Lambda Calculus - Recursions (9A)

Young Won Lim
12/26/24

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Fixed-point combinator (1)

a **fixed-point combinator**

(or fixpoint combinator),

denoted **fix**, is a higher-order function

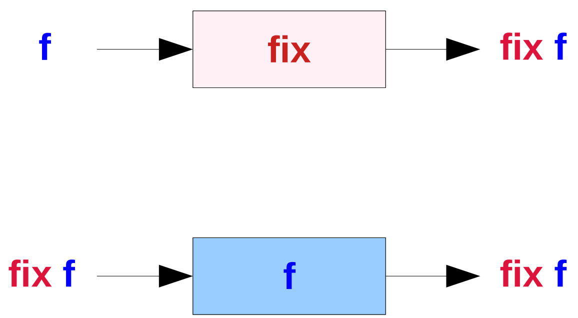
which takes a function **f** as argument

that returns some fixed point **(fix f)**

(a value that is mapped to itself)

of its argument function **f**, if one exists.

fix f = f (fix f),



https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (2)

some fixed point **(fix f)** of its argument function **f**, if one exists.

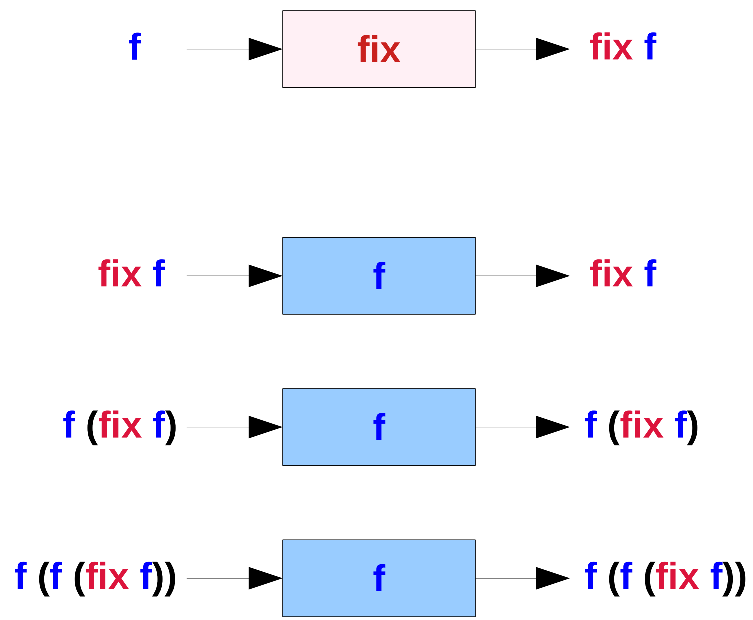
Formally, if the **function f** has one or more fixed points, then

fix f = f (fix f),

and hence, by repeated application,

fix f = f (f (… f (fix f) …))

fix f	fixed point
fix	fixed point combinator



Y = (λf. (λx.f (x x)) (λx. f (x x)))

Y g = g (Y g)

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (3)

Every recursively defined function can be seen
as a fixed point of some suitably defined function
closing over the recursive call with an extra argument,

and therefore, using **Y**, every recursively defined function
can be expressed as a lambda expression.

In particular, we can now cleanly define
the subtraction, multiplication and comparison predicate
of natural numbers recursively.

Y = λf. (λx.f (x x)) (λx. f (x x))

Y g = g (Y g)

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Fixed-point combinator (4)

In the classical untyped lambda calculus,
every function has a fixed point.

A particular implementation of **fix** is
Curry's paradoxical **combinator Y**, represented by

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

In functional programming, the **Y combinator** can be used
to formally define recursive functions in a programming language
that does not support recursion.

https://en.wikipedia.org/wiki/Fixed-point_combinator

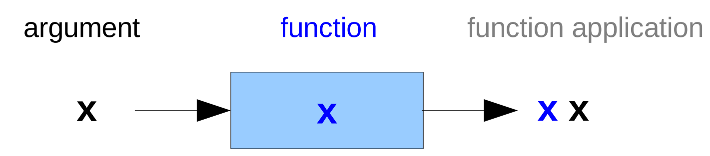
Fixed-point combinator (5)

Y = λf. (λx. f (x x)) (λx. f (x x))

Y is a function that takes one argument **f** and returns the entire expression following the first period;

(λx. f (x x)) (λx. f (x x))

the expression **(λx. f (x x))** denotes a function that takes one argument **x**, thought of as a function, and returns the expression **f (x x)**,



(λx. x x)

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (6)

Y = λf. (λx. f (x x)) (λx. f (x x))

the expression **(λx. f (x x))** denotes a function

that takes one argument **x**,

which is thought of as a function,

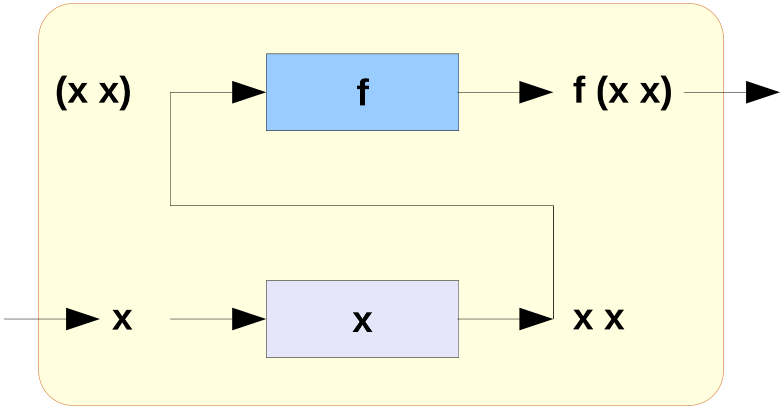
and returns the expression **f (x x)**,

where **(x x)** denotes

a function **x** applied to itself as an argument.

Juxtaposition of expressions denotes function application,

is left-associative, and has higher precedence than the period.)



(λx. f (x x))

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (7)

The following calculation verifies that **Y g** is indeed a fixed point of the function **g** :

