Lambda Calculus Review

Why Study Lambda Calculus?

- It is a “core” language
  - Very small but still Turing complete
- But with it can explore general ideas
  - Language features, semantics, proof systems, algorithms, …
- Plus, higher-order, anonymous functions (aka *lambdas*) are now very popular!
  - C++ (C++11), PHP (PHP 5.3.0), C# (C# v2.0), Delphi (since 2009), Objective C, Java 8, Swift, Python, Ruby (Procs), … (and functional languages like OCaml, Haskell, F#, …)

Lambda Calculus Syntax

- A lambda calculus expression is defined as

e ::= x	**variable**
| λx.e	**abstraction** (func def)
| e e	**application** (func call)

  - This grammar describes ASTs; not for parsing
  - Lambda expressions also known as lambda **terms**

- λx.e is like `(fun x -> e)` in OCaml

That's it! Nothing but (higher-order) functions

Lambda Calculus Semantics

- Evaluation: All that's involved are function calls (λx.e1) e2
  - Evaluate e1 with x replaced by e2
- This application is called beta reduction
  - (λx.e1) e2 → e1{e2/x}
    - e1{e2/x} is e1 with occurrences of x replaced by e2
    - This operation is called *substitution*
      - **Replace** formal parameters with actual arguments
- When a term cannot be reduced further it is in beta normal form, e.g., x, λx.e, x x, x (λx.e).

Beta Reduction Example

- (λx.λz.x z) y

  → (λx.(λz.(x z))) y   // since λ extends to right

  → (λx.(λz.(x z))) y   // apply (λx.e1) e2 → e1{e2/x}

  // where e1 = λz.(x z), e2 = y

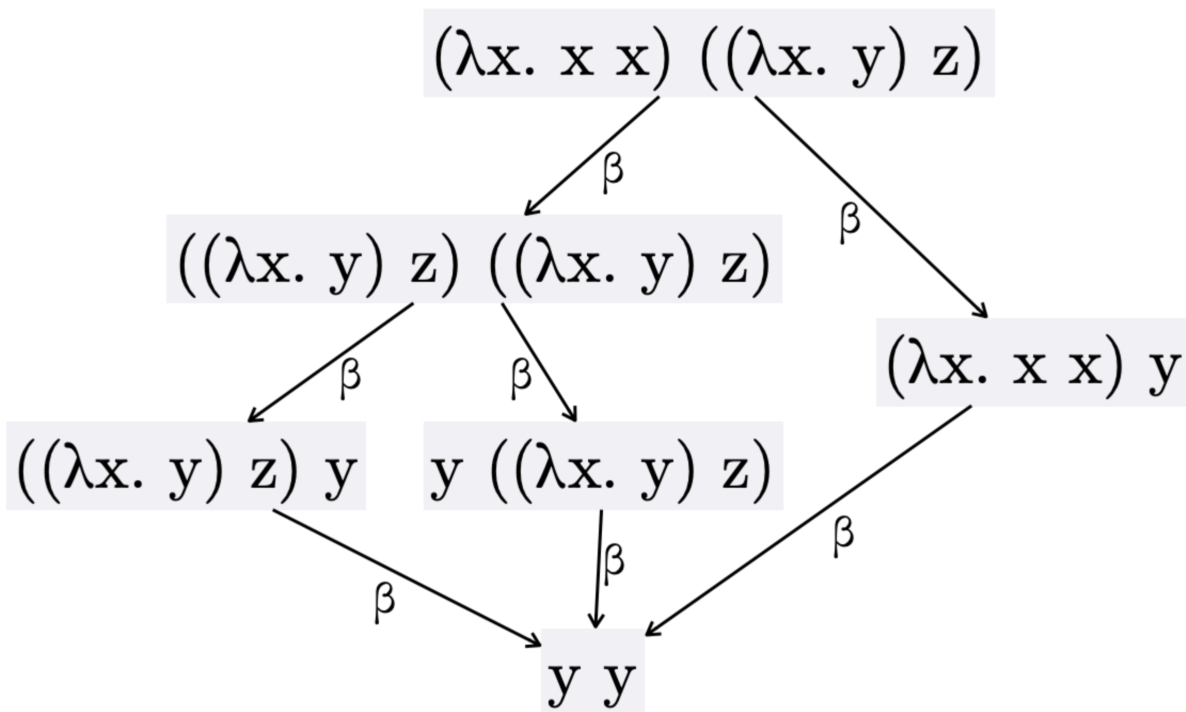
  → λz.(y z)   // final result

Parameters
- Formal
- Actual

- Equivalent OCaml code
  - (fun x -> (fun z -> (x z))) y → fun z -> (y z)

Confluence

- We allow reductions to occur *anywhere* in a term
  - Order reductions are applied does not affect final value!



Termination

- May or may not terminate based on the applications chosen to reduce.

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

$$\rightarrow_\beta \quad y$$

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

$$\rightarrow_\beta \quad (\lambda x.\, y)\ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x))$$

$$\rightarrow_\beta \quad \ldots$$

Call-by-name vs. Call-by-value

- Sometimes we have a choice about where to apply beta reduction. Before call (i.e., argument):
  - $(\lambda z.z)\ ((\lambda y.y)\ x) \rightarrow (\lambda z.z)\ x \rightarrow x$
- Or after the call:
  - $(\lambda z.z)\ ((\lambda y.y)\ x) \rightarrow (\lambda y.y)\ x \rightarrow x$
- The former strategy is called call-by-value (CBV)
  - Evaluate any arguments before calling the function
- The latter is called call-by-name (CBN)
  - Delay evaluating arguments as long as possible

Call-by-name vs. Call-by-value

- Call-by-name

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

$$\rightarrow_\beta \quad y$$

- Call by value

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

$$\rightarrow_\beta \quad (\lambda x. y)\ ((\lambda x. x\ x)\ (\lambda x. x\ x))$$

$$\rightarrow_\beta \quad \ldots$$

Definitional Interpreter as Semantics

```
let rec reduce e =
  match e with
     (λx. e1) e2 -> e1{e2/x}
    | e1 e2 ->
      let e1' = reduce e1 in
      if e1' <> e1 then e1' e2
      else e1 (reduce e2)
    | λx. e -> λx. (reduce e)
    | _ -> e
```

Straight β rule

Reduce lhs of app

Reduce rhs of app

Reduce function body

Already in a normal form nothing to do

Partial Evaluation

- It is also possible to evaluate within a function (without calling it):
  - $(\lambda y.(\lambda z.z)\ y\ x) \rightarrow (\lambda y.y\ x)$
- Called partial evaluation
  - Can combine with CBN or CBV (as in the interpreter)
  - In practical languages, this evaluation strategy is employed in a limited way, as compiler optimization

```
int foo(int x) {
  return 0+x;
}
```

→

```
int foo(int x) {
  return x;
}
```

Summary

- Lambda calculus is a core model of computation
  - We can encode familiar language constructs using only functions
    - E.g., Booleans, control-flows, recursive functions.

- Useful for understanding how languages work
  - Ideas of types, evaluation order, termination, proof systems, etc. can be developed in lambda calculus,
    - then scaled to full languages