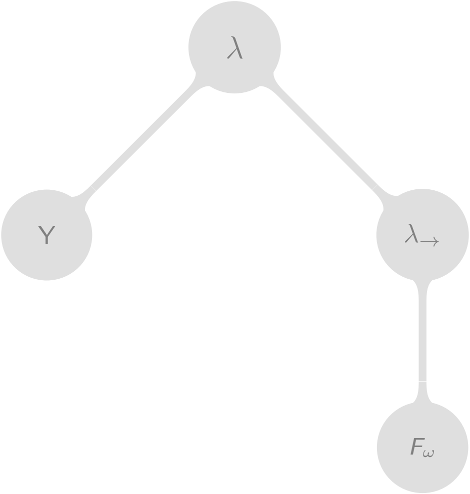
Introduction to Lambda Calculus

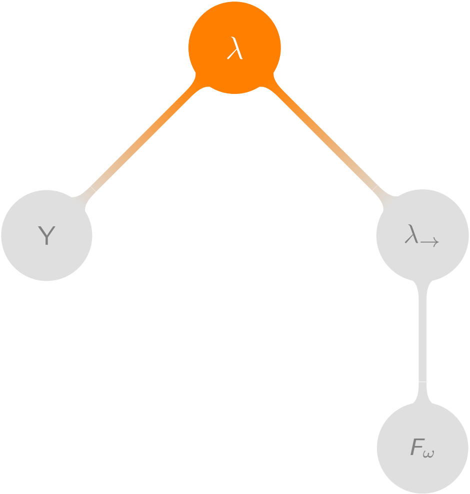
Maciek Makowski (@mmakowski)

7th December 2014

The Plan



Basic Lambda Calculus



Intuition

$$f(x) = 3 * x + 2$$
$$(\texttt{x} : \texttt{Int}) => 3 * \texttt{x} + 2$$

Intuition

$$f(x) = 3 * x + 2$$
$$(\mathtt{x} : \mathtt{Int}) => 3 * \mathtt{x} + 2$$
$$\lambda x. + \; (* \, 3 \, x) \, 2$$

Syntax

$$
\begin{array}{rcll}
\langle \mathit{term} \rangle & ::= & x & \text{(variable)} \\
& | & (\lambda x.\langle \mathit{term} \rangle) & \text{(abstraction)} \\
& | & (\langle \mathit{term} \rangle\ \langle \mathit{term} \rangle) & \text{(application)}
\end{array}
$$

where $x \in \mathbb{X}$ – the set of variables

Syntax

v_1

Syntax

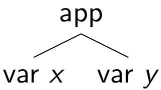
v_1 var v_1

Syntax

$x \ y$

Syntax

$x\ y$



Syntax

$\lambda a.b$

Syntax

$\lambda a.b$

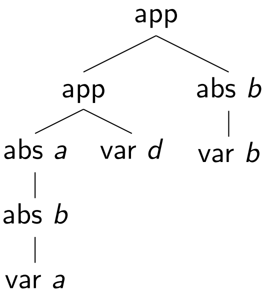
$$\begin{tikzcd}[arrows=dash] \mathsf{abs}\ a \arrow[d] \\ \mathsf{var}\ b \end{tikzcd}$$

