Lambda Calculus

Overview

- Motivation
- Syntax
- Semantics
  - Axiomatic
  - Operational
- Reduction
  - Normal
  - Applicative
- Properties
  - Confluence

Motivation

- Lambda calculus
  - Minimalist approach
  - Preserve expressive power
  - Very simple
  - Untyped *or* typed
- What we gain
  - Easier to study
    - Simple semantics
  - Tool to formalize semantics of other languages

Lambda Calculus Syntax

- Very small

```
Expr :== <Expr> <Expr>
     :== λ<Id>.<Expr>
     :== ( <Expr> )
     :== <Id>
```

- Examples

```
x
y
s
(x)
x y
λ x. x
λ x. y
(λ x. y) z
```

Associativity

- Application associates to the left

```
f a b c d
```

```
(((f a) b) c) d
```

Scoping

- Scope of abstractions
  - Extent to the right as far as possible

```
λx.λy.M N

λx.(λy.(M N))
```

Abstraction

- The abstraction
  - Represents a function definition

    ```
    λx.M
    ```

  - **λ** Is the abstraction symbol
  - The formal is the identifier *x*
  - The body is the expression *M*

Application

- The application
  - The expression

```
M N
```

  - Is a function *call*
- *Intuitive* semantics
  - M is an abstraction

```
λx.P
```

  - N is the argument
  - The application consist of
    - Replace the formal *x* by the argument N in the body P

Free & Bound Variables

- Consider an example

$$\boxed{\lambda x.\,(\underline{x}\;\;\underline{y})}$$

Bound = {x}　　　　Free = {y}

- Can we define it formally ?

Examples

- Compute the free set for:

```
λx.x y          =
λx.λy.(x y z)   =
λx.λy.(xzλz.z)  =
```

Substitution

- We write

```
[N/x]M
```

- To denote the expression resulting from
  - Replacing all free occurrences of *x in M by N*
- Examples

```
[z/x](x y)   =

[λx.x/y](x y z)   =

[λx.xy/z](λy.(zy))  =
```

Substitution

- A Catch...
  - We must make sure that
    - Free variables in N do not become bound.
- How to do it ?

Axiomatic Semantics

- Axiomatic semantics
  - Three *axioms*
    - α-equivalence
    - β-equivalence
    - η-equivalence
- Role of semantics
  - Establish logical consequences of equations

α-Equivalence

- Intuitively
  - Two expressions are equivalent up to a renaming
- Formally

```
λx.M = λy.[y/x]M   y not free in M
```

- Examples

```
λx.x    =
λx.y    =
λx.x  y =
```

β-Equivalence

- Intuitively
  - The application of a function to an argument is equivalent to the result of the application obtained from substituting the argument for the formal in the body.
- Formally

```
(λx.M) N = [N/x]M
```

- Examples

```
(λx.x) y       =
(λx.x z) y     =
(λx.λy.x) a b  =
```

η-Equivalence

- Intuitively
  - Proxy functions can be added or removed without affecting the semantics.
- Formally

```
λx.(f x)   = f
```

if x is not free in *f* then *f* is a function

- Example

```
λx.(λy.y x)   =
```

Operational Semantics

- What is it ?
  - A description of the mechanics of an “interpreter”
- Relationship ?
  - Operational semantics give
    - Reduction rules
    - Reduction rules are related to equivalence axioms
      - α-reduction
      - β-reduction
      - η-reduction ?

β-Reduction

- The most useful rule!
  - It defines the result of function application
  - Related to β-Equivalence
- Formally

$$\boxed{(\lambda x.M)\ N \rightarrow_{\beta} [N/x]M}$$

- Subtlety
  - Reduction is defined up to α-Equivalence.

α-Reduction

- Purpose
  - Prevent incorrect binding in β-Reduction.
- Formally

$$(\lambda x.M) \rightarrow_{\alpha} (\lambda y.[y/x]M) \quad \textbf{s.t. Y not in FV(M)}$$

η-Reduction

- Purpose
  - Eliminate or introduce proxies
- Do we need that in an interpreter ?

Computation With Reduction

- A Computation is
  - A sequence of reductions
  - Starting from an initial lambda term
  - Ending...
    - In a *normal* form
    - Or not at all
- What is a normal form ?
  - A lambda term with no applications left.
  - β-Reduction can no longer be applied.
- Examples

```
x
x y
(λx.x x)
z(λx.x)y
```

Applicative Reduction

- Objective
  - Reduce a lambda term through application
  - Rule of thumb
    - Evaluate the argument first
    - Pass the normal form of the argument to the function.
- Examples

```
(λx.xx)((λy.y)z)
```

```
(λx.y)((λx.xx)(λx.xx))
```

Applicative Reduction

- Applicative reduction
  - May not terminate
  - Corresponds to eager evaluation
  - Corresponds to call by value

Normal Order Reduction

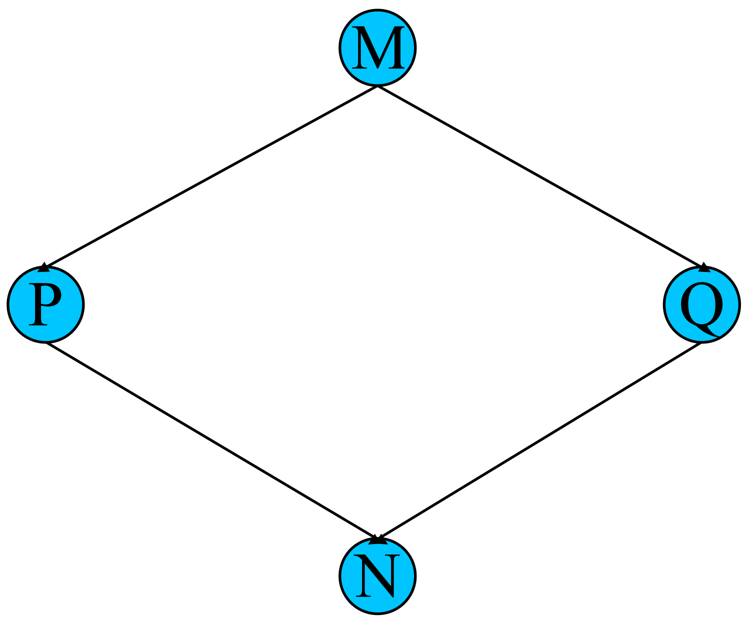
- Objective
  - Reduce a lambda term
    - Leftmost, outermost abstraction first.
- Examples

```
(λx.xx)((λy.y)z)
```

```
(λx.y)((λx.xx)(λx.xx))
```

Soundness

- So....
  - Depending on terms
    - Applicative order may terminate or not
    - Normal order may produce a different sequence of terms
- Question
  - Is the following diagram true ?



Church-Rosser Theorem

- Some good news....

$$\forall M, P, Q \in \Lambda : M \Rightarrow_* P, M \Rightarrow_* Q$$
$$\longrightarrow \exists N \in \Lambda : P \Rightarrow_* N \wedge Q \Rightarrow_* N$$

- The result of a computation does not depend on the order in which reductions are applied.
- The end result is a normal form (if one exists).
- For all terminating computation there is a unique normal form.

λ-calculus

- As a programming language!
  - We can now write all our favorite programs!
  - The λ-calculus is Turing complete
- A few question though....
  - How to
    - Write constants like
      - True, false, 0, 1, 2, ...
    - Write branching instructions like
      - If .... then ... else ....
    - Write Iterative statements ?
    - Write recursive statements ?