

Operational Semantics

Computation rule:

$$(\lambda x.t_{12}) v_2 \longrightarrow [x \mapsto v_2]t_{12}$$
 (E-APPABS)

 $[x \mapsto v_2]t_{12}$ is "the term that results from substituting occurrences of x in t_{12} with v_2 ."

Operational Semantics

Computation rule:

 $(\lambda x.t_{12}) v_2 \longrightarrow [x \mapsto v_2]t_{12}$ (E-APPABS)

 $[x \mapsto v_2]t_{12}$ is "the term that results from substituting occurrences of x in t_{12} with v_2 ."

Congruence rules:

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ t_2 \longrightarrow t'_1 \ t_2}$$
(E-APP1)
$$\frac{t_2 \longrightarrow t'_2}{v_1 \ t_2 \longrightarrow v_1 \ t'_2}$$
(E-APP2)

Terminology

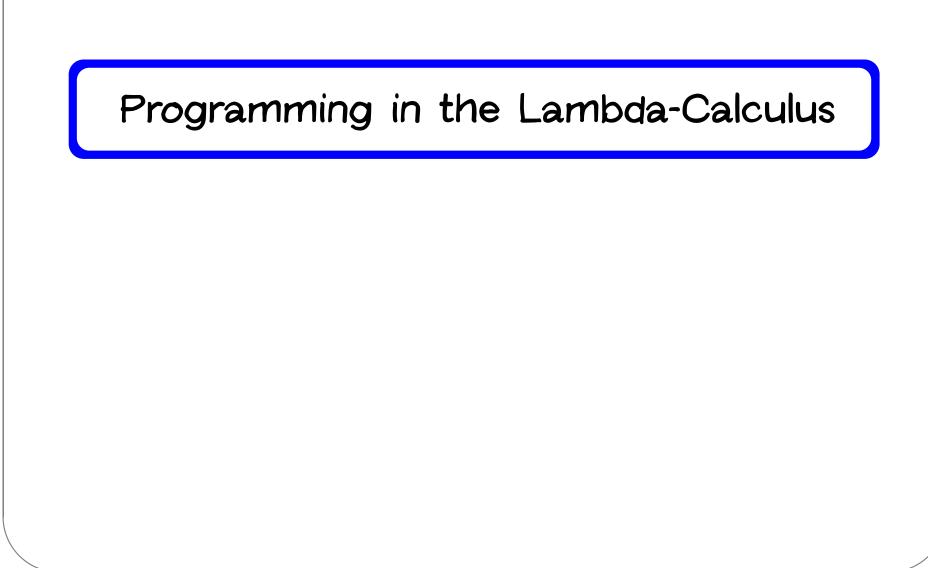
A term of the form $(\lambda x.t) v$ — that is, a λ -abstraction applied to a value — is called a redex (from "reducible expression").

Alternative evaluation strategies

The evaluation strategy we have chosen — called call by value — reflects standard conventions found in most mainstream languages.

Some other common ones:

- Full beta-reduction
- Normal order (leftmost/outermost)
- Call by name (cf. Haskell)



Multiple arguments

On Monday, we wrote a function double that returns a function as an argument.

```
double = \lambda f. \lambda y. f (f y)
```

This idiom — a λ -abstraction that does nothing but immediately yield another abstraction — is very common in the λ -calculus.

In general, λx . λy . t is a function that, given a value v for x, yields a function that, given a value u for y, yields t with v in place of x and u in place of y.

That is, λx . λy . t is a two-argument function.

Aside: Currying

The transformation from a function taking a pair of arguments (in a language like OCaml that provides pairs) to a one-argument function returning another one-argument function is called currying.

It is considered good style in OCaml to define functions in curried style whenever possible.

Syntactic conventions

Since λ -calculus provides only one-argument functions, all multi-argument functions must be written in curried style.

