Chapter 7

Introduction to the λ-Calculus

History

- Invented in 1934 by Alonzo Church
- Proven equivalent to Turing-machines

Alan Turing was PhD student of Alonzo Church!

- Direct inspiration for Lisp in 1958 (John McCarthy)
- Numerous extensions exist
- Continuations were discovered for translating Algol to the λ-calculus.

What is it?

formal model for computations

a "rewrite" system

The λ-calculus consists of

- An infinite set of strings called λ-expressions.
- A relation $=_\beta$ that expresses computation by rewriting.

Inspired by high school algebra

rewrite

$$1 + 3 \times 4 = 1 + 12 = 13$$

rewrite

λ-Expressions

Let V be an infinite set of variables.

The set of λ-expressions is recursively defined as:

- $v \in V$ is a λ-expression
- If E is a λ-expression and if $x \in V$, then $\lambda x.E$ is a λ-expression (abstraction)
- If E and E' are λ-expressions, then so is (E E') (application)

Examples:

identity function

λf. f

λf. (f (f (f (f x))))

x

(λg. g λf. h)

((x y) z)

(x (y z))

Free Variables

The set of free variables FV of a λ-expression is defined/computed as follows:

$$FV(\ x\ ) = \{\ x\ \}$$
$$FV(\ \lambda x.E\ ) = FV(\ E\ ) \setminus \{\ x\ \}$$
$$FV(\ (E\ E')\ ) = FV(\ E\ ) \cup FV(\ E'\ )$$

A λ-expression E is called closed if FV(E) = ∅

"Useful" programs

a.k.a. combinators

Only the first one is closed

λf. f
λf. (f (f (f (f x))))
x
(λg. g λf. h)
((x y) z)
(x (y z))

Remember the substitution model of evaluation

Substitution

The workhorse of λ-calculus

Let E and E' be λ-expressions and let x be some variable. The substitution of E' for x in E, denoted $[E' \to x]E$ is defined by case-analysis on E:

- $[E' \to x]\ x \quad = E'$
- $[E' \to x]\ y \quad = y$
- $[E' \to x]\ \lambda x.\ E_1 \quad = \lambda x.\ E_1$
- $[E' \to x]\ \lambda y.\ E_1 \quad = \lambda y.\ [E' \to x]\ E_1 \qquad$ if $y \notin FV(E')$
- $[E' \to x]\ \lambda y.\ E_1 \quad = \lambda z.\ [E' \to x]\ [z \to y]\ E_1 \qquad$ if $y \in FV(E'),\ z \notin FV(E')$
- $[E' \to x]\ (E_1\ E_2) \quad = ([E' \to x]E_1\ [E' \to x]E_2)$

Lexical scoping!

Lexical scoping!

Pick one

Examples

$[f \to x]\ \lambda y.\ x = \lambda y.\ [f \to x]\ x = \lambda y.\ f$

$[f \to x]\ \lambda y.\ y = \lambda y.\ [f \to x]\ y = \lambda y.\ y$

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

$[f \to x]\ (\lambda f.\ (f\ x)\ x)$
$= ([f \to x]\lambda f.\ (f\ x)\ [f \to x]x)$
$= (\lambda z.\ [f \to x]\ [z \to f](f\ x)\ f)$
$= (\lambda z.\ [f \to x](z\ x)\ f)$
$= (\lambda z.\ (z\ f)\ f)$

β - equality

The basic rewrite rule

The $=_\beta$ relation between λ-expressions is defined as follows:

A "reducible expression" aka "redex"

- $(\lambda x.E'\ E) =_\beta [E \rightarrow x]\ E'$
- $\lambda x.E =_\beta \lambda y.[y \rightarrow x]E$
- $E =_\beta E$
- if $E_1 =_\beta E_2$ and $E_2 =_\beta E_3$ then $E_1 =_\beta E_3$
- if $E_1 =_\beta E'_1$ and $E_2 =_\beta E'_2$ then $(E_1\ E_2) =_\beta (E'_1\ E'_2)$
- if $E =_\beta E'$ then $\lambda x.E =_\beta \lambda x.E'$

Examples

$(\lambda x.\, x\ y) =_\beta y$

$(\lambda x.\, (\lambda x.\, x\ x)\ y) =_\beta (\lambda x.\, x\ y) =_\beta y$

$(\lambda x.\, x\ (y\ \lambda z.\, z)) =_\beta (y\ \lambda z.\, z)$

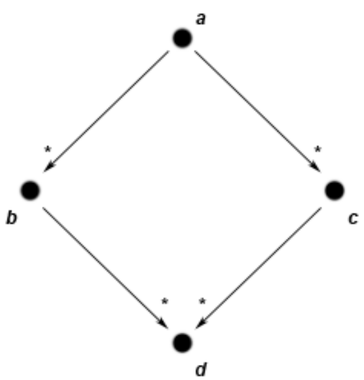
$(\lambda x.\, (x\ x)\ \lambda x.\, (x\ x))$
$=_\beta (\lambda x.\, (x\ x)\ \lambda x.\, (x\ x))$
$=_\beta (\lambda x.\, (x\ x)\ \lambda x.\, (x\ x))$
$=_\beta (\lambda x.\, (x\ x)\ \lambda x.\, (x\ x))$
$=_\beta \ldots$

You can work “normal order”, “applicative order or “whatever order”

Some order may terminate, another may not.

