Lambda calculus for dummies

David Luposchainsky

2018-01-26

Lambdas today

Everyone has them

```
// Javascript
addOne = function(x) { return x+1; } // Old style
addOne = x => x+1;                   // ES6 style
```

Everyone has them

```
// Javascript
addOne = function(x) { return x+1; } // Old style
addOne = x => x+1;                    // ES6 style

// Java
boilerplate addOne = x -> return x+1;
```

Everyone has them

```
// Javascript
addOne = function(x) { return x+1; } // Old style
addOne = x => x+1;                   // ES6 style

// Java
boilerplate addOne = x -> return x+1;

// C++
auto addOne = [](int x) { return x+1; }
```

Everyone has them

```
// Javascript
addOne = function(x) { return x+1; } // Old style
addOne = x => x+1;                   // ES6 style

// Java
boilerplate addOne = x -> return x+1;

// C++
auto addOne = [](int x) { return x+1; }

-- Haskell
addOne = \x -> x+1
```

But why?

- Functions as first class values
- Long forgotten/ignored, then rediscovered

But why?

- Functions as first class values
- Long forgotten/ignored, then rediscovered

```
map (\x -> x+1) [1,2,3] -- [2,3,4]
```

But why?

- Functions as first class values
- Long forgotten/ignored, then rediscovered

```
map (\x -> x+1) [1,2,3] -- [2,3,4]
```

```
http.createServer((request, response) => {
    doStuff(request);
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('Hello World\n');
}).listen(1337, '127.0.0.1');
```

Back a century

Same thing, different views

Researching foundations of mathematics and computation

- Turing: imperative machine emulates mathematician
- Church: mathematical construct emulates mathematics

Turing machines

- Technical and somewhat complicated
- Powerful
- Theoretical use: immeasurably high
- Practical use: saying wrong or useless things on the internet

Definition of a Turing machine

A Turing machine consists of

$$\begin{aligned}
Q &\ \text{finite set of states} \\
\Gamma &\ \text{finite set of tape symbols} \\
b \in \Gamma &\ \text{blank symbol} \\
\Sigma \subseteq \Gamma \setminus \{b\} &\ \text{set of input symbols} \\
q_0 \in Q &\ \text{initial state} \\
F \subseteq Q &\ \text{set of final states} \\
\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\} &\ \text{transition function}
\end{aligned}$$

Lambda calculus

- Extremely simple
- Just as powerful
- Theoretical use: immeasurably high
- Practical use: incrementing lists of numbers

Lambda calculus

Definition of Lambda Calculus

Terms
- x (variable)
- $(\lambda x.T)$ (abstraction)
- $(T_1 \ \ T_2)$ (application)

Definition of Lambda Calculus

Terms

- x (variable)
- $(\lambda x.T)$ (abstraction)
- $(T_1 \;\; T_2)$ (application)

Examples

- $(\lambda x.\ x)$
- $(\lambda x\ (\lambda f.\ f\ (f\ x)))$

Definition of Lambda Calculus

Terms

- x (variable)
- $(\lambda x. T)$ (abstraction)
- $(T_1 \;\; T_2)$ (application)

Examples

- $(\lambda x.\ x)$
- $(\lambda x\ (\lambda f.\ f\ (f\ x)))$

Evaluation

- $(\lambda x. T_x) \equiv (\lambda y. T_y)$ (renaming variables)
- $(\lambda x. T_x) y \rightsquigarrow T_x[x \rightarrow y]$ (function application)
- $(\lambda x.(f\ x)) \equiv f$ (η reduction)

