Class 11:

Curry-Howard correspondence, I

In this class and the next one we'll look at a fascinating connection between logic and computation, and more specifically between intuitionistic logic and typed lamba calculi; the latter are systems designed as abstract models of computation, which also underlie some programming languages (Pascal, Haskell, ...) and form the backbone of foundational frameworks for constructive mathematics such as Martin Löf's type theory. This connection was not discovered at once, but gradually came to be appreciated more and more, and currently plays an important role in theoretical computer science.

The first observations about this correspondence were made by Haskell Curry as early as 1937, relative to a small fragment of the language. This was taken up again and generalized by Howard (1969), whence the name of "Curry-Howard correspondence". For reasons that we will see, other names for the same connection are "propositions-as-types correspondence" and "proofs-as-programs correspondence". The correspondence is a general phenomenon which has many incarnations. We will look at it in the simplest setting where it can be appreciated, namely, in the setting of the simply typed λ-calculus.

In this class I will introduced the simply typed λ-calculus. Then, in the next class we will see how it is related to intuitionistic logic.

The <u>simply typed λ-calculus</u> is a system developed in the 30s by Alonso Church and Haskell Curry as a restricted version of a more general model of computation: Church's <u>untyped λ-calculus</u>.

This is a very simple model of computation. We have terms, denoting programs, that look like this:

$\lambda x.x$ $\rightsquigarrow$ the program that, given an input x, returns x

$\lambda x.\lambda y.xy$ $\rightsquigarrow$ the program that, given x and y, returns the result of applying x to y

Such terms can be applied to each other freely. Computation happens by substitution:

$$(\lambda x.\lambda y.xy)(\lambda x.x) \longrightarrow \lambda y.(\lambda x.x)y \longrightarrow \lambda y.y$$

$\Downarrow$

output of the computation

It might seem that in this model we cannot do very much, since there is so little structure to work with. In fact, however, we can code the natural numbers and <u>all</u> computable functions $f: \mathbb{N}^k \to \mathbb{N}$. I.e., the untyped λ-calculus is <u>Turing-complete</u>.

This makes it a very expressive model, but it also means that it lacks certain desirable properties. In particular, we cannot be sure that a computation will eventually terminate and yield an output.

Here is a simple example:

$\Omega := \lambda x.\, xx$

Consider Ω applied to itself:

$\Omega\Omega = (\lambda x.\, xx)\Omega \to \Omega\Omega$

This means that when we try to compute the value of Ω on itself we end up in a loop and never get a value.

$\Omega\Omega$ loop

In the <u>typed</u> λ-calculus, terms are associated with types which determine which terms they can take in input and what kind of output they return.

Terms in typed λ-calculus look like this:

$\lambda x_{\alpha\to\beta}.\, \lambda y_{\alpha}.\, xy$ $\rightsquigarrow$ the program that, given an input of type $\alpha\to\beta$ and an input of type α, returns the result of applying the first to the second.

Such a term cannot be applied to any term, but only to a term of the type it expects ($\alpha\to\beta$).

The typing system guarantees that computations always terminate. But this also means, by general results of computability theory, that not all computable functions can be expressed in this system.

That is the idea – now let us formally introduce the typed λ-calculus and see how it works.

Def (Types)

Given a set A of atomic types, the set T_A of types over A is given by:

$$\tau ::= \alpha \mid \tau \to \tau \qquad \text{where } \alpha \in A$$

We call this system the <u>simply typed</u> λ-calculus because $\to$ is the only type constructor; more complex λ-calculi have a richer repertoire of constructors.

Think of atomic types as standing for arbitrary sets, and of arrow types as standing for the set of functions from the source type to the target type.

<u>Examples</u>: $A = \{\alpha, \beta\}$

The following are types: α, $\alpha \to \alpha$, $\alpha \to (\beta \to \alpha)$

Each type comes with an infinite stock of variables: $Var_\tau = \{x_\tau^0, x_\tau^1, x_\tau^2, \dots\}$

Next we are going to populate the universe of types. The "inhabitants" of a type are called <u>terms</u> of that type.

Def (Terms)

The set of terms of a type is given by the following inductive rules:

(Variables) if $x \in Var_\tau$ then $x : \tau$

(Abstraction) if $x \in Var_\sigma$ and $M : \tau$ then $(\lambda x. M) : \sigma \to \tau$

(Application) if $M : \sigma \to \tau$, $N : \sigma$ then $(MN) : \tau$

Intuitively, the second rule allows us to build functions, while the third allows us to apply a function to an argument of the appropriate type.

We take an operator 'λx' to bind the variable x in the term to which it is applied. The set $FV(M)$ of variables free in M is defined as usual.
If $FV(M) = \emptyset$ we say that M is a <u>closed</u> term.
If $M : \tau$ for some closed M we say τ is <u>inhabited</u>.

<u>Ex.</u> For every type τ, the type $\tau \to \tau$ is <u>inhabited</u>, since given $x \in Var_\tau$ we have $\lambda x.x : \tau \to \tau$

<u>Ex.</u> For every τ, σ, the type $\tau \to (\sigma \to \tau)$ is inhabited, since given $x \in Var_\tau$, $y \in Var_\sigma$: $\lambda x.\lambda y.x : \tau \to (\sigma \to \tau)$

We regard two terms M and M' as the same if they can be converted into each other by a renaming of bound variables. Thus, eg., if $x, y \in Var_\tau$ then $\lambda x.x = \lambda y.y$.

