CS611 Lecture 34 Curry-Howard Isomorphism November 15, 2000
Scribe: K. O'Neill and M. Kobyakov Lecturer: Andrew Myers

1 Introduction

The Curry-Howard isomorphism is a correspondence between logical formulas and types. The central idea is that each type τ corresponds to a logical formula ϕ, and that a proof that some expression e is well-formed with respect to type τ, (i.e., $\vdash e : \tau$) corresponds to a proof that the logical formula ϕ is logically valid.

Many variations of classical logic have been developed, especially in the last century. We're most interested in establishing an isomorphism between types and logical formulas in *constructive*, or *intuitionistic* logic. In constructive logic, formulas of the form $\phi \vee \neg\phi$ and $\neg\neg\phi \Leftrightarrow \phi$, for example, are not valid. To demonstrate that a formula is valid in constructive logic, it is not enough to demonstrate that its negation is not valid.

2 Logic

To present a formal version of constructive logic, we will use the system of proof construction called "natural deduction", introduced by Gerhard Gentzen in 1934, because it corresponds closely to the way we write type derivations.

A logical formula ϕ is defined inductively as follows:

$$\phi ::= \phi \wedge \phi \mid \phi \vee \phi \mid X \mid \phi \Rightarrow \phi \mid T \mid F \mid \neg\phi \mid \forall X.\phi \mid \exists X.\phi$$

(Note that $\neg\phi$ can also be represented as $\phi \Rightarrow F$.)

To prove a logical formula is true, we have the following inference rules. Here $\Gamma \vdash \phi$ means "ϕ is provable from the set of assumptions Γ". If Γ is empty, $\Gamma \vdash \phi$ simply means "ϕ is valid". We can also write $\emptyset \vdash \phi$, $\vdash \phi$, or simply ϕ to mean the same thing.

Here are some of the inference rules for constructive logic:

$$\frac{\Gamma \vdash \phi_1 \quad \Gamma \vdash \phi_2}{\Gamma \vdash \phi_1 \wedge \phi_2}\ \wedge-\textbf{introduction} \qquad \frac{\Gamma \vdash \phi_1 \wedge \phi_2}{\Gamma \vdash \phi_1}\ \wedge-\textbf{elimination}$$

(The case for ϕ_2 in $\wedge$-elimination is symmetric.)

$$\frac{\Gamma \vdash \phi_1}{\Gamma \vdash \phi_1 \vee \phi_2}\ \vee-\textbf{introduction} \qquad \frac{\Gamma \vdash \phi_1 \vee \phi_2 \quad \Gamma \vdash \phi_1 \Rightarrow \phi_3 \quad \Gamma \vdash \phi_2 \Rightarrow \phi_3}{\Gamma \vdash \phi_3}\ \vee-\textbf{elimination}$$

(The case for ϕ_2 in $\vee$-introduction is symmetric.)

$$\frac{\Gamma, \phi_1 \vdash \phi_2}{\Gamma \vdash \phi_1 \Rightarrow \phi_2}\ \Rightarrow-\textbf{introduction} \qquad \frac{\Gamma \vdash \phi_1 \Rightarrow \phi_2 \quad \Gamma \vdash \phi_1}{\Gamma \vdash \phi_2}\ \Rightarrow-\textbf{elimination}$$

(The common name for the $\Rightarrow$-elimination rule is "modus ponens".)

$$\frac{\Gamma, x \in S \vdash \phi \quad x \text{ fresh in } \Gamma}{\Gamma \vdash \forall x \in S.\phi}\ \forall-\textbf{introduction} \qquad \frac{\Gamma \vdash \forall x \in S.\phi \quad \Gamma \vdash A \in S}{\Gamma \vdash \phi\{A/x\}}\ \forall-\textbf{elimination}$$

Finally, we present the truth axiom:

$$\frac{}{T}$$

which states that T is always valid.

3 Using Assumptions

In a proof tree, we represent an assumption ϕ as $[\phi]$. If we assume ϕ_1 and manage to prove ϕ_2, we can deduce $\phi_1 \Rightarrow \phi_2$. Similarly, if we assume that x is an arbitrary member of some set S and can prove ϕ under this assumption, ϕ holds for all x in the set S. (This is $\forall$-introduction.)

Using the notation of natural deduction, we write $\phi_1, \phi_2, \ldots, \phi_n \vdash \phi$ to mean that ϕ is provable given $\phi_1, \phi_2, \ldots, \phi_n$ as assumptions. As before, we can represent a set of assumptions by Γ, and simply write $\Gamma \vdash \phi$. Here Γ keeps track of what assumptions ϕ could be introduced above this point in the proof tree.

The following rules (including two we've seen before) let us use assumptions to derive valid formulas:

$$\frac{}{\phi \vdash \phi} \qquad \frac{\Gamma, \phi_1 \vdash \phi_2}{\Gamma \vdash \phi_1 \Rightarrow \phi_2} \qquad \frac{\Gamma, x \in S \vdash \phi}{\Gamma \vdash \forall x \in S. \phi}$$

We will now provide a proof that implication is transitive, i.e.,

$$\forall X, Y, Z. (X \Rightarrow Y) \wedge (Y \Rightarrow Z) \Rightarrow (X \Rightarrow Z)$$

Our set of assumptions Γ will have two parts: the current set of variables in scope (e.g. X, Y, Z) and the set of logical assumptions that have been made.

