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

COS 326

Presented by: Andrew W. Appel

Princeton University

WHAT CAN'T WE COMPUTE?

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, ⌈e⌉ is its representation in **`exp`**

⌈x_i⌉ = **`Var i`**

⌈e1 e2⌉ = **`App`** ⌈e1⌉ ⌈e2⌉

⌈λ x_i . e1⌉ = **`Fun i`** ⌈e1⌉

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,

also 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 = ⌈e⌉

Impossible:

Consider e1 = (λx.x)y and e2=y

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

⌈e1⌉ = **`App(Fun(i,Var i),Var j)`**

⌈e2⌉ = **`Var j`**

⌈e1⌉≠⌈e2⌉

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.

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 functions, you can implement in H-G 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 "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.

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!)

Conclusion 1

All these models of computation can simulate each other, thus they have equivalent power to express mathematical functions.

(Some of these models “run faster” than others.)

They can express functions not imagined by Church, Godel, etc: for example, the Amazon app on your Samsung smartphone running Google’s operating system that’s an open-source derivative of Linux . . .

Conclusion 1

All these models of computation can simulate each other, thus they have equivalent power to express mathematical functions.

(Some of these models "run faster" than others.)

They can express functions not imagined by Church, Godel, etc: for example, the Amazon app on your Samsung smartphone running Google's operating system that's an open-source derivative of Linux . . .

. . . but in 1950, Turing imagined computers of the year 2000 with billions of bits of memory, that could conduct intelligent-seeming computations.

Conclusion 2

All these models of computation can simulate each other, thus they have equivalent power to express mathematical functions.

But some functions are not "computable" by *any* of these models: in particular,

- For any program P, does P halt? (yes or no)
- For any program P, does P compute the right answer?
- For any program P, what's the fastest (most optimized) machine-language program that implements it?

Caveat

Not computable:

- For any program P, does P halt? (yes or no)
- For any program P, does P compute the right answer?
- For any program P, what's the fastest (most optimized) machine-language program that implements it?

Caveat

Not computable:

- For any program P, does P halt? (yes or no)
- For any program P, does P compute the right answer?
- For any program P, what's the fastest (most optimized) machine-language program that implements it?

Computable:

- Does this program halt? let f(i)=if i=0 then 1 else 2
- Does this program halt? let rec f(i) = f(i+1)
- Does this program compute a+b ?

```
let rec g(a,b) = if a>0 then g(a-1,b+1)
                 else if a<0 then g(a+1,b-1)
                 else b
```