IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

The λ-calculus

Mathematical modeling of functions

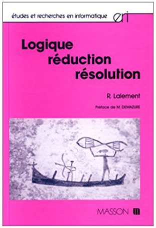
Fabien Dagnat

ELU 610 – C5

1st semester 2019

The λ-calculus [Church]

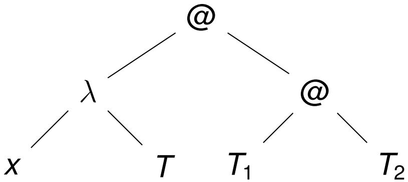
- A formal language proposed by Alonzo Church in the 1930s to model the notion of function
- http://en.wikipedia.org/wiki/Lambda_calculus
- We will use it to
  - illustrate the notion of formal language
  - understand fundamentals of formal reasoning
  - introduce the functional paradigm

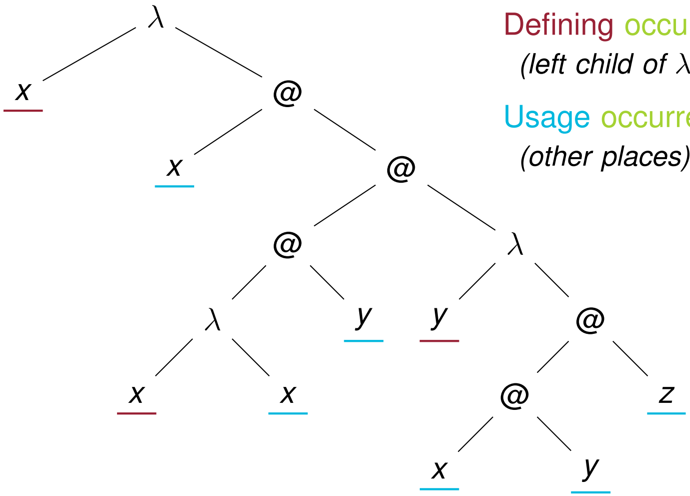


René Lalement
Logique Réduction Résolution
ERI Masson, 1990
Book translated in english *Computation as logic*, Prentice-Hall in 1993, ISBN 9780137700097

- The set $\Lambda_{\mathcal{X}}$ of the terms defined by
  - variables x, y, ... from a denumerable set $\mathcal{X}$
  - applications $(T_1 T_2)$ of a term T_1 (the function) to a term T_2 (the argument)
  - functions $(\lambda x.T)$ of a variable x (the parameter) and a term T (the body)
- BNF: $T ::= x \mid (TT) \mid (\lambda x.T)$
- Parenthesis may be omitted
  - outer: $(T_1 T_2) = T_1 T_2$ and $(\lambda x.T) = \lambda x.T$
  - application is left associative: $T_1 T_2 T_3 = (T_1 T_2) T_3$
  - λ is right associative: $\lambda x.\lambda y.T = \lambda x.(\lambda y.T)$ and $\lambda x.T_1 T_2 = \lambda x.(T_1 T_2)$
- Some well-known λ-terms
  - $\lambda x.x = \mathbf{I}$ $\lambda x.\lambda y.x = \mathbf{K}$ $\lambda x.\lambda y.\lambda z.((xz)(yz)) = \mathbf{S}$

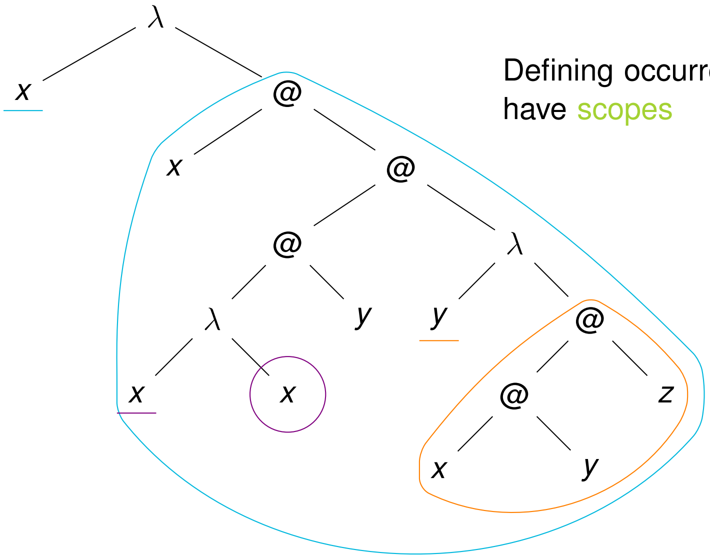
- $\Lambda_{\mathcal{X}} = T_{\{@,\lambda\}}[\mathcal{X}]$ with
  - @ is the only constructor and $ar(@) = 2$
  - λ is the only binder and $ar(\lambda) = 1$
- Terms are trees
  - variables are leaves
  - constructors and binders are nodes
- ex: $(\lambda x.T)(T_1 T_2)$



