Computability

COS 326
Andrew W. Appel
Princeton University
FUNCTIONAL PROGRAMMING AS A MODEL OF COMPUTATION
Untyped lambda-calculus

\[ e ::= \lambda x.e_1 \mid x \mid e_1 e_2 \]  
\[ \lambda x.e_1 \quad \text{means same as} \quad \text{fun} \ x \rightarrow e_1 \]

---

**big-step call-by-value evaluation**

\[ \lambda x.e \downarrow \lambda x.e \]

\[ e_1 \downarrow \lambda x.e \quad \begin{array}{l} e_2 \downarrow v_2 \quad e[v_2/x] \downarrow v \end{array} \]

\[ e_1 e_2 \downarrow v \]

\[ e_1 \downarrow \text{rec} \ f \ x = e \quad e_2 \downarrow v_2 \quad e[\text{rec} \ f \ x = e/f][v_2/x] \downarrow v_3 \]

\[ e_1 e_2 \downarrow v_3 \]

---

**small-step general evaluation**

\[ (\lambda x.e_1) \ e_2 \rightarrow e_1[e_2/x] \]

\[ e_1 \rightarrow e_1' \quad e_2 \rightarrow e_2' \]

\[ e_1 e_2 \rightarrow e_1' e_2 \quad e_1 e_2 \rightarrow e_1 e_2' \]

\[ \lambda x.e_1 \rightarrow \lambda x.e_1' \]

---

Let’s use small-step general evaluation for a while . . .
What can we program with just $\lambda$?

(a,b) \quad (\lambda x.xab)

pair \quad (\lambda a.\lambda b.\lambda x.xab) \quad \text{pair a b} \approx (a,b)

fst \quad (\lambda p. p(\lambda xy.x))

snd \quad (\lambda p. p(\lambda xy.y))

fst(pair a b) = a

snd(pair a b) = b

\[
\text{fst (pair a b)} = (\lambda p. p(\lambda xy.x))((\lambda a.\lambda b.\lambda x.xab)ab) \\
\quad \rightarrow (\lambda p. p(\lambda xy.x))((\lambda b.\lambda x.xab)b) \\
\quad \rightarrow (\lambda p. p(\lambda xy.x))(\lambda x.xab) \\
\quad \rightarrow (\lambda x.xab)(\lambda xy.x) \\
\quad \rightarrow (\lambda y.a)b \\
\quad \rightarrow a
\]
Booleans

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

true $\quad (\lambda xy.x)$
false $\quad (\lambda xy.y)$

if $\quad (\lambda xab.xab)$

if true a b = a
if false a b = b

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

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

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

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

--> a
Lists

nil \ (\lambda cn.n) \quad \text{nil} \approx []
cons \ (\lambda ht.\lambda cn.cht) \quad \text{cons } h \ t \approx h::t
match \ (\lambda acn.acn) \quad \text{match } a \ c \ n \approx \text{match } a \text{ with}
\quad \mid h::t \to c \ h \ t
\quad \mid [] \to n

\left(\text{match } (\text{cons } x \ y) \text{ with}
\quad \mid \text{cons } h \ t \to f \ h \ t
\quad \mid \text{nil} \to g\right)
\quad = f \ x \ y
\left(\text{match } (\text{cons } x \ y) \ f \ g
\quad = (\lambda acn.acn)((\lambda ht.\lambda cn.cht)xy)fg
\quad \to (\lambda acn.acn)(\lambda cn.cxy)fg
\quad \to (\lambda cn. (\lambda cn.cxy)cn) \ fg
\quad \to (\lambda n. fxy)g
\quad \to fxy\right)
Lists (nil case)

| nil      | (λcn.n)  | nil ≈ []   |
| cons    | (λht.λcn.cht) | cons h t ≈ h::t |
| match   | (λacn.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
= (λacn.acn) (λcn.n) fg
--> (λcn. (λcn.n) cn) fg
--> (λcn.n) fg
--> (λn.n) g
--> g
type t = A of t1 | B of t2 | C | D