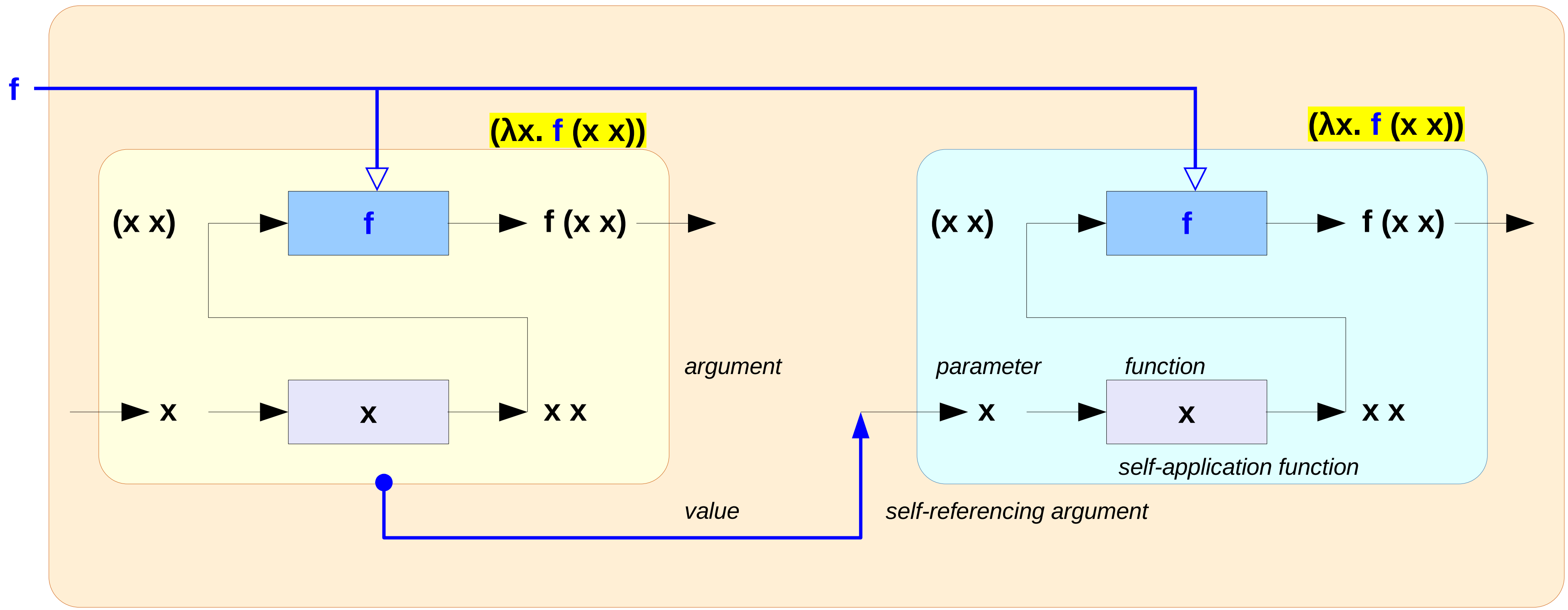
Y g	**= (λf. (λx.f (x x)) (λx. f (x x))) g**	by the definition of **Y**
	= (λx. g (x x)) (λx. g (x x))	by β-reduction: replacing the formal argument **f** of **Y** with the actual argument **g**
	= g ((λx. g (x x)) (λx. g (x x)))	by β-reduction: replacing the formal argument x of the first function with the actual argument **(λx. g (x x))**
	= g (Y g)	by second equality, above

Y	**= λf. (λx. f (x x)) (λx. f (x x))**
Y g	**= (λx. g (x x)) (λx. g (x x))**
g (Y g)	**= g ((λx. g (x x)) (λx. g (x x)))**

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (8)

$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$ *combinator*



fixed point

$Y\ g = g\ (Y\ g)$

$fix\ f = f\ (fix\ f)$

$fix\ F = F\ (fix\ F)$

Y combinator + argument function

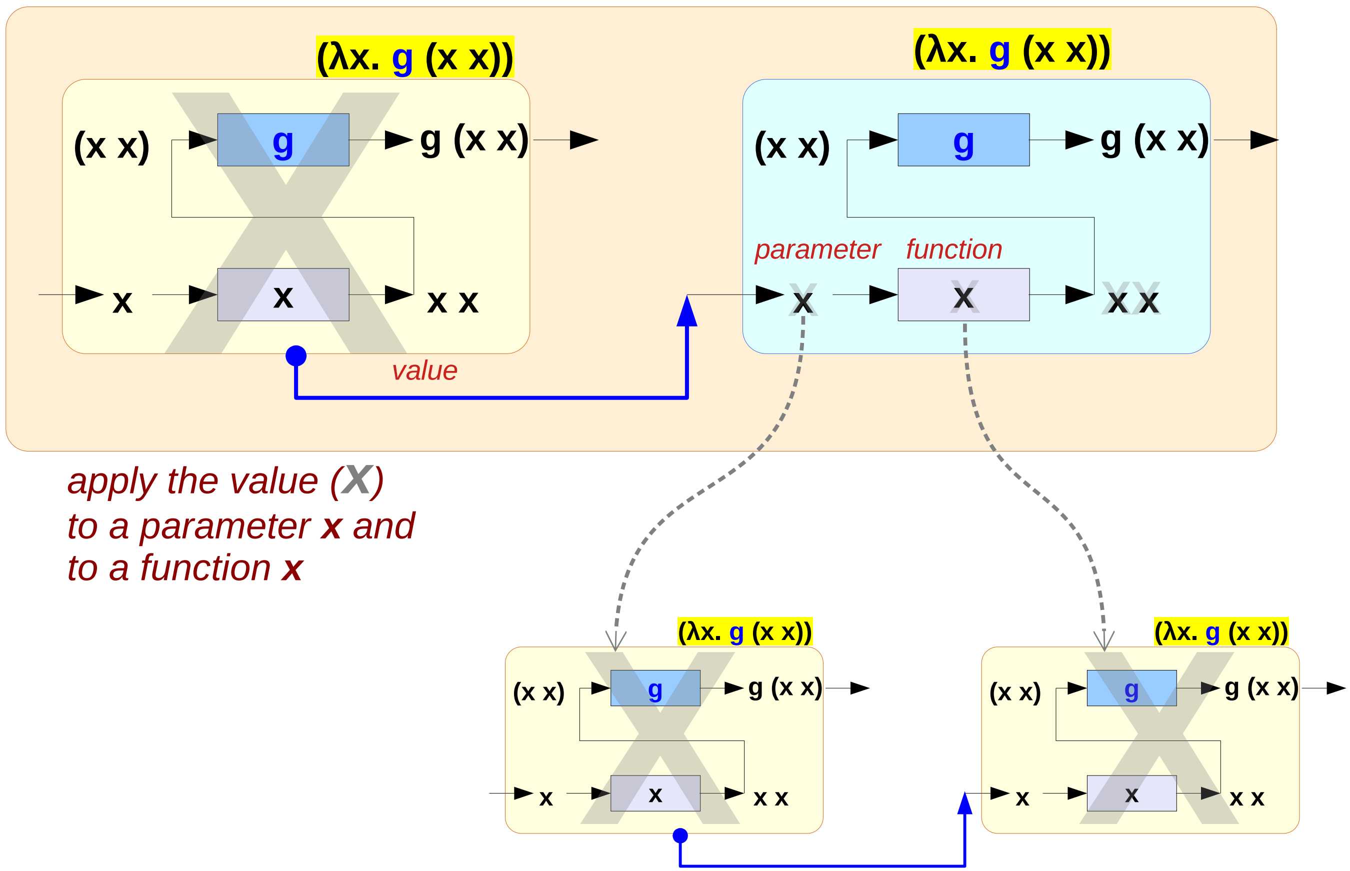
$Y\ g = (\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))$