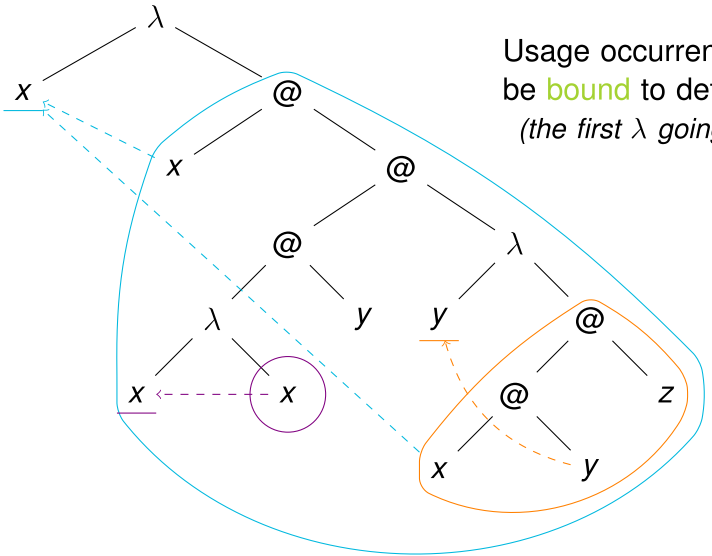


Defining occurrences
(left child of λ)

Usage occurrences
(other places)

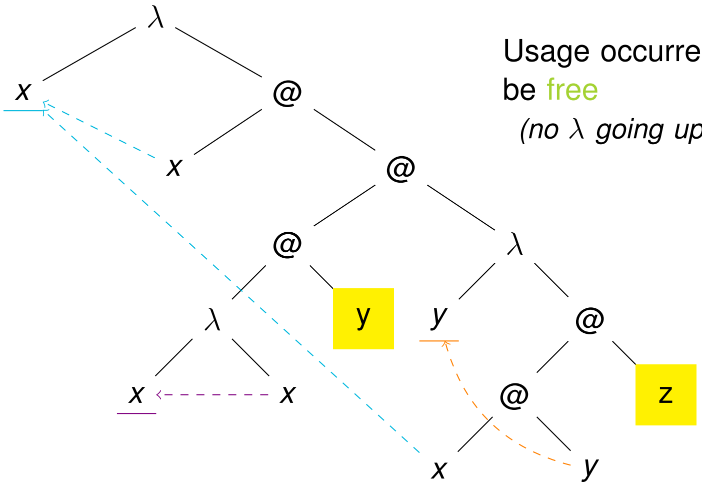


Defining occurrences have scopes



Usage occurrences may be bound to defining ones

(the first λ going up)



Usage occurrences may be free

(no λ going up)

- A free variable is defined outside the term
  - a kind of *global variable* (for the term)
  - its name is essential and cannot be modified
  - $\lambda x.y$ is different from $\lambda x.z$
- A bound variable is intern to the term
  - a kind of *local variable* (for the term)
  - its name can be modified (the defining occurrence and all its depending bound occurrences)
  - $\lambda x.x$ is identical to $\lambda y.y$
  - known as α-conversion (see later for the mathematical definition)
  - the name of a bound variable has no importance, only the link to its binder[1]
- A term with free variables is open
- A term with no free variables is closed (*a.k.a.* combinators)

[1] there exists notations without names, see for example [Bou08]

- One can define a function f on $\mathbb{N}$ recursively by
  1. defining $f(0)$
  2. defining $f(n+1)$ in terms of $f(n)$

  for example, factorial
  1. $0! = 1$
  2. $(n+1)! = (n+1)n!$
- One can prove a property P on $\mathbb{N}$ by
  1. proving $P(0)$
  2. proving that if $P(n)$ holds, $P(n+1)$ is true

  for example, if $P(n)$ is $0 + 1 + \cdots + n = \frac{n(n+1)}{2}$
  1. $0 = 0$
  2. $0 + 1 + \cdots + n + (n+1) = \frac{n(n+1)}{2} + (n+1) = (n+1)(\frac{n}{2} + 1)$
     $= \frac{(n+1)(n+2)}{2}$
- Variants: starting at k or $P(0), \ldots, P(n) \Rightarrow P(n+1)$

- (Structural) induction is a method of definition or proof on the set of terms $T_\Sigma[\mathcal{X}]$
- One can define a function f on $\Lambda_\mathcal{X}$ inductively by
  1. defining f on $\mathcal{X}$ (leaves)
  2. defining $f(T_1 T_2)$ in terms of $f(T_1)$ and $f(T_2)$
  3. defining $f(\lambda x.T)$ in terms of $f(T)$
- For example, the set of free variables FV is defined by
  1. $FV(x) = \{x\}$
  2. $FV(T_1 T_2) = FV(T_1) \cup FV(T_2)$
  3. $FV(\lambda x.T) = FV(T) \setminus \{x\}$
- For example, the size of a λ-term is defined by
  1. $\mathtt{size}(x) = 1$
  2. $\mathtt{size}(T_1 T_2) = \mathtt{size}(T_1) + \mathtt{size}(T_2) + 1$
  3. $\mathtt{size}(\lambda x.T) = \mathtt{size}(T) + 1$

- One can prove a property P on $\Lambda_{\mathcal{X}}$ inductively by
  1. proving P on $\mathcal{X}$
  2. proving $P(T_1 T_2)$ supposing $P(T_1)$ and $P(T_2)$ are true
  3. proving $P(\lambda x.T)$ supposing $P(T)$ is true
- Prove $\forall T \in \Lambda_{\mathcal{X}}, card(FV(T)) \leq \texttt{size}(T)$

