# CS618: Program Analysis
## 2016-17 $I^{st}$ Semester

# The Untyped Lambda Calculus

Amey Karkare
karkare@cse.iitk.ac.in
karkare@cse.iitb.ac.in

Department of CSE, IIT Kanpur/Bombay



## Reference Book

Types and Programming Languages by Benjamin C. Pierce



## The Abstract Syntax

$$\begin{array}{lll} \mathrm{t} & := & x & \text{– Variable} \\ & & \mid \lambda x.\mathrm{t} & \text{– Abstraction} \\ & & \mid \mathrm{t}\ \mathrm{t} & \text{– Application} \end{array}$$

Parenthesis, $(\ldots)$, can be used for grouping and scoping.



## Conventions

- $\lambda x.\mathrm{t}_1\mathrm{t}_2\mathrm{t}_3$ is an abbreviation for $\lambda x.(\mathrm{t}_1\mathrm{t}_2\mathrm{t}_3)$, i.e., the scope of $x$ is as far to the right as possible until it is
  - terminated by a **)** whose matching **(** occurs to the left pf $\lambda$, OR
  - terminated by the end of the term.
- Applications associate to the left: $\mathrm{t}_1\mathrm{t}_2\mathrm{t}_3$ to be read as $(\mathrm{t}_1\mathrm{t}_2)\mathrm{t}_3$ and not as $\mathrm{t}_1(\mathrm{t}_2\mathrm{t}_3)$
- $\lambda xyz.\mathrm{t}$ is an abbreviation for $\lambda x\lambda y\lambda z.\mathrm{t}$ which in turn is abbreviation for $\lambda x.(\lambda y.(\lambda z.\mathrm{t}))$.

## $\alpha$-renaming

- The name of a bound variable has no meaning except for its use to identify the bounding $\lambda$.
- Renaming a $\lambda$ variable, including all its bound occurrences, does not change the meaning of an expression. For example, $\lambda x.x\ x\ y$ is equivalent to $\lambda u.u\ u\ y$
  - But it is not same as $\lambda x.x\ x\ w$
  - Can not change free variables!



## $\beta$-reduction (Execution Semantics)

- if an abstraction $\lambda x.t_1$ is applied to a term $t_2$ then the result of the application is
  - the body of the abstraction $t_1$ with all free occurrences of the formal parameter $x$ replaced with $t_2$.
- For example,

$$(\lambda f \lambda x.f\ (f\ x))\ g \xrightarrow{\beta} \lambda x.g\ (g\ x)$$



## Caution

- During $\beta$-reduction, make sure a free variable is not captured inadvertently.
- The following reduction is **WRONG**

$$(\lambda x \lambda y.x)(\lambda x.y) \xrightarrow{\beta} \lambda y.\lambda x.y$$

- Use $\alpha$-renaming to avoid variable capture

$$(\lambda x \lambda y.x)(\lambda x.y) \xrightarrow{\alpha} (\lambda u \lambda v.u)(\lambda x.y) \xrightarrow{\beta} \lambda v.\lambda x.y$$



## Exercise

- Apply $\beta$-reduction as far as possible

1. $(\lambda x\ y\ z.\ x\ z\ (y\ z))\ (\lambda x\ y.\ x)\ (\lambda y.y)$
2. $(\lambda x.\ x\ x)(\lambda x.\ x\ x)$
3. $(\lambda x\ y\ z.\ x\ z\ (y\ z))\ (\lambda x\ y.\ x)\ ((\lambda x.\ x\ x)(\lambda x.\ x\ x))$

## Church-Rosser Theorem

- Multiple ways to apply $\beta$-reduction
- Some may not terminate
- However, if two different reduction sequences terminate then they always terminate in the same term
  - Also called the *Diamond Property*
- Leftmost, outermost reduction will find the normal form if it exists



## Programming in $\lambda$ Calculus

- Where is the other stuff?
- Constants?
  - Numbers
  - Booleans
- Complex Types?
  - Lists
  - Arrays
- Don't we need data?

Abstractions act as functions as well as data!



## Numbers: Church Numerals

- We need a "Zero"
  - "Absence of item"
- And something to count
  - "Presence of item"
- Intuition: Whiteboard and Marker
  - Blank board represents Zero
  - Each mark by marker represents a count.
  - However, other pairs of objects will work as well
- Lets translate this intuition into $\lambda$-expressions



## Numbers

- Zero = $\lambda m\ w.\ w$
  - No mark on the whiteboard
- One = $\lambda m\ w.\ m\ w$
  - One mark on the whiteboard
- Two = $\lambda m\ w.\ m\ (m\ w)$
- ...
- What about operations?
  - add, multiply, subtract, divide, ...?

