{snd}\ p)) \to \Sigma[x:A]\ \Sigma[y:B]\ C\,x\,y$$

Exercises

Let $\Gamma := A : \mathcal{U}\,,\ B : A \to \mathcal{U}\,,\ C : A \to \mathcal{U}$. Derive

$$\Gamma\ \vdash\ \_ : (\Pi[x:A]\ B\,x \times C\,x) \leftrightarrow (\Pi[y:A]\ B\,y) \times (\Pi[z:A]\ C\,z)$$

$$\Gamma\ \vdash\ \_ : (\Sigma[x:A]\ B\,x + C\,x) \leftrightarrow (\Sigma[y:A]\ B\,y) + (\Sigma[z:A]\ C\,z)$$

What about

$$\Gamma\ \vdash\ \_ : (\Pi[x:A]\ B\,x + C\,x) \leftrightarrow (\Pi[y:A]\ B\,y) + (\Pi[z:A]\ C\,z)$$

$$\Gamma\ \vdash\ \_ : (\Sigma[x:A]\ B\,x \times C\,x) \leftrightarrow (\Sigma[y:A]\ B\,y) \times (\Sigma[z:A]\ C\,z)$$

?

Now let $\Gamma := A : \mathcal{U}\,,\ B : \mathcal{U}\,,\ R : A \to B \to \mathcal{U}$. Prove the *axiom of choice*, i.e., derive

$$\Gamma\ \vdash\ \_ : (\Pi[x:A]\ \Sigma[y:B]\ R\,x\,y) \to \Sigma[f:A \to B]\ \Pi[z:A]\ R\,z\,(f\,z)$$

Universes

In our current theory, to form types like $A \to \mathcal{U}$ where $A : \mathcal{U}$, we need to assume $\mathcal{U} : \mathcal{U}$. However, this assumption — called *impredicativity* — was shown to lead to inconsistency by Jean-Yves Girard. (This result is commonly referred to as *Girard's paradox*.)

We thus need to introduce a *predicative* hierarchy of universes $\mathcal{U}_0$, $\mathcal{U}_1$, ..., up to infinity.

- Smaller universes are elements of larger universes.

$$\frac{}{\Gamma\ \vdash\ \mathcal{U}_i : \mathcal{U}_{i+1}}\ (\mathcal{U}\mathrm{F})$$

- Elements of smaller universes are also elements of larger universes. This is called *cumulativity*.

$$\frac{\Gamma\ \vdash\ A : \mathcal{U}_i}{\Gamma\ \vdash\ A : \mathcal{U}_{i+1}}\ (\mathrm{cum})$$

Universe polymorphism and typical ambiguity

Formation rules and elimination rules have to be revised to be *universe-polymorphic*.

- For example, the formation rule for coproducts becomes

$$\frac{\Gamma\ \vdash\ A : \mathcal{U}_i \qquad \Gamma\ \vdash\ B : \mathcal{U}_j}{\Gamma\ \vdash\ A + B : \mathcal{U}_{max\,i\,j}}\ (+\mathrm{F})$$

- The elimination rule for coproducts is

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

  for which we implicitly assume the following judgements:

$$\Gamma\ \vdash\ A : \mathcal{U}_i \qquad \Gamma\ \vdash\ B : \mathcal{U}_j \qquad \Gamma\ \vdash\ C : \mathcal{U}_k$$

In practice, however, we can drop the indices (as if assuming $\mathcal{U} : \mathcal{U}$) because they can be inferred most of the time. (This is called *typical ambiguity*.)

Computation

Let $\Gamma := A : \mathcal{U}\,,\ B : A \to \mathcal{U}\,,\ C : A \to \mathcal{U}$. Try to derive

$$\Gamma\ \vdash\ \_ : (\Pi[p:\Sigma\,A\,B]\ C\,(\mathtt{fst}\ p)) \to \Pi[x:A]\ B\,x \to C\,x$$

... and you should notice some problems.