- One can prove a property P on $\Lambda_{\mathcal{X}}$ inductively by
  1. proving P on $\mathcal{X}$
  2. proving $P(T_1 T_2)$ supposing $P(T_1)$ and $P(T_2)$ are true
  3. proving $P(\lambda x.T)$ supposing $P(T)$ is true
- Prove $\forall T \in \Lambda_{\mathcal{X}}, card(FV(T)) \leq \texttt{size}(T)$
  1. $card(FV(x)) = card(\{x\}) = 1 = \texttt{size}(x)$

- One can prove a property P on $\Lambda_{\mathcal{X}}$ inductively by
  1. proving P on $\mathcal{X}$
  2. proving $P(T_1 T_2)$ supposing $P(T_1)$ and $P(T_2)$ are true
  3. proving $P(\lambda x.T)$ supposing $P(T)$ is true
- Prove $\forall T \in \Lambda_{\mathcal{X}}, card(FV(T)) \leq \texttt{size}(T)$
  1. $card(FV(x)) = card(\{x\}) = 1 = \texttt{size}(x)$
  2. let's suppose $card(FV(T_i)) \leq \texttt{size}(T_i)$ for i in $\{1, 2\}$ (IH)

- One can prove a property P on $\Lambda_{\mathcal{X}}$ inductively by
  1. proving P on $\mathcal{X}$
  2. proving $P(T_1\,T_2)$ supposing $P(T_1)$ and $P(T_2)$ are true
  3. proving $P(\lambda x.T)$ supposing $P(T)$ is true
- Prove $\forall T \in \Lambda_{\mathcal{X}}, card(FV(T)) \leq \texttt{size}(T)$
  1. $card(FV(x)) = card(\{x\}) = 1 = \texttt{size}(x)$
  2. let's suppose $card(FV(T_i)) \leq \texttt{size}(T_i)$ for i in $\{1, 2\}$ (IH)

$$
\begin{aligned}
card(FV(T_1\,T_2)) &= card(FV(T_1) \cup FV(T_2)) && \text{def of } FV \\
&\leq card(FV(T_1)) + card(FV(T_2)) && \text{prop of } card \\
&\leq \texttt{size}(T_1) + \texttt{size}(T_2) && \text{IH} \\
&\leq \texttt{size}(T_1\,T_2) && \text{def of } \texttt{size}
\end{aligned}
$$

- One can prove a property P on $\Lambda_{\mathcal{X}}$ inductively by
  1. proving P on $\mathcal{X}$
  2. proving $P(T_1\,T_2)$ supposing $P(T_1)$ and $P(T_2)$ are true
  3. proving $P(\lambda x.T)$ supposing $P(T)$ is true
- Prove $\forall T \in \Lambda_{\mathcal{X}}, \mathit{card}(FV(T)) \leq \mathtt{size}(T)$
  1. $\mathit{card}(FV(x)) = \mathit{card}(\{x\}) = 1 = \mathtt{size}(x)$
  2. let's suppose $\mathit{card}(FV(T_i)) \leq \mathtt{size}(T_i)$ for i in $\{1, 2\}$ (IH)

$$
\begin{aligned}
\mathit{card}(FV(T_1\,T_2)) &= \mathit{card}(FV(T_1) \cup FV(T_2)) && \text{def of } FV \\
&\leq \mathit{card}(FV(T_1)) + \mathit{card}(FV(T_2)) && \text{prop of } \mathit{card} \\
&\leq \mathtt{size}(T_1) + \mathtt{size}(T_2) && \text{IH} \\
&\leq \mathtt{size}(T_1\,T_2) && \text{def of } \mathtt{size}
\end{aligned}
$$

  3. let's suppose $\mathit{card}(FV(T)) \leq \mathtt{size}(T)$ (IH)

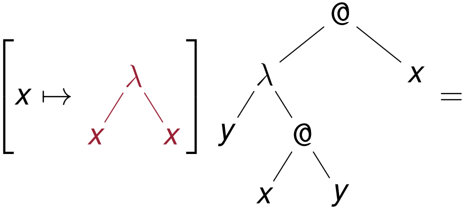
- One can prove a property P on $\Lambda_{\mathcal{X}}$ inductively by
  1. proving P on $\mathcal{X}$
  2. proving $P(T_1\,T_2)$ supposing $P(T_1)$ and $P(T_2)$ are true
  3. proving $P(\lambda x.T)$ supposing $P(T)$ is true
- Prove $\forall T \in \Lambda_{\mathcal{X}}, \mathit{card}(FV(T)) \leq \mathtt{size}(T)$
  1. $\mathit{card}(FV(x)) = \mathit{card}(\{x\}) = 1 = \mathtt{size}(x)$
  2. let's suppose $\mathit{card}(FV(T_i)) \leq \mathtt{size}(T_i)$ for i in $\{1, 2\}$ (IH)

$$\begin{aligned} \mathit{card}(FV(T_1\,T_2)) &= \mathit{card}(FV(T_1) \cup FV(T_2)) && \text{def of } FV \\ &\leq \mathit{card}(FV(T_1)) + \mathit{card}(FV(T_2)) && \text{prop of } \mathit{card} \\ &\leq \mathtt{size}(T_1) + \mathtt{size}(T_2) && \text{IH} \\ &\leq \mathtt{size}(T_1\,T_2) && \text{def of } \mathtt{size} \end{aligned}$$