Evaluation: example 1

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

Evaluation: example 1

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

$$= ((\lambda y.\ x)[x \to 1])\ 2$$

Evaluation: example 1

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

$$= ((\lambda y.\ x)[x \to 1])\ 2$$

$$= (\lambda y.\ 1)2$$

Evaluation: example 1

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

$$= ((\lambda y.\ x)[x \rightarrow 1])\ 2$$

$$= (\lambda y.\ 1)2$$

$$= 1[y \rightarrow 2]$$

Evaluation: example 1

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

$$= ((\lambda y.\ x)[x \to 1])\ 2$$

$$= (\lambda y.\ 1)2$$

$$= 1[y \to 2]$$

$$= 1$$

Evaluation: example 2

$$((\lambda f.\ (\lambda x.\ (f\ x)))\ \text{double})\ 4$$

Evaluation: example 2

$$((\lambda f.\ (\lambda x.\ (f\ x)))\ \text{double})\ 4$$

$$= ((\lambda x.\ (f\ x))[f \to \text{double}])\ 4$$

Evaluation: example 2

$$((\lambda f.\ (\lambda x.\ (f\ x)))\ \text{double})\ 4$$

$$= ((\lambda x.\ (f\ x))[f \rightarrow \text{double}])\ 4$$

$$= (\lambda x.\ (\text{double}\ x))\ 4$$

Evaluation: example 2

$$((\lambda f.\ (\lambda x.\ (f\ x)))\ \text{double})\ 4$$

$$= ((\lambda x.\ (f\ x))[f \to \text{double}])\ 4$$

$$= (\lambda x.\ (\text{double}\ x))\ 4$$

$$= (\text{double}\ x)[x \to 4]$$

Evaluation: example 2

$$((\lambda f.\ (\lambda x.\ (f\ x)))\ \text{double})\ 4$$

$$= ((\lambda x.\ (f\ x))[f \to \text{double}])\ 4$$

$$= (\lambda x.\ (\text{double}\ x))\ 4$$

$$= (\text{double}\ x)[x \to 4]$$

$$= \text{double}\ 4$$

Too many parentheses!

$$
\begin{aligned}
\text{const} &= \lambda x.\ (\lambda y.\ x) \\
&= \lambda x\ y.\ x
\end{aligned}
$$

$$
\begin{aligned}
\text{apply} &= \lambda f.\ (\lambda x.\ (f\ x)) \\
&= \lambda f\ x.\ f\ x
\end{aligned}
$$

$$
\lambda x.\ ((f\ x)\ y)\ z = \lambda x.\ f\ x\ y\ z
$$

Numbers

$=$ repeated function application

- $0 := \lambda f\, x.\ x$
- $1 := \lambda f\, x.\ f\, x$
- $2 := \lambda f\, x.\ f\, (f\, x)$

Numbers

$=$ repeated function application

- $0 := \lambda f\, x.\ x$
- $1 := \lambda f\, x.\ f\, x$
- $2 := \lambda f\, x.\ f\, (f\, x)$

$$\text{succ} := \lambda n.\ (\lambda f\, x.\ f\, (n\ f\, x))$$

Numbers

$=$ repeated function application

- $0 := \lambda f\, x.\ x$
- $1 := \lambda f\, x.\ f\, x$
- $2 := \lambda f\, x.\ f\, (f\, x)$

$$\text{succ} := \lambda n.\ (\lambda f\, x.\ f\, (n\ f\, x))$$

$$\text{add} := \lambda m\ n.\ (\lambda f\, x.\ m\ f\, (n\ f\, x))$$

Numbers

$=$ repeated function application

- $0 := \lambda f\, x.\ x$
- $1 := \lambda f\, x.\ f\, x$
- $2 := \lambda f\, x.\ f\, (f\, x)$

$$\text{succ} := \lambda n.\ (\lambda f\, x.\ f\, (n\ f\, x))$$

$$\text{add} := \lambda m\ n.\ (\lambda f\, x.\ m\ f\, (n\ f\, x))$$

$$\begin{aligned}\text{mul} :&= \lambda m\ n.\ (\lambda f.\ m\ (n\ f))\\ &= \lambda m\ n.\ (\lambda f\, x.\ m\ (n\ f)\ x)\end{aligned}$$

Booleans

$=$ ignore one branch

$$\begin{aligned} \text{true} &:= \lambda x\ y.\ x \\ \text{false} &:= \lambda x\ y.\ y \end{aligned}$$