Syntax

$$(\lambda a.\lambda b.a)\ d\ (\lambda b.b)$$

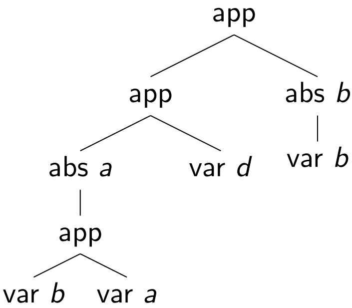
Syntax

$(\lambda a.\lambda b.a)\ d\ (\lambda b.b)$



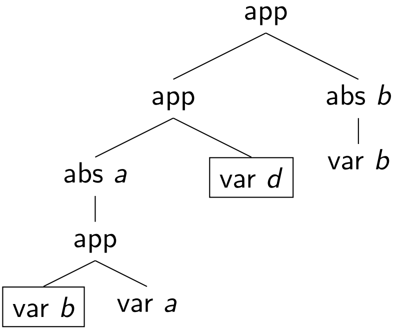
Syntax

$(\lambda a.b\ a)\ d\ (\lambda b.b)$



Syntax

$(\lambda a.\underline{b}\ a)\ \underline{d}\ (\lambda b.b)$



Syntax

- terms: trees consisting of
  - variables
  - abstractions
  - applications
- variables are *bound* by abstraction; otherwise *free*

β-reduction

$$(\lambda x.M)\ N \longrightarrow_\beta M[x/N]$$

β-reduction

$$(\lambda x.M)\ N \longrightarrow_\beta M[x/N]$$

$$(\lambda x.x\,y)\ (\lambda z.z) \longrightarrow_\beta (\lambda z.z)\ y \longrightarrow_\beta y$$

β-reduction

$$(\lambda x.M)\ N \longrightarrow_\beta M[x/N]$$

$$(\lambda a.a\ (\lambda a.a))\ b \longrightarrow_\beta b\ (\lambda a.a)$$

β-reduction

$$(\lambda x.M)\ N \longrightarrow_\beta M[x/N]$$

$$(\lambda a.(\lambda b.b) \ ① \ a) \ ③ \ ((\lambda c.\lambda d.d) \ ② \ \lambda f.f)$$

Church-Rosser

$$
\begin{tikzcd}
 & S \arrow[dl] \arrow[dr] & \\
T \arrow[dr, dotted] & & U \arrow[dl, dotted] \\
 & V &
\end{tikzcd}
$$

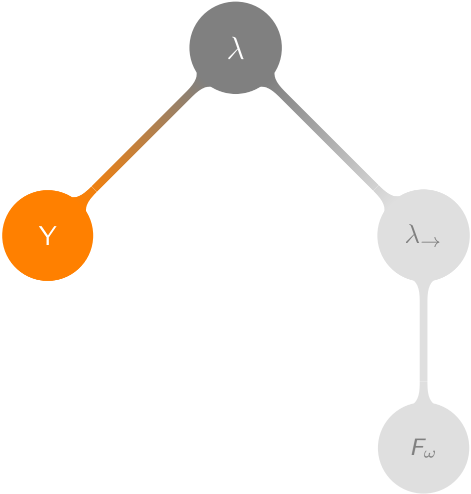
Termination

$$\Omega = (\lambda x.xx)(\lambda x.xx)$$

Semantics

- abstraction: function definition
- application: function application
- β-reduction: function evaluation

Programming in Lambda Calculus



Conditionals

$$\texttt{if}\ C\ \texttt{then}\ T\ \texttt{else}\ F$$

Conditionals

$$\texttt{if } C \texttt{ then } T \texttt{ else } F$$

$$\begin{aligned} \texttt{true} &= \lambda t.\lambda f.t \\ \texttt{false} &= \lambda t.\lambda f.f \end{aligned}$$

Conditionals

$$\texttt{if}\ C\ \texttt{then}\ T\ \texttt{else}\ F$$

$$\begin{aligned}
\texttt{true} &= \lambda t.\lambda f.t \\
\texttt{false} &= \lambda t.\lambda f.f \\
\texttt{test} &= \lambda c.\lambda t.\lambda f.c\ t\ f \\
\texttt{if}\ C\ \texttt{then}\ T\ \texttt{else}\ F &= \texttt{test}\ C\ T\ F
\end{aligned}$$

Numbers

$$
\begin{aligned}
0 &= \lambda s.\lambda z.z \\
\mathtt{succ} &= \lambda n.\lambda s.\lambda z.s\,(n\,s\,z)
\end{aligned}
$$

