CS 6110 Lecture 32 — Propositions as Types — 17 April 2009

Lecturer: Andrew Myers

1 Intuitionistic Logic and Constructive Mathematics

It turns out that there is there is a deep connection between the type systems we have been exploring for the lambda calculus, and proof systems for a variety of logic known as intuitionistic logic. Intuitionistic logic is the basis of *constructive mathematics*, which takes a more conservative view of truth than classical mathematics. Constructive mathematics is concerned less with *truth* than with *provability*. Its main proponents were Kronecker and Brouwer around the beginning of the last century. Their views at the time generated great controversy in the mathematical world.

In constructive mathematics, not all deductions of classical logic are considered valid. For example, to prove in classical logic that there exists an object having a certain property, it is enough to assume that no such object exists and derive a contradiction. Intuitionists would not consider this argument valid. Intuitionistically, you must actually construct the object and prove that it has the desired property.

Intuitionists do not accept the law of double negation: $P \leftrightarrow \neg\neg P$. They do believe that $P \rightarrow \neg\neg P$, that is, if P is true then it is not false; but they do not believe $\neg\neg P \rightarrow P$, that is, even if P is not false, then that does not automatically make it true.

Similarly, intuitionists do not accept the law of the excluded middle $P \vee \neg P$. In order to prove $P \vee \neg P$, you must prove either P or $\neg P$. It may well be that neither is provable, in which case the intuitionist would not accept that $P \vee \neg P$.

For intuitionists, the implication $P \rightarrow Q$ has a much stronger meaning than merely $\neg P \vee Q$, as in classical logic. To prove $P \rightarrow Q$, one must show how to construct a proof of Q from any given proof of P. So a proof of $P \rightarrow Q$ is a (computable) function from proofs of P to proofs of Q. Similarly, to prove $P \wedge Q$, you must prove both P and Q; thus a proof of $P \wedge Q$ is a pair consisting of a proof of P and a proof of Q.

1.1 Example

Here is an example of a nonconstructive proof, which would not be accepted by an intuitionist.

Theorem There exist irrational numbers a and b such that a^b is rational.

Proof. Either $\sqrt{2}^{\sqrt{2}}$ is rational or not. If it is, take $a = b = \sqrt{2}$ and we are done. If it is not, take $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$; then $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$, and again we are done. $\square$

Now an intuitionist would not like this, because we haven't actually constructed a definite a and b with the desired property. We have used the law of the excluded middle, which is cheating.

2 Syntax

Syntactically, formulas $\phi, \psi, \ldots$ of intuitionistic logic look the same as their classical counterparts. At the propositional level, we have propositional variables $P, Q, R, \ldots$ and formulas

$$\phi \quad ::= \quad \top \ \mid \ \bot \ \mid \ P \ \mid \ \phi_1 \Rightarrow \phi_2 \ \mid \ \phi_1 \vee \phi_2 \ \mid \ \phi_1 \wedge \phi_2 \ \mid \ \neg\phi.$$

Here $\top$ is "true" and $\bot$ is "false". We might also add a second-order quantifier $\forall P$ ranging over propositions P:

$$\phi \quad ::= \quad \cdots \ \mid \ \forall P\,.\,\phi.$$

3 Natural Deduction (Gentzen, 1943)

Intuitionistic logic uses a sequent calculus to derive the truth of formulas. Assertions are judgements of the form $\phi_1, \ldots, \phi_n \vdash \phi$, which means that ϕ can be derived from the assumptions $\phi_1, \ldots, \phi_n$. If $\vdash \phi$ without assumptions, then ϕ is a theorem of intuitionistic logic. The system is called *natural deduction.*

As we write down the proof rules, it will be clear that they correspond exactly to the typing rules of the pure simply-typed λ-calculus $\lambda^{\to}$ (and with quantifiers, System F). We will show them side by side. There are generally *introduction* and *elimination* rules for each operator.