Booleans

$=$ ignore one branch

$$
\begin{aligned}
\mathsf{true} &:= \lambda x\ y.\ x \\
\mathsf{false} &:= \lambda x\ y.\ y
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{ifThenElse} :=& \lambda p\ t\ f.\ (p\ t)\ f \\
\equiv& \lambda p\ t.\ p\ t \\
\equiv& \lambda p.\ p \\
\equiv& \mathsf{id}
\end{aligned}
$$

Booleans

$=$ ignore one branch

$$\begin{aligned}
\text{true} &:= \lambda x\ y.\ x \\
\text{false} &:= \lambda x\ y.\ y \\
\text{ifThenElse} &:= \lambda p\ t\ f.\ (p\ t)\ f \\
&\equiv \lambda p\ t.\ p\ t \\
&\equiv \lambda p.\ p \\
&\equiv \text{id} \\
\text{not} &:= \lambda p.\ p\ \text{false}\ \text{true} \\
\text{and} &:= \lambda p\ q.\ p\ q\ \text{false} \\
\text{or} &:= \lambda p\ q.\ p\ \text{true}\ q
\end{aligned}$$

Pairs/lists

= nil/cons like in Lisp

$$\begin{aligned}
\text{cons} &:= \lambda x\ y\ f.\ f\ x\ y \\
\text{first} &:= \lambda p.\ p\ \text{true} \\
\text{second} &:= \lambda p.\ p\ \text{false} \\
\text{nil} &:= \lambda x.\ \text{true} \\
\text{null} &:= \lambda p.\ p\ (\lambda x\ y.\ \text{false})
\end{aligned}$$

Recursion

...but how?

Recursion

...but how?
Idea: emulate module system

$$(\lambda \mathit{helper}.\langle\text{program}\rangle)$$
$$(\langle\text{value for helper}\rangle)$$

Recursion

...but how?
Idea: emulate module system

$$(\lambda \textit{helper}. \langle \text{program} \rangle)$$
$$(\langle \text{value for helper} \rangle)$$

$$(\lambda \textit{add}.\ \textit{add}\ 3\ 4)$$
$$(\lambda m\ n.\ (\lambda f\ x.\ m\ f\ (n\ f\ x)))$$

Infinite modules, yaay

$$(\lambda \mathit{infiniteLoop}.\ \mathit{infiniteLoop})$$
$$(\mathit{infiniteLoop}\textcolor{red}{\mathit{BZZZT}})$$

Infinite modules, yaay

$$(\lambda \mathit{infiniteLoop}.\ \mathit{infiniteLoop})$$
$$(\mathit{infiniteLoop}\,\textcolor{red}{\mathit{BZZZT}})$$

```
(\inf. inf)
    (inf BZZZT)
```

Infinite modules, yaay

$$(\lambda \mathit{infiniteLoop}.\ \mathit{infiniteLoop})$$
$$(\mathit{infiniteLoop}\textcolor{red}{\mathit{BZZZT}})$$

```
(\inf. inf)
    (inf BZZZT)
```

```
(\inf. inf)
    ((\inf2. inf2)
        (inf2 BZZZT))
```

Infinite modules, yaay

$$(\lambda \mathit{infiniteLoop}.\ \mathit{infiniteLoop})$$
$$(\mathit{infiniteLoop}\,\mathit{BZZZT})$$

```
(\inf. inf)
    (inf BZZZT)

(\inf. inf)
    ((\inf2. inf2)
        (inf2 BZZZT))

(\inf. inf)
    ((\inf2. inf2)
        ((\inf3. inf3)
            ((\inf4. inf4)
                (inf4 BZZZT))))
```

Fixed point combinator!

$$\begin{aligned} \mathsf{Y}\ f &= f\,(\mathsf{Y}\ f) \\ &= f\,(f\,(\mathsf{Y}\ f)) \\ &= f\,(f\,(f\,(\ldots))) \end{aligned}$$

Fixed point combinator!