A \( \lambda x.\lambda abcd.ax \)
B \( \lambda y.\lambda abcd.by \)
C \( \lambda abcd.c \)
D \( \lambda abcd.d \)

match_t \( \lambda uababcd.uababcd \)

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

\[ e ::= \lambda x.e_1 | x | e_1 e_2 \]

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

\[ \Omega = (\lambda x.xx) (\lambda x.xx) \]
\[ \quad \rightarrow (\lambda x.xx) (\lambda x.xx) \]

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

\[ Y \quad \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)) \]

\[ Yg = (\lambda f. (\lambda x. f(xx))(\lambda x. f(xx)))g \]

\[ \rightarrow (\lambda x. g(xx))(\lambda x. g(xx)) \]

\[ \rightarrow g((\lambda x. g(xx))(\lambda x. g(xx))) \]

\[ = g(Yg) \]
Let \( f(x) = 1/x \)

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

Answer: \(-1\)

\[
\begin{align*}
f(-1) &= 1/(-1) = -1
\end{align*}
\]
Recursive functions

\[ Y = \lambda f. (\lambda x. f(xx))(\lambda x. f(xx)) \]

\[ Yg = (\lambda f. (\lambda x. f(xx))(\lambda x. f(xx)))g \]

\[ \rightarrow (\lambda x. g(xx))(\lambda x. g(xx)) \]

\[ \rightarrow g((\lambda x. g(xx))(\lambda x. g(xx))) \]

\[ = g(Yg) \]

\[ Yg \text{ is a fixed point of } g, \text{ that is } g(Yg) = Yg \]
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, \( \text{add} \ 2 \ b = S(S \ b) \)

\[
\text{add} = \lambda ab. \ Y(\lambda f. \ \lambda a. \ \text{match} \ a \ \text{with} \ O \to b \mid S \ a' \to S(f \ a' \ b))a
\]

\[
\text{add} \ (S(SO))b = (\lambda ab. \ Yga)(S(SO))b
\]

\[
= Yg(S(SO))
\]

\[
= g(Yg)(S(SO))
\]

\[
= \text{match} \ S(SO) \ \text{with} \ O \to b \mid S \ a' \to S(Yga')
\]

\[
= S(Yg(SO))
\]

\[
= S(\text{match} \ SO \ \text{with} \ O \to b \mid S \ a' \to S(Yga'))
\]

\[
= S(S(YgO))
\]

\[
= S(\text{S(match} \ O \ \text{with} \ O \to b \mid S \ a' \to S(Yga')))\]

\[
= S(S \ b)
\]
Theorem: add 1 2 = 3

type int = O | S of int

O = \lambda xy.x
S = \lambda n.\lambda xy.yn

add (SO) (S(SO)) --\* S(S(SO))
--\> (\lambda n.\lambda xy.yn)((\lambda n.\lambda xy.yn)((\lambda n.\lambda xy.yn)(\lambda xy.x)))
--\> (\lambda n.\lambda xy.yn)((\lambda n.\lambda xy.yn)(\lambda xy.y(\lambda xy.x)))
--\> (\lambda n.\lambda xy.yn)(\lambda xy.y(\lambda xy.y(\lambda xy.x)))
--\> \lambda xy.y(\lambda xy.y(\lambda xy.y(\lambda xy.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
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

\begin{align*}
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
\end{align*}
Now we have everything!

tuples, Booleans, if-statements, lists, integers, induction data types, recursive functions . . .

We can implement a substitution-based interpreter.

[paste in lecture 6 here . . . ]

type var = int
type exp = Fun of var*exp | Var of var | App of exp*exp
Some meta-notation

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

We want to talk about the AST of a given term:
When $e$ is a $\lambda$-expression, $[e]$ is its representation in $\text{exp}$

$[x_i] = \text{Var } i$
$[e_1 e_2] = \text{App }[e_1][e_2]$
$[\lambda x_i.e_1] = \text{Fun } i [e_1]$
type var = int
type exp = Fun of var*exp | Var of var | App of exp*exp

1. Write a λ-function \texttt{interp} such that

For any expression \( e \)

that evaluates in \( \lambda \)-calculus to a normal form \( e' \),

(that is, \( e \rightarrow* e' \) and \( e' \) cannot take a step)

\[
\texttt{interp} [e] \rightarrow* [e']
\]

(Yes, this is just a version of the substitution-based interpreter from lecture 6, and homework 4)
What can we compute?