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

Fixed-point combinator (9)

$\mathbf{Y}\ g = (\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))$

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



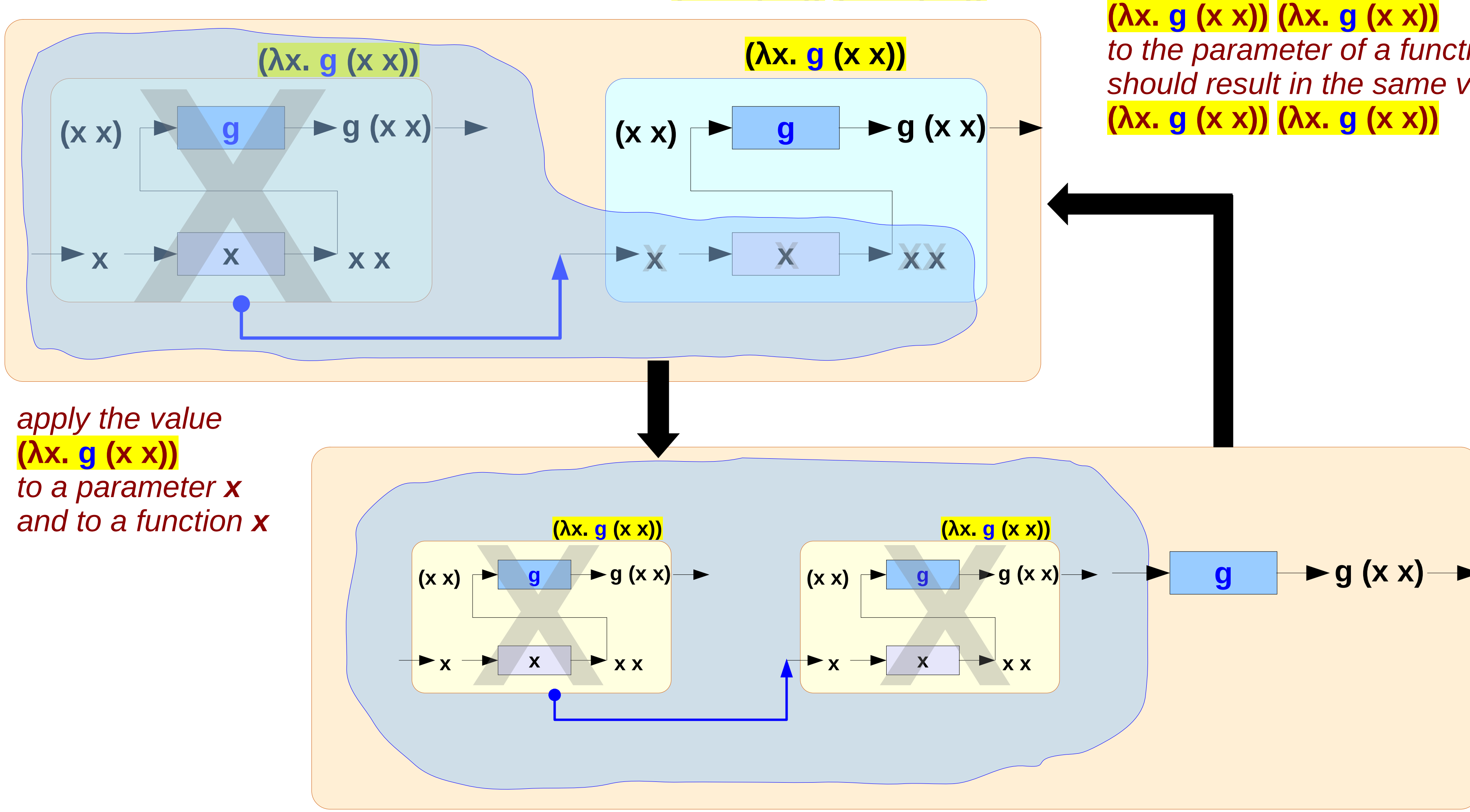
*apply the value (**X**)*
*to a parameter **x** and*
*to a function **x***

Fixed-point combinator (10)

Y g = (λx.g (x x)) (λx.g (x x))

Y = λf. (λx. f (x x)) (λx. f (x x))

Applying the value
(λx. g (x x)) (λx. g (x x))
to the parameter of a function **g**
should result in the same value
(λx. g (x x)) (λx. g (x x))



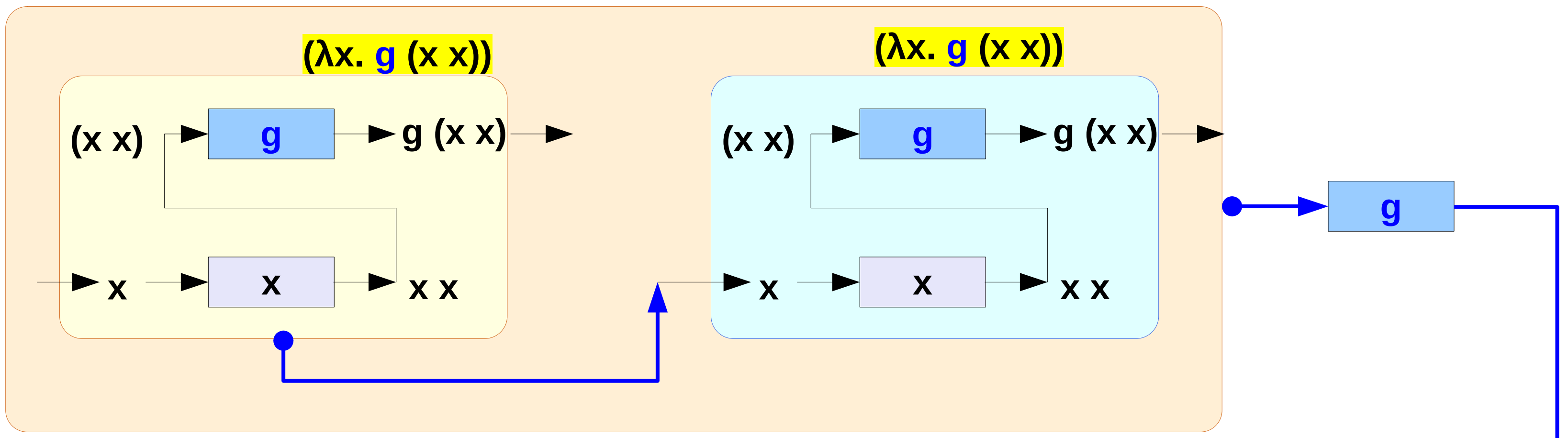
apply the value
(λx. g (x x))
to a parameter ***x***
and to a function ***x***

g (Y g) = g ((λx. g (x x)) (λx. g (x x)))

