Introductory Course
on Logic and Automata Theory

Introduction to the lambda calculus

Polyvios.Pratikakis@imag.fr

Based on slides by Jeff Foster, UMD

History

- Formal mathematical system
- Simplest programming language
- Intended for studying functions, recursion
- Invented in 1936 by Alonzo Church (1903-1995)
  - Church's Thesis:
    - "*Every effectively calculable function (effectively decidable predicate) is general recursive*"
    - i.e. can be computed by lambda calculus
  - Church's Theorem:
    - First order logic is undecidable

Syntax

- Simple syntax:

$$\begin{array}{rcll} e & ::= & x & \text{Variables} \\ & | & \lambda x.e & \text{Functions} \\ & | & e\ e & \text{Function applications} \end{array}$$

- Pure lambda calculus: only functions
  - Arguments are functions
  - Returned value is function
  - A function on functions is *higher-order*

Semantics

- Evaluating function application: $(\lambda x.e_1)\ e_2$
  - Replace every x in e_1 with e_2
  - Evaluate the resulting term
  - Return the result of the evaluation
- Formally: "β-reduction"
  - $(\lambda x.e_1)\ e_2 \rightarrow_\beta e_1[e_2/x]$
  - A term that can be β-reduced is a *redex*
  - We omit β when obvious

Convenient assumptions

- Syntactic sugar for declarations
  - let $x = e_1$ in e_2
- Scope of λ extends as far to the right as possible
  - $\lambda x.\lambda y.x\,y$ is $\lambda x.(\lambda y.(x\,y))$
- Function application is left-associative
  - $x\,y\,z$ means $(x\,y)\,z$

Scoping and parameter passing

- β-reduction is not yet well-defined:
  - $(\lambda x.e_1)\ e_2 \rightarrow e_1[e_2/x]$
  - There might be many x defined in e_1
- Example
  - Consider the program
    let $x = a$ in
    let $y = \lambda z.x$ in
    let $x = b$ in
    $y\ x$
  - Which x is bound to a, and which to b?

Lexical scoping

- Variable refers to closest definition
- We can rename variables to avoid confusion:
  let $x = a$ in
  let $y = \lambda z.x$ in
  let $w = b$ in
  $y\ w$

Free/bound variables

- The set of *free variables* of a term is

$$
\begin{aligned}
FV(x) &= x \\
FV(\lambda x.e) &= FV(e) \setminus \{x\} \\
FV(e_1\, e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}
$$

- A term e is *closed* if $FV(e) = \emptyset$
- A variable that is not free is *bound*

α-conversion

- Terms are equivalent up to renaming of bound variables
  - $\lambda x.e = \lambda y.e[y/x]$ if $y \notin FV(e)$
  - Renaming of bound variables is called α-conversion
  - Used to avoid having duplicate variables, capturing during substitution

Substitution

- Formal definition

$$
\begin{array}{rcll}
x[e/x] & = & e & \\
y[e/x] & = & y & \text{when } x \neq y \\
(e_1\ e_2)[e/x] & = & (e_1[e/x]\ e_2[e/x]) & \\
(\lambda y.e_1)[e/x] & = & \lambda y.(e_1[e/x]) & \text{when } y \neq x \text{ and } y \notin FV(e)
\end{array}
$$

- Example
  - $(\lambda x.y\ x)\ x =_\alpha (\lambda w.y\ w)\ x \rightarrow_\beta y\ x$
  - We omit writing α-conversion

Functions with many arguments

- We can't yet write functions with many arguments
  - For example, two arguments: $\lambda(x,y).e$
- Solution: take the arguments, one at a time
  - $\lambda x.\lambda y.e$
  - A function that takes x and returns another function that takes y and returns e
  - $(\lambda x.\lambda y.e)\ a\ b \rightarrow (\lambda y.e[a/x])\ b \rightarrow e[a/x][b/y]$
  - This is called *Currying*
  - Can represent any number of arguments

Representing booleans

- true $= \lambda x.\lambda y.x$
- false $= \lambda x.\lambda y.y$
- if a then b else $c = a\ b\ c$
- For example:
  - if true then b else $c \longrightarrow (\lambda x.\lambda y.x)\ b\ c \longrightarrow (\lambda y.b)\ c \longrightarrow b$
  - if false then b else $c \longrightarrow (\lambda x.\lambda y.y)\ b\ c \longrightarrow (\lambda y.y)\ c \longrightarrow c$

Combinators

- Any closed term is also called a *combinator*
  - true and false are combinators
- Other popular combinators:
  - $I = \lambda x.x$
  - $K = \lambda x.\lambda y.x$
  - $S = \lambda x.\lambda y.\lambda z.x\ z\ (y\ z)$
  - We can define calculi in terms of combinators
    - The SKI-calculus
    - SKI-calculus is also Turing-complete