So far we have been concentrating on the *statics* of type theory — how to match program structure with type structure.

Here we need to invoke the *dynamics* of type theory — how to *reduce* (rewrite) programs to other programs.

Equality judgements

We introduce a new kind of judgements stating that two terms should be regarded as the same during typechecking:

$$\Gamma \vdash t = u \in A$$

in which we require that A and everything appearing on the right of the colons in Γ are judged to be sets, and t and u are judged to be elements of A.

Computation rules

For each set, (when applicable) we specify additional *computation rules* stating how to eliminate an introductory term. This is the type-theoretic manifestation of *Gentzen's inversion principle*: elimination rules should be justified in terms of introduction rules.

For example, for product types we have two computation rules:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \mathtt{fst}\,(a, b) = a \in A}\,(\times\mathrm{CL}) \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash \mathtt{snd}\,(a, b) = b \in B}\,(\times\mathrm{CR})$$

More computation rules

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.\, t)\, a = t[a/x] \in A \to B}\,(\to\mathrm{C})$$

Notation. The term $t[a/x]$ is the result of *substituting* the term a for all "free occurrences" of the variable x in the term t.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : A \to C \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash \mathtt{case}\,(\mathtt{left}\ a)\, f\, g = f\, a \in C}\,(+\mathrm{CL})$$

$$\frac{\Gamma \vdash b : B \qquad \Gamma \vdash f : A \to C \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash \mathtt{case}\,(\mathtt{right}\ b)\, f\, g = g\, b \in C}\,(+\mathrm{CR})$$

More computation rules

$$\frac{\Gamma, x : A \vdash t : B\, x \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.\, t)\, a = t[a/x] \in B\, a}\,(\Pi\mathrm{C})$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B\, a}{\Gamma \vdash \mathtt{fst}\,(a, b) = a \in A}\,(\Sigma\mathrm{CL})$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B\, a}{\Gamma \vdash \mathtt{snd}\,(a, b) = b \in B\, a}\,(\Sigma\mathrm{CR})$$

Congruence rules

We need a congruence rule for each constant we introduce:

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash b = b' \in B}{\Gamma \vdash (a\,,\,b) = (a'\,,\,b') \in A \times B}$$

$$\frac{\Gamma \vdash p = p' \in A \times B}{\Gamma \vdash \mathtt{fst}\; p = \mathtt{fst}\; p' \in A} \qquad \frac{\Gamma \vdash p = p' \in A \times B}{\Gamma \vdash \mathtt{snd}\; p = \mathtt{snd}\; p' \in B}$$

$$\frac{\Gamma, x : A \vdash t = t' \in B}{\Gamma \vdash \lambda x.\, t = \lambda x.\, t' \in A \to B}$$

$$\frac{\Gamma \vdash f = f' \in A \to B \qquad \Gamma \vdash a = a' \in A}{\Gamma \vdash f\, a = f'\, a' \in B}$$

... and similar rules for `left`, `right`, `case`, and `absurd`.

Equivalence rules

Judgemental equality is an equivalence relation.

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t \in A}$$

$$\frac{\Gamma \vdash t = u \in A}{\Gamma \vdash u = t \in A}$$

$$\frac{\Gamma \vdash t = u \in A \qquad \Gamma \vdash u = v \in A}{\Gamma \vdash t = v \in A}$$

Conversion rule

Once we establish that two sets are judgementally equal, we can transfer terms between the two sets.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B \in \mathcal{U}}{\Gamma \vdash t : B}\ (\text{conv})$$

Exercise. Finish deriving

$\Gamma \vdash \_ : (\Pi[p : \Sigma\, A\, B]\ C\,(\mathtt{fst}\; p)) \to \Pi[x : A]\ B\, x \to C\, x$

(where $\Gamma := A : \mathcal{U}\,,\, B : A \to \mathcal{U}\,,\, C : A \to \mathcal{U}$).