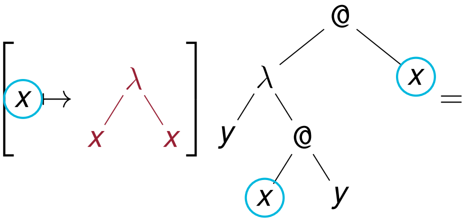
  3. let's suppose $\mathit{card}(FV(T)) \leq \mathtt{size}(T)$ (IH)

$$\begin{aligned} \mathit{card}(FV(\lambda x.T)) &= \mathit{card}(FV(T) \setminus \{x\}) && \text{def of } FV \\ &\leq \mathit{card}(FV(T)) && \text{prop of } \mathit{card} \\ &\leq \mathtt{size}(T) && \text{IH} \\ &\leq \mathtt{size}(\lambda x.T) && \text{def of } \mathtt{size} \end{aligned}$$

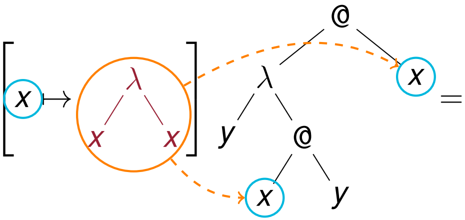
- Giving a meaning to a free variable is done by substitution
- Substitution is a function associating a term to
  - a variable (the substituted variable) and
  - two terms (the replacement term and the term on which substitution operates)
- $[x \mapsto T_1]\,T_2$ is the term defined by replacing **all** free occurrences of x within T_2 by T_1



- Giving a meaning to a free variable is done by substitution
- Substitution is a function associating a term to
  - a variable (the substituted variable) and
  - two terms (the replacement term and the term on which substitution operates)
- $[x \mapsto T_1]T_2$ is the term defined by replacing **all** free occurrences of x within T_2 by T_1

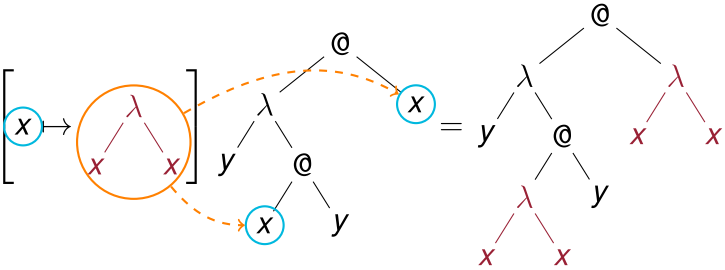


- Giving a meaning to a free variable is done by substitution
- Substitution is a function associating a term to
  - a variable (the substituted variable) and
  - two terms (the replacement term and the term on which substitution operates)
- $[x \mapsto T_1]\,T_2$ is the term defined by replacing **all** free occurrences of x within T_2 by T_1



Substitutions

- Giving a meaning to a free variable is done by substitution
- Substitution is a function associating a term to
  - a variable (the substituted variable) and
  - two terms (the replacement term and the term on which substitution operates)
- $[x \mapsto T_1]\,T_2$ is the term defined by replacing **all** free occurrences of x within T_2 by T_1



- Defined inductively

$$\begin{cases} [x \mapsto T]x &= T & \\ [x \mapsto T]y &= y & \text{if } x \neq y \\ [x \mapsto T]T_1 T_2 &= [x \mapsto T]T_1 [x \mapsto T]T_2 & \\ [x \mapsto T]\lambda y.T' &= \lambda y.[x \mapsto T]T' & \text{if } x \neq y, y \notin FV(T) \end{cases}$$

  the last condition prevent captures of a free y in T
- The definition is incomplete *e.g.* $[x \mapsto T]\lambda x.T'$, $[x \mapsto y]\lambda y.T$
- α-conversion (a.k.a. α-equivalence) is defined by
  $\lambda x.T =_\alpha \lambda y.[x \mapsto y]T \quad$ if $y \notin FV(T)$ (*freshness condition*)
- The definition of substitution is complete modulo renaming
  - if $x = y$ or $y \in FV(M)$, we rename the bound y
- We always work on $\Lambda_{\mathcal{X}}/=_\alpha$ (modulo renaming)

$$\begin{cases} (1)\ [x \mapsto T]x = T & \\ (2)\ [x \mapsto T]y = y & \text{if } x \neq y \\ (3)\ [x \mapsto T]T_1 T_2 = [x \mapsto T]T_1 [x \mapsto T]T_2 & \\ (4)\ [x \mapsto T]\lambda y.T' = \lambda y.[x \mapsto T]T' & \text{if } x \neq y \text{ and } y \notin FV(T) \\ & \\ (\alpha)\ \lambda x.T = \lambda y.[x \mapsto y]T & \text{if } y \notin FV(T) \end{cases}$$

- $[z \mapsto \lambda x.xy]\lambda z.x\ (\lambda y.zy) =$

$$\begin{cases} (1)\ [x \mapsto T]x = T & \\ (2)\ [x \mapsto T]y = y & \text{if } x \neq y \\ (3)\ [x \mapsto T]T_1 T_2 = [x \mapsto T]T_1 [x \mapsto T]T_2 & \\ (4)\ [x \mapsto T]\lambda y.T' = \lambda y.[x \mapsto T]T' & \text{if } x \neq y \text{ and } y \notin FV(T) \\ & \\ (\alpha)\ \lambda x.T = \lambda y.[x \mapsto y]T & \text{if } y \notin FV(T) \end{cases}$$