Encoding pairs

- $(a,b) = \lambda x.\text{if } x \text{ then } a \text{ else } b$
- $\text{fst} \ = \lambda p.p \text{ true}$
- $\text{snd} \ = \lambda p.p \text{ false}$
- Then
  - $\text{fst } (a,b) \rightarrow \ldots \rightarrow a$
  - $\text{snd } (a,b) \rightarrow \ldots \rightarrow b$

Natural numbers (Church)

- $0 = \lambda x.\lambda y.y$
- $1 = \lambda x.\lambda y.xy$
- $2 = \lambda x.\lambda y.x\ (x\ y)$
- i.e. $n = \lambda x.\lambda y.\langle\text{apply } x\ n \text{ times to } y\rangle$
- $\mathsf{succ} = \lambda z.\lambda x.\lambda y.x\ (z\ x\ y)$
- $\mathsf{iszero} = \lambda z.z\ (\lambda y.\mathsf{false})\ \mathsf{true}$

Natural numbers (Scott)

- $0 = \lambda x.\lambda y.x$
- $1 = \lambda x.\lambda y.y\ 0$
- $2 = \lambda x.\lambda y.y\ 1$
- i.e. $n = \lambda x.\lambda y.y\ (n-1)$
- $\mathsf{succ}\ = \lambda z.\lambda x.\lambda y.yz$
- $\mathsf{pred}\ = \lambda z.z\ 0\ (\lambda x.x)$
- $\mathsf{iszero}\ = \lambda z.z\ \mathsf{true}\ (\lambda x.\mathsf{false})$

Nondeterministic semantics

$$\frac{}{(\lambda x.e_1)\ e_2 \longrightarrow e_1[e_2/x]} \qquad \frac{e \longrightarrow e'}{(\lambda x.e) \longrightarrow (\lambda x.e')}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{e_1\ e_2 \longrightarrow e_1\ e_2'}$$

Question: why is this semantics non-deterministic?

Example

- We can apply reduction anywhere in the term
  - $(\lambda x.(\lambda y.y)\ x\ ((\lambda z.w)\ x) \longrightarrow \lambda x.(x\ ((\lambda z.w)\ x) \longrightarrow \lambda x.x\ w$
  - $(\lambda x.(\lambda y.y)\ x\ ((\lambda z.w)\ x) \longrightarrow \lambda x.(\lambda y.y)\ x\ w \longrightarrow \lambda x.x\ w$
- Does the order of evaluation matter?

The Church-Rosser Theorem

- Lemma (The Diamond Property):
  - If $a \longrightarrow b$ and $a \longrightarrow c$, then there exists d such that $b \longrightarrow^{*} d$ and $c \longrightarrow^{*} d$
- Church-Rosser theorem:
  - If $a \longrightarrow^{*} b$ and $a \longrightarrow^{*} c$, then there exists d such that $b \longrightarrow^{*} d$ and $c \longrightarrow^{*} d$
  - Proof by diamond property
- Church-Rosser also called *confluence*

Normal form

- A term is in *normal form* if it cannot be reduced
  - Examples: $\lambda x.x$, $\lambda x.\lambda y.z$
- By the Church-Rosser theorem, every term reduces to at most one normal form
  - Only for pure lambda calculus with non-deterministic evaluation
- Notice that for function application, the argument need not be in normal form

β-equivalence

- Let $=_\beta$ be the reflexive, symmetric, transitive closure of $\longrightarrow$
  - E.g., $(\lambda x.x)\, y \longrightarrow y \longleftarrow (\lambda z.\lambda w.z)\, y\, y$ so all three are β-equivalent
- If $a =_\beta b$, then there exists c such that $a \longrightarrow^* c$ and $b \longrightarrow^* c$
  - Follows from Church-Rosser theorem
- In particular, if $a =_\beta b$ and both are normal forms, then they are equal

Not every term has a normal form

- Consider
  - $\Delta = \lambda x.x\,x$
  - Then $\Delta\,\Delta \rightarrow \Delta\,\Delta \rightarrow \cdots$
- In general, *self application* leads to loops
- ...which is good if we want recursion

Fixpoint combinator

- Also called a paradoxical combinator
  - $Y = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$
  - There are many versions of the Y combinator
- Then, $Y\ F =_\beta F\ (Y\ F)$
  - $Y\ F = (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ F$
  - $\longrightarrow (\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x))$
  - $\longrightarrow F\ ((\lambda x.F\ (x\ x))\ (\lambda x.F\ (x\ x)))$
  - $\longleftarrow F\ (Y\ F)$

