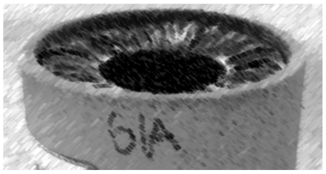

CS61A Lecture 38

Robert Huang
UC Berkeley
April 17, 2013

Announcements

- HW12 due Wednesday
- Scheme project, contest out

Review: Program Generator

A computer program is just a sequence of bits

It is possible to enumerate all bit sequences

```
from itertools import product

def bitstrings():
    size = 0
    while True:
        tuples = product(('0', '1'), repeat=size)
        for elem in tuples:
            yield ''.join(elem)
        size += 1

>>> [next(bs) for _ in range(0, 10)]
['', '0', '1', '00', '01', '10', '11', '000', '001', '010']
```

Review: Function Streams

Given a stream of 1-argument functions, we can construct a function that is not in the stream, *assuming that all functions in the stream terminate*

```
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

Inputs

```
           [F] T   T   T   T   F   T   F   T   F  . . .
            T [T]  F   F   F   F   F   T   F   T  . . .
            T  F  [T]  F   T   F   T   F   T   T  . . .
            T  F   F  [T]  T   F   F   T   F   T  . . .
            T  F   T   T  [F]  T   F   T   F   T  . . .
            F  F   F   F   T  [F]  F   F   T   T  . . .
            T  F   T   F   F   F  [F]  T   T   T  . . .
            F  T   F   T   T   F   T  [F]  F   T  . . .
            T  F   T   F   F   T   T   F  [F]  T  . . .
Functions   F  T   T   T   T   T   T   T   T  [F] . . .
                            . . .

            T  F   F   F   T   T   T   T   T   T  . . .
```

Programs and Mathematical Functions

A mathematical function *f(x)* maps elements from its input *domain D* to its output *range R*

$$f : \mathbb{N} \rightarrow \{0, 1\}, \;\; f(x) = x^2 \mod 2$$

A Python function **func** *computes* a mathematical function *f* if the following conditions hold:

- **func** has the same number of parameters as inputs to *f*
- **func** terminates on every input in *D*
- The return value of **func(x)** is the same as *f(x)* for all *x* in *D*

```
def func(x):
    return (x * x) % 2
```

A mathematical function *f* is *computable* if there exists a program (i.e. a Python function) **func** that computes it

Computability

Are all functions computable?

More specifically, we hate infinite loops

So if we have a program that computes the following function, we can run it on our programs to determine if they have infinite loops:

$$haltsonallinputs : Programs \rightarrow \{0, 1\},$$
$$haltsonallinputs(P) = \begin{cases} 1 & \text{if } P \text{ halts on all inputs} \\ 0 & \text{otherwise} \end{cases}$$

Halts

Let's be less ambitious; we'll take a program that computes whether or not another program halts on a specific non-negative integer input:

$$halts : Programs \times \mathbb{N} \to \{0, 1\},$$
$$halts(P, n) = \begin{cases} 1 & \text{if } P \text{ halts on input } n \\ 0 & \text{otherwise} \end{cases}$$

Is this function computable?

It's not as simple as just running the program *P* on *n* to see if it terminates

How long do we let it run before deciding that it won't terminate?

However long we let it run before declaring it that it won't terminate, it might just need a little more time to finish its computation

Thus, we have to do something more clever, analyzing the program itself

Turing

Let's assume that we have a Python function **`halts`** that computes the mathematical function *halts*, written by someone more clever than us

Remember, we can pass a function itself as its argument. Thus, we can consider **`halts(f, f)`**; in other words, does function **`f`** halt when given itself as an argument? (This is just a thought experiment.)

We can then define a new function, **`turing`**, which takes in 1 argument.

```
def turing(f):
    if halts(f, f):
        while True:     # infinite loop
            pass
    else:
        return True     # halts
```

`turing` will go into an infinite loop if **`f`** halts when given itself as an argument. Otherwise, **`turing`** returns **`True`**.

Turing... what?