More congruence rules

We can state congruence rules for dependent products and sums only after we have the conversion rule. Why?

$$\frac{\Gamma \vdash a = a' \in A \qquad \Gamma \vdash b = b' \in B\, a}{\Gamma \vdash (a\,,\,b) = (a'\,,\,b') \in \Sigma\, A\, B}$$

$$\frac{\Gamma \vdash p = p' \in \Sigma\, A\, B}{\Gamma \vdash \mathtt{fst}\; p = \mathtt{fst}\; p' \in A} \qquad \frac{\Gamma \vdash p = p' \in \Sigma\, A\, B}{\Gamma \vdash \mathtt{snd}\; p = \mathtt{snd}\; p' \in B\,(\mathtt{fst}\; p)}$$

$$\frac{\Gamma, x : A \vdash t = t' \in B\, x}{\Gamma \vdash \lambda x.\, t = \lambda x.\, t' \in \Pi\, A\, B}$$

$$\frac{\Gamma \vdash f = f' \in \Pi\, A\, B \qquad \Gamma \vdash a = a' \in A}{\Gamma \vdash f\, a = f'\, a' \in B\, a}$$

Decidability of judgemental equality

Our judgemental equality is *decidable*: for any equality judgement we can decide whether it has a derivation or not.

(As a consequence, typechecking is also decidable.)

Decidability is achieved by reducing terms to their *normal forms* and see if the normal forms match.

There are various *reduction strategies*, and judgemental equality is formulated without reference to any particular reduction strategy — it captures the notion of computation only abstractly.

Canonical vs non-canonical elements

Introduction rules specify *canonical* — or *immediately recognisable* — elements of a set.

A complex construction may not be immediately recognisable as belonging to a set, but as long as we can see that it *computes* to a canonical element, we accept it as a *non-canonical* element of the set.

Remark. It follows that all computations in type theory must terminate, because from a non-canonical proof we should be able to get a canonical one in finite time.

Property (canonicity). If $\vdash t : A$, then $\vdash t = c \in A$ for some canonical element c.

Non-derivable proposition

Let $\Gamma := A : \mathcal{U}, B : A \to \mathcal{U}$. We can derive

$$\Gamma \vdash \_ : (\Sigma[x : A]\ B\,x) \to (\neg\Pi[x : A]\ \neg B\,x)$$

but not

$$\Gamma \vdash \_ : (\neg\Pi[x : A]\ \neg B\,x) \to (\Sigma[x : A]\ B\,x)$$

Exercise. Assuming that there is an additional rule

$$\frac{\Gamma \vdash X : \mathcal{U}}{\Gamma \vdash \texttt{LEM}\,X : X + \neg X}\ (\text{LEM})$$

derive

$$\Gamma \vdash \_ : (\neg\Pi[x : A]\ \neg B\,x) \to (\Sigma[x : A]\ B\,x)$$

Classical logic as an extension of intuitionistic logic

By including the LEM rule, our logic system can now derive intuitionistically unprovable but classically provable propositions.

The LEM rule breaks canonicity, however (why?) — the system is no longer computationally meaningful.

We are actually abusing the intuitionistic system, though: classical disjunction and existence are semantically weaker than their intuitionistic counterparts, so naively using the latter to state classical facts does not really make much sense.

Classical logic as a sub-system of intuitionistic logic

The *Gödel–Gentzen negative translation* embeds classical logic into intuitionistic logic by

- putting double negation in front of "atomic propositions",
- representing existential quantification by '$\neg\Pi[x:A]\ \neg\ldots$', and
- representing disjunction by '$\neg(\neg\ldots\times\neg\ldots)$'.

A proposition is classically provable if and only if its Gödel–Gentzen negative translation is intuitionistically provable.

