CS611 Lecture 34 — Curry-Howard Isomorphism — 17 November, 1999

Scribe: Shan-leung Woo — Lecturer: Andrew Myers

1 More on parametric polymorphism

First we need to wrap up our previous discussion on parametric polymorphism. Last lecture we saw the following language constructs:

Functions	$\lambda x.\,\tau e$
Applications	$(e_0\ e_1)$
Type Abstraction	$\Lambda\,\alpha(e)$
Type Instantiation	$e[\tau]$

We want to be able to talk about types in our language. So we introduce $\Gamma\colon \mathbf{var} \rightarrow \tau$ and Δ, which is the set of type variables. Now a type judgement looks like:

$$\Delta; \Gamma \vdash e : \tau$$

To avoid writing types that don't make sense, we need a new kind of judgement, called **well-formed types** judgement:

$$\frac{}{\Delta \cup \{\alpha\} \vdash \alpha} \quad \frac{}{\Delta \vdash \mathbf{bool}} \quad \frac{\Delta \vdash \tau_0 \qquad \Delta \vdash \tau_1}{\Delta \vdash \tau_0 \rightarrow \tau_1} \quad \frac{\Delta \cup \{\alpha_1, \ldots, \alpha_n\} \vdash \tau}{\Delta \vdash \forall \alpha_1, \ldots, \alpha_n.\tau}$$

We also need new rules to talk about $\Lambda\,\alpha(e)$ etc.:

$$\frac{\Delta; \Gamma[x \mapsto \tau] \vdash e : \tau'}{\Delta; \Gamma \vdash \lambda x.\,\tau e : \tau \rightarrow \tau'}$$

$$\frac{\Delta \cup \alpha_1, \ldots, \alpha_n; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\,\alpha_1, \ldots, \alpha_n(e) : \forall \alpha_1, \ldots, \alpha_n.\tau}$$

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha_1, \ldots, \alpha_n.\tau}{\Delta; \Gamma \vdash e[\tau_1, \ldots, \tau_n] : \tau[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]}$$

To finish this section, we note that even though this language is more expressive than simple typed lambda calculus, it is still strongly normalizing and thus not universal. Moreover, type inference for this language is in general not decidable.

2 Polymorphic type equivalence

Our intuition tells us that polymorphic types like $\forall \alpha_1, \ldots, \alpha_n.\tau$ and $\forall \alpha'_1, \ldots, \alpha'_n.\tau$ are equivalent. However, consider the variable $x : \forall \alpha'.\alpha' \rightarrow \alpha'$ and the function $f : (\forall \alpha.\alpha \rightarrow \alpha) \rightarrow \mathbf{bool}$. We cannot just say $(f\ x)$ because $\alpha \neq \alpha'$. Instead, we need to say $(f\ (\Lambda\,\alpha(x[\alpha])))$. This is awkward! So we introduce new typing judgement to relax this restriction. We say $\Delta \vdash \tau_1 = \tau_2$ when τ_1 is equivalent to τ_2 in Δ. Here are the inference rules that specifies this relaxation:

$$\frac{}{\Delta \vdash \mathbf{bool} = \mathbf{bool}}$$

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha = \alpha}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \qquad \Delta \vdash \tau_1 = \tau_2}{\Delta \vdash e : \tau_2}$$

$$\frac{\Delta \vdash \tau_1 = \tau_3 \qquad \Delta \vdash \tau_2 = \tau_4}{\Delta \vdash \tau_1 \to \tau_3 = \tau_2 \to \tau_4}$$

$$\frac{\Delta \cup \{\alpha_1, \ldots, \alpha_n\} \vdash \tau'[\alpha'_1 \mapsto \alpha_1, \ldots, \alpha'_n \mapsto \alpha_n] = \tau \qquad \alpha_i \notin \mathrm{FTV}(\tau')}{\Delta \vdash \forall \alpha_1, \ldots, \alpha_n. = \forall \alpha'_1, \ldots, \alpha'_n.\tau'}$$

3 A set-theoretic model for the polymorphic lambda calculus?

We end this section by noting that in the denotational semantics of our polymorphic language, we can no longer map all types to sets. In the past we have function T that let us denote a type τ by $T[\![\tau]\!]$. But when we have polymorphic type like $\forall\alpha.\tau$, we have trouble constructing the corresponding mapping. Note that $\forall\alpha.\tau$ is a function from set to set. As we know, a function is just a set of ordered pairs. Suppose our T function is:

$$\{\langle T_1, T'_1\rangle, \langle T_2, T'_2\rangle, \ldots\}$$

Consider the type $I = \forall\alpha.\alpha \to \alpha$ and the function $\Lambda\,\alpha(\lambda x.\,\alpha x) : I$. We can apply it to a variable of type I. But this means for some i, T will appear as T'_i in the set T. In other words we are allowing a set to contain itself which is the origin of the famous Russell's Paradox. So we cannot naively write down types as sets of their corresponding values. To develop a model that explains what it going on in this language we will need more sophisticated models than those based on set theory. Category theory, which we will not cover in this course, will allow us to develop models for this and other fancier mechanisms for polymorphism.

4 Curry-Howard isomorphism

The idea behind Curry-Howard isomorphism is that there is an isomorphism between type systems and logic systems: an expression of type τ corresponds to a formula that is provably true in *constructive logic*.

