Computability

COS 326

Presented by: Andrew W. Appel
Princeton University

FUNCTIONAL PROGRAMMING AS A MODEL OF COMPUTATION

Untyped lambda-calculus

$e ::= \lambda x.e_1 \mid x \mid e_1\ e_2$ $\lambda x.e_1$ *means same as* fun x -> e_1

big-step call-by-value evaluation

$$\frac{}{\lambda x.e \Downarrow \lambda x.e}$$

$$\frac{e1 \Downarrow \lambda x.e \qquad e2 \Downarrow v2 \qquad e[v2/x] \Downarrow v}{e1\ e2 \Downarrow v}$$

$$\frac{e1 \Downarrow \text{rec } f\ x = e \qquad e2 \Downarrow v2 \qquad e[\text{rec } f\ x = e/f][v2/x] \Downarrow v3}{e1\ e2 \Downarrow v3}$$

small-step general evaluation

$$\frac{}{(\lambda x.e1)\ e2 \dashrightarrow e1[e2/x]}$$

$$\frac{e1 \dashrightarrow e1'}{e1\ e2 \dashrightarrow e1'\ e2} \qquad \frac{e2 \dashrightarrow e2'}{e1\ e2 \dashrightarrow e1\ e2'}$$

$$\frac{e1 \dashrightarrow e1'}{\lambda x.e1 \dashrightarrow \lambda x.e1'}$$

Let's use small-step general evaluation for a while . . .

What can we program with just λ ?

(a,b)	$(\lambda x.xab)$	
pair	$(\lambda a.\lambda b.\lambda x.xab)$	pair a b $\approx$ (a,b)
fst	$(\lambda p.p(\lambda xy.x))$	
snd	$(\lambda p.p(\lambda xy.y))$	

fst(pair a b) = a

snd(pair a b) = b

fst (pair a b)

$= (\lambda p.p(\lambda xy.x))((\lambda a.\lambda b.\lambda x.xab)ab)$

$\rightarrow (\lambda p.p(\lambda xy.x))((\lambda b.\lambda x.xab)b)$

$\rightarrow (\lambda p.p(\lambda xy.x))(\lambda x.xab)$

$\rightarrow (\lambda x.xab)(\lambda xy.x)$

$\rightarrow (\lambda xy.x)ab$

$\rightarrow (\lambda y.a)b$

$\rightarrow a$

Booleans

Henceforth, abbreviate: $\lambda xy.E$ means $\lambda x.\lambda y.E$

true	$(\lambda xy.x)$
false	$(\lambda xy.y)$
if	$(\lambda xab.xab)$

if true a b = a

if false a b = b

if true a b

= $(\lambda xab.xab)\ (\lambda xy.x)$ a b

--> $(\lambda ab.\ (\lambda xy.x)ab)$ a b

--> $(\lambda b.\ (\lambda xy.x)ab)$ b

--> $(\lambda xy.x)ab$

--> $(\lambda y.a)b$

--> a

Lists

nil	$(\lambda cn.n)$	nil ≈ []
cons	$(\lambda ht.\lambda cn.cht)$	cons h t ≈ h::t
match	$(\lambda acn.acn)$	match a c n ≈ match a with | h::t -> c h t | [] -> n

(match (cons x y) with
 | cons h t -> f h t
 | nil -> g)
 = f x y

match (cons x y) f g

$= (\lambda acn.acn)((\lambda ht.\lambda cn.cht)xy)fg$

$\to (\lambda acn.acn)(\lambda cn.cxy)fg$

$\to (\lambda cn. (\lambda cn.cxy)cn)\ fg$

$\to (\lambda n.fxy)g$

$\to fxy$

Lists (nil case)

nil	$(\lambda cn.n)$	nil ≈ []
cons	$(\lambda ht.\lambda cn.cht)$	cons h t ≈ h::t
match	$(\lambda acn.acn)$	match a c n ≈ match a with
		| h::t -> c h t
		| [] -> n

(match nil with
 | cons h t -> f h t
 | nil -> g)
  = g

match nil f g
= $(\lambda acn.acn)\ (\lambda cn.n)$ fg
--> $(\lambda cn.\ (\lambda cn.n)\ cn)$ fg
--> $(\lambda cn.n)$ fg
--> $(\lambda n.n)$ g
--> g

General inductive datatypes

type t = A of t1 | B of t2 | C | D

A	λx.λabcd.ax
B	λy.λabcd.by
C	λabcd.c
D	λabcd.d

match_t   λuabcd.uabcd

(match B z with A x -> a x | B y -> b y | C -> c | D -> d)

= b z

Integers

type int = O | S of int

add = (rec add a b -> match a with O -> b | S a’ -> S(add a’ b))

. . . if only we had recursive functions!