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdots \vdash (X \Rightarrow Y) \wedge (Y \Rightarrow Z)}{\cdots \vdash Y \Rightarrow Z} \quad \dfrac{\dfrac{\cdots \vdash (X \Rightarrow Y) \wedge (Y \Rightarrow Z)}{\cdots \vdash X \Rightarrow Y} \quad \cdots \vdash X}{\cdots \vdash Y}}{X, Y, Z; (X \Rightarrow Y) \wedge (Y \Rightarrow Z), X \vdash Z}}{X, Y, Z; (X \Rightarrow Y) \wedge (Y \Rightarrow Z) \vdash X \Rightarrow Z}}{X, Y, Z; \emptyset \vdash (X \Rightarrow Y) \wedge (Y \Rightarrow Z) \Rightarrow (X \Rightarrow Z)}$$

These proof trees look a lot like typing rule derivation trees, except that ϕ's are supposed to be τ's, and we have to blur our eyes a little bit to ignore the e's. Γ signifies our assumptions of truths, and type expressions are logical statements to prove.

Here are some rules involving pairs and sums, compare them to $\wedge$-introduction, $\wedge$-elimination, $\vee$-introduction and $\vee$-elimination.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \pi_1 e : \tau_1}$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}_{\tau_1 + \tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}_{\tau_1 + \tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \quad \Gamma \vdash e_1 : \tau_1 \to \tau_3 \quad \Gamma \vdash e_2 : \tau_2 \to \tau_3}{\Gamma \vdash (\mathsf{case}\ e_0\ e_1\ e_2) : \tau_3}$$

The rules for $\forall$ introduction and elimination also correspond to the logical rules, when Δ is the set of type variables:

$$\frac{\Delta, X; \Gamma \vdash e : \tau \quad X \notin \Delta}{\Delta; \Gamma \vdash e : \forall X. \tau} \qquad \frac{\Delta; \Gamma \vdash e : \forall x. \tau \quad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e[\tau'] : \tau[\tau'/x]}$$

The modus ponens rule is simply function application in type derivation rules. Similarly, logical implication is abstraction.

$$\frac{\Gamma \vdash e_0 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_0\ e_1 : \tau_2} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}$$

With type derivation, any assumption is automatically true via typing a variable.

$$\frac{}{\Gamma, \phi \vdash \phi} \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}$$

The result of the above correspondences is that any proof that a program is well-typed corresponds to a proof that some formula is valid in constructive logic. This is sometimes called "proofs for free". To translate proofs between the two systems, we can use the following table:

Types	Formulas
well-formed expressions/type derivation	proof
$*, \times$	$\wedge$
$+$	$\vee$
$\rightarrow$	$\Rightarrow$
$\forall, \prod$	$\forall$
unit/B	T
0	F
$\tau \rightarrow 0$	$\neg\phi$
$\cong$	$\Leftrightarrow$

Here the symbol 0 is equivalent of False; we have not seen this type yet. The reason for this is because we never had to deal with types of expressions that never return control to their context functions. Expressions that take an argument of type τ and never return are said to be of type $\tau \rightarrow 0$ in this context.

We have seen expressions like this before in CPS conversion. A continuation is something that we can call and forget about because it does not return control to the caller. We can also give non-terminating computations the type 0 if we can show through the type system that they do not terminate (which we usually cannot do).

Now we are in position to provide a different proof of $\Rightarrow$ being transitive. If in the logical case we had to write a proof tree for $(x \Rightarrow y) \wedge (y \Rightarrow z) \Rightarrow (x \Rightarrow z)$, here we can write an equivalent typed expression with its type derivation. The expression is

$$\Lambda X, Y, Z.\, \lambda p : (X \rightarrow Y) * (Y \rightarrow Z).\, \lambda x : X.\, (\pi_2 p)((\pi_1 p)x)$$

The type derivation for this term has exactly the same structure as the natural-deduction proof given earlier for the logic formula that corresponds to its type. In other words, the structure of the term tells us at each step in the proof of the logical formula which rule to apply! This compact encoding of proofs as terms is actually used in Proof Carrying Code (PCC), where checking that proofs of code safety holds becomes a process of type-checking an appropriate term that encodes the proof.

4 Some DeMorgan tricks

A formula is true only if we can produce a proof for it. A type corresponding to a true formula is a type of the result of some well-typed program. Untrue formulas, like $F \Rightarrow T$, correspond to types for which we can never produce a value. Thus, a type τ can be interpreted as an assertion that the type is *inhabited* by some value other than $\perp$: $\exists v.\ v \neq \perp\ \&\ v \in \mathcal{T}[\![\tau]\!]$. If two formulas are equivalent logically, it says that we can construct a bijection between elements of the two types. For example, $(A \wedge B \Rightarrow C) \Leftrightarrow (A \Rightarrow (B \Rightarrow C))$ is true and the corresponding bijection is curry/uncurry, which demonstrate the isomorphism $(A * B \rightarrow C) \cong (A \rightarrow (B \rightarrow C))$. Other logical equivalences demonstrate even more interesting type equivalences.

For example, here is what happens when apply DeMorgan's Law to implication.

$$\begin{aligned} (A \Rightarrow B) &\Leftrightarrow (\neg A \vee B) \\ &\Leftrightarrow \neg(A \wedge \neg B) \\ &\Leftrightarrow (A \wedge (B \Rightarrow F)) \Rightarrow F \end{aligned}$$

which translates into

$$A \rightarrow B \quad \cong \quad (A * (B \rightarrow 0)) \rightarrow 0$$

This equivalence corresponds to CPS conversion, which converts a function that takes a type A and returns type B, into an equivalent continuation that takes an argument of type A and another continuation to send the result of B to, and never returns.

Similarly, $A \wedge B \Leftrightarrow \neg(\neg A \vee \neg B)$ translates into $A * B \cong ((A \to 0) + (B \to 0)) \to 0$, which suggests (correctly) that there is a twisted way to implement pairs as sums (and vice versa).

On the homework you will explore the equivalence of $\exists x.\phi$ and $\neg\forall x.\neg\phi$.