Numbers

$$\begin{aligned} 0 &= \lambda s.\lambda z.z \\ \texttt{succ} &= \lambda n.\lambda s.\lambda z.s\,(n\,s\,z) \end{aligned}$$

$$\begin{array}{lll} 0 = & & \lambda s.\lambda z.z \\ 1 = & \texttt{succ}\;\; 0 = & \lambda s.\lambda z.s\,z \\ 2 = & \texttt{succ}\;\; 1 = & \lambda s.\lambda z.s\,(s\,z) \\ 3 = & \texttt{succ}\;\; 2 = & \lambda s.\lambda z.s\,(s\,(s\,z)) \\ \vdots & & \\ \texttt{n} = & & \lambda s.\lambda z\,\underbrace{s\,(\ldots s\,(s}_{n}\;z)\ldots) \end{array}$$

Numbers

$$\begin{aligned} \mathtt{0} &= \lambda s.\lambda z.z \\ \mathtt{succ} &= \lambda n.\lambda s.\lambda z.s\,(n\,s\,z) \end{aligned}$$

$$\begin{aligned} \mathtt{plus} &= \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z) \\ \mathtt{times} &= \lambda m.\lambda n.m\ (\mathtt{plus}\ n)\ \mathtt{0} \end{aligned}$$

Recursion

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n * (n-1)! & \text{otherwise.} \end{cases}$$

Recursion

$$\mathtt{Y} = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

Recursion

$$\mathtt{Y} = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

$$\begin{aligned}\mathtt{g} &= \lambda f.\lambda n.\mathtt{if\ eq}\ n\ 0\ \mathtt{then\ 1\ else}\ (\mathtt{times\ n}\,(f\,(\mathtt{pred\ n})))\\ \mathtt{factorial} &= \mathtt{Y\ g}\end{aligned}$$

Recursion

$$\mathtt{Y} = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

$$\mathtt{g} = \lambda f.\lambda n.\mathtt{if\ eq}\ n\ \mathtt{0\ then\ 1\ else}\ (\mathtt{times\ n}\,(f\,(\mathtt{pred\ n})))$$

$$\mathtt{factorial} = \mathtt{Y\ g}$$

$\mathtt{factorial\ 3}$

$\mathtt{Y\ g\ 3}$

$(\mathtt{h\ h})\ \mathtt{3}$ where $\mathtt{h} = \lambda x.\mathtt{g}\ (x\ x)$

$\mathtt{g}\ (\mathtt{h\ h})\ \mathtt{3}$

$\mathtt{g\ fct\ 3}$ where $\mathtt{fct} = \mathtt{h\ h}$

$\mathtt{if\ eq\ 3\ 0\ then\ 1\ else}\ (\mathtt{times\ 3}\,(\mathtt{fct}\,(\mathtt{pred\ 3})))$

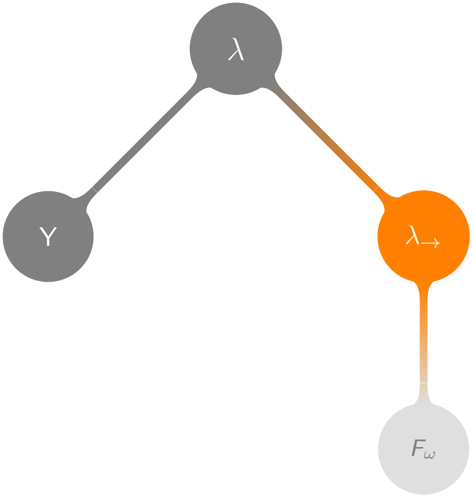
$\mathtt{times\ 3}\,(\mathtt{fct}\,(\mathtt{pred\ 3}))$

$\mathtt{times\ 3}\,((\mathtt{h\ h})\,\mathtt{2})$

Programming

- Church encoding: everything as lambda expression
- lambda calculus as a programming language
- small: makes formal proofs easier
- we can rely on intuition about mathematical functions