Exercise. Let $\Gamma := A : \mathcal{U}\,,\ a : A\,,\ D : A \to \mathcal{U}$. (Note that A is an inhabited set.) Derive

$$\Gamma\ \vdash\ \_ : \neg\Pi[x:A]\ \neg(\neg\neg D\,x \to \Pi[y:A]\ \neg\neg D\,y)$$

where the proposition is the Gödel–Gentzen negative translation of

$$\exists[x:A]\ D\,x \to \forall[y:A]\ D\,y$$

FLOLAC'14

Type theory and logic

Lecture III: natural number arithmetic

3 July 2014

柯向上

Department of Computer Science
University of Oxford

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

Natural numbers

- Formation:

$$\frac{}{\Gamma \vdash \mathbb{N} : \mathcal{U}}\ (\mathbb{N}\mathrm{F})$$

- Introduction:

$$\frac{}{\Gamma \vdash \mathtt{zero} : \mathbb{N}}\ (\mathbb{N}\mathrm{IZ}) \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{suc}\ n : \mathbb{N}}\ (\mathbb{N}\mathrm{IS})$$

- Elimination:

$$\frac{\begin{array}{l}\Gamma \vdash P : \mathbb{N} \to \mathcal{U} \\ \Gamma \vdash z : P\ \mathtt{zero} \\ \Gamma \vdash s : \Pi[x : \mathbb{N}]\ P\,x \to P\,(\mathtt{suc}\ x) \\ \Gamma \vdash n : \mathbb{N}\end{array}}{\Gamma \vdash \mathtt{ind}\ P\,z\,s\,n : P\,n}\ (\mathbb{N}\mathrm{E})$$

Logically this is the *induction principle*; computationally this is *primitive recursion*.

Natural numbers — computation rules

- Computation:

$$\frac{\begin{array}{l}\Gamma \vdash P : \mathbb{N} \to \mathcal{U} \\ \Gamma \vdash z : P\ \mathtt{zero} \\ \Gamma \vdash s : \Pi[x : \mathbb{N}]\ P\,x \to P\,(\mathtt{suc}\ x)\end{array}}{\Gamma \vdash \mathtt{ind}\ P\,z\,s\ \mathtt{zero} = z \in P\ \mathtt{zero}}\ (\mathbb{N}\mathrm{CZ})$$

$$\frac{\begin{array}{l}\Gamma \vdash P : \mathbb{N} \to \mathcal{U} \\ \Gamma \vdash z : P\ \mathtt{zero} \\ \Gamma \vdash s : \Pi[x : \mathbb{N}]\ P\,x \to P\,(\mathtt{suc}\ x) \\ \Gamma \vdash n : \mathbb{N}\end{array}}{\Gamma \vdash \mathtt{ind}\ P\,z\,s\,(\mathtt{suc}\ n) = s\,n\,(\mathtt{ind}\ P\,z\,s\,n) \in P\,(\mathtt{suc}\ n)}\ (\mathbb{N}\mathrm{CS})$$

Exercise. Define addition and multiplication with `ind`.

Inductively defined sets

The set of natural numbers is *inductively defined*.

In general, every element of an inductively defined set is recursively constructed in a finite number of steps.

Accompanying every inductively defined set is an induction principle, which says that the recursive structure of an element can guide computation, and is formulated as elimination and computation rules in type theory.

In Agda, every datatype can be thought of as being inductively defined, and a structurally recursive function on a datatype (usually using pattern matching) can be thought of as syntactic sugar for an invocation of its induction principle.

Identity types

Identity types are also called *propositional equality*, especially when contrasted with judgemental equality.

- Formation:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathtt{Id}\ A\ t\ u : \mathcal{U}}\ (\mathtt{IdF})$$

- Introduction:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathtt{refl}\ t : \mathtt{Id}\ A\ t\ t}\ (\mathtt{IdI})$$

Exercise. Assume $\Gamma \vdash t = u \in A$ and derive $\Gamma \vdash \mathtt{refl}\ t : \mathtt{Id}\ A\ t\ u$.

