Lecture 5: The Untyped λ-Calculus

Syntax and basic examples

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Programming Languages

Motivation

- Common programming languages are complex
  - ANSI C99: 538 pages
  - ANSI C++: 714 pages
  - Java 2.0: 505 pages
- Not ideal for teaching and understanding principles of languages and program analysis
- Ideal: a "core language" with
  - Essential features enough to express all computation
  - No redundancy: encode extra features as "syntactic sugar"

Lambda Calculus

- Core language for sequential programming
- Can express all computation
  - Still extremely simple and minimal
  - Can encode many extensions as syntactic sugar
- Easy to extend with additional features
- Simple to understand
  - Whole definition in one slide

History

- Invented in the 1930s by Alonzo Church (1903-1995)
- Princeton Mathematician
- Lectures on λ-calculus published in 1941
- Also known for
  - Church's Thesis:
    - "*Every effectively calculable (decidable) function can be expressed by recursive functions*"
    - i.e. can be computed by λ-calculus
  - Church's Theorem:
    - The first order logic is undecidable

Syntax

- Simple syntax:

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

- Functions are the only language construct
  - The argument is a function
  - The result is a function
  - Functions of functions are *higher-order*

Semantics

- To evaluate the term $(\lambda x.e_1)\ e_2$
  - Replace every x in e_1 with e_2
    - Written as $e_1[e_2/x]$, pronounced "e_1 with e_2 for x"
    - Also written $e_1[x \mapsto e_2]$
  - Evaluate the resulting term
  - Return the result
- Formally called "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 really means $(\lambda x.e_2)\ e_1$
- 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?

Static (Lexical) Scope

- 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
  $y\ w$
- Renaming variables without changing the program meaning is called "α-conversion"

Free/bound variables

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

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

- 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 $y \notin FV(e)$
  - Used to avoid having duplicate variables, capturing during substitution
  - This is called α-conversion, used implicitly

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
  - $(\lambda x.y\, x)\ x =_\alpha (\lambda w.y\, w)\ x \to_\beta y\, x$
  - 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 (like we do in OCaml)
  - $\lambda x.\lambda y.e$
  - A function that takes x and returns another function that takes y and returns e
  - $(\lambda x.\lambda y.e)\ a\ b \to (\lambda y.e[a/x])\ b \to e[a/x][b/y]$
  - 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 $c \rightarrow (\lambda x.\lambda y.x)\ b\ c \rightarrow (\lambda y.b)\ c \rightarrow b$
  - if false then b else $c \rightarrow (\lambda x.\lambda y.y)\ b\ c \rightarrow (\lambda y.y)\ c \rightarrow c$

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)

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

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. $n = \lambda x.\lambda y.y\, (n-1)$
- $\text{succ} \;= \lambda z.\lambda x.\lambda y.y\, z$
- $\text{pred} \;= \lambda z.z\, 0\ (\lambda x.x)$
- $\text{iszero} \;= \lambda z.z\, \text{true}\ (\lambda x.\text{false})$

Nondeterministic semantics

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

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

Question: why are these rules non-deterministic?

Example

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

The Church-Rosser Theorem

- Lemma (The Diamond Property):
  - If $a \rightarrow b$ and $a \rightarrow c$, then there exists d such that $b \rightarrow^* d$ and $c \rightarrow^* d$
- Church-Rosser theorem:
  - If $a \rightarrow^* b$ and $a \rightarrow^* c$, then there exists d such that $b \rightarrow^* d$ and $c \rightarrow^* d$
  - 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 $\rightarrow$
  - E.g., $(\lambda x.x)\ y \rightarrow y \leftarrow (\lambda z.\lambda w.z)\ y\ y$ so all three are β-equivalent
- If $a =_\beta b$, then there exists c such that $a \rightarrow^* c$ and $b \rightarrow^* c$
  - 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 this combinator
- Then, $Y\,F =_\beta F\,(Y\,F)$
  - $Y\,F = (\lambda f.(\lambda x.f\,(x\,x))\;(\lambda x.f\,(x\,x)))\;F$
  - $\rightarrow (\lambda x.F\,(x\,x))\;(\lambda x.F\,(x\,x))$
  - $\rightarrow F\,((\lambda x.F\,(x\,x))\;(\lambda x.F\,(x\,x)))$
  - $\leftarrow F\,(Y\,F)$

Example

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

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 language 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

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

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

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

Eager Fixpoint

- The Y combinator works for lazy semantics
  - $Y = \lambda f.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x))$
- The Z combinator does the same for eager (call-by-value) semantics
  - $Z = \lambda f.(\lambda x.f\,(\lambda y.x\,x\,y))(\lambda x.f\,(\lambda y.x\,x\,y))$
  - Why doesn't the Y combinator work for call-by-value?
  - Why does Z do the same thing for call-by-value?

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#, Java, C++) are everywhere in functional languages
  - Many object-oriented abstraction principles come from ML's module system
  - ...