<u>Def</u> Let M, N be terms, x a variable of the same type as N. Then $M[N/x]$ is the term obtained by replacing each free occurrence of x in M by N, taking care that no $y \in FV(N)$ ends up bound in the process.

<u>Computation</u>: $\underbrace{(\lambda x.M)N}_{\text{redex}} \to_\beta \underbrace{M[N/x]}_{\text{reduct}}$

basic β-reduction step

<u>Def</u> (β-reduction)

- $M \to_\beta N$ if N is obtained from M by replacing a redex with its reduct.

  Notice that in general $M \to_\beta N$ for several N, since M may contain multiple redexes.

- $M \twoheadrightarrow_\beta N$ if there is a chain $M \to_\beta \cdots \to_\beta N$ (possibly of length 0, so we have $M \twoheadrightarrow_\beta M$).

- M is in <u>normal form</u> if $\forall N : M \not\to_\beta N$

<u>Example</u> let $z, z', x, y \in Var_\alpha$.

$$((\lambda x.\lambda y.x)z)z' \to_\beta (\lambda y.z)z' \to_\beta z$$

We think of the β-reduction process as the <u>computation</u> of a term, and of the normal form reached in this way as the <u>output</u>.

The following are key results of the theory of the simply typed λ-calculus.

<u>Theor</u> (Church-Rosser)

If $M \twoheadrightarrow_\beta M'$ and $M \twoheadrightarrow_\beta M''$ then for some term N, $M' \twoheadrightarrow_\beta N$ and $M'' \twoheadrightarrow_\beta N$.

This expresses a <u>confluence property</u> of the β-reduction process:

$$\begin{tikzcd}
 & M' \arrow[dr, dotted] & \\
M \arrow[ur, dotted] \arrow[dr, dotted] & & N \\
 & M'' \arrow[ur, dotted] &
\end{tikzcd}$$

<u>Theor</u> (Weak normalization)

For any M there is always N in normal form such that $M \twoheadrightarrow_\beta N$

This says that given a term, there is always a way to get to its value (in finitely many steps). That is, computations can always terminate if we reduce in a suitable way.

As a corollary of weak normalization and Church-Rosser we have that the "value" that we get from a computation of a term is uniquely determined.

<u>Cor</u> (Uniqueness of normal form)

If $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta N'$, and if N and N' are in normal form, then $N = N'$.

Denote the unique N in n.f. s.t. $M \twoheadrightarrow_\beta N$ as $nf(M)$. Think of this as the <u>value</u> of the term M.

Finally, the following result says that any chain of β-reduction from a term will eventually terminate, leading to its value.

<u>Theor</u> (Strong normalization)

There are no infinite chains $M \to_\beta M' \to_\beta \cdots$

<u>Illustration</u>

Let's look at how natural numbers and a computable function like product can be encoded in typed λ-calculus.

$$\mathbb{N} := (\alpha \to \alpha) \to (\alpha \to \alpha) \quad \text{for } \alpha \in A$$

$$\bar{n} := \lambda f_{\alpha\to\alpha}.\ \lambda x_\alpha.\ \underbrace{f(\cdots(fx))}_{n \text{ occurrences of } f}$$

$$\mathrm{Prod} : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

$$\mathrm{Prod} := \underbrace{\lambda x_{\mathbb{N}}.\ \lambda y_{\mathbb{N}}.\ \lambda f_{\alpha\to\alpha}.\ \underbrace{x\underbrace{(yf)}_{\alpha\to\alpha}}_{\alpha\to\alpha}}_{\mathbb{N}}$$

Let us try to compute $\mathrm{Prod}(1)(2)$

$$\begin{aligned}
(\mathrm{Prod}\ \bar{1})\ \bar{2} &= ((\lambda x.\lambda y.\lambda f.x(yf))\ \bar{1})\ \bar{2} \\
&\to_\beta (\lambda y.\lambda f.\bar{1}(yf))\ \bar{2} \\
&\to_\beta \lambda f.\bar{1}(\bar{2}f) \\
&\twoheadrightarrow_\beta \lambda f.\bar{1}(\lambda x.f(fx)) \qquad \text{(see first } \beta\text{-reduction on the next sheet)} \\
&\twoheadrightarrow_\beta \lambda f.\lambda x.f(fx) = \bar{2}
\end{aligned}$$

This shows how the product of 1 and 2 indeed computes to 2, as we expect.

$$\bar{2}f = (\lambda f.\lambda x.f(fx))\,f$$

$$= (\lambda g.\lambda x.g(gx))\,f \quad \text{rename bound variable for convenience}$$

$$\rightarrow_\beta \lambda x.f(fx)$$

$$\bar{1}(\lambda x.f(fx)) = (\lambda f.\lambda x.fx)(\lambda x.f(fx))$$

$$= (\lambda g.\lambda y.gy)(\lambda x.f(fx)) \quad \text{rename bound vars for convenience}$$

$$\rightarrow_\beta \lambda y.((\lambda x.f(fx))\,y)$$

$$\rightarrow_\beta \lambda y.f(fy)$$

$$= \lambda x.f(fx) \quad \text{rename bound variables for convenience}$$