Can we infinite loop?

$e ::= \lambda x.e_1 \mid x \mid e_1\ e_2$

no recursive functions!   Can we infinite-loop without loops?

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

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

$\rightarrow (\lambda x.xx)\ (\lambda x.xx)$

That doesn't typecheck!
But who said anything about types, this is *untyped* lambda-calculus

Recursive functions

$Y \quad \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

$Yg = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))g$

$\to \quad (\lambda x.g(xx))(\lambda x.g(xx))$

$\to \quad g((\lambda x.g(xx))(\lambda x.g(xx)))$

$= \quad g(Yg)$

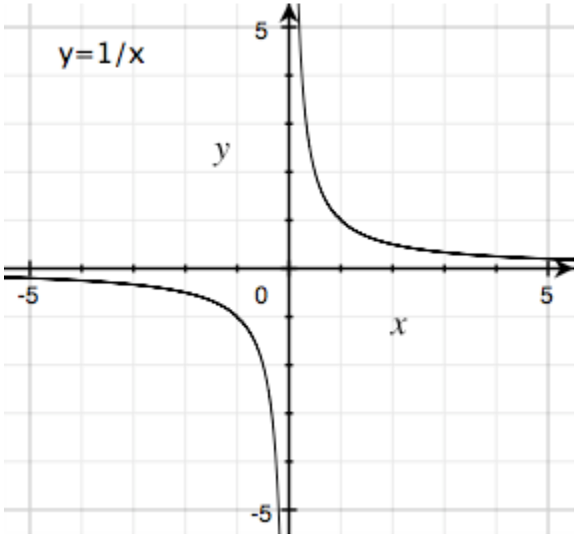
Fixed points

Let f(x)=1/x

Find a fixed point of f,
that is, a value z such that f(z)=z

Answer: -1

f(-1) = 1/(-1) = -1



Recursive functions

Y $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

Yg = $(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))g$

--> $(\lambda x.g(xx))(\lambda x.g(xx))$

--> $g((\lambda x.g(xx))(\lambda x.g(xx)))$

= $g(Yg)$

Yg is a fixed point of g, that is $g(Yg)=Yg$

Recursive add function

type int = O | S of int

add = (rec add a b -> match a with O -> b | S a’ -> S(add a’ b))

. . . if only we had recursive functions!

add = (rec f a b -> match a with O -> b | S a’ -> S(f a’ b))

add = λab.(rec f a -> match a with O -> b | S a’ -> S(f a’))

add = λab. Y(λf. λa. match a with O -> b | S a’ -> S(f a’))a

Theorem: for all b,   add 2 b = S(S b)

$$add = \lambda ab.\ Y\underbrace{(\lambda f.\ \lambda a.\ \text{match } a \text{ with } O \rightarrow b \mid S\ a' \rightarrow S(f\ a'\ b))}_{g}a$$

$add\ (S(SO))b$

$= (\lambda ab.\ Yga)(S(SO))b$

$= Yg(S(SO))b$

$= g(Yg)(S(SO))b$

$= \text{match } S(SO) \text{ with } O \rightarrow b \mid S\ a' \rightarrow S(Yga')$

$= S(Yg(SO)b)$

$= S(\text{match } SO \text{ with } O \rightarrow b \mid S\ a' \rightarrow S(Yga'))$

$= S(S(YgOb))$

$= S(S(\text{match } O \text{ with } O \rightarrow b \mid S\ a' \rightarrow S(Yga')))$

$= S(S\ b)$

Theorem: add 1 2 = 3

type int = O | S of int   O=λxy.x   S= λn.λxy.yn

add (SO) (S(SO)) -->*   S(S(SO))

--> (λn.λxy.yn) ((λn.λxy.yn)((λn.λxy.yn)(λxy.x)))

--> (λn.λxy.yn) ((λn.λxy.yn)(λxy.y(λxy.x)))

--> (λn.λxy.yn) (λxy.y(λxy.y(λxy.x)))

--> λxy.y(λxy.y(λxy.y(λxy.x)))

None of our small-step evaluation rules apply here, so this must be the “answer,” also called the “normal form” of add (SO) (S(SO)).