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

2. Write a quoting function such that \( kwoht \ e = [e] \)

Impossible:

Consider \( e_1 = (\lambda x.x)y \) and \( e_2 = y \)
\[
kwoht e_1 = kwoht ((\lambda x.x)y) = kwoht y = kwoht e_2
\]
\[
[e_1] = App(Fun(i,Var \ i),Var \ j)
\]
\[
[e_2] = Var \ j
\]
\[
[e_1] \neq [e_2]
\]
What can we compute?

```
type var = int

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

3. Write a quoting function such that \( \text{quote } [e] = [[[e]]] \)

Easy:

```
let rec quote e =
    match e with
    | Fun(i,e1) -> App (App [Fun] i) (quote e1)
    | Var i -> App [Var] i
    | App(e1,e2) -> App (App [App] (quote e1)) (quote e2)
```
What can we compute?

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

4. Write a \( \lambda \)-function \texttt{halts} such that

For any expression \( e \),
if \( e \rightarrow^* e' \) and \( e' \) cannot step, then \( \text{halts}[e] = \text{true} \)
if \( e \) infinite loops no matter which reductions you do,
then \( \text{halts}[e] = \text{false} \)

Claim: you cannot write such a function
```
What can we compute?

Proof by contradiction. Suppose there exists a \( \lambda \)-expression \texttt{halts} such that for any expression \( e \),

\[
\text{if } e \rightarrow^* e' \text{ and } e' \text{ cannot step, then } \texttt{halts}[e] = \text{true}
\]
\[
\text{if } e \text{ infinite loops no matter which reductions you do, then } \texttt{halts}[e] = \text{false}
\]

Then we can write the \( \lambda \)-expression

\[
f = \lambda x. \text{if } \texttt{halts}(\text{App } x (\text{quote } x)) \text{ then } \Omega \text{ else true}
\]

Now, either \( f[f] \) halts, or it doesn’t.

\[
f[f] = \text{if } \texttt{halts}(\text{App } f (\text{quote } f )) \text{ then } \Omega \text{ else true}
\]
What can we compute?

Suppose: For any expression e,

- if $e \rightarrow^* e'$ and $e'$ cannot step, then $\text{halts } [e] = \text{true}$
- if $e$ infinite loops no matter which reductions you do, then $\text{halts } [e] = \text{false}$

Write a quoting function such that $\text{quote } [e] = \lceil [e] \rceil$

$$f = \lambda x. \text{ if } \text{halts } (\text{App } x (\text{quote } x)) \text{ then } \Omega \text{ else true}$$

$$f [f] = \text{ if } \text{halts } (\text{App } [f] (\text{quote } [f] )) \text{ then } \Omega \text{ else true}$$

$$\text{App } [f] (\text{quote } [f] ) = \text{quote } (f [f]) = \lceil f [f] \rceil$$

If $f [f]$ halts, then $f [f]$ doesn’t halt.

If $f [f]$ doesn’t halt, then $f [f]$ halts.

But we only made one hypothetical assumption so far: that is, one can implement a “halts” function. That leads to a contradiction. So therefore, the “halts” function cannot be implemented.
Models of computation

• Herbrand-Gödel recursive functions (1935)
  developed by Kleene from ideas by Herbrand and Gödel
• λ-calculus (1935)
  developed by Church with his students Rosser & Kleene
• Turing machine (1936)
  developed by Turing
Theorem (1935, Kleene): any function you can implement in H-G recursive functions, you can implement in $\lambda$-calculus.
Proof: previous slides—all those data structures, numbers, recursion, etc.