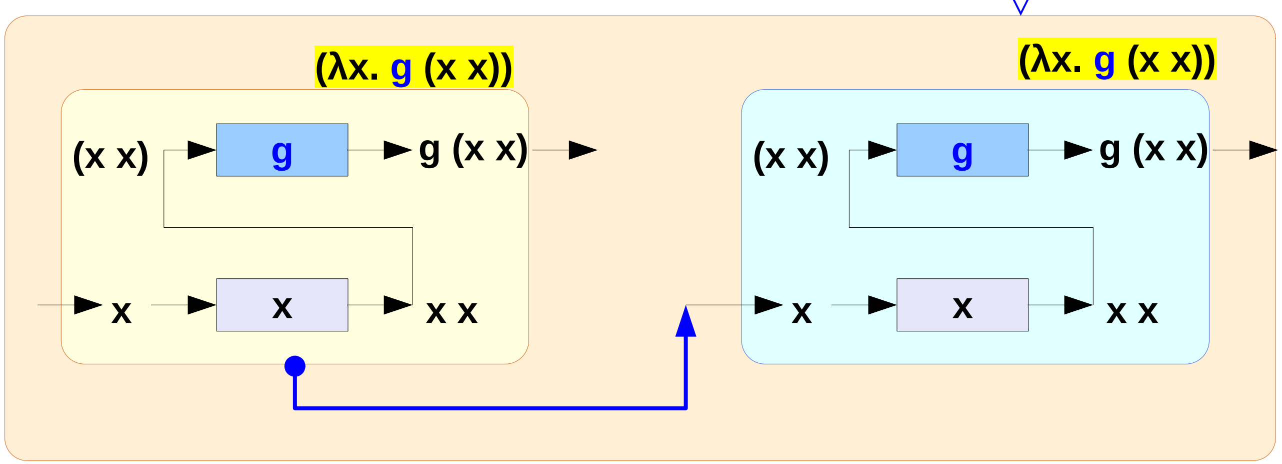
Fixed-point combinator (11)

Y g = (λx.g (x x)) (λx.g (x x))

Y = λf. (λx. f (x x)) (λx. f (x x))



Y g = (λx.g (x x)) (λx.g (x x)) ⬅ g (Y g) = g ((λx. g (x x)) (λx. g (x x)))



Fixed-point combinator (12)

The following calculation verifies that **Y g** is indeed a fixed point of the function **g** :

Y g	**= (λf. (λx.f (x x)) (λx. f (x x))) g**	by the definition of **Y**
	= g (Y g)	by second equality, above

The lambda term **g (Y g)** may <u>not</u>,

in general, β-reduce to the term **(Y g)** .

However, both terms β-reduce to the same term, as shown.

$$\mathbf{Y\ g} \xrightarrow{} \mathbf{g\ (Y\ g)}$$

Y g fixed point **g (Y g) = Y g**

Y g = **(λx. g (x x)) (λx. g (x x))**

g (Y g) = **g ((λx. g (x x)) (λx. g (x x)))**

Y g = g (Y g)

fix f = f (fix f)

fix F = F (fix F)

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (13)

This combinator may be used in implementing **Curry's paradox**.

The heart of Curry's paradox is
that untyped lambda calculus is unsound as a deductive system,

and the **Y combinator** demonstrates this
by allowing an anonymous expression
to represent zero, or even many **values**.

This is inconsistent in mathematical logic.

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (14)

Applied to a function with one variable,
the **Y combinator** usually does not terminate.

More interesting results are obtained
by applying the **Y combinator** to functions of two or more variables.

the additional variables may be used as a counter, or index.

the resulting function behaves like a **while** or a **for** loop
in an imperative language.

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

$$\mathbf{Y}\ g = g\ (\mathbf{Y}\ g)$$

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (15)

Used in this way, the **Y combinator** implements simple recursion.

The lambda calculus does not allow
a function to appear as a term in its own definition
as is possible in many programming languages,

but a function can be passed as an argument
to a higher-order function that applies it in a recursive manner.

Y = (λf. (λx.f (x x)) (λx. f (x x)))

Y g = g (Y g)

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (16)

Every recursively defined function can be seen
as a fixed point of some suitably defined function
closing over the recursive call with an extra argument,

and therefore, using **Y**, every recursively defined function
can be expressed as a lambda expression.

In particular, we can now cleanly define
the subtraction, multiplication and comparison predicate
of natural numbers recursively.

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (17)

Applied to a function with one variable,

the **Y combinator** usually does not terminate.

More interesting results are obtained

by applying the **Y combinator** to functions of two or more variables.

The additional variables may be used as a counter, or index.

The resulting function behaves like a **while** or a **for** loop

in an imperative language.

Used in this way, the **Y combinator** implements simple recursion.

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (18)

In the lambda calculus, it is not possible
to refer to the definition of a function
inside its own body by name.

Recursion though may be achieved
by obtaining the same function passed in as an argument,
and then using that argument to make the recursive call,
instead of using the function's own name,
as is done in languages which do support recursion natively.

The **Y combinator** demonstrates this style of programming.

https://en.wikipedia.org/wiki/Fixed-point_combinator

Fixed-point combinator (19)

An example implementation of **Y combinator** in two languages is presented below.

```
# Y Combinator in Python

Y=lambda f: (lambda x: f(x(x)))(lambda x: f(x(x)))

Y(Y)
```

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (1)

The **factorial function** provides a good example of how a fixed-point combinator may be used to define recursive functions.

The standard recursive definition of the factorial function in mathematics can be written as

$$\textbf{fact n} = \begin{cases} \mathbf{1} & \textbf{if} \quad \mathbf{n = 0} \\ \mathbf{n \cdot fact\ (n\text{-}1)} & \textbf{otherwise.} \end{cases}$$

where **n** is a non-negative integer.

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (2)

If we want to implement this in lambda calculus,

- integers are represented using Church encoding,

the problem is that the lambda calculus

does not allow the name of a function ('**fact**')

to be used in the function's definition.

this problem can be circumvented

using a fixed-point combinator **fix** as follows.

fix f = **f** (**fix f**),

fix f fixed point

fix fixed point combinator

Y g = **g** (**Y g**)

Y = (λ**f**. (λx.**f** (x x)) (λx. **f** (x x)))

Y g = (λx.**g** (x x)) (λx.**g** (x x))

= **g** ((λx. **g** (x x)) (λx. **g** (x x)))

= **g** (**Y g**)

fixed point

Y g = **g** (**Y g**)

fix f = **f** (**fix f**)

fix F = **F** (**fix F**)

Y combinator + argument function

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (3)

using a fixed-point combinator **fix**

fix f = f (fix f)

fix F = F (fix F),

Let the fixed point of **F, (fix F)** as **fact**

fact ≡ **fix F**

(fix F) = F (fix F)

(fact) = F (fact) fixed-point **fact**

(fact n) = F (fact n)

fix F = F (fix F),

fix F fixed point

fix fixed point combinator

fact n = F fact n

= (IsZero n) 1

(multiply n (fact (pred n)))

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (4)

a fixed-point combinator **fix**

fix F = **F** (**fix F**),

the fixed point of **F**, **(fix F)** as **fact**

(fact n) = F (fact n)

define a function **F** of two arguments **f** and **n**:

F f n = (IsZero n) 1 (multiply n (f (pred n)))

F fact n = (IsZero n) 1 (multiply n (fact (pred n)))

fix F = F (fix F),

fix F fixed point

fix fixed point combinator

fact n = F fact n

= (IsZero n) 1

(multiply n (fact (pred n)))

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (5)

(fact n) = F (fact n)

F f n ≡ (IsZero n) 1 (multiply n (f (pred n)))

the definition of the function F

F fact n ≡ (IsZero n) 1 (multiply n (fact (pred n)))

the recursive relation of fact

fact n = (IsZero n) 1 (multiply n (fact (pred n)))

n * fact (n-1)

fix F = F (fix F)

fix fixed point combinator ➡ Y

(λf. (λx.f (x x)) (λx. f (x x)))

fix F fixed point ➡ fact

((λx.F (x x)) (λx. F (x x)))

fixed point

Y g = g (Y g)

fix f = f (fix f)

fix F = F (fix F)

Y combinator + argument function

https://en.wikipedia.org/wiki/Fixed-point_combinator

The factorial function (6)

fact n = F fact n
= (IsZero n) 1 (multiply n (fact (pred n)))

here **(IsZero n)** is a function
that takes two arguments **1 (multiply n (fact (pred n)))**
and returns
its first argument **1** if **n=0**,
otherwise its second argument **(multiply n (f (pred n)))**

pred n evaluates to **n-1**

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

https://en.wikipedia.org/wiki/Fixed-point_combinator

Recursion (1)

recursion:

the definition of a function using the function itself.

A function definition containing itself inside itself, by value,

leads to the whole value being of infinite size.

Other notations which support recursion natively overcome this

by referring to the function definition by name.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (2)

Lambda calculus cannot express this
(referring to the function definition by name for recursion)

all functions are anonymous in lambda calculus,
so we can't refer by name to a value
which is yet to be defined,
inside the lambda term defining that same value.

in lambda calculus

- all functions are anonymous

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (3)

however, a lambda expression can <u>receive</u> itself

as its own argument, for example in **(λx.x x) E**.

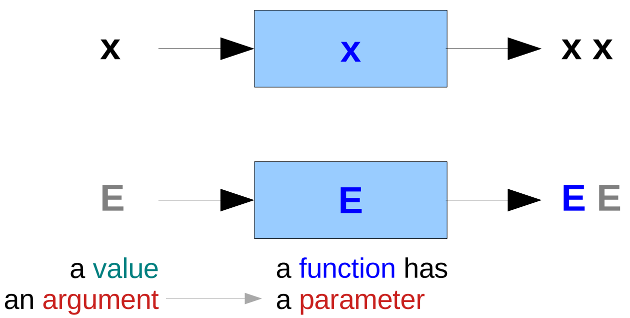
Here **E** should be an abstraction,

<u>applying</u> its parameter to a value to express recursion.

in lambda calculus

- all functions are anonymous
- but a function can receive itself

  as an argument



https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (4)

Consider the factorial function **fact(n)** recursively defined by

fact(n) = 1, if n = 0;   else n * fact(n-1).

in the lambda expression

which is to represent the function **fact(n)**,

typically, the first parameter will be assumed

to receive the lambda expression itself as its value,

so that calling it (applying it to an argument)

will amount to recursion.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (5)

Thus to achieve *recursion*,

the *intended-as-self-referencing* argument

(called r here) must always be <u>passed</u> to itself   r r

within the function body, at a call point: r r (n−1)

G := λr. λn. (1, if n = 0;   else n × (r r (n−1)))

with **r r n = F n = G r n** to hold,

so **r = G** and

F := G G = (λx.x x) G

fix G = G (fix G)

fix G fixed point **fact**

fix fixed point combinator

r r n =
F n =
G r n

r r n =
F n =
G G n

r = G

F = G G = r r

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (6) – with a self-referencing argument

G is a recursive factorial function

G := λr. λn. (1, if n = 0;   else n × (r r (n−1)))

G must have two arguments   **r n**   **G r n**

in the body of **G**, self-referencing argument **r**
must always be passed to **r**, for recursion   **r r n**

F is the top level **fact** function with one argument   **n**   **F n**

with **G r n = r r n = F n** to hold   **r = G**

F := G G = (λx.x x) G

G r n =	**G G n =**
r r n =	**r r n =**
F n	**F n**

r = G

F = G G = r r

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (7)

G := λr. λn. (1, if n = 0; else n × (r r (n−1)))

with r r n = F n = G r n to hold, so r = G and

F := G G = (λx.x x) G

fact n = F fact n

= (IsZero n) 1 (multiply n (fact (pred n)))

r r n = G r n

= (1, if n=0; else n × (r r (n-1)))

fact fact n = G fact n

= (1, if n=0; else n × (fact fact (n-1)))

fix G = G (fix G)

fix G fixed point fact

fix fixed point combinator

r r n =	r r n =
F n =	F n =
G r n	G G n

r = G

F = G G = r r

r = G = fact

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (8)

G := λr. λn. (1, if n = 0;   else n × (r r (n−1)))

with r r n = F n = G r n to hold, so r = G and

F := G G = (λx.x x) G

r r n = G r n

= (1, if n=0; else n × (r r (n-1)))

r = G = fact

fact fact n = G fact n = G G n

= (1, if n=0; else n × (fact fact (n-1)))

F n = G G n

= (1, if n=0; else n × (F (n-1)))

fact n = F fact n

= (IsZero n) 1 (multiply n (fact (pred n)))

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

fix G = G (fix G)

fix G fixed point fact

fix fixed point combinator

r r n =
F n =
G r n

r r n =
F n =
G G n

r = G = fact

F = G G = r r = fact fact

Recursion (9)

The self-application achieves replication here,

passing the function's lambda expression

on to the next invocation as an argument value, **r r**

making it available to be referenced and called there.