Church-Rosser Theorem:

If two orders terminate, then their result is the same



Functions of Multiple Arguments

```
(define plus
  (lambda (n m)
    (+ n m)))

(define curried-plus
  (lambda (n)
    (lambda (m)
      (+ n m))))
```

Currying is the technique that allows us to program functions of multiple parameters

```
> (plus 1 2)
3
> ((curried-plus 1) 2)
3
```

Notice that + is not λ-calculus!

The application to *a* arguments is simulated by *a* applications

Church Numerals

Notational Shorthand (definable by induction):

$$(E^n\ E') = (E\ (E\ (E\ (E\ \ldots\ (E\ E')))))$$

Church Numerals

Encoding natural numbers in the λ-calculus

$C_0 = \lambda f.\ \lambda x.\ x$

$C_n = \lambda f.\ \lambda x.\ (f^n\ x)$

Examples:

$C_0 = \lambda f.\ \lambda x.\ x$

$C_1 = \lambda f.\ \lambda x.\ (f\ x)$

$C_4 = \lambda f.\ \lambda x.\ (f\ (f\ (f\ (f\ x))))$

λ-Definability

Can we construct a "program" that computes f?

A function $f : \mathbb{N}^a \longrightarrow \mathbb{N}$ of a arguments is called λ-definable if there exists a λ-expression F such that:

$$(\ldots(F\ c_{n1})\ c_{n2})\ \ldots\ c_{na}) =_\beta c_{f(n1,\ldots,na)}$$

Using currying for multiple parameters

Examples: $(\text{inc}\ c_{40}) =_\beta c_{41}$

$((\text{plus}\ c_4)\ c_6) =_\beta c_{4+6}$

Church Numerals

Arithmetic

$C_0 = \lambda f.\ \lambda x.\ x$
$C_n = \lambda f.\ \lambda x.\ (f^n\ x)$

Arithmetic Operators:

$\text{inc} = \lambda n.\ \lambda f.\ \lambda x.(f\ ((n\ f)\ x))$
$\text{plus} = \lambda n.\ \lambda m.\ \lambda f.\ \lambda x.((n\ f)\ ((m\ f)\ x))$
$\text{times} = \lambda n.\ \lambda m.\ \lambda f.\ (n\ (m\ f))$
$\text{power} = \lambda n.\ \lambda m.\ (m\ n)$
$\text{dec} = ?$
$\text{minus} = ?$

(+1), +, * and ^ are λ-definable.

How do we "remove" an application of f?

Booleans

Definitions:

true = λt. λf. t
false = λt. λf. f

if = λt. λc. λa. ((t c) a)

not = λt. (((if t) false) true)
and = λp. λq. (((if p) q) p)
or = λp. λq. (((if p) p) q)

Example:

(((if true) C) A)
= (((if λt. λf. t) C) A)
= (((λt. λc. λa. ((t c) a) λt. λf. t) C) A)
$=_\beta$ ((λc. λa. ((λt. λf. t c) a) C) A)
$=_\beta$ ((λc. λa. (λf. c a) C) A)
$=_\beta$ ((λc. λa. c C) A)
$=_\beta$ (λa. C A)
$=_\beta$ C

It does what we expect it to do!

Pairs

Definitions:

cons = λa. λd. λs. ((s a) d)
car = λp. (p λa. λd. a)
cdr = λp. (p λa. λd. d)

(car ((cons true) C_0))
$=_\beta$ (car ((λa. λd. λs. ((s a) d) true) C_0))
$=_\beta$ (car (λd. λs. ((s true) d) C_0)
$=_\beta$ (car λs. ((s true) C_0)
$=_\beta$ (λp. (p λa. λd. a) λs. ((s true) C_0)
$=_\beta$ (λs. ((s true) C_0) λa. λd. a)
$=_\beta$ ((λa. λd. true) C_0)
$=_\beta$ (λd. true C_0)
$=_\beta$ true

Subtraction

Definitions: nextpair = λp. ((cons (cdr p)) (inc (cdr p)))

dec = λn. (car ((n nextpair) ((cons c_0) c_0)))

(dec c_3)

= (car ((c_3 nextpair) ((cons c_0) c_0)))