- $[z \mapsto \lambda x.xy]\lambda z.x\ (\lambda y.zy) =$

(3) $= [z \mapsto \lambda x.xy]\lambda z.x\ ([z \mapsto \lambda x.xy]\lambda y.zy)$

$$\begin{cases} (1)\ [x \mapsto T]x = T & \\ (2)\ [x \mapsto T]y = y & \text{if } x \neq y \\ (3)\ [x \mapsto T]T_1 T_2 = [x \mapsto T]T_1 [x \mapsto T]T_2 & \\ (4)\ [x \mapsto T]\lambda y.T' = \lambda y.[x \mapsto T]T' & \text{if } x \neq y \text{ and } y \notin FV(T) \\ & \\ (\alpha)\ \lambda x.T = \lambda y.[x \mapsto y]T & \text{if } y \notin FV(T) \end{cases}$$

- $[z \mapsto \lambda x.xy]\lambda z.x\ (\lambda y.zy) =$

$$\begin{aligned} &(3) && = [z \mapsto \lambda x.xy]\lambda z.x\ ([z \mapsto \lambda x.xy]\lambda y.zy) \\ &(\alpha)(\alpha) && = [z \mapsto \lambda x.xy](\lambda t.[z \mapsto t]x)\ ([z \mapsto \lambda x.xy](\lambda u.[y \mapsto u]zy)) \end{aligned}$$

$$\begin{cases} (1)\ [x \mapsto T]x = T & \\ (2)\ [x \mapsto T]y = y & \text{if } x \neq y \\ (3)\ [x \mapsto T]T_1 T_2 = [x \mapsto T]T_1 [x \mapsto T]T_2 & \\ (4)\ [x \mapsto T]\lambda y.T' = \lambda y.[x \mapsto T]T' & \text{if } x \neq y \text{ and } y \notin FV(T) \\ & \\ (\alpha)\ \lambda x.T = \lambda y.[x \mapsto y]T & \text{if } y \notin FV(T) \end{cases}$$

- $[z \mapsto \lambda x.xy]\lambda z.x\ (\lambda y.zy) =$

$$\begin{aligned} &\text{(3)} && = [z \mapsto \lambda x.xy]\lambda z.x\ ([z \mapsto \lambda x.xy]\lambda y.zy) \\ &(\alpha)(\alpha) && = [z \mapsto \lambda x.xy](\lambda t.[z \mapsto t]x)\ ([z \mapsto \lambda x.xy](\lambda u.[y \mapsto u]zy)) \\ &\text{(2)(3,2+1)} && = [z \mapsto \lambda x.xy]\lambda t.x\ ([z \mapsto \lambda x.xy]\lambda u.zu) \end{aligned}$$

$$\begin{cases} (1)\ [x \mapsto T]x = T & \\ (2)\ [x \mapsto T]y = y & \text{if } x \neq y \\ (3)\ [x \mapsto T]T_1 T_2 = [x \mapsto T]T_1 [x \mapsto T]T_2 & \\ (4)\ [x \mapsto T]\lambda y.T' = \lambda y.[x \mapsto T]T' & \text{if } x \neq y \text{ and } y \notin FV(T) \\ & \\ (\alpha)\ \lambda x.T = \lambda y.[x \mapsto y]T & \text{if } y \notin FV(T) \end{cases}$$

- $[z \mapsto \lambda x.xy]\lambda z.x\ (\lambda y.zy) =$

$$\begin{array}{ll} \text{(3)} & = [z \mapsto \lambda x.xy]\lambda z.x\ ([z \mapsto \lambda x.xy]\lambda y.zy) \\ (\alpha)(\alpha) & = [z \mapsto \lambda x.xy](\lambda t.[z \mapsto t]x)\ ([z \mapsto \lambda x.xy](\lambda u.[y \mapsto u]zy)) \\ \text{(2)(3,2+1)} & = [z \mapsto \lambda x.xy]\lambda t.x\ ([z \mapsto \lambda x.xy]\lambda u.zu) \\ \text{(4)(4)} & = \lambda t.[z \mapsto \lambda x.xy]x\ (\lambda u.[z \mapsto \lambda x.xy]zu) \end{array}$$

$$\begin{cases} (1)\ [x \mapsto T]x = T & \\ (2)\ [x \mapsto T]y = y & \text{if } x \neq y \\ (3)\ [x \mapsto T]T_1 T_2 = [x \mapsto T]T_1 [x \mapsto T]T_2 & \\ (4)\ [x \mapsto T]\lambda y.T' = \lambda y.[x \mapsto T]T' & \text{if } x \neq y \text{ and } y \notin FV(T) \\ & \\ (\alpha)\ \lambda x.T = \lambda y.[x \mapsto y]T & \text{if } y \notin FV(T) \end{cases}$$