Identity types — (simplified) elimination and computation

- Elimination:

$$\frac{\begin{array}{l}\Gamma \vdash P : A \to \mathcal{U} \\ \Gamma \vdash t : A \\ \Gamma \vdash p : P\ t \\ \Gamma \vdash u : A \\ \Gamma \vdash q : \mathtt{Id}\ A\ t\ u\end{array}}{\Gamma \vdash \mathtt{transport}\ P\ p\ q : P\ u}\ (\mathtt{IdE})$$

- Computation:

$$\frac{\Gamma \vdash P : A \to \mathcal{U} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash p : P\ t}{\Gamma \vdash \mathtt{transport}\ P\ p\ (\mathtt{refl}\ t) = p \in P\ t}\ (\mathtt{IdC})$$

Exercise. Prove

$$\begin{array}{l}\Pi[A : \mathcal{U}]\ \Pi[B : \mathcal{U}]\ \Pi[f : A \to B] \\ \quad \Pi[x : A]\ \Pi[y : A]\ \mathtt{Id}\ A\ x\ y \to \mathtt{Id}\ B\ (f\,x)\ (f\,y)\end{array}$$

Id is an equivalence relation

Id is obviously reflexive — we can derive

$$\vdash \lambda A. \lambda x. \mathtt{refl}\ x : \Pi[A : \mathcal{U}]\ \Pi[x : A]\ \mathtt{Id}\ A\ x\ x$$

Exercise. Prove that Id is symmetric and transitive, i.e.,

$$\Pi[A : \mathcal{U}]\ \Pi[x : A]\ \Pi[y : A]\ \mathtt{Id}\ A\ x\ y \to \mathtt{Id}\ A\ y\ x$$

and

$$\begin{array}{l}\Pi[A : \mathcal{U}]\ \Pi[x : A]\ \Pi[y : A]\ \Pi[z : A] \\ \quad \mathtt{Id}\ A\ x\ y \to \mathtt{Id}\ A\ y\ z \to \mathtt{Id}\ A\ x\ z\end{array}$$

Identity types — general elimination and computation

- Elimination:

$$\frac{\begin{array}{l}\Gamma \vdash P : \Pi[x : A]\ \Pi[y : A]\ \mathtt{Id}\ A\ x\ y \to \mathcal{U} \\ \Gamma \vdash p : \Pi[z : A]\ P\ z\ z\ (\mathtt{refl}\ z) \\ \Gamma \vdash t : A \\ \Gamma \vdash u : A \\ \Gamma \vdash q : \mathtt{Id}\ A\ t\ u\end{array}}{\Gamma \vdash \mathtt{J}\ P\ p\ q : P\ t\ u\ q}\ (\mathtt{IdE})$$

- Computation:

$$\frac{\begin{array}{l}\Gamma \vdash P : \Pi[x : A]\ \Pi[y : A]\ \mathtt{Id}\ A\ x\ y \to \mathcal{U} \\ \Gamma \vdash p : \Pi[z : A]\ P\ z\ z\ (\mathtt{refl}\ z) \\ \Gamma \vdash t : A\end{array}}{\Gamma \vdash \mathtt{J}\ P\ p\ (\mathtt{refl}\ t) = p\ t \in P\ t\ t\ (\mathtt{refl}\ t)}\ (\mathtt{IdC})$$

Peano axioms

Peano axioms specify an *equational theory* of natural number arithmetic; all of them are provable in type theory.

