Computability

COS 326
Andrew W. Appel
Princeton University

FUNCTIONAL PROGRAMMING AS A MODEL OF COMPUTATION

Untyped lambda-calculus

$e ::= \lambda x.e_1 \mid x \mid e_1\ e_2$ $\qquad$ $\lambda x.e_1$ *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) (λx.xab)

pair (λa.λb.λx.xab)

fst (λp.p(λxy.x))

snd (λp.p(λxy.y))

fst(pair a b) = a

snd(pair a b) = b

pair a b ≈ (a,b)

fst (pair a b)

= (λp.p(λxy.x))((λa.λb.λx.xab)ab)

--> (λp.p(λxy.x))((λb.λx.xab)b)

--> (λp.p(λxy.x))(λx.xab)

--> (λx.xab)(λxy.x)

--> (λxy.x)ab

--> (λy.a)b

--> 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

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

$\dashrightarrow (\lambda ab.\ (\lambda xy.x)ab)\ a\ b$

$\dashrightarrow (\lambda b.\ (\lambda xy.x)ab)\ b$

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

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

$\dashrightarrow a$

Lists

nil (λcn.n)

cons (λht.λcn.cht)

match (λacn.acn)

nil ≈ []

cons h t ≈ h::t

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

= (λacn.acn)((λht.λcn.cht)xy)fg

--> (λacn.acn)(λcn.cxy)fg

--> (λcn. (λcn.cxy)cn) fg

--> (λn.fxy)g

--> fxy)