G := λr. λn. (1, if n = 0; else n × (r r (n−1)))

with **r r n = F n = G r n** to hold, so **r = G**

fact fact n = G fact n = G G n

= (1, if n=0; else n × (fact fact (n-1)))

This solves it but requires re-writing

each recursive call as self-application. **r r (n−1)**

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (10)

G := λr. λn. (1, if n = 0; else n × ($\boxed{\text{r r}}$ (n−1)))

with $\boxed{\text{r r}}$ n = **F n = G r n** to hold, so **r = G**

We would like to have a generic solution,

without a need for any re-writes:

G := λr. λn. (1, if n = 0; else n × ($\boxed{\text{r}}$ (n−1)))

with $\boxed{\text{r}}$ **n = F n = G r n** to hold, so **r = G r =: FIX G** and

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

r r n =
F n =
G r n

r r n =
F n =
G G n

r = G

F = G G = r r

r n =
F n =
G r n

r n =
F n =
G r n

Recursion (11) without a self-referencing argument

G := λr. λn. (1, if n = 0; else n × (r (n−1)))

with r n = F n = G r n to hold, so r = G r =: FIX G and

FIX G = G (FIX G)

FIX fixed point combinator

Y = (λf. (λx. f (x x)) (λx. f (x x)))

FIX G fixed point

fact = ((λx. G (x x)) (λx. G (x x)))

r n =
F n =
G r n

r n =
F n =
G r n

r = G r

F = G r = r

FIX G = G (FIX G)

r = FIX G
fixed point

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (12) without a self-referencing argument

G is a recursive factorial function

G := λr. λn. (1, if n = 0; else n × (r (n−1)))

G must have two arguments	**r n**	**G r n**
in the body of **G**, no self-referencing argument	**r**	**r n**
F is the top level function with a single argument	**n**	**F n**

with **r n = F n = G r n** to hold — **r = G r**

G r n =	**G r n =**
r n =	**r n =**
F n	**G r n**

r = G r

F = G r

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (13)

G := λr. λn. (1, if n = 0; else n × (r (n−1)))

with r n = F n = G r n to hold, so r = G r =: FIX G

Let r = FIX G,

r	= G r	r	= g r
(FIX G)	= G (FIX G)	(FIX g)	= g (FIX g)

F := FIX G where FIX g := (r where r = g r) = g (FIX g)

F = r = G r = FIX G

F = G F

(FIX G) = G (FIX G)

G r n =
r n =
F n

G r n =
r n =
G r n

r = G r
F = G r

FIX G = G (FIX G)

FIX fixed point combinator

Y = (λf. (λx.f (x x)) (λx. f (x x)))

FIX G fixed point

fact = ((λx.G (x x)) (λx. G (x x)))

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (14)

G := λr. λn. (1, if n = 0; else n × (r (n−1)))

with r x = F x = G r x to hold, so r = G r =: FIX G and

Let r = FIX G, (thus F = FIX G)

(FIX G) = G (FIX G)

r = G r

F = G F

(FIX g) = G (FIX g)

r = g r

F := FIX G where FIX g := (r where r = g r) = g (FIX g)

FIX G = G (FIX G) = (λn. (1, if n = 0; else n × ((FIX G) (n−1))))

r x =	r x =
F x =	F x =
G r x	G r x

r = G r

F = G r = r

fix F = F (fix F)

fix F fixed point fact

fix fixed point combinator

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (15)

FIX G = G (FIX G) = (λn. (1, if n = 0; else n × ((FIX G) (n−1))))

Given a lambda term with first argument
representing recursive call (e.g. **G** here),
the fixed-point combinator **FIX** will return
a self-replicating lambda expression
representing the recursive function (here, **F**).

The function does not need to be
explicitly passed to itself at any point,
for the self-replication is arranged in advance,
when it is created, to be done each time it is called.

FIX F = F (FIX F),

FIX F fixed point
FIX fixed point combinator

FIX F = F (FIX F) = fact
(fact) = F (fact)
(fact n) = F (fact n)

F f n = (IsZero n) 1
(multiply n (f (pred n)))

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (16)

Thus the original lambda expression (**FIX G**) is
re-created inside itself, at call-point, achieving self-reference.

In fact, there are many possible definitions for this **FIX** operator,
the simplest of them being:

$$\mathbf{Y := \lambda g.(\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))}$$

$$\mathbf{Y\ g = (\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))}$$
$$\mathbf{= g\ (\lambda x.\ (x\ x))\ (\lambda x.g\ (x\ x))}$$

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (17)

In the lambda calculus, **Y g** is a fixed-point of **g**, as it expands to:

Y g

(λh.(λx.h (x x)) (λx.h (x x))) g

(λx.g (x x)) (λx.g (x x))

g ((λx.g (x x)) (λx.g (x x)))

g (Y g)

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (18)

Now, to perform our recursive call to the factorial function,
we would simply call **(Y G) n**, where **n** is the number
we are calculating the factorial of.

Given **n = 4**, for example, this gives:

(Y G) 4
G (Y G) 4
(λr.λn.(1, if n = 0; else n × (r (n−1)))) (Y G) 4
(λn.(1, if n = 0; else n × ((Y G) (n−1)))) 4
1, if 4 = 0; else 4 × ((Y G) (4−1))
4 × (G (Y G) (4−1))

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (19)

4 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (4−1))

4 × (1, if 3 = 0; else 3 × ((Y G) (3−1)))

4 × (3 × (G (Y G) (3−1)))

4 × (3 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (3−1)))

4 × (3 × (1, if 2 = 0; else 2 × ((Y G) (2−1))))

4 × (3 × (2 × (G (Y G) (2−1))))

4 × (3 × (2 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (2−1))))

4 × (3 × (2 × (1, if 1 = 0; else 1 × ((Y G) (1−1)))))

4 × (3 × (2 × (1 × (G (Y G) (1−1)))))

4 × (3 × (2 × (1 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (1−1)))))

4 × (3 × (2 × (1 × (1, if 0 = 0; else 0 × ((Y G) (0−1))))))

4 × (3 × (2 × (1 × (1))))

24

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (20)

Every recursively defined function can be seen
as a fixed point of some suitably defined function
closing over the recursive call with an extra argument,
and therefore, using **Y**, every recursively defined function
can be expressed as a lambda expression.

In particular, we can now cleanly define the subtraction, multiplication
and comparison predicate of natural numbers recursively.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Encoding Conditionals (1)

consider how to encode a conditional expression of the form:

if P then A else B

i.e., the value of the whole expression is either **A** or **B**,

depending on the value of **P**

this conditional expression can be represented by

using a lambda expression as follows

COND P A B

where **COND**, **P**, **A** and **B** are all lambda expressions.

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Encoding Conditionals (2)

COND P A B

COND is a function of 3 arguments

that works by applying **P** to (**A** and **B**)

(i.e., **P** itself chooses **A** or **B**):

COND == λp.λa.λb.p a b

(where == means "is defined to be").

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Encoding Conditionals (3)

To make this definition work correctly,

we must define the representations of **true** and **false** carefully

since the lambda expression **P**

that **COND** applies to its arguments **A** and **B**

will reduce to either **TRUE** or **FALSE**

when **TRUE** is applied to **a** and **b** we want it to return **a** (first)

when **FALSE** is applied to **a** and **b** we want it to return **b**. (second)

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Encoding Conditionals (4)

let **TRUE** be a function of two arguments
that ignores the second argument
and returns the first argument,

let **FALSE** be a function of two arguments
that ignores the first argument
and returns the second argument:

TRUE == λx.λy.x
FALSE == λx.λy.y

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Encoding Conditionals (5)

COND TRUE M N

Note that this expression should evaluate to **M**.

substituting our definitions for **COND** and **TRUE**,

and evaluating the resulting expression

the sequence of beta-reductions is shown below

in each case, the redex about to be reduced is indicated

by *underlining* the formal parameter and

the argument that will be substituted in for that parameter. NO

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Encoding Conditionals (6)

(<u>λp</u>.λa.λb. p a b) (<u>λx.λy. x</u>) M N →β

(<u>λa</u>.λb. (λx.λy.x) a b) <u>M</u> N →β

(<u>λb</u>. (λx.λy.x) M b) <u>N</u> →β

(<u>λx</u>.λy. x) <u>M</u> N →β

(<u>λy</u>. M) <u>N</u> →β

M

https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.LAMBDA-CALCULUS-PART2.html#cond

Division (1-1)

Division of natural numbers may be implemented by,

$$n / m \;=\; \text{if}\;\; n \geq m \;\; \text{then} \;\; 1 + (n - m) / m$$
$$\text{else} \;\; 0$$

Calculating **n – m** takes many beta reductions.