$$
\begin{aligned}
\mathrm{Y}\ f &= f\,(\mathrm{Y}\ f) \\
&= f\,(f\,(\mathrm{Y}\ f)) \\
&= f\,(f\,(f\,(\ldots)))
\end{aligned}
$$

$$Y = \lambda f.\ (\lambda x.\ f\,(x\ x))\ (\lambda x.\ f\,(x\ x))$$

Y combinator usage

$$\begin{aligned} \text{factorialStep} := \lambda rec\ n.\ \text{ifThenElse}\ & (n > 0) \\ & (n * rec\ (n - 1)) \\ & 1 \end{aligned}$$

Y combinator usage

$$\begin{aligned}\text{factorialStep} := \lambda rec\ n.\ \text{ifThenElse}\ &(n > 0)\\ &(n * rec\ (n-1))\\ &1\end{aligned}$$

$$\text{factorial} := \text{Y factorialStep}$$

Example

$$
\begin{aligned}
\text{factorial } 3 &= (\text{Y factorialStep})\ 3 \\
&= \text{factorialStep } (\text{Y factorialStep})\ 3 \\
&= (\lambda \mathit{rec}.\ \text{ifThenElse } (3 > 0)\ (3 * \mathit{rec}\ (3 - 1))\ 1)\ (\text{Y factorialStep}) \\
&= (\lambda \mathit{rec}.\ (3 * \mathit{rec}\ 2))\ (\text{Y factorialStep}) \\
&= 3 * \text{factorial } 2 \\
&= 3 * 2 * 1 * \text{factorial } 0 \\
&= 6 * \text{Y factorialStep } 0 \\
&= 6 * (\lambda \mathit{rec}.\ \text{ifThenElse } (0 > 0)\ (0 * \mathit{rec}\ (0 - 1))\ 1)\ (\text{Y factorialStep}) \\
&= 6 * (\lambda \mathit{rec}.\ 1)\ (\text{Y factorialStep}) \\
&= 6 * 1 \\
&= 6
\end{aligned}
$$

What now?

- Invent Lisp
- Add types (simply typed LC, Hindley/Milner)
- CPS transform your whole program to make maintenance a nightmare

Questions?

```
fibo :=
    (\pred true false Y 1 2.
        (\isZero sub.
            (\leq.
                Y (\rec n. (leq n 2)
                           n
                           (add (rec (sub n 1))
                                (rec (sub n 2)))))
                (\m n. isZero (sub m n)))
            (\n. n (\x. false) true)
            (\m n. n pred m))
        (\n f x. n (\g h. h (g f)) (\u. x) (\u. u))
        (\x y. x)
        (\x y. y)
        (\f. (\x. f (x x)) (\x. f (x x)))
        (\f x. f x)
        (\f x. f (f x))
```

Church-Rosser theorem, evaluation strategies

»Normal forms are unique«

Church-Rosser theorem, evaluation strategies

»Normal forms are unique«

Applicative order

- Reduce the rightmost-innermost redex first
- Used by eager languages (Lisps, C, Java, …)
- Might not find normal form even though it exists:

$$(\lambda x\ y.\ x)\ 1\ \underbrace{((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))}_{\Omega} \rightsquigarrow \bot$$

Church-Rosser theorem, evaluation strategies

»Normal forms are unique«

Applicative order

- Reduce the rightmost-innermost redex first
- Used by eager languages (Lisps, C, Java, …)
- Might not find normal form even though it exists:

$$(\lambda x\ y.\ x)\ 1\ \underbrace{((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))}_{\Omega} \leadsto \bot$$

Normal order

- Reduce the leftmost-outermost redex first
- Used by lazy languages (Haskell, Miranda)
- If there is a normal form, it will be reached

$$(\lambda x\ y.\ x)\ 1\ \underbrace{((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))}_{\Omega} \leadsto 1$$

SKI calculus

»Lambda calculus is too complicated!«

$$K\ x\ y \longrightarrow x$$
$$S\ f\ g\ x \longrightarrow f\ x\ (g\ x)$$

Universal iota

»SKI calculus is too complicated!«

$$\iota := \lambda f.\ f\ S\ K = S\ (S\ I\ (K\ S))\ (K\ K)$$

$$K = (\iota(\iota(\iota\iota)))$$

$$S = (\iota(\iota(\iota(\iota\iota))))$$