```
def turing(f):
    if halts(f, f):
        while True:     # infinite loop
            pass
    else:
        return True     # halts
turing(turing)          # * what?
```

If this sounds fishy, it should. Should the call **`turing(turing)`** halt or go into an infinite loop?

- **`turing(turing)`** loops → **`halts(turing, turing)`** returns true
  - However, **`turing(turing)`** should have halted
- **`turing(turing)`** halts → **`halts(turing, turing)`** returns false
  - However, **`turing(turing)`** should not have halted

We have a contradiction! Our assumption that **`halts`** exists is false.

Bitstrings and Functions

Let's develop another proof, assuming that we have a **`halts`** program that computes the mathematical function *halts*

Let's create a stream of all 1-argument Python functions, then use **`halts`** to filter out non-terminating programs from that stream

Assume we have the following Python functions:

```
def is_valid_python_function(bitstring):
    """Determine whether or not a bitstring represents a
    syntactically valid 1-argument Python function."""

def bitstring_to_python_function(bitstring):
    """Coerce a bitstring representation of a Python
    function to the function itself."""
```

Bitstrings and Functions

Let's develop another proof, assuming that we have a **`halts`** program that computes the mathematical function *halts*

Let's create a stream of all 1-argument Python functions, then use **`halts`** to filter out non-terminating programs from that stream

Then the following produces all valid 1-argument Python functions:

On HW12

```
def function_stream():
    """Return a stream of all valid 1-argument Python
    functions."""
    bitstring_stream = iterator_to_stream(bitstrings())
    valid_stream = filter_stream(is_valid_python_function,
                                 bitstring_stream)
    return map_stream(bitstring_to_python_function,
                      valid_stream)
```

Filtering Out Non-Terminating Programs

With **`halts`**, we can't filter out programs that don't halt on all input

But we can filter out programs that don't halt on a specific input

Specifically, let's make sure that a program halts on its index in the resulting stream of programs

```
def make_halt_checker():
    index = 0
    def halt_checker(fn):
        nonlocal index
        if halts(fn, index):
            index += 1
            return True
        return False
    return halt_checker

programs = filter_stream(make_halt_checker(),
                         function_stream())
```

Developing a Contradiction

We now have a stream of programs that halt when given their own index as input

```
programs = filter_stream(make_halt_checker(),
                         function_stream())
```

Recall the following function that produces a function that is not in a given stream:

```
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

Consider the following:

```
church = func_not_in_stream(programs)
```

Does **`church`** appear anywhere in **`programs`**?

Developing a Contradiction

```
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

```
church = func_not_in_stream(programs)
```

Does **`church`** appear anywhere in **`programs`**?

Every element in **`programs`** halts when given its own index as input

Thus, **`church`** halts on all inputs **`n`**, since it calls the **`n`**th element in **`programs`** on **`n`**

So **`halt_checker`** returns true on **`church`**, which means that **`church`** is in **`programs`**

If **`church`** is in **`programs`**, it has an index **`m`**; so what does **`church(m)`** do?

Developing a Contradiction

```
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

```
church = func_not_in_stream(programs)
```

Does **`church`** appear anywhere in **`programs`**?

Every element in **`programs`** halts when given its own index as input

Thus, **`church`** halts on all inputs **`n`**, since it calls the **`n`**th element in **`programs`** on **`n`**

If **`church`** is in **`programs`**, it has an index **`m`**; so what does **`church(m)`** do?

It calls the **`m`**th element in **`programs`**, which is **`church`** itself, on **`m`**

This results in an infinite loop, which means **`halt_checker`** will return false on **`church`**, since it does not halt given its own index

Developing a Contradiction

```
def func_not_in_stream(s):
    return lambda n: not s[n](n)
```

```
church = func_not_in_stream(programs)
```

We have a contradiction!