It is our *representation* of 3

$$\frac{}{(\lambda x.e1)\ e2 \ \text{-->}\ e1[e2/x]}$$

$$\frac{e1 \ \text{-->}\ e1'}{e1\ e2 \ \text{-->}\ e1'\ e2} \qquad \frac{e2 \ \text{-->}\ e2'}{e1\ e2 \ \text{-->}\ e1\ e2'}$$

$$\frac{e1 \ \text{-->}\ e1'}{\lambda x.e1 \ \text{-->}\ \lambda x.e1'}$$

Try it again: factorial

$g = \lambda f.\ \lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n\cdot f(n-1)$

$fact = Yg$

$fact\ 3 = Yg3$

$= g(Yg)3$

$= (\lambda f.\ \lambda n.\ \text{if } n=0 \text{ then } 1 \text{ else } n\cdot f(n-1))\ (Yg)\ 3$

$= \text{if } 3=0 \text{ then } 1 \text{ else } 3\cdot((Yg)(3-1))$

$= 3\cdot(Yg2)$

$= 3\cdot\ (g(Yg)2) = 3\cdot(\text{if } 2=0 \text{ then } 1 \text{ else } 2\cdot(Yg(2-1)))$

$= 3\cdot(2\cdot(Yg1)) = 3\cdot(2\cdot(g(Yg)1))$

$= 3\cdot(2\cdot(\text{if } 1=0 \text{ then } 1 \text{ else } 1\cdot(Yg(1-1)))) = 3\cdot(2\cdot(1\cdot Yg0))$

$= 3\cdot(2\cdot(1\cdot \text{if } 0=0 \text{ then } 1 \text{ else } 0\cdot(Yg(0-1)))) = 3\cdot(2\cdot(1\cdot 1)) = 6$

Now we have everything!

tuples, Booleans, if-statements, lists, integers,

inductive data types, recursive functions . . .

We can implement a substitution-based interpreter.

[paste in "Interpreter" lectures here . . .]

```
type var = int
type exp = Fun of var*exp | Var of var | App of exp*exp
```

Expressive power of untyped λ-calculus

Could you write in OCaml,

. . . a Turing machine simulator?

Then surely you could express in λ-calculus

. . . a Turing machine simulator!

Expressive power of untyped λ-calculus

Could you write in OCaml,

. . . a Turing machine simulator?

. . . an x86 instruction simulator?

Then surely you could express in λ-calculus

. . . a Turing machine simulator!

. . . an x86 instruction simulator!

Expressive power of untyped λ-calculus

Could you write in OCaml,

. . . a Turing machine simulator?

. . . an x86 instruction simulator?

. . . a simulation of *any* program that would run on your laptop computer?

Then surely you could express in λ-calculus

. . . a Turing machine simulator!

. . . an x86 instruction simulator!

. . . *any* program that would run on your laptop computer!

Summary

$$e ::= \lambda x.e_1 \mid x \mid e_1\, e_2$$

$$\frac{}{(\lambda x.e1)\ e2\ \text{-->}\ e1[e2/x]}$$

$$\frac{e1\ \text{-->}\ e1'}{e1\ e2\ \text{-->}\ e1'\ e2} \qquad \frac{e2\ \text{-->}\ e2'}{e1\ e2\ \text{-->}\ e1\ e2'}$$

$$\frac{e1\ \text{-->}\ e1'}{\lambda x.e1\ \text{-->}\ \lambda x.e1'}$$

The untyped lambda calculus, although an extremely *simple* model of computation, can *express* any function that's computable by any computer we know how to build.

Question

Q. Is there any mathematical function that the untyped lambda calculus *can't* express?

A. Yes. See the next lecture!