The following conventions make the linear forms of terms easier to read and write:

- Application associates to the left
- Bodies of λ abstractions extend as far to the right as possible

The "Church Booleans"

tru =	λ t. λ f. t		
fls =	λ t. λ f. f		
		tru v w	
	=	$(\lambda t.\lambda f.t) v w$	by definition
	\longrightarrow	<u>(λf. v) w</u>	reducing the underlined redex
	\longrightarrow	V	reducing the underlined redex
		fls v w	
	=	$(\lambda t.\lambda f.f) v w$	by definition
	\longrightarrow	<u>(λf. f) w</u>	reducing the underlined redex
	\longrightarrow	W	reducing the underlined redex

Functions on Booleans

not = λb . b fls tru

That is, not is a function that, given a boolean value v, returns fls if v is tru and tru if v is fls.

Functions on Booleans

and = λb . λc . b c fls

That is, and is a function that, given two boolean values v and w, returns w if v is tru and fls if v is fls

Thus and v w yields tru if both v and w are tru and fls if either v or w is fls.

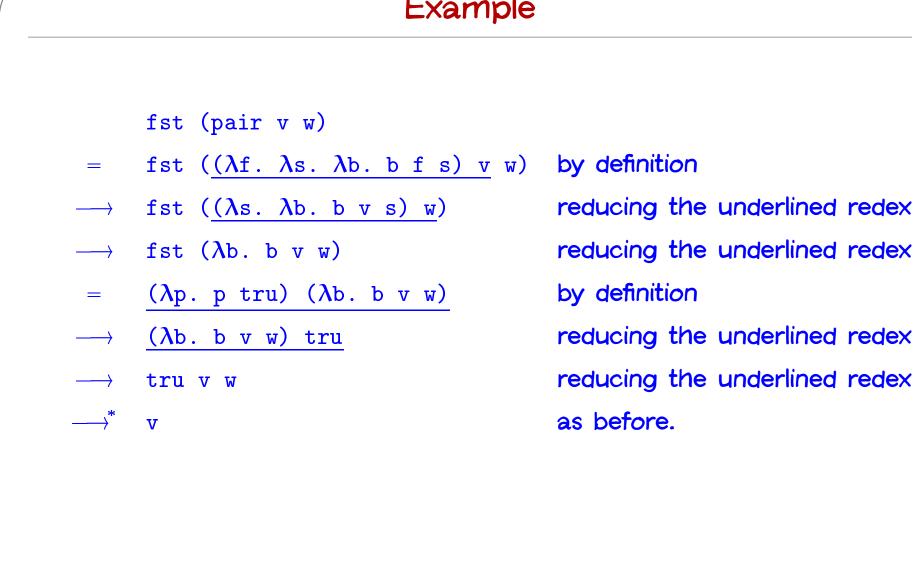
Pairs

```
pair = \lambda f. \lambda s. \lambda b. b f s
fst = \lambda p. p tru
snd = \lambda p. p fls
```

That is, pair v w is a function that, when applied to a boolean value b, applies b to v and w.

By the definition of booleans, this application yields v if b is tru and w if b is fls, so the first and second projection functions fst and snd can be implemented simply by supplying the appropriate boolean.

Example



Church numerals

Idea: represent the number ${\boldsymbol{n}}$ by a function that "repeats some action ${\boldsymbol{n}}$ times."

 $c_{0} = \lambda s. \lambda z. z$ $c_{1} = \lambda s. \lambda z. s z$ $c_{2} = \lambda s. \lambda z. s (s z)$ $c_{3} = \lambda s. \lambda z. s (s (s z))$

That is, each number n is represented by a term c_n that takes two arguments, s and z (for "successor" and "zero"), and applies s, n times, to z.