	intuitionistic logic	*$\lambda^{\to}$ or System F type system*
(axiom)	$\Gamma, \phi \vdash \phi$	$\Gamma, x : \tau \vdash x : \tau$
($\to$-intro)	$\dfrac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \Rightarrow \psi}$	$\dfrac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma .\, e) : \sigma \to \tau}$
($\to$-elim)	$\dfrac{\Gamma \vdash \phi_1 \Rightarrow \phi_2 \quad \Gamma \vdash \phi_1}{\Gamma \vdash \phi_2}$	$\dfrac{\Gamma \vdash e_0 : \sigma \to \tau \quad \Gamma \vdash e_1 : \sigma}{\Gamma \vdash (e_0\ e_1) : \tau}$
($\wedge$-intro)	$\dfrac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}$	$\dfrac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1, e_2) : \sigma * \tau}$
($\wedge$-elim)	$\dfrac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \quad \dfrac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}$	$\dfrac{\Gamma \vdash e : \sigma * \tau}{\Gamma \vdash \#1\, e : \sigma} \quad \dfrac{\Gamma \vdash e : \sigma * \tau}{\Gamma \vdash \#2\, e : \tau}$
($\vee$-intro)	$\dfrac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi} \quad \dfrac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi}$	$\dfrac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathbf{inl}_{\sigma+\tau}\ e : \sigma + \tau} \quad \dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{inr}_{\sigma+\tau}\ e : \sigma + \tau}$
($\vee$-elim)	$\dfrac{\Gamma \vdash \phi \vee \psi \quad \Gamma \vdash \phi \to \chi \quad \Gamma \vdash \psi \to \chi}{\Gamma \vdash \chi}$	$\dfrac{\Gamma \vdash e : \sigma + \tau \quad \Gamma, x{:}\sigma \vdash e_1 : \rho \quad \Gamma, y{:}\tau \vdash e_2 : \rho}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ x.e_1 \mid y.e_2 : \rho}$
($\forall$-intro)	$\dfrac{\Gamma, P \vdash \phi}{\Gamma \vdash \forall P .\, \phi}$	$\dfrac{\Delta, \alpha; \Gamma \vdash e : \tau \quad \alpha \notin FV(\Gamma)}{\Delta; \Gamma \vdash (\Lambda\alpha .\, e) : \forall \alpha .\, \tau}$
($\forall$-elim)	$\dfrac{\Gamma \vdash \forall P .\, \phi}{\Gamma \vdash \phi\{\psi/P\}}$	$\dfrac{\Delta; \Gamma \vdash e : \forall \alpha .\, \tau \quad \Delta \vdash \sigma}{\Delta; \Gamma \vdash (e\ \sigma) : \tau\{\sigma/\alpha\}}$

The elimination rule for $\Rightarrow$ is often called *modus ponens.*

4 The Curry–Howard Isomorphism

The fact that propositions in intuitionistic logic correspond to types in our λ-calculus type systems is known as the *Curry–Howard isomorphism* or the *propositions-as-types* principle. The analogy is far-reaching:

	type theory		*logic*
τ	type	ϕ	proposition
τ	inhabited type	ϕ	theorem
e	well-typed program	π	proof
$\to$	function space	$\to$	implication
$*$	product	$\wedge$	conjunction
$+$	sum	$\vee$	disjunction
$\forall$	type quantifier	$\forall$	2nd order quantifier
B	inhabited type	$\top$	truth
0	uninhabited type	$\perp$	falsity

A proof in intuitionistic logic is a construction, which is essentially a program (λ-term). Saying that a proposition has an intuitionistic or constructive proof says essentially that the corresponding type is inhabited by a λ-term. Since System F is sound and strongly normalizing, that term will evaluate to a value of the same type.

If we are given a well-typed term in System F or $\lambda^{\to}$, then its proof tree will look exactly like the proof tree for the corresponding formula in intuitionistic logic. This means that every well-typed program proves something, i.e. is a proof in constructive logic. Conversely, every theorem in constructive logic corresponds to an inhabited type. Several automated deduction systems (e.g. Nuprl, Coq) are based on this idea.

5 Theorem proving and type checking

We have seen that *type inference* is the process of inferring a type for a given λ-term. Under the Curry–Howard isomorphism, this is the same as determining what theorem a given proof proves. Theorem proving, on the other hand, is going in the opposite direction: Given a formula, does it have a proof? Equivalently, given a type, is it inhabited?

For example, consider the formula expressing transitivity of implication:

$$\forall P, Q, R\,.\ ((P \to Q) \wedge (Q \to R)) \ \to\ (P \to R)$$

Under the Curry–Howard isomorphism, this is related to the type

$$\forall \alpha, \beta, \gamma\,.\ (\alpha \to \beta) * (\beta \to \gamma) \ \to\ (\alpha \to \gamma).$$

If we can construct a term of this type, we will have proved the theorem in intuitionistic logic. The program

$$\Lambda\alpha, \beta, \gamma.\, \lambda p : (\alpha \to \beta) * (\beta \to \gamma).\, \lambda x : \alpha.\, (\#2\, p)\ ((\#1\, p)\ x)$$

does it. This is a function that takes a pair of functions as its argument and returns their composition. The proof tree that establishes the typing of this function is essentially an intuitionistic proof of the transitivity of implication.

Here is another example. Consider the formula

$$\forall P, Q, R\,.\ (P \wedge Q \to R) \ \leftrightarrow\ (P \to Q \to R)$$

The double implication $\leftrightarrow$ is an abbreviation for the conjunction of the implications in both directions. It says that the two formulas on either side are propositionally equivalent. The typed expressions corresponding to each side of the formula above are

$$\alpha * \beta \to \gamma \qquad\qquad \alpha \to \beta \to \gamma.$$

We know that any term of the first type can be converted to one of the second by *currying*, and we can go in the opposite direction by *uncurrying*. The two λ-terms that convert a function to its curried form and back constitute a proof of the logical statement.

6 Uninhabited types

Since the proposition $\perp$ is not provable, it follows that if it corresponds to a type 0, that type must be uninhabited: there is no term with that type. Of course, $\perp$ is not the only uninhabited type; for example, the type $\forall\alpha\,.\,\alpha$ also corresponds to logical falsity and must be uninhabited as well.