Theorem (1935, Kleene): any function you can implement in $\lambda$-calculus, you can implement in H-G recursive functions.

Theorem (1936, Church): There’s a mathematical function \textit{not} implementable in $\lambda$-calculus (the “halts” function).

Theorem (1936, Turing, ): There’s a mathematical function \textit{not} implementable in Turing machines (the “halts” function). (Dang! Church published first!)

Theorem (1936, Turing): any function you can implement in $\lambda$-calculus, you can implement in Turing machines.
Proof: Turing machine can simulate the substitution-based interpreter.

Theorem (1936, Turing): any function you can implement in Turing machines, you can implement in $\lambda$-calculus.
Proof: Program Turing-machine simulator in $\lambda$-calculus.
Theorem (1936, Turing): any function you can implement in $\lambda$-calculus, you can implement in Turing machines.
Proof: Turing machine can simulate the substitution-based interpreter.

Do you believe this proof?
You’ve seen the substitution-based interpreter in Ocaml; could that be programmed to run on a von Neumann machine?

(There’s strong evidence for “yes”, it’s called “ocamlc.opt”, the compiler)

(but a von Neumann machine is not a Turing machine, one has to simulate a von Neumann machine on a Turing machine – not difficult.)
Theorem (1936, Turing): any function you can implement in Turing machines, you can implement in \( \lambda \)-calculus.
Proof: Program Turing-machine simulator in \( \lambda \)-calculus.

Do you believe this proof?
Could you write a pure functional Ocaml program that simulates a Turing machine?

(Of course you could!)
WHY IT’S IMPORTANT TO PRUNE CLOSURE ENVIRONMENTS
let zeros n = if n=0 then [] else 0 :: zeros(n-1)

let h (n: int) : int =
  let f x =
    let k = List.length x in
    fun () -> k
  in
  let rec g i : (unit->int) list =
    if i=0 then [] else f (zeros n) :: g (i-1)
  in
  let bigdata = g n
  in List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000
let zeros i = if i=0 then [] else 0 :: s(i-1)

let h (n: int) : int =
let f x =
  let k = List.length x in
  fun () -> k
in
let rec g i : (unit->int) list =
  if i=0 then [] else f (zeros n) :: g (i-1)
in let bigdata = g n
in List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000

5 words of memory versus 3 words, what’s the big deal?

What are the free variables of this function?
let zeros i = if i=0 then [] else 0 :: s(i-1)

let h (n: int) : int =
  let f x =
    let k = List.length x in
    fun () -> k
  in
  let rec g i : (unit->int) list =
    if i=0 then [] else f (zeros n) :: g (i-1)
  in let bigdata = g n
  in List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000

Run the program to here, and what is in memory?

```
bigdata
    +---+---+---+
   n
   +---+---+---+
    fun() => k
    +---+---+---+
    fun() => k
    +---+---+---+
    fun() => k
```

let zeros i = if i=0 then [] else 0 :: s(i-1)

let h (n: int) : int =
  let f x =
    let k = List.length x in
    fun () -> k
  in
  let rec g i : (unit->int) list =
    if i=0 then [] else f (zeros n) :: g (i-1)
  in
  let bigdata = g n
  in
  List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000

n closures for (fun()->k), each is a list of length n, total space usage n^2
let zeros i = if i=0 then [] else 0 :: s(i-1)

let h (n: int) : int =
  let f x =
    let k = List.length x in
    fun () -> k
  in
  let rec g i : (unit->int) list =
    if i=0 then [] else f (zeros n) :: g (i-1)
  in
  let bigdata = g n
  in
  List.fold_left (fun s u -> u()+s) 0 bigdata

let a = h 1000

What are the free variables of this function?

n closures for (fun()¬→k),
each is just a number k,
total space usage O(n)
Therefore

Closures should represent *only* the free variables of a function (not *all the variables currently in scope*),

otherwise the compiled program may use *asymptotically more space*,

such as *O*(n^2) instead of *O*(n)