Unless doing the reduction by hand,
this doesn't matter that much,
but it is preferable to not have to do
this calculation **(n – m)** twice.

9 / 3 = 1 + (9 – 3) / 3
= 1 + (1 + (6 – 3) / 3)
= 1 + (1 + (1 + 0 / 3))
= 1 + (1 +(1 + 0))

n / m = if n ≥ m then 1 + (n – m) / m
else 0

computing the condition **(n ≥ m)**
involves **(n – m)** calculation

https://en.wikipedia.org/wiki/Church_encoding

Division (1-2)

The simplest predicate for testing numbers is **IsZero**

so consider the condition.

IsZero (minus n m)

But this condition is equivalent to **n ≤ m**, not **n < m**.

minus n m = m pred n = 0 if n ≤ m

If this expression is used

then the mathematical definition of division given above

is translated into function on Church numerals as,

minus m n = n pred m

minus 4 3 = 3 pred 4
= (pred (pred (pred 4)))
= (pred (pred 3))
= (pred 2)
= 1

IsZero (minus 3 1) = 0	**3 > 1**	**2**
IsZero (minus 3 2) = 0	**3 > 2**	**1**
IsZero (minus 3 3) = 1	**3 = 3**	**0**
IsZero (minus 3 4) = 1	**3 < 4**	**0**
IsZero (minus 3 5) = 1	**3 < 5**	**0**

https://en.wikipedia.org/wiki/Church_encoding

Division (2-1)

n / m	=	if	n ≥ m	then	1 + (n − m) / m
				else	0
n / m	=	if	n < m	then	0
				else	1 + (n − m) / m
(n-1)/m	=	if	n ≤ m	then	0
				else	1 + (n − m) / m

If **IsZero** **(minus n m)** is used

a single call to **(minus n m)** is possible

but the result gives the value of **(n-1) / m**.

(minus n m) can be utilized

in computing **1 + (n − m) / m**

correct condition: **n < m**

modified condition: **n ≤ m**

https://en.wikipedia.org/wiki/Church_encoding

Division (2-2)

divide1 n m f x =

(λd. IsZero d (0 f x) (f (divide1 d m f x))) (minus n m)

d ← n − m

IsZero d ⟹ IsZero (minus n m)

TRUE ⟹ (λx.λy.x) (0 f x) (f (divide1 d m f x))

= (0 f x)

FALSE ⟹ (λx.λy.y) (0 f x) (f (divide1 d m f x))

= (f (divide1 d m f x))

(n-1)/m = if n ≤ m then 0

else 1 + (n − m) / m

https://en.wikipedia.org/wiki/Church_encoding

Division (2-3)

divide1 n m f x =
    (λd. IsZero d (0 f x) (f (divide1 d m f x)))   (minus n m)

divide1 9 3 f x
    = IsZero 6 (0 f x) (f (divide1 6 3 f x)) = (f (divide1 6 3 f x))
divide1 6 3 f x
    = IsZero 3 (0 f x) (f (divide1 3 3 f x)) = (f (divide1 3 3 f x))
divide1 3 3 f x
    = IsZero 0 (0 f x) (f (divide1 0 3 f x)) = (0 f x) = x

9 / 3 = 1 + (9 – 3) / 3
       = 1 + (1 + (6 – 3) / 3)
       = 1 + (1 + (1 + 0 / 3))
       = 1 + (1 +(1 + 0))

divide1 9 3 f x
       = (f (divide1 6 3 f x))

       = (f (f (divide1 3 3 f x)))

       = (f (f (0 f x)))

       = (f (f x))

https://en.wikipedia.org/wiki/Church_encoding

Division (3-1)

add **1** to **n** before calling **divide**.

divide n = divide1 (succ n)

divide1 10 3 f x

= IsZero 7 (0 f x) (f (divide1 7 3 f x)) = (f (divide1 7 3 f x))

divide1 7 3 f x

= IsZero 4 (0 f x) (f (divide1 4 3 f x)) = (f (divide1 4 3 f x))

divide1 4 3 f x

= IsZero 1 (0 f x) (f (divide1 1 3 f x)) = (f (divide1 1 3 f x))

divide1 1 3 f x

= IsZero 0 (0 f x) (f (divide1 1 3 f x)) = (0 f x) = x

divide1 9 3 f x

= (f (divide1 7 3 f x))

= (f (f (divide1 4 3 f x)))

= (f (f (f (divide1 1 3 f x))))

= (f (f (f x)))

https://en.wikipedia.org/wiki/Church_encoding

Division (3-2)

add **1** to **n** before calling **divide.**

divide n = divide1 (succ n)

divide1 is a recursive definition.

divide1 n m f x =

(λd. IsZero d (0 f x) (f (divide1 d m f x)))   (minus n m)

https://en.wikipedia.org/wiki/Church_encoding

Division (4)

The **Y combinator** may be used to implement the recursion.

Create a new function called **div** by;

In the left hand side **divide1** → **div c**

In the right hand side **divide1** → **c**

divide1 n m f x =

(λd. IsZero d (0 f x) (f (divide1 d m f x)) (minus n m)

div = λc. λn. λm. λf. λx.

(λd. IsZero d (0 f x) (f (c d m f x))) (minus n m)

div c = λn. λm. λf. λx.