Successor:

Successor:

scc = λ n. λ s. λ z. s (n s z)

Successor:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$

Addition:

Successor:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$

Addition:

plus = λm . λn . λs . λz . m s (n s z)

Successor:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$

Addition:

plus = λm . λn . λs . λz . m s (n s z)

Multiplication:

Successor:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$

Addition:

plus = λm . λn . λs . λz . m s (n s z)

Multiplication:

times = λ m. λ n. m (plus n) c₀

Successor:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$

Addition:

plus = λm . λn . λs . λz . m s (n s z)

Multiplication:

times = λm . λn . m (plus n) c₀

Zero test:

Successor:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$

Addition:

plus = λm . λn . λs . λz . m s (n s z)

Multiplication:

times = λ m. λ n. m (plus n) c₀

Zero test:

iszro = λ m. m (λ x. fls) tru

Successor:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$

Addition:

plus = λm . λn . λs . λz . m s (n s z)

Multiplication:

times = λm . λn . m (plus n) c₀

Zero test:

iszro = λ m. m (λ x. fls) tru

What about predecessor?

CIS 500, 25 September

Predecessor

 $zz = pair c_0 c_0$ ss = λp . pair (snd p) (scc (snd p))

Predecessor

```
zz = pair c_0 c_0
ss = \lambda p. pair (snd p) (scc (snd p))
prd = \lambdam. fst (m ss zz)
```

Normal forms

A normal form is a term that cannot take an evaluation step.

A stuck term is a normal form that is not a value.

Are there any stuck terms in the pure λ -calculus?

Prove it.

Normal forms

A normal form is a term that cannot take an evaluation step.

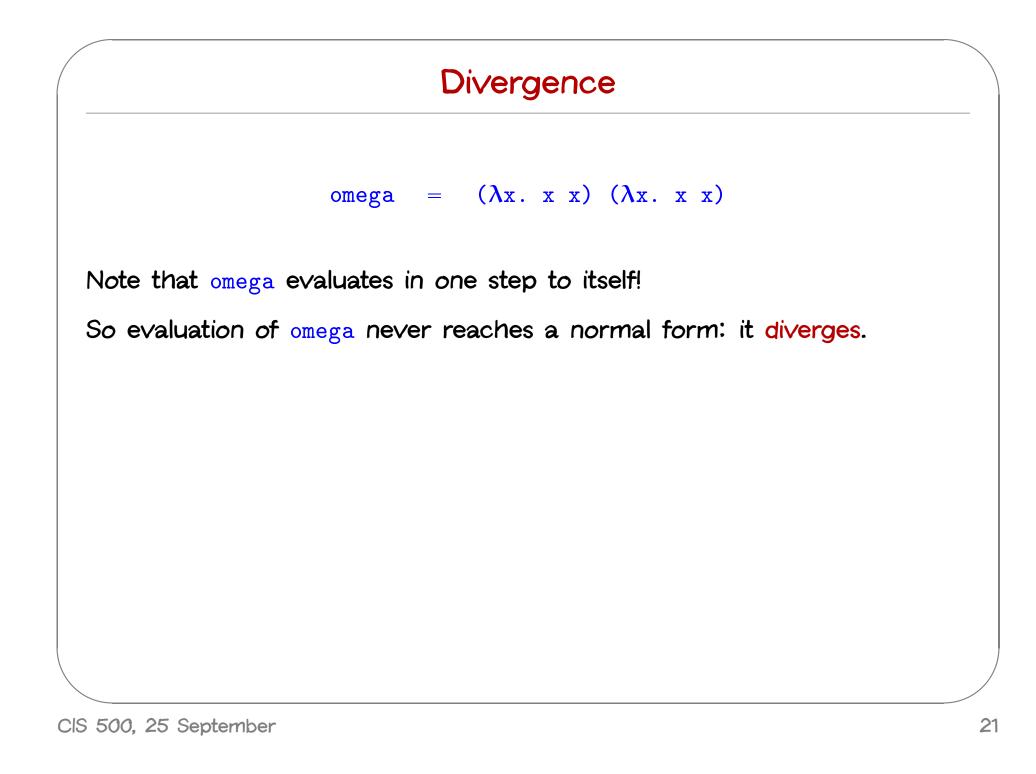
A stuck term is a normal form that is not a value.