- $[z \mapsto \lambda x.xy]\lambda z.x\ (\lambda y.zy) =$

$$\begin{array}{ll} (3) & = [z \mapsto \lambda x.xy]\lambda z.x\ ([z \mapsto \lambda x.xy]\lambda y.zy) \\ (\alpha)(\alpha) & = [z \mapsto \lambda x.xy](\lambda t.[z \mapsto t]x)\ ([z \mapsto \lambda x.xy](\lambda u.[y \mapsto u]zy)) \\ (2)(3,2{+}1) & = [z \mapsto \lambda x.xy]\lambda t.x\ ([z \mapsto \lambda x.xy]\lambda u.zu) \\ (4)(4) & = \lambda t.[z \mapsto \lambda x.xy]x\ (\lambda u.[z \mapsto \lambda x.xy]zu) \\ (2)(3,1{+}2) & = \lambda t.x\ (\lambda u.(\lambda x.xy)u) \end{array}$$

$$\begin{cases} (1)\ [x \mapsto T]x = T & \\ (2)\ [x \mapsto T]y = y & \text{if } x \neq y \\ (3)\ [x \mapsto T]T_1 T_2 = [x \mapsto T]T_1 [x \mapsto T]T_2 & \\ (4)\ [x \mapsto T]\lambda y.T' = \lambda y.[x \mapsto T]T' & \text{if } x \neq y \text{ and } y \notin FV(T) \\ & \\ (\alpha)\ \lambda x.T = \lambda y.[x \mapsto y]T & \text{if } y \notin FV(T) \end{cases}$$

- $[z \mapsto \lambda x.xy]\lambda z.x\ (\lambda y.zy) =$

$$\begin{array}{ll} (3) & = [z \mapsto \lambda x.xy]\lambda z.x\ ([z \mapsto \lambda x.xy]\lambda y.zy) \\ (\alpha)(\alpha) & = [z \mapsto \lambda x.xy](\lambda t.[z \mapsto t]x)\ ([z \mapsto \lambda x.xy](\lambda u.[y \mapsto u]zy)) \\ (2)(3{,}2{+}1) & = [z \mapsto \lambda x.xy]\lambda t.x\ ([z \mapsto \lambda x.xy]\lambda u.zu) \\ (4)(4) & = \lambda t.[z \mapsto \lambda x.xy]x\ (\lambda u.[z \mapsto \lambda x.xy]zu) \\ (2)(3{,}1{+}2) & = \lambda t.x\ (\lambda u.(\lambda x.xy)u) \end{array}$$

- Everyone should be comfortable with such rewritings…

- The usual function call can be modeled by

$$\underbrace{(\lambda x.T_1)}_{(1)}\underbrace{T_2}_{(2)} \to \underbrace{[x \mapsto T_2]T_1}_{(3)}$$

  where (1) is the function, (2) the argument and (3) the result
- For example $\mathbf{II} = \lambda x.x\ \lambda x.x \to [x \mapsto \lambda x.x]x = \lambda x.x = \mathbf{I}$
- This rule is called β-reduction (def later)
- It can be applied anywhere within a term
- A location in a term where it can be applied is called a β-redex

- A judgment is a logical assertion, here[2]: $Term \rightarrow OtherTerm$
- An inference rule is a set of judgments $J_1, \ldots, J_n, J$ such that $J_1 \wedge \ldots \wedge J_n \Rightarrow J$
  - $J_1, \ldots, J_n$ are the premises, J is the conclusion
  - written $$\frac{J_1 \quad \cdots \quad J_n}{J}$$
  - An axiom is an inference rule with no premise
- A derivation is a tree of such rules where the leaves are axioms

$$\dfrac{\dfrac{}{J_1} \quad \dfrac{}{J_2} \quad \cdots \quad \dfrac{\dfrac{}{J_3} \quad \cdots \quad \dfrac{}{J_4}}{J_5}}{J_6}$$

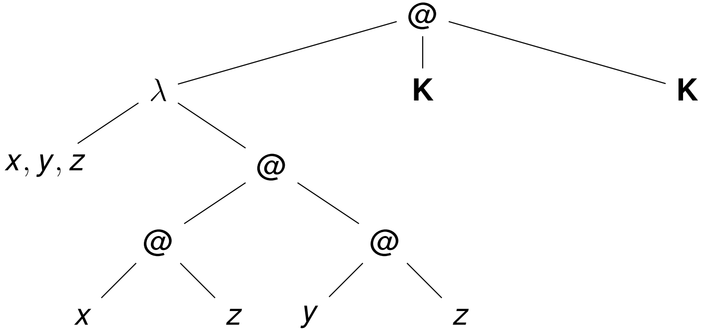
see http://en.wikipedia.org/wiki/Inference_rule

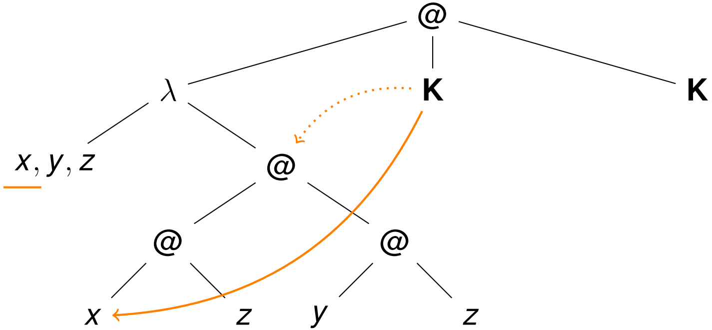
[2]There exists various other forms of judgment