(λd. IsZero d (0 f x) (f (c d m f x))) (minus n m)

https://en.wikipedia.org/wiki/Church_encoding

Division (5)

Then,

divide = λn. divide1 (succ n)

where,

divide1 = Y div succ = λn. λf. λx. f (n f x) Y

= λf. (λx. F (x x)) (λx. f (x x)) 0

= λf. λx. x IsZero

= λn. N (λx. False) true

true ≡ **λa. λb. a false** ≡ **λa. λb. b**

minus = λm. λn. n pred m pred

= λn. λf. λx. n (λg. λh. h (g f)) (λu. x) (λu. u)

https://en.wikipedia.org/wiki/Church_encoding

Division (6)

Gives,

```
divide =
    λn. ((λf. (λx. x x) (λx. f (x x)))
         (λc. λn. λm. λf. λx.
              (λd. (λn. n (λx. (λa. λb. b)) (λa. λb . a))
                        d ((λf. λx. x) f x) (f (c d m f x)))
              ((λm. λn. n (λn. λf. λx . n (λg. λh. h (g f))
                                   (λu. x) (λu. u)) m) n m)
         ))
        ((λn. λf. λx. f (n f x)) n)
```

Or as text, using \ for λ,

```
divide =
(\n.((\f.(\x.x x) (\x.f (x x)))
    (\c.\n.\m.\f.\x.
        (\d.(\n.n (\x.(\a.\b.b)) (\a.\b.a))
                d ((\f.\x.x) f x) (f (c d m f x)))
        ((\m.\n.n (\n.\f.\x.n (\g.\h.h (g f))
                        (\u.x) (\u.u)) m) n m)
    ))
    ((\n.\f.\x. f (n f x)) n))
```

https://en.wikipedia.org/wiki/Church_encoding

Division (6)

Gives,

divide = λn. ((λf. (λx. x x) (λx. f (x x))) (λc. λn. λm. λf. λx. (λd. (λn. n (λx. (λa. λb. b)) (λa. λb. a)) d ((λf. λx. x) f x) (f (c d m f x))) ((λm. λn. n (λn. λf. λx. n (λg. λh. h (g f)) (λu. x) (λu. u)) m) n m))) ((λn. λf. λx. f (n f x)) n)

Or as text, using \ for λ,

divide = (\n.((\f.(\x.x x) (\x.f (x x))) (\c.\n.\m.\f.\x.(\d.(\n.n (\x.(\a.\b.b)) (\a.\b.a)) d ((\f.\x.x) f x) (f (c d m f x))) ((\m.\n.n (\n.\f.\x.n (\g.\h.h (g f)) (\u.x) (\u.u)) m) n m))) ((\n.\f.\x. f (n f x)) n))

https://en.wikipedia.org/wiki/Church_encoding

Division (7)

For example, 9/3 is represented by

divide (\f.\x.f (f (f (f (f (f (f (f (f x))))))))) (\f.\x.f (f (f x)))

Using a lambda calculus calculator,

the above expression reduces to 3, using normal order.

(\f.\x.f (f (f (x))))

https://en.wikipedia.org/wiki/Church_encoding

Recursion (1-1)

recursion:

the definition of a function using the function itself.

A function definition containing itself inside itself, by value,

leads to the whole value being of infinite size.

Other notations which support recursion natively overcome this

by referring to the function definition by name.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (1-2)

Lambda calculus cannot express this:

all functions are anonymous in lambda calculus,

so we can't refer by name to a value which is yet to be defined,

inside the lambda term defining that same value.

however, a lambda expression can receive itself

as its own argument, for example in **(λx.x x) E**.

Here **E** should be an abstraction,

applying its parameter to a value to express recursion.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (1-3)

Consider the factorial function **F(n)** recursively defined by

F(n) = 1, if n = 0;   else n * F(n-1).

In the lambda expression which is to represent the function **F(n)**,
a parameter (typically the first one) will be assumed
to receive the lambda expression itself as its value,
so that calling it - applying it to an argument
will amount to recursion.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (2-1)

Thus to achieve recursion,

the *intended-as-self-referencing* argument

(called **r** here) must always be passed to itself

within the function body, at a call point:

G := λr. λn. (1, if n = 0;   else n × (r r (n−1)))

with **r r x = F x = G r x** to hold,

so **r = G** and

F := G G = (λx.x x) G

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (2-2)

F(n) = 1, if n = 0; else n × F(n − 1).

G := λr. λn.(1, if n = 0; else n × (r r (n−1)))

with **r r x = F x = G r x** to hold, so **r = G** and

F := G G = (λx.x x) G

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (3-1)

The self-application achieves replication here,
passing the function's lambda expression
on to the next invocation as an argument value,
making it available to be referenced and called there.

This solves it but requires re-writing
each recursive call as self-application.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (3-2)

We would like to have a generic solution,

<u>without</u> a need for any <u>re-writes</u>:

G := λr. λn.(1, if n = 0; else n × (r (n−1)))

with **r x = F x = G r x** to hold, so **r = G r =: FIX G** and

F := FIX G where **FIX g :=** (**r** where **r = g r**) **= g (FIX g)**

so that

FIX G = G (FIX G) = (λn.(1, if n = 0; else n × ((FIX G) (n−1))))

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (4)

Given a lambda term with first argument
representing recursive call (e.g. **G** here),
the fixed-point combinator **FIX** will return
a self-replicating lambda expression
representing the recursive function (here, **F**).

The function does not need to be
explicitly passed to itself at any point,
for the self-replication is arranged in advance,
when it is created, to be done each time it is called.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (5)

Thus the original lambda expression (**FIX G**) is
re-created inside itself, at call-point, achieving self-reference.

In fact, there are many possible definitions for this **FIX** operator,
the simplest of them being:

$$\mathbf{Y := \lambda g.(\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))}$$

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

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (6)

In the lambda calculus, **Y g** is a fixed-point of **g**, as it expands to:

Y g

(λh.(λx.h (x x)) (λx.h (x x))) g

(λx.g (x x)) (λx.g (x x))

g ((λx.g (x x)) (λx.g (x x)))

g (Y g)

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (7)

Now, to perform our recursive call to the factorial function,
we would simply call **(Y G) n**, where **n** is the number
we are calculating the factorial of.

Given **n = 4**, for example, this gives:

(Y G) 4

G (Y G) 4

(λr.λn.(1, if n = 0; else n × (r (n−1)))) (Y G) 4

(λn.(1, if n = 0; else n × ((Y G) (n−1)))) 4

1, if 4 = 0; else 4 × ((Y G) (4−1))

4 × (G (Y G) (4−1))

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (8)

4 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (4−1))

4 × (1, if 3 = 0; else 3 × ((Y G) (3−1)))

4 × (3 × (G (Y G) (3−1)))

4 × (3 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (3−1)))

4 × (3 × (1, if 2 = 0; else 2 × ((Y G) (2−1))))

4 × (3 × (2 × (G (Y G) (2−1))))

4 × (3 × (2 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (2−1))))

4 × (3 × (2 × (1, if 1 = 0; else 1 × ((Y G) (1−1)))))

4 × (3 × (2 × (1 × (G (Y G) (1−1)))))

4 × (3 × (2 × (1 × ((λn.(1, if n = 0; else n × ((Y G) (n−1)))) (1−1)))))

4 × (3 × (2 × (1 × (1, if 0 = 0; else 0 × ((Y G) (0−1))))))

4 × (3 × (2 × (1 × (1))))

24

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

Recursion (9)

Every recursively defined function can be seen

as a fixed point of some suitably defined function

closing over the recursive call with an extra argument,

and therefore, using **Y**, every recursively defined function

can be expressed as a lambda expression.

In particular, we can now cleanly define the subtraction, multiplication

and comparison predicate of natural numbers recursively.

https://en.wikipedia.org/wiki/Lambda_calculus#Formal_definition

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf

[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf