Lecture 29:

- λ-calculus (cont)

Announcements:

- HW-6 out
- Extra Credit proposal due
- Exam 2 Mon

From λ-Calculus to Functional Programming

TMs are (roughly) the MoC for imperative languages
... λ-calculus is (roughly) the MoC for 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 ... $(\lambda x.(\lambda y.x))ab \Rightarrow (\lambda y.a)b \Rightarrow a$
- ... where $\Rightarrow$ denotes a one-step *application*

The λ-Calculus

(3) Expressions

- Either a function, an application, a variable, or a constant
- General form of a function: $\lambda x.e$ where x is a variable 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 an expression (function application) such as:

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

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

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

The λ-Calculus

Can represent "true" and "false" as expressions (function applications)

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

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

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

$$\text{AND} \equiv \lambda x.(\lambda y.xy(\lambda u.(\lambda v.v))) \equiv \lambda x.(\lambda y.xyF)$$

$$\text{OR} \equiv \lambda x.(\lambda y.x(\lambda u.(\lambda v.u))y) \equiv \lambda x.(\lambda y.xTy)$$

$$\text{NOT} \equiv \lambda x.x(\lambda u.(\lambda v.v))(\lambda y.(\lambda z.y)) \equiv \lambda x.xFT$$

The λ-Calculus

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

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

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

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

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

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 an expression $(c\ e_1\ e_2)$ to represent: `IF` c `THEN` e_1 `ELSE` e_2

- e.g., $T\, e_1\, e_2$ means `IF` T `THEN` e_1 `ELSE` e_2

The λ-Calculus

Can also express recursion ... called a "**Y combinator**"

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

Basic idea: 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)

... can stop recursion using conditionals

The λ-Calculus

As the examples show:

- λ calculus is inherently ***higher order*** – functions passed as arguments
- all functions are single argument ... enables ***currying***
- allows for ***partial function application*** ... e.g.: `add_one` $\equiv (\lambda x.(\lambda y. + x\, y))\, 1$

Different paradigms, same power ...

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