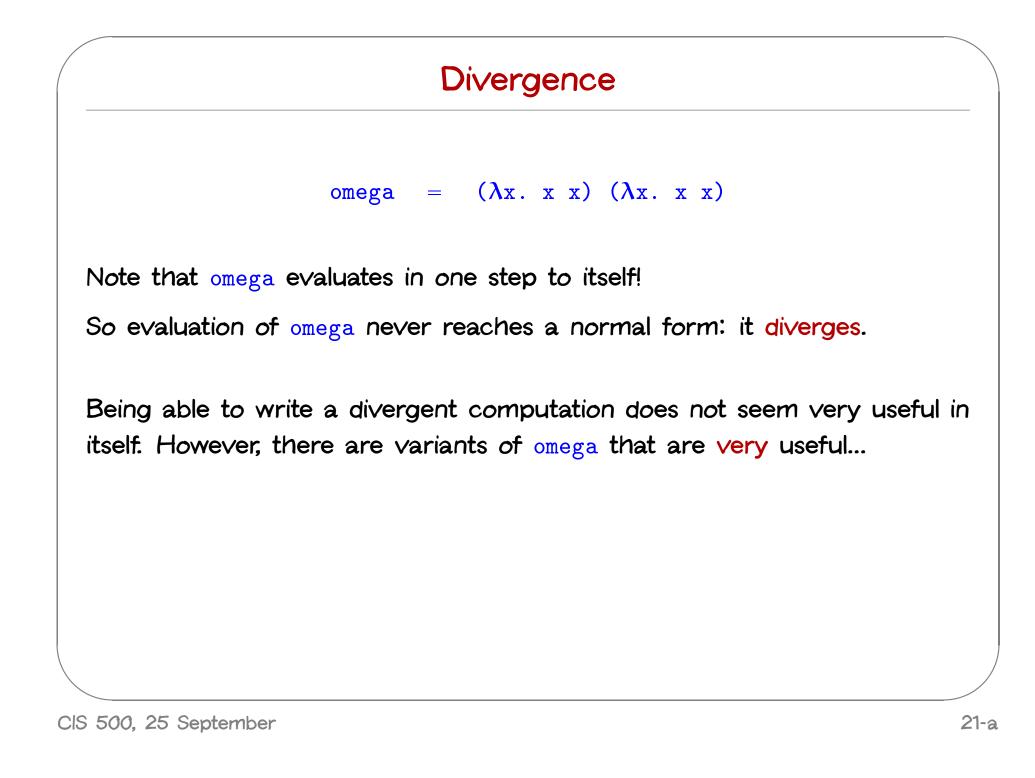
Are there any stuck terms in the pure λ -calculus?

Prove it.

Does every term evaluate to a normal form?

Prove it.





Iterated Application

Suppose f is some λ -abstraction, and consider the following term:

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

Iterated Application

Suppose f is some λ -abstraction, and consider the following term:

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

Now the "pattern of divergence" becomes more interesting:

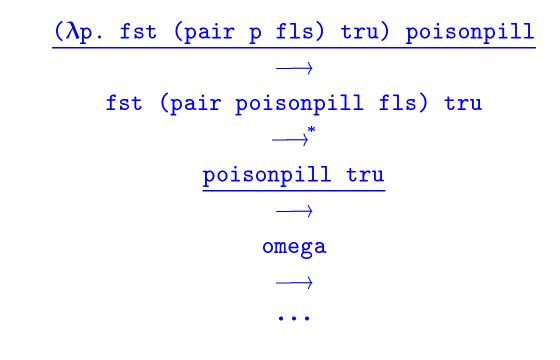
 Y_f is still not very useful, since (like omega), all it does is diverge.

Is there any way we could "slow it down"?



poisonpill = λy . omega

Note that poisonpill is a value — it it will only diverge when we actually apply it to an argument. This means that we can safely pass it as an argument to other functions, return it as a result from functions, etc.



A delayed variant of omega

Here is a variant of omega in which the delay and divergence are a bit more tightly intertwined:

omegav =
$$\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y$$

Note that omegav is a normal form. However, if we apply it to any argument v, it diverges:

omegav v

$$= \frac{(\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y) v}{\longrightarrow}$$

$$= \frac{(\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) v}{\longrightarrow}$$

$$(\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y) v$$

$$= 0$$
omegav v

Another delayed variant

Suppose f is a function. Define

 $Z_f \Rightarrow \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y$

This term combines the "added f" from Y_f with the "delayed divergence" of omegav.

If we now apply Z_f to an argument v, something interesting happens:

$$Z_{f} v$$

$$=$$

$$(\lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y) v$$

$$\longrightarrow$$

$$(\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) v$$

$$\longrightarrow$$

$$f (\lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y) v$$

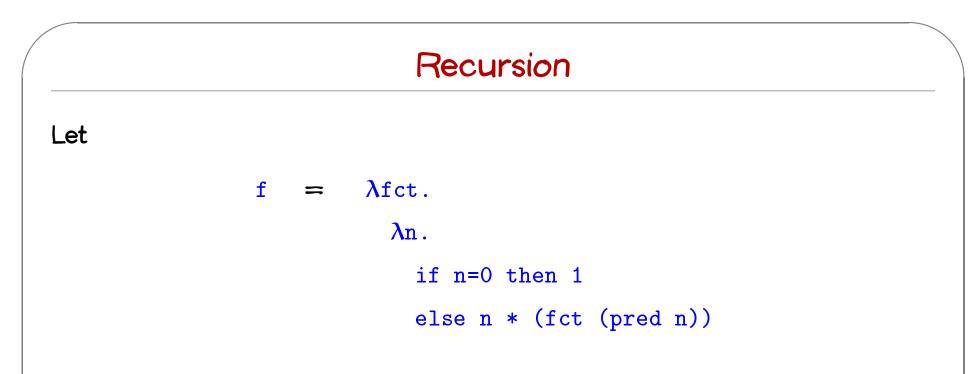
$$=$$

$$f Z_{f} v$$

Since Z_f and v are both values, the next computation step will be the reduction of $f Z_f$ — that is, before we "diverge," f gets to do some computation.

Now we are getting somewhere.

CIS 500, 25 September



f looks just the ordinary factorial function, except that, in place of a recursive call in the last time, it calls the function fct, which is passed as a parameter.

N.b.: for brevity, this example uses "real" numbers and booleans, infix syntax, etc...

We can use z to "tie the knot" in the definition of f and obtain a real recursive factorial function:

$$Z_{f} 3$$

$$\longrightarrow^{*}$$

$$f Z_{f} 3$$

$$=$$

$$(\lambda fct. \lambda n. ...) Z_{f} 3$$

$$\longrightarrow \longrightarrow$$
if 3=0 then 1 else 3 * (Z_{f} (pred 3)))
$$\longrightarrow^{*}$$

$$3 * (Z_{f} (pred 3)))$$

$$\longrightarrow$$

$$3 * (Z_{f} 2)$$

$$\longrightarrow^{*}$$

$$3 * (f Z_{f} 2)$$

$$\cdots$$

A Generic Z

If we define

$$Z = \lambda f \cdot Z_f$$

i.e.,

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

then we can obtain the behavior of Z_f for any f we like, simply by applying Z to f.

 $Z f \longrightarrow Z_f$

N.b.:

The term Z here is essentially the same as the fix discussed the book.

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

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

Z is hopefully slightly easier to understand, since it has the property that $Z f v \longrightarrow^* f (Z f) v$, which fix does not (quite) share.