Example

- $fact(n) = \text{if } (n = 0) \text{ then } 1 \text{ else } n * fact(n-1)$
- Let $G = \lambda f.\lambda n.\text{if } (n = 0) \text{ then } 1 \text{ else } n * f(n-1)$
- $Y\ G\ 1 =_\beta G\ (Y\ G)\ 1$
  - $=_\beta (\lambda f.\lambda n.\text{if } (n = 0) \text{ then } 1 \text{ else } n * f(n-1))\ (Y\ G)\ 1$
  - $=_\beta \text{if } (1 = 0) \text{ then } 1 \text{ else } 1 * ((Y\ G)\ 0)$
  - $=_\beta \text{if } (1 = 0) \text{ then } 1 \text{ else } 1 * (G\ (Y\ G)\ 0)$
  - $=_\beta \text{if } (1 = 0) \text{ then } 1 \text{ else } 1 * (\lambda f.\lambda n.\text{if } (n = 0) \text{ then } 1 \text{ else } n * f(n-1)\ (Y\ G)\ 0)$
  - $=_\beta \text{if } (1 = 0) \text{ then } 1 \text{ else } 1 * (\text{if } (0 = 0) \text{ then } 1 \text{ else } 0 * ((Y\ G)\ 0))$
  - $=_\beta 1 * 1 = 1$

In other words

- The Y combinator "unrolls" or "unfolds" its argument an infinite number of times
  - $Y\ G = G\ (Y\ G) = G\ (G\ (Y\ G)) = G\ (G\ (G\ (Y\ G))) = \ldots$
  - G needs to have a "base case" to ensure termination
- But, only works because we follow call-by-name
  - Different combinator(s) for call-by-value
  - $Z = \lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))\ (\lambda x.f\ (\lambda y.x\ x\ y))$
  - Why is this a fixed-point combinator? How does its difference from Y work for call-by-value?

Why encodings

- It's fun!
- Shows that the language is expressive
- In practice, we add constructs as languages primitives
  - More efficient
  - Much easier to analyze the program, avoid mistakes
  - Our encodings of 0 and `true` are the same, we may want to avoid mixing them, for clarity

Lazy and eager evaluation

- Our non-deterministic reduction rule is fine for theory, but awkward to implement
- Two deterministic strategies:
  - *Lazy*: Given $(\lambda x.e_1)\ e_2$, do not evaluate e_2 if e_1 does not need x anywhere
    - Also called left-most, call-by-name, call-by-need, applicative, normal-order evaluation (with slightly different meanings)
  - *Eager*: Given $(\lambda x.e_1)\ e_2$, always evaluate e_2 to a normal form, before applying the function
    - Also called call-by-value

Lazy operational semantics

$$\frac{}{(\lambda x.e_1) \rightarrow^l (\lambda x.e_1)}$$

$$\frac{e_1 \rightarrow^l \lambda x.e \qquad e[e_2/x] \rightarrow^l e'}{e_1\ e_2 \rightarrow^l e'}$$

- The rules are deterministic, *big-step*
  - The right-hand side is reduced "all the way"
- The rules do not reduce under λ
- The rules are normalizing:
  - If a is closed and there is a normal form b such that $a \rightarrow^* b$, then $a \rightarrow^l d$ for some d

Eager (big-step) semantics

$$\frac{}{(\lambda x.e_1) \longrightarrow^e (\lambda x.e_1)}$$

$$\frac{e_1 \longrightarrow^e \lambda x.e \qquad e_2 \longrightarrow^e e' \qquad e[e'/x] \longrightarrow^e e''}{e_1\ e_2 \longrightarrow^e e''}$$

- This big-step semantics is also deterministic and does not reduce under λ
- But is not normalizing!
  - Example: let $x = \Delta\,\Delta$ in $(\lambda y.y)$

Lazy vs eager in practice

- Lazy evaluation (call by name, call by need)
  - Has some nice theoretical properties
  - Terminates more often
  - Lets you play some tricks with “infinite” objects
  - Main example: Haskell
- Eager evaluation (call by value)
  - Is generally easier to implement efficiently
  - Blends more easily with side-effects
  - Main examples: Most languages (C, Java, ML, . . .)

Functional programming

- The λ calculus is a prototypical functional programming language
  - Higher-order functions (lots!)
  - No side-effects
- In practice, many functional programming languages are not "pure": they permit side-effects
  - But you're supposed to avoid them...

Functional programming today

- Two main camps
  - Haskell – Pure, lazy functional language; no side-effects
  - ML (SML, OCaml) – Call-by-value, with side-effects
- Old, still around: Lisp, Scheme
  - Disadvantage/feature: no static typing

Influence of functional programming

- Functional ideas move to other langauges
  - Garbage collection was designed for Lisp; now most new languages use GC
  - Generics in C++/Java come from ML polymorphism, or Haskell type classes
  - Higher-order functions and closures (used in Ruby, exist in C#, proposed to be in Java soon) are everywhere in functional languages
  - Many object-oriented abstraction principles come from ML's module system
  - ...