`halt_checker(church)` returns true, which means that **`church`** is in **`programs`**

But if **`church`** is in **`programs`**, then **`church(m)`**, where **`m`** is **`church`**'s index in **`programs`**, is an infinite loop, so **`halt_checker(church)`** returns false

So we made a false assumption somewhere

False Assumption

We assumed we had the following Python functions:

- **`halts`**
- **`is_valid_python_function`**
- **`bitstring_to_python_function`**

Everything else we wrote ourselves

The latter two functions can be built using components of the interpreter

Thus, it is our assumption that there is a Python function that computes *halts* that is invalid

$$halts : Programs \times \mathbb{N} \rightarrow \{0, 1\},$$
$$halts(P, n) = \begin{cases} 1 & \text{if } P \text{ halts on input } n \\ 0 & \text{otherwise} \end{cases}$$

The Halting Problem

The question of whether or not a program halts on a given input is known as *the halting problem*.

In 1936, Alan Turing proved that the halting problem is unsolvable by a computer

That is, the mathematical function *halts* is uncomputable

$$halts : Programs \times \mathbb{N} \rightarrow \{0, 1\},$$
$$halts(P, n) = \begin{cases} 1 & \text{if } P \text{ halts on input } n \\ 0 & \text{otherwise} \end{cases}$$

We proved that *halts* is uncomputable in Python, but our reasoning applies to all languages

It is a fundamental limitation of all computers and programming languages

Uncomputable Functions

It gets worse; not only can we not determine programmatically whether or not a given program halts, we can't determine *anything* "interesting" about the *behavior* of a program in general

For example, suppose we had a program **`prints_something`** that determines whether or not a given program prints something to the screen when run on a specific input:

Then we can write **`halts`**:

```
def halts(fn, i):
    delete all print calls from fn
    replace all returns in fn with prints
    return prints_something(fn, i)
```

Since we know we can't write **`halts`**, our assumption that we can write **`prints_something`** is false

Consequences

There are vast consequences from the impossibility of computing *halts*, or any other sufficiently interesting mathematical functions on programs

The best we can do is approximation

For example, perfect anti-virus software is impossible

- Anti-virus software must either miss some viruses (false negatives), mark some innocent programs as viruses (false positives), or fail to terminate on others

We can't write perfect security analyzers, optimizing compilers, etc.

Incompleteness Theorem

In 1931, Kurt Gödel proved that any mathematical system that contains the theory of non-negative integers must be either *incomplete* or *inconsistent*

- A system is *incomplete* if there are true facts that cannot be proven
- A system is *inconsistent* if there are false claims that can be proven

A proof is just a sequence of statements, which can be represented as bits

- We can generate all proofs the same way we generated all programs

It is also possible to check the validity of a proof using a computer

- Given a finite set of axioms and inference rules, a program can check that each statement in a proof follows from the previous ones

Thus, if a valid proof exists for a mathematical formula, then a computer can find it

Incompleteness Theorem

Given a sufficiently powerful mathematical system, we can write the following formula, which is a predicate form of the *halts* function:

$$H(P, n) = \text{“program } P \text{ halts on input } n\text{”}$$

If *H(P, n)* is provable or disprovable for all *P* and *n*, then we can write a program to prove or disprove it by generating all proofs and checking each one to see if it proves or disproves *H(P, n)*

But then this program would solve the halting problem, which is impossible

Thus, there must be values of *P* and *n* for which *H(P, n)* is neither provable nor disprovable, or for which an incorrect result can be proven

Thus, there are fundamental limitations not only to computation, but to mathematics itself!

Interpretation in Python

`eval`: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.

`exec`: Executes a statement in the current environment. Doing so may affect the environment.

```
eval('2 + 2')

exec('def square(x): return x * x')
```

`os.system('python <file>')`: Directs the operating system to invoke a new instance of the Python interpreter.