## Operations on Numbers

- succ = $\lambda x\ m\ w.\ m\ (x\ m\ w)$
  - Verify: succ N = N + 1
- add = $\lambda x\ y\ m\ w.\ x\ m\ (y\ m\ w)$
  - Verify: add M N = M + N
- mult = $\lambda x\ y\ m\ w.\ x\ (y\ m)\ w$
  - Verify: mult M N = M * N



## More Operations

- pred = $\lambda x\ m\ w.\ x\ (\lambda g\ h.\ h\ (g\ m))(\lambda u.\ w)(\lambda u.\ u)$
  - Verify: pred N = N - 1
- nminus = $\lambda x\ y.\ y$ pred $x$
  - Verify: nminus M N = max(0, M - N) – natural subtraction



## Church Booleans

- True and False
- Intuition: Selection of one out of two (complementary) choices
- True = $\lambda x\ y.\ x$
- False = $\lambda x\ y.\ y$
- Predicate:
  - isZero = $\lambda x.\ x\ (\lambda u.\text{False})$ True



## Operations on Booleans

- Logical operations

$$\begin{aligned} and &= \lambda p\ q.\ p\ q\ p \\ or &= \lambda p\ q.\ p\ p\ q \\ not &= \lambda p\ t\ f.p\ f\ t \end{aligned}$$

- The conditional operator *if*
  - *if* $c\ e_t\ e_f$ reduces to $e_t$ if $c$ is True, and to $e_f$ if $c$ is False

$$if = \lambda c\ e_t\ e_f.\ (c\ e_t\ e_f)$$

## More...

- More such types can be found at `https://en.wikipedia.org/wiki/Church_encoding`
- It is fun to come up with your own definitions for constants and operations over different types
- or to develop understanding for existing definitions.



## We are missing something!!

- The machinery described so far does not allow us to define Recursive functions
  - Factorial, Fibonacci, ...
- There is no concept of “named” functions
  - So no way to refer to a function “recursively”!
- Fix-point computation comes to rescue



## Fix-point and $Y$-combinator

- A fix-point of a function $f$ is a value $p$ such that $f\ p = p$
- Assume existence of a magic expression, called $Y$-combinator, that when applied to a $\lambda$-expression, gives its fixed point
$$Y\ f = f\ (Y\ f)$$
- $Y$-combinator gives us a way to apply a function recursively



## Recursion Example: Factorial

$$\begin{aligned} \text{fact} &= \lambda n.\ \textit{if}\ (\text{isZero}\ n)\ \text{One}\ (\text{mult}\ n\ (\text{fact}\ (\text{pred}\ n))) \\ &= (\lambda f\ n.\ \textit{if}\ (\text{isZero}\ n)\text{One}\ (\text{mult}\ n\ (f\ (\text{pred}\ n))))\ \text{fact} \end{aligned}$$

$$\textit{fact} = \textit{gfact}$$

- fact is a fixed point of the function
$$g = (\lambda f\ n.\ \textit{if}\ (\text{isZero}\ n)\text{One}\ (\text{mult}\ n\ (f\ (\text{pred}\ n))))$$
- Using Y-combinator,
$$\text{fact} = Y\ g$$

## Factorial: Verify

$$
\begin{aligned}
\text{fact } 2 &= (Y\ g)\ 2 \\
&= g\ (Y\ g)\ 2 \quad \text{– by definition of Y-combinator} \\
&= (\lambda fn.\ \mathit{if}\ (\text{isZero } n)\ 1\ (\text{mult } n\ (f\ (\text{pred } n))))\ (Y\ g)\ 2 \\
&= (\lambda n.\ \mathit{if}\ (\text{isZero } n)\ 1\ (\text{mult } n\ ((Y\ g)\ (\text{pred } n))))\ 2 \\
&= \mathit{if}\ (\text{isZero } 2)\ 1\ (\text{mult } 2\ ((Y\ g)(\text{pred2}))) \\
&= (\text{mult } 2\ ((Y\ g)\ 1)) \\
&\quad \ldots \\
&= (\text{mult } 2\ (\text{mult } 1\ (\mathit{if}\ (\text{isZero } 0)\ 1\ (\ldots)))) \\
&= (\text{mult } 2\ (\text{mult } 1\ 1)) \\
&= 2
\end{aligned}
$$



## Recursion and $Y$-combinator

- Y-combinator allows to unroll the body of loop once—similar to one unfolding of recursive call
- Sequence of $Y$-combinator applications allow complete unfolding of recursive calls

BUT, what about the existence of $Y$-combinator?



## $Y$-combinators

- Many candidates exist

$$Y_1 = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

$$Y = \lambda abcdefghijklmnopqstuvwxwzr.r(thisisafixedpointcombinator)$$

$$Y_{\text{funny}} = TTTTT\ TTTTT\ TTTTT\ TTTTT\ TTTTT\ T$$

- Verify that $(Y\ f) = f\ (Y\ f)$ for each



## Summary

- A cursory look at $\lambda$-calculus
- Functions are data, and Data are functions!
- Not covered but important to know: The power of $\lambda$ calculus is equivalent to that of Turing Machine ("Church Turing Thesis")