$=_\beta$ (car ((λf. λx. (f (f (f x))) nextpair) ((cons c_0) c_0)))

$=_\beta$ (car (nextpair (nextpair (nextpair ((cons c_0) c_0))))

$=_\beta$ (car (nextpair (nextpair ((cons c_0) c_1))))

$=_\beta$ (car (nextpair ((cons c_1) c_2)))

$=_\beta$ (car ((cons c_2) c_3)))

$=_\beta$ c_2

Recursion

This is an infinite program

$$\text{fac} \stackrel{?}{=} \lambda n.\ (((\text{if}\ ((=\ n)\ c_0)\ c_1)\ ((\text{times}\ n)\ (\text{fac}\ (\text{dec}\ n))))$$

Can we find an expression that behaves as expected?

$$\text{fac} =_\beta \lambda n.\ (((\text{if}\ ((=\ n)\ c_0)\ c_1)\ ((\text{times}\ n)\ (\text{fac}\ (\text{dec}\ n))))$$

$$\Rightarrow \quad \text{fac} =_\beta (\lambda f.\ \lambda n.\ (((\text{if}\ ((=\ n)\ c_0)\ c_1)\ ((\text{times}\ n)\ (f\ (\text{dec}\ n))))\ \text{fac})$$

$$\Rightarrow \quad \text{fac} =_\beta (F\ \text{fac})$$

Seems like we are looking for the fixed point of the green guy

$$\Rightarrow \quad \text{fac} = (Y\ \lambda f.\ \lambda n.\ (((\text{if}\ ((=\ n)\ c_0)\ c_1)\ ((\text{times}\ n)\ (f\ (\text{dec}\ n)))))$$

$$Y = \lambda F.\ (\lambda x.\ (F\ (x\ x))\ \lambda x.\ (F\ (x\ x)))$$

Fixed-point Combinator

Mathematical Interpretation of Y

For every λ-expression F, (Y F) is the fixed-point of F.

Proof:

$$
\begin{aligned}
&(Y\ F)\\
&= (\lambda F.\ (\lambda x.\ (F\ (x\ x))\ \lambda x.\ (F\ (x\ x)))\ F)\\
&= (\lambda x.\ (F\ (x\ x))\ \lambda x.\ (F\ (x\ x)))\\
&= (F\ (\lambda x.\ (F\ (x\ x))\ \lambda x.\ (F\ (x\ x))))\\
&= (F\ (Y\ F))
\end{aligned}
$$

Operational Interpretation of Y

$$
\begin{aligned}
& Y \\
&= \lambda F.\,(\lambda x.\,(F\,(x\,x))\ \lambda x.\,(F\,(x\,x))) \\
&=_\beta \lambda F.\,(F\,(\lambda x.\,(F\,(x\,x))\ \lambda x.\,(F\,(x\,x)))) \\
&=_\beta \lambda F.\,(F\,(F\,(\lambda x.\,(F\,(x\,x))))\ \lambda x.\,(F\,(x\,x))))) \\
&=_\beta \lambda F.\,(F\,(F\,(F\,(\lambda x.\,(F\,(x\,x)))))\ \lambda x.\,(F\,(x\,x))))))) \\
&=_\beta \lambda F.\,(F\,(F\,(F\,(F\,(\lambda x.\,(F\,(x\,x))\ \lambda x.\,(F\,(x\,x)))))))
\end{aligned}
$$

Xerox

$$
\begin{aligned}
& (\mathrm{fac}\ c_3) = ((Y\ F)\ c_3) \\
&= (\lambda n.\,(\mathrm{if}\ \ldots\ ((F\,(\lambda x.\,(F\,(x\,x))\ \lambda x.\,(F\,(x\,x))))\ (\mathrm{dec}\ n))\ \ldots)\ c_3) \\
&=_\beta (\mathrm{if}\ \ldots\ ((F\,(\lambda x.\,(F\,(x\,x))\ \lambda x.\,(F\,(x\,x))))\ c_2)\ \ldots) \\
&=_\beta (\mathrm{if}\ \ldots\ (\lambda n.\,(\mathrm{if}\ \ldots\ ((F\,(\lambda x.\,(F\,(x\,x))\ \lambda x.\,(F\,(x\,x))))\ (\mathrm{dec}\ n))\ \ldots)\ c_2)\ \ldots) \\
&=_\beta (\mathrm{if}\ \ldots\ (\mathrm{if}\ \ldots\ ((F\,(\lambda x.\,(F\,(x\,x))\ \lambda x.\,(F\,(x\,x))))\ c_1)\ \ldots)\ c_2)\ \ldots) \\
&=_\beta \ldots
\end{aligned}
$$

Last ‘if’ doesn’t use the Xerox anymore

Chapter 7