In constructive logic, all proofs must only use the "intuitionistic" part of natural deduction. More precisely, the *reductio ad absurdum* (prove by contradiction) rule is absent. It turns out that the intuitionistic logic are still valid under this interpretation of deduction. But the *law of excluded middle* ($A \vee \neg A$) and thus double negation ($\neg\neg A \Rightarrow A$), which are both valid in classical logic and embody the idea of indirect proof, are in fact not. A consequence of their absence is that in proving $\exists x.A$ we cannot say $\neg(\forall x.\neg A)$; instead we need to give some uniform method to produce x, i.e. specify x *constructively*.

Here are some example of the correspondences:

Type Inferences	Logic Deductions
$\dfrac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$	$\dfrac{A_1 \qquad A_2}{A_1 \wedge A_2}$
$\dfrac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{left } e : \tau_1}$	$\dfrac{A_1 \qquad A_2}{A_1}$
$\dfrac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{inl}^{\tau_1+\tau_2} e : \tau_1 + \tau_2}$	$\dfrac{A_1}{A_1 \wedge A_2}$
$\dfrac{\Gamma \vdash e_0 : \tau_1 + \tau_2 \qquad \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_3 \qquad \Gamma \vdash e_2 : \tau_2 \rightarrow \tau_3}{\Gamma \vdash \text{case } e_0\, e_1\, e_2 : \tau_3}$	$\dfrac{A_1 \vee A_2 \qquad A_1 \Rightarrow A_3 \qquad A_2 \Rightarrow A_3}{A_3}$
$\dfrac{\Gamma \vdash e_0 : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (e_0\ e_1) : \tau_2}$	$\dfrac{A_1 \Rightarrow A_2 \qquad A_1}{A_2}$

But how about this type inference?

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, \tau_1 e : \tau_1 \rightarrow \tau_2}$$

We need something in our logic system to store the logical assumptions we have made up so far in a proof. For technical reasons we use Γ to store labeled logical assumptions. Then the corresponding logic deduction rule is:

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \Rightarrow A_2}$$

The rules for constructive logic are typically written without explicitly representing the set of current assumptions. The rule for proving an implication takes the following form:

$$\dfrac{\begin{matrix} [A_1] \\ \vdots \\ A_2 \end{matrix}}{A_1 \Rightarrow A_2}$$

where brackets are written around A_1 to indicate that it is an assumption that is justified by the use of the rule, and the vertical dots indicate an arbitrary proof tree underneath some occurrences of the assumption. Sometimes matching superscripts are written next to the brackets and the horizontal line of the corresponding rule to show from which rule an assumption originates, as in the following proof of the transitivity of implication:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A]^3 \quad [A \Rightarrow B]^1}{B} \qquad [B \Rightarrow C]^2}{C}}{A \Rightarrow C}\;(3)}{((B \Rightarrow C) \Rightarrow (A \Rightarrow C))}\;(2)}{(A \Rightarrow B) \Rightarrow ((B \Rightarrow C) \Rightarrow (A \Rightarrow C))}\;(1)$$

This notation is equivalent to maintaining the list of legal assumptions that may be asserted in the proof in Γ. Note that the same proof can be expressed much more compactly using the following expression, whose typing proof will be identical to that above:

$$\lambda x\!:\!A \rightarrow B.\, \lambda y\!:\!B \rightarrow C.\, \lambda z\!:\!A.\, (y\ (x\ z))$$

To summarize, we note the following correspondences:

$$
\begin{array}{rcl}
\text{type} & \approx & \text{formula} \\
\text{expression (lambda term)} & \approx & \text{proof of formula} \\
\text{reduction} & \approx & \text{proof normalization} \\
\rightarrow & \approx & \Rightarrow \\
+ & \approx & \vee \\
\times & \approx & \wedge \\
\forall\alpha.\tau & \approx & \forall\alpha.A \\
\textbf{unit (1)} & \approx & \textbf{true} \\
\mathbf{0} & \approx & \textbf{false}
\end{array}
$$

Note that in logic $\neg A$ is just $A \Rightarrow \textbf{false}$. This corresponds to a function of type $\tau \rightarrow 0$ that you can call but it never returns. The type 0 represents a type that has no values; viewed as a set of values, it is the empty set. We can think of a continuation that accepts values of type τ as having this type.

We give some examples of proofs here.

$$(\lambda x\!:\!\tau.\, x)\!:\!\tau \rightarrow \tau \quad \approx \quad A \Rightarrow A$$

$$\Lambda\, \alpha(\lambda x\!:\!\alpha.\, x) : \forall\alpha \rightarrow \alpha \quad \approx \quad \forall\alpha.\alpha \Rightarrow \alpha$$

We can also show that the types $\tau_1 \times \tau_2 \rightarrow \tau_3$ and $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$ are equivalent by constructing the function $\lambda f\!:\!\tau_1 \times \tau_2 \rightarrow \tau_3.\, \lambda a\!:\!\tau_1.\, \lambda b\!:\!\tau_2.\, f\langle a, b\rangle$, and a counterpart function that maps in the other directions.

In classical logic, as mentioned earlier, we would have some additional rules that do not make sense from the standpoint of programming languages. It is actually possible to design a type system whose proofs correspond to proofs in classical logic, but the resulting language is quite unusual.