Lecture 31:

- λ-calculus cont

Announcements:

- HW-6 out

The Lambda (λ) Calculus

From λ-calculus to functional programming

- TMs are (roughly) the computation model behind imperative languages
- λ-calculus is (roughly) the computation model behind functional languages

Basic idea of λ-calculus

1. Unnamed, single-variable functions (λ "functions" aka "abstractions")
   - $\lambda x.x$ takes an x and returns an x
   - $\lambda x.(\lambda y.x)$ takes x and returns a function that takes y and returns x
   - shorthand for multi-argument functions: $\lambda xy.x$
2. Function application
   - $(\lambda x.x)0$ applies the identity function to 0 (resulting in 0)
   - $(\lambda x.(\lambda y.x))ab$ reduces to a ... $(\lambda x.(\lambda y.x))ab \Rightarrow (\lambda y.a)b \Rightarrow a$
3. Expressions
   - Either a function, an application, a variable, or a constant
   - A function has the form: $\lambda x.e$ where x is a name and e an expression
   - An application has the form: $e_1 e_2$ where both e's are expressions

Computation in λ-calculus is via function application

- Given a function application such as:

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

- An application is evaluated by substituting x's in the function body with y:

$$(\lambda x.x)y = [y/x]x = y$$

Representing the values "true" and "false" (here as substitutions):

$$T \equiv \lambda x.(\lambda y.x) \qquad \text{(True)}$$
$$F \equiv \lambda x.(\lambda y.y) \qquad \text{(False)}$$

We can use these to define basic logical operators (AND, OR, NOT):

$$\text{AND} \equiv \lambda x.(\lambda y.xy(\lambda u.(\lambda v.v))) \equiv \lambda x.(\lambda y.xyF)$$
$$\text{OR} \equiv \lambda x.(\lambda y.x(\lambda u.(\lambda v.u))y) \equiv \lambda x.(\lambda y.xTy)$$
$$\text{NOT} \equiv \lambda x.x(\lambda u.(\lambda v.v))(\lambda y.(\lambda z.y)) \equiv \lambda x.xFT$$

Examples: ... note prefix notation, e.g., AND T T

$$\text{NOT } T \Rightarrow (\lambda x.xFT)T \Rightarrow TFT \Rightarrow (\lambda x.(\lambda y.x))FT \Rightarrow (\lambda y.F)T \Rightarrow F$$

$$\text{NOT } F \Rightarrow (\lambda x.xFT)F \Rightarrow FFT \Rightarrow (\lambda x.(\lambda y.y))FT \Rightarrow (\lambda y.y)T \Rightarrow T$$

$$\text{AND } T\ T \Rightarrow (\lambda x.(\lambda y.xyF))TT \Rightarrow (\lambda y.TyF)T \Rightarrow TTF \Rightarrow (\lambda x.(\lambda y.x))TF$$
$$\Rightarrow (\lambda y.T)F = T$$

$$\text{AND } T\ F \Rightarrow (\lambda x.(\lambda y.xyF))TF \Rightarrow (\lambda y.TyF)F \Rightarrow TFF \Rightarrow (\lambda x.(\lambda y.x))FF$$
$$\Rightarrow (\lambda y.F)F = F$$

$$\text{OR } F\ T \Rightarrow (\lambda x.(\lambda y.xTy))FT \Rightarrow (\lambda y.FTy)T \Rightarrow FTT \Rightarrow (\lambda x.(\lambda y.y))TT$$
$$\Rightarrow (\lambda y.y)T \Rightarrow T$$

Note: Can use for conditionals ($c\ e_1\ e_2$) representing: `IF` c `THEN` e_1 `ELSE` e_2

Can express recursion using λ-calculus ... called a "**Y combinator**"

$$R \equiv (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))$$

- The basic idea is that R calls a function y then "regenerates" itself
- For example, applying R to a function g yields:

$$\begin{aligned}
R_g &= (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))g && (1)\\
&= (\lambda x.g(xx))(\lambda x.g(xx)) && (2)\\
&= g((\lambda x.g(xx))(\lambda x.g(xx))) && (3)\\
&= g(R_g) && (4)\\
&= g(g(R_g)) && (5)\\
&= \text{ and so on} && (6)
\end{aligned}$$

- Note in (4) that $g(R_g)$ since $R_g = (\lambda x.g(xx))(\lambda x.g(xx))$ from (2)
- We can stop recursion using conditional functions (similar to Boolean ops)

As the examples show:

- λ calculus is inherently "higher order" – functions can be passed as arguments
- all functions can be thought of as having a single argument (called "*currying*")
- allows for "partial" function application ... e.g.: $(\lambda x.(\lambda y.+\,x\,y))\,3\,4$

Different paradigms, same power ...:

λ-calculus and Turing Machines have the same expressive power!