- Zero is a natural number. If n is a natural number, so is the successor of n.
  — The introduction rules.
- Equality on natural numbers is an equivalence relation.
  — We use `Id`, which has been proved to be an equivalence relation.
- The successor operation is an injective function, i.e.,

$$\Pi[\,m:\mathbb{N}\,]\ \Pi[\,n:\mathbb{N}\,]\ \texttt{Id}\ \mathbb{N}\ m\ n \leftrightarrow \texttt{Id}\ \mathbb{N}\ (\texttt{suc}\ m)\ (\texttt{suc}\ n)$$

- The successor operation never yields zero, i.e.,

$$\Pi[\,n:\mathbb{N}\,]\ \neg\texttt{Id}\ \mathbb{N}\ (\texttt{suc}\ n)\ \texttt{zero}$$

Peano axioms

- Addition satisfies

$$\Pi[\,n:\mathbb{N}\,]\ \texttt{Id}\ \mathbb{N}\ (\texttt{zero}+n)\ n$$

  and

$$\Pi[\,m:\mathbb{N}\,]\ \Pi[\,n:\mathbb{N}\,]\ \texttt{Id}\ \mathbb{N}\ ((\texttt{suc}\ m)+n)\ (\texttt{suc}\ (m+n))$$

- Multiplication satisfies

$$\Pi[\,n:\mathbb{N}\,]\ \texttt{Id}\ \mathbb{N}\ (\texttt{zero}\times n)\ \texttt{zero}$$

  and

$$\Pi[\,m:\mathbb{N}\,]\ \Pi[\,n:\mathbb{N}\,]\ \texttt{Id}\ \mathbb{N}\ ((\texttt{suc}\ m)\times n)\ (n+m\times n)$$

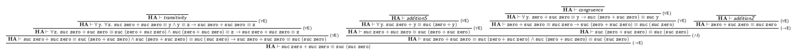
- The induction principle holds for natural numbers.
  — The elimination rule.

Computational foundation

In type theory, Peano “axioms” (formulated as propositions) are merely consequences, and do not play an important role in actual theorem proving.

We now have a more natural foundation of mathematics based on the idea of typed computation.

The infamous proof of $1+1=2$



is just an automatic check by computation in type theory.

FLOLAC'14

Type theory and logic

Lecture IV: meta-theoretical reasoning

4 July 2014

柯向上

Department of Computer Science
University of Oxford

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

Meta-language vs object language

Types and programs form a language, which are talked about by a separate language of judgements and derivations. In this case we call the former the *object language*, and the latter the *meta-language*.

What we write down as types and programs are nothing more than certain syntax trees by themselves; then, at a higher level, we organise and relate these syntax trees with judgements and derivations.

For example, equality judgements are a meta-theoretic notion and cannot be used inside the theory to state equations as provable propositions — we need identity types instead.

Type theory should eat itself

Judgements and derivations can also be regarded as syntax trees to be reasoned about. For example, consistency is a statement in which judgements and derivations are the object language and English is the meta-language. (Canonicity is another example.)

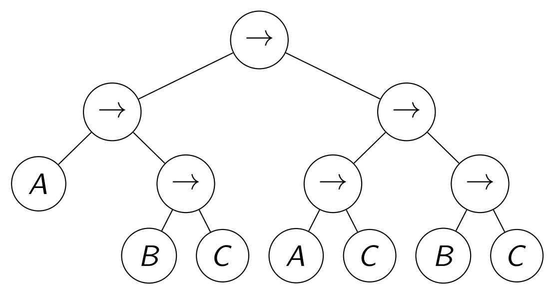
We can also use Agda as the meta-language!

Implicational fragment of propositional logic

Today we consider only propositions formed with implication.

Each of these propositions is a finite tree whose internal nodes are implications and whose leaves are atomic propositions, which are elements of a given set $Var = \{A, B, C, \ldots\}$.

Example. The proposition $(A \to B \to C) \to (A \to C) \to B \to C$ is represented as



Natural deduction

Natural deduction is the type part of simple type theory (and we are considering only the implicational fragment).

$$\frac{}{\Gamma \vdash p}\,\text{(assum)} \quad \text{when } p \in \Gamma$$

$$\frac{\Gamma, p \vdash q}{\Gamma \vdash p \to q}\,(\to\text{I})$$

$$\frac{\Gamma \vdash p \to q \qquad \Gamma \vdash p}{\Gamma \vdash q}\,(\to\text{E})$$

Untyped λ-calculus

A λ-term is either a variable, an abstraction, or an application.

We usually assume *α-equivalence* of λ-terms, i.e., the names of *bound variables* do not matter.

- Change of bound variable names is called *α-conversion*, which has to be *capture-avoiding*, i.e., *free variables* must not become bound after a name change.
- In formalisation, we prefer not to deal with α-equivalence explicitly, and one way is to use *de Bruijn indices* — λ's are nameless, and a bound variable is represented as a natural number indicating to which λ it is bound.

Simply typed λ-calculus (à la Curry)

λ-calculus was designed to model function abstraction and application in mathematics. In untyped λ-calculus, however, we can write nonsensical terms like $\lambda x.\, x\, x$.

We can use the implicational fragement of propositional logic as a type language for λ-calculus, ruling out nonsensical terms.

$$\frac{}{\Gamma \vdash x : p}\,\text{(var)} \quad \text{when } x : p \in \Gamma$$

$$\frac{\Gamma, x : p \vdash t : q}{\Gamma \vdash \lambda x.\, t : p \to q}\,\text{(abs)}$$

$$\frac{\Gamma \vdash t : p \to q \qquad \Gamma \vdash u : p}{\Gamma \vdash t\, u : q}\,\text{(app)}$$

Curry–Howard isomorphism

Derivations in natural deduction and well-typed λ-terms are in one-to-one correspondence.

That is, we can write two functions,

- one mapping a logical derivation in natural deduction to a λ-term and its typing derivation, and
- the other mapping a λ-term with a typing derivation to a logical derivation in natural deduction,

and can prove that the two functions are inverse to each other.

This result is historically significant: two formalisms are developed separately from logical and computational perspectives, yet they coincide perfectly.

Simply typed λ-calculus à la Church

The Curry–Howard isomorphism points out that derivations in natural deduction are actually λ-terms in disguise.

These λ-terms are intrinsically typed, so every term we are able to write down is necessarily well-behaved, whereas in simply typed λ-calculus à la Curry, we can write arbitrary λ-terms, and only rule out ill-behaved ones via typing later.

Semantics

After defining a language (like the implicational fragment of propositional logic), which consists of a bunch of syntax trees, we need to specify what these trees mean.

Judgements and derivations (which form a *deduction system*) assign meaning to the propositional language by specifying how it is used in formal reasoning.

We can also translate the syntax trees into entities in a well understood semantic domain. In the case of propositional logic, we can translate propositional trees to functions on truth values. (This is the classical treatment.)

Two-valued semantics of propositional logic

- Define $\texttt{Bool} := \{\texttt{false}\,, \texttt{true}\}$.
- An *assignment* is a function of type $V \to \texttt{Bool}$.
- A proposition p is translated into a function $[\![p]\!] : (V \to \texttt{Bool}) \to \texttt{Bool}$ mapping assignments to truth values.
- An assignment σ *models* a proposition p exactly when $[\![p]\!]\ \sigma$ is true, and *models* a context Γ exactly when it models every proposition in Γ.

Two-valued semantics of propositional logic

- A proposition p is *satisfiable* exactly when there exists an assignment that models p, and is *valid* exactly when every assignment models p.
- A proposition p is a *semantic consequence* of a context Γ (written $\Gamma \models p$) exactly when every assignment that models Γ also models p.

Exercise. Show that $(p \to p \to q) \to (p \to q)$ is valid for any propositions p and q.

Exercise. Show that a proposition p is valid if and only if p is a semantic consequence of the empty context.

Relationship between deduction systems and semantics

Natural deduction is *sound* with respect to the two-valued semantics: whenever we can deduce $\Gamma \vdash p$, it must be the case that $\Gamma \models p$.

The implicational fragment of propositional logic is also *(semantically) complete* with respect to the two-valued semantics: if $\Gamma \models p$, then we can construct a derivation of $\Gamma \vdash p$.