Simple Types



Simple Types

$$\lambda x.x : \sigma \to \sigma$$

$$\lambda f.\lambda x.f(f\,x) : (\sigma \to \sigma) \to \sigma \to \sigma$$

Simple Types

$$\frac{M : \sigma \to \tau \qquad N : \sigma}{M\,N : \tau} \qquad \text{(application)}$$

$$\frac{\begin{array}{c}\cancel{x : \sigma} \\ \vdots \\ M : \tau\end{array}}{\lambda x.M : \sigma \to \tau} \qquad \text{(abstraction)}$$

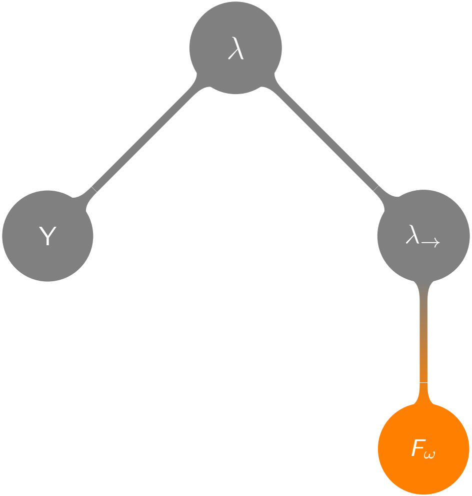
Simple Types

$$\cfrac{\cfrac{\cfrac{\cancel{f : \sigma \to \sigma} \qquad \cfrac{\cancel{f : \sigma \to \sigma} \quad \cancel{x : \sigma}}{f\,x : \sigma}\ (\mathit{app})}{f(f\,x) : \sigma}\ (\mathit{app})}{\lambda x.f(f\,x) : \sigma \to \sigma}\ (\mathit{abs})}{\lambda f.\lambda x.f(f\,x) : (\sigma \to \sigma) \to \sigma \to \sigma}\ (\mathit{abs})$$

Simple Types

- new calculus, $\lambda_{\rightarrow}$: if a term cannot be assigned a type, it is invalid
- simple types are very restrictive
- more complex type systems permit more programs

More Types



The Lambda Cube

$$\begin{tikzcd}
\lambda 2 & & \\
& \lambda\underline{\omega} & \\
\lambda_{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P
\end{tikzcd}$$

The Lambda Cube

$$
\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda P\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu, dotted] \arrow[rr, dotted] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda_{\to} \arrow[uu] \arrow[ur, dotted] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}
$$

Subtyping

$$
\begin{tikzcd}
 & & \lambda\omega \arrow[rr] & & \lambda P\omega \\
 & \lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & & \lambda\underline{\omega} \arrow[uu, dotted] \arrow[rr, dotted] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda_{<:} & \lambda_{\to} \arrow[l] \arrow[uu] \arrow[ur, dotted] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}
$$

Subtyping

$$\begin{tikzcd}
& F^{\omega}_{<:} & & \lambda\omega \arrow[ll] \arrow[rr] & & \lambda P\omega \\
F_{<:} \arrow[ur] & & \lambda 2 \arrow[ll] \arrow[rr] \arrow[ur] & & \lambda P2 \arrow[ur] & \\
& & & \lambda\underline{\omega} \arrow[uu, dotted] \arrow[rr, dotted] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda_{<:} \arrow[uu] & & \lambda_{\rightarrow} \arrow[ll] \arrow[uu] \arrow[ur, dotted] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

More Types

- many typed calculi
- orthogonal concepts can be combined into more complex calculi
- programming language researchers prove properties of calculi

Further Reading

- Benjamin C. Pierce, *Types and Programming Languages*
- Morten Heine B. Sørensen, Paweł Urzyczyn, *Lectures on the Curry-Howard Isomorphism*
- Henk Berendregt, Erik Barendsen, *Introduction to Lambda Calculus*
- Henk Berendregt *The Lambda Calculus, its Syntax and Semantics*

Notes

- https://github.com/mmakowski/introlambda/blob/scalax/notes.pdf
- https://github.com/mmakowski/introlambda/blob/scalax/slides.pdf