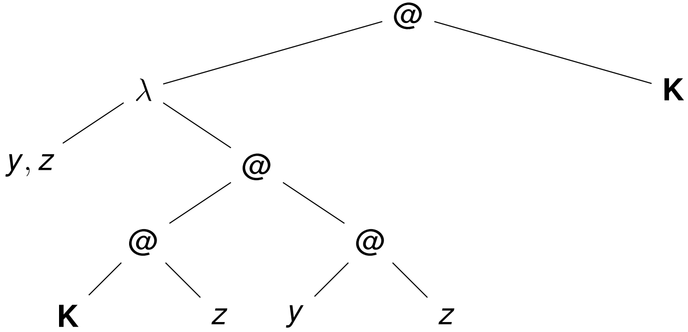
$$(1)\ (\lambda x.T_1)T_2 \rightarrow [x \mapsto T_2]T_1 \qquad (2)\ \frac{T \rightarrow T'}{\lambda x.T \rightarrow \lambda x.T'}$$

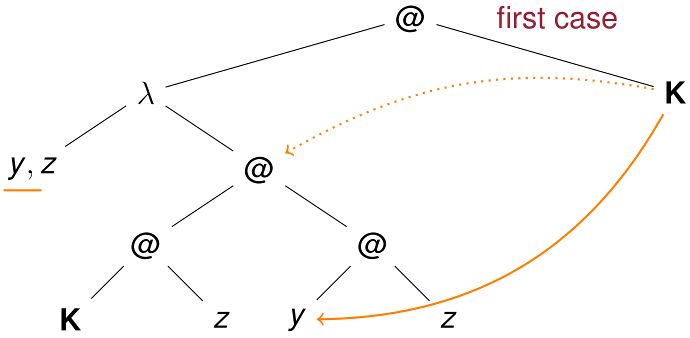
$$(3)\ \frac{T_1 \rightarrow T_1'}{T_1 T_2 \rightarrow T_1' T_2} \qquad (4)\ \frac{T_2 \rightarrow T_2'}{T_1 T_2 \rightarrow T_1 T_2'}$$

$$\begin{aligned}
\mathbf{SKK} &= \underline{\lambda x}.\lambda y.\lambda z.((xz)(yz))\underline{\mathbf{K}}\mathbf{K} && \text{def of } \mathbf{S} \\
&\rightarrow \underline{\lambda y}.\lambda z.((\mathbf{K}z)(yz))\underline{\mathbf{K}} && (1) \\
&\rightarrow \lambda z.((\mathbf{K}z)(\mathbf{K}z)) && (1) \\
&\rightarrow \lambda z.(((\underline{\lambda x}.\lambda y.x)\underline{z})(\mathbf{K}z)) && \text{def of } \mathbf{K} \\
&\rightarrow \lambda z.(\underline{\lambda y}.z\underline{(\mathbf{K}z)}) && (1) \\
&\rightarrow \lambda z.z && (1) \\
&\rightarrow \mathbf{I} && \text{def of } \mathbf{I}
\end{aligned}$$


@
λ
K
K
x, y, z
@
@
@
x
z
y
z

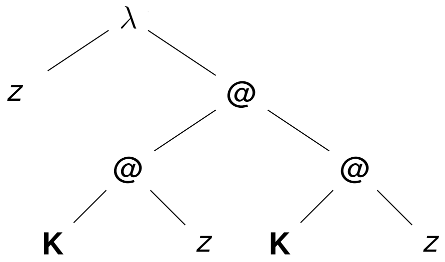

@
λ
K
K
x, y, z
@
@
@
x
z
y
z


@
λ
K
y, z
@
@
@
K
z
y
z


@
first case
λ
K
y, z
@
@
@
K
z
y
z

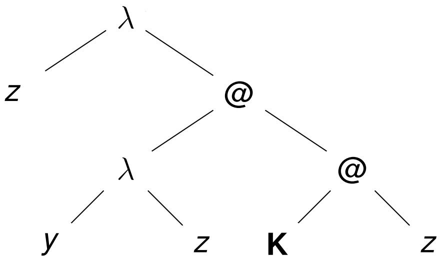
Another look

first case



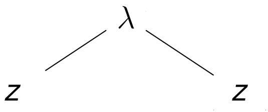
- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$