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

```
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

General inductive datatypes

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 $\lambda uabcd.uabcd$

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

= b y

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?

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

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

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

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

Recursive functions

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

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

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

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

= $g(Yg)$

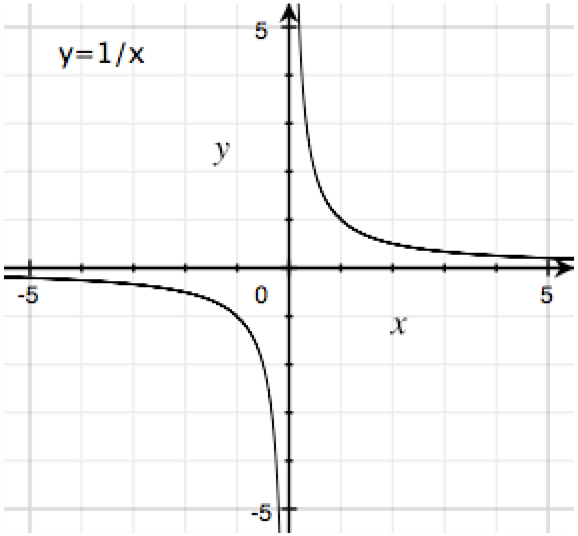
Fixed points

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)

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

add (S(SO))b

= (λab. Yga)(S(SO))b

= Yg(S(SO))

= g(Yg)(S(SO))

= match S(SO) with O -> b | S a’ -> S(Yga’)

= S(Yg(SO))

=S(match SO with O -> b | S a’ -> S(Yga’))

=S(S(YgO))

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

=S(S b)

Theorem: add 1 2 = 3

type int = O | S of int $\quad$ O=λxy.x $\quad$ 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

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

fact = Yg

fact 3 = Yg3

= g(Yg)3

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

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

= $3 \cdot (Yg2)$

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

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

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

= $3 \cdot (2 \cdot (1 \cdot$if 0=0 then 1 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 lecture 6 here . . .]

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

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

Models of computation

Theorem (1935, Kleene): any function you can implement in H-G recursive functions, you can implement in λ-calculus.
Proof: previous slides—all those data structures, numbers, recursion, etc.

Theorem (1935, Kleene): any function you can implement in λ-calculus, you can implement in Herbrand-Gödel recursive functions.

Theorem (1936, Church): There's a mathematical function *not* implementable in λ-calculus (the "halts" function).

Theorem (1936, Turing,): There's a mathematical function *not* implementable in Turing machines (the "halts" function). (Dang! Church published first!)

Theorem (1936, Turing): any function you can implement in λ-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 λ-calculus.
Proof: Program Turing-machine simulator in λ-calculus.

Models of computation

Theorem (1936, Turing): any function you can implement in λ-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 "the OCaml 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.

Models of computation

Theorem (1936, Turing): any function you can implement in Turing machines, you can implement in λ-calculus.
Proof: Program Turing-machine simulator in λ-calculus.

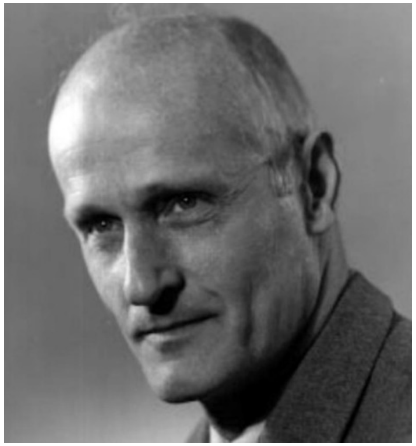
Do you believe this proof?
Could you write a pure functional Ocaml program that simulates a Turing machine?

(Of course you could!)

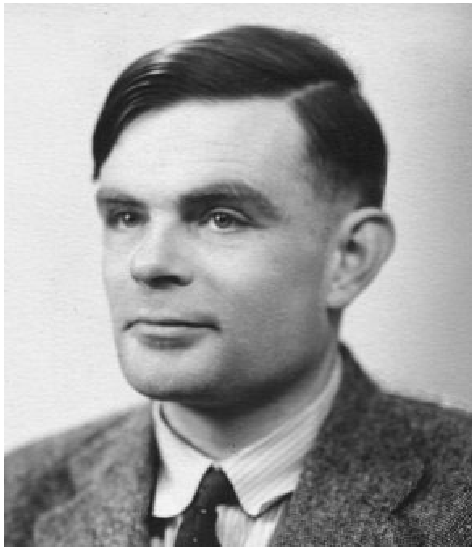
Summary:

Programming Languages = Computers

Church



Kleene*



Turing

Von Neumann

Princeton, New Jersey

Models of computation

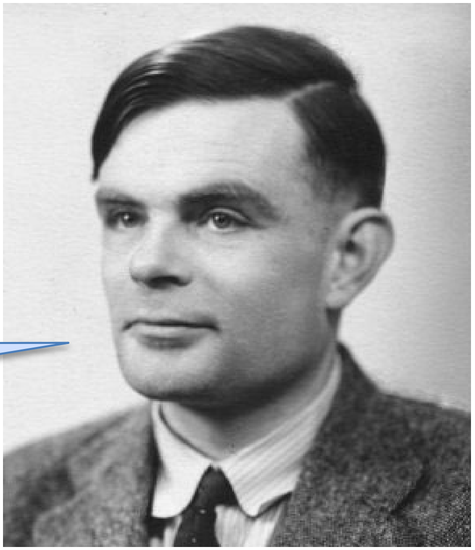
In 1950, Turing even made the far-fetched claim

that by the year 2000,

a computer might have a billion bits of memory

and might be able to simulate human conversation.

Hey Siri,
what's the
"Turing Test"?



Uncomputability:
What we **can't** compute

Entscheidungsproblem (1928)

Is there a mathematical function that **cannot** be computed

- by a Turing machine?
- by an expression in λ-calculus?
- by a von Neumann machine?
- by an OCaml program?
- by any kind of mechanical process?

Answer: Yes indeed. Let's define that function and then show that it can't be implemented

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 λ-expression, $\lceil e \rceil$ is its representation in **`exp`**

$\lceil x_i \rceil$ = **`Var i`**

$\lceil e1\ e2 \rceil$ = **`App`** $\lceil e1 \rceil\ \lceil e2 \rceil$

$\lceil \lambda\, x_i .\, e1 \rceil$ = **`Fun i`** $\lceil e1 \rceil$

Datatype representation

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

This data type can also be expressed in pure λ-calculus:

Fun = $\lambda v \lambda e\ \lambda abc.ave$

Var = $\lambda v\ \lambda abc.bv$

App = $\lambda e_1 e_2\ \lambda abc.ce_1e_2$

What can we compute?

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

1. Write a λ-function **interp** such that

   For any expression e

   that evaluates in λ-calculus to a normal form e',

   (that is, e -->* e' and e' cannot take a step)

   interp [e] -->* [e']

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

What will **interp** do on infinite loops?

Suppose e never gets to a normal form, that is,

e --> e' --> e'' --> e'' … forever

Then

interp [e] --> … --> … --> … --> … --> … --> …

interp [e] also does not have a normal form,

that is,

interp [e] infinite loops.

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 = $\lceil e \rceil$

Impossible:

Consider e1 = (λx.x)y and e2=y

kwoht e1 = kwoht ((λx.x)y) = kwoht y = kwoht e2

$\lceil e1 \rceil$ = **`App(Fun(i,Var i),Var j)`**

$\lceil e2 \rceil$ = **`Var j`**

$\lceil e1 \rceil \neq \lceil e2 \rceil$

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 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?

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

4. Write a λ-function **halts** such that

For any expression e,

if e -->* e' and e' cannot step, then halts [e] = true

if e infinite loops no matter which reductions you do,

then halts [e] = false

Claim: you cannot write such a function

What can we compute?

Proof by contradiction. Suppose there exists a λ-expression **halts** such that for any expression e,

if e -->* e' and e' cannot step, then halts ⌈e⌉ = true

if e infinite loops no matter which reductions you do, then halts ⌈e⌉ = false

Then we can write the λ-expression

f = λx. if halts (App x (quote x)) then Ω else true

Now, either f ⌈f⌉ halts, or it doesn't.

f ⌈f⌉ = if halts (App ⌈f⌉ (quote ⌈f⌉)) then Ω else true

What can we compute?

Suppose: For any expression e,

if e -->* e' and e' cannot step, then halts ⌈e⌉ = true

if e infinite loops no matter which reductions you do, then halts ⌈e⌉ = false

Write a quoting function such that quote ⌈e⌉ = ⌈⌈e⌉⌉

f = λx. if halts (App x (quote x)) then Ω else true

f ⌈f⌉ = if halts (App ⌈f⌉ (quote ⌈f⌉)) then Ω else true

App ⌈f⌉ (quote ⌈f⌉) = quote (f ⌈f⌉) = ⌈ f ⌈f⌉ ⌉

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,

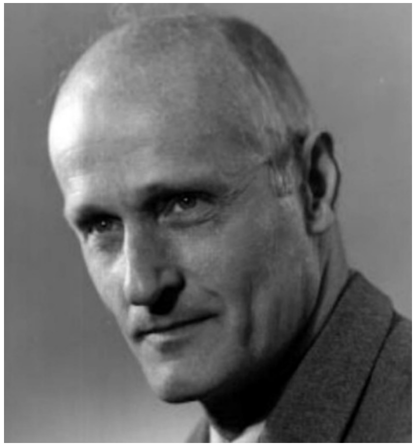
one can implement a "halts" function. That leads to a contradiction.

So therefore, the "halts" function cannot be implemented.

That's what Alonzo Church proved in 1936

(with ideas from Kleene)

Church



Kleene*

Princeton, New Jersey