Note that we can produce terms with these types if we have recursive functions, as in the following term with type 0:

(rec f: int→0. λx: int. f(x)) 42

However, the typing rule for recursive functions corresponds to a logic rule that makes the logic inconsistent: it assumes what it wants to prove!

$$\frac{\Gamma, y\!:\!\tau \to \tau', x\!:\!\tau \vdash e : \tau'}{\Gamma \vdash \textbf{rec}\ y\!:\!\tau \to \tau'.\lambda x\!:\!\tau.\, e : \tau \to \tau'} \qquad \frac{\Gamma, \phi \Rightarrow \phi', \phi \vdash \phi'}{\Gamma \vdash \phi \Rightarrow \phi'}$$

Thus, we can think of 0 as the type of a term that doesn't actually return to its surrounding context.

7 Continuations and negation

What is the significance of negation? We know that logically $\neg\phi$ is equivalent to $\phi \Rightarrow \bot$, which suggests that we can think of $\neg\phi$ as corresponding to a function $\tau \to 0$. We have seen functions that accept a type and don't return a value before: continuations have that behavior. If ϕ corresponds to τ, a reasonable interpretation of $\neg\phi$ is as a continuation expecting a τ. Negation corresponds to turning outputs into inputs.

As we saw above with currying and uncurrying, meaning-preserving program transformations can have interesting logical interpretations. What about conversion to continuation-passing style? We represent a continuation k expecting a value of type τ as a function with type $\tau \to 0$.

We can then define CPS conversion as a type-preserving translation $[\![\Gamma \vdash e : \tau]\!]$. It is type-preserving in the sense that a well-typed source term ($\Gamma \vdash e : \tau$) translates to a well-typed target term: $\mathcal{G}[\![\Gamma]\!] \vdash [\![\Gamma \vdash e : \tau]\!] : \mathcal{T}[\![\tau]\!]$. In this case the translation of a typing context simply translates all the contained variables: $\mathcal{G}[\![x_1 \!:\! \tau_1, \ldots, x_n \!:\! \tau_n]\!] = x_1 \!:\! \mathcal{T}[\![\tau_1]\!], \ldots, x_n \!:\! \mathcal{T}[\![\tau_n]\!]$. The soundness of the translation can be seen by induction on the typing derivation.

$$\begin{array}{rcl}
[\![\Gamma, x\!:\!\tau \vdash x : \tau]\!] & = & \lambda k\!:\!\mathcal{T}[\![\tau]\!] \to 0.\, k\ x \\
[\![\Gamma \vdash \lambda x\!:\!\tau.\, e : \tau \to \tau']\!] & = & \lambda k\!:\!\mathcal{T}[\![\tau \to \tau']\!] \to 0.\, k\ (\lambda k'\!:\!\mathcal{T}[\![\tau']\!] \to 0.\, \lambda x\!:\!\mathcal{T}[\![\tau]\!].\, [\![\Gamma, x\!:\!\tau \vdash e : \tau']\!]k') \\
[\![\Gamma \vdash e_0\ e_1 : \tau']\!] & = & \lambda k\!:\!\mathcal{T}[\![\tau']\!] \to 0.\, [\![\Gamma \vdash e_0 : \tau \to \tau']\!](\lambda f\!:\!\mathcal{T}[\![\tau \to \tau']\!].\, [\![\Gamma \vdash e_1 : \tau]\!](\lambda v\!:\!\mathcal{T}[\![\tau]\!].\, f\ k\ v))
\end{array}$$

To make this type-check, we define the type translation $\mathcal{T}[\![\cdot]\!]$ as follows:

$$\begin{array}{rcl}
\mathcal{T}[\![B]\!] & = & B \\
\mathcal{T}[\![\tau \to \tau']\!] & = & (\mathcal{T}[\![\tau']\!] \to 0) \to (\mathcal{T}[\![\tau]\!] \to 0)
\end{array}$$

Notice that the logical interpretation of the translation of a function type corresponds to the use of the contrapositive: $(\phi \Rightarrow \psi) \implies (\neg\psi \Rightarrow \neg\phi)$.

By induction on the typing derivation, we can see that CPS conversion converts a source term of type τ into a target term of type $(\mathcal{T}[\![\tau]\!] \to 0) \to 0$. Since programs correspond to proofs, CPS conversion shows how to convert a proof of proposition ϕ into a proof of proposition $\neg\neg\phi$. In other words, CPS conversion proves the admissibility in constructive logic of the rule for introducing double negation:

$$\frac{\phi}{\neg\neg\phi}\ (\textsc{dni})$$

However, we are unable to invert CPS translation, and similarly we are unable (constructively) to *remove* double negation.

8 Other logics

If 2nd-order constructive predicate logic corresponds to System F, do other logics correspond to new kinds of programming language features? This has been an avenue of fruitful exploration over the last couple of decades, with programming-language researchers deriving insights from classical logic, higher-order, and linear logics that help guide the design of useful language features.