first case

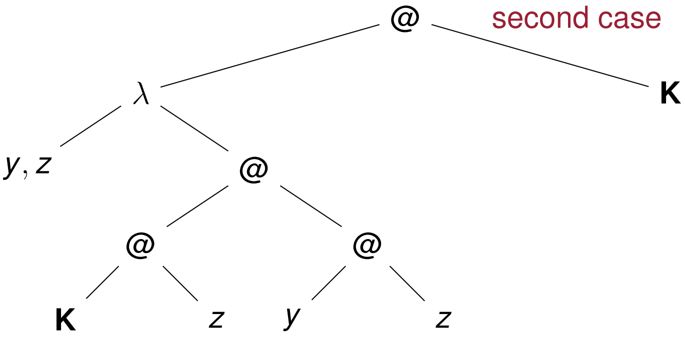


- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$

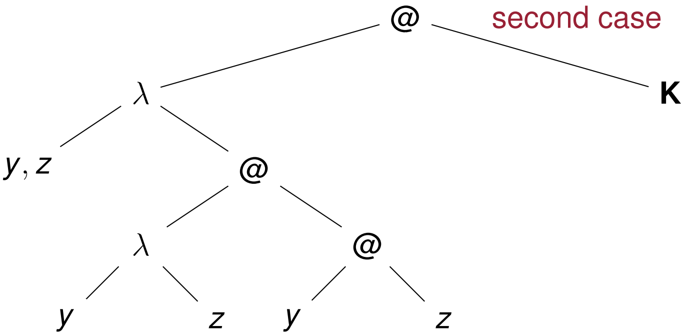
first case



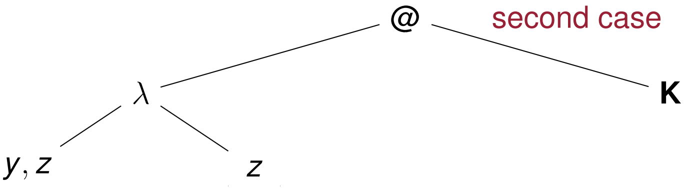
- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$



- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$

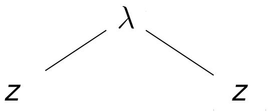


- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$



- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$

second case



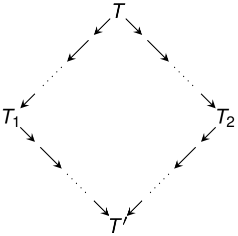
same result!

- $\mathbf{K}z = (\lambda x.\lambda y.x)z = \lambda y.z$
- $(\lambda y.z)T = z$

- A term T is irreducible or normal, if there exist no term it can reduce to ($T \not\to$)
- If T reduces to T' normal, T' is called a normal form of T
- A reduction sequence is a sequence $T_1 \to \cdots \to T_n$
  - denoted $T_1 \to^n T_n$
  - denoted $T_1 \to^* T_n$ if you don't care about the number of steps
- Often, there is several reduction sequences starting from a term (*e.g.* **SKK**)
- A reduction (resp. a term) is
  - (strongly) normalizing if all (resp. its) reduction sequences are finite
  - weakly normalizing if all terms have (resp. it has) a normal form
- $\Omega = (\lambda x.xx)(\lambda x.xx) \to \Omega$

⚠ β-reduction is not weakly normalizing for $\Lambda_{\mathcal{X}}$

- If T reduces to T_1 and T_2 there exists T' such that T_1 and T_2 both reduce to T'



- It shows that the path of computation is not important
- A term has at most one normal form

Church-Rosser theorem

β-reduction is confluent on $\Lambda_{\mathcal{X}}$

⚠ Some terms reduces indefinitely but has a normal form:
$\mathbf{KI}\Omega \to \mathbf{KI}\Omega$ or $\mathbf{KI}\Omega \to^2 \mathbf{I}$

Reduction strategy

- A reduction strategy is a way to choose the β-redex to reduce
- Standard orders
  - Normal order
    - the leftmost outermost reduction
    - always finds the normal form if it exists
  - Applicative order
    - the leftmost innermost reduction
    - only finds the normal form for normalizing terms
  - but both reduce inside functions (rule (2))
- Two other classical strategies (not using rule (2))
  - call by name: resolve application before evaluating the arguments
    - may duplicate computations
  - call by value: evaluate argument before application
    - optimal for sharing of computations

- In theory, yes as everything can be encoded as a λ
  - Turing has proved all computable functions can be written in $\Lambda_{\mathcal{X}}$
- In practice not usable, what is this term[3]?
  - $\lambda xyzu.(x(yzu)u)\lambda xy.(y(yx))\lambda xy.(yx)$

- We extend its core with
  - basic datatypes (integer, boolean, ...)
  - data structures (pairs, lists, ...)
  - recursion
  - ...

  It's the functional core of Ocaml! http://caml.inria.fr/ocaml

[3] We use $\lambda xyz.$ for $\lambda x.\lambda y.\lambda z.$, this notation si called currying

- In theory, yes as everything can be encoded as a λ
  - Turing has proved all computable functions can be written in $\Lambda_{\mathcal{X}}$
- In practice not usable, what is this term[3]?
  - $\lambda xyzu.(x(yzu)u)\lambda xy.(y(yx))\lambda xy.(yx)$
  - 2 + 1
- We extend its core with
  - basic datatypes (integer, boolean, . . .)
  - data structures (pairs, lists, . . .)
  - recursion
  - . . .

It's the functional core of Ocaml! http://caml.inria.fr/ocaml

[3]We use $\lambda xyz.$ for $\lambda x.\lambda y.\lambda z.$, this notation si called currying

Conclusion

- The λ-calculus
  - anything that is computable can be expressed
  - is often used to study sequential computation
  - close to a programing language (Caml)
  - for the interested [Lal90]
- Used to illustrate fundamental notions
  - variables, scope
  - induction
  - substitution
  - reduction
- Starting point to learn functional programming

N. Bourbaki.
Théorie des ensembles.
Eléments de mathématique. Springer, 2008.

René Lalement.
Logique Réduction Résolution.
ERI Masson, 1990.
The book has been translated in english under the title *Computation as logic* and edited by Prentice-Hall in 1993, ISBN 9780137700097.