IMPOSSIBLE PROGRAMS

@tomstuart / QCon London / 2014-03-05

PROGRAMS

CAN'T DO

EVERYTHING

how can a

PROGRAM

be

IMPOSSIBLE?

WE DEMAND

UNIVERSAL SYSTEMS

Compare two programming languages, say Python and Ruby.

We can translate any Python program into Ruby.
We can translate any Ruby program into Python.

We can implement a Python **interpreter** in Ruby.
We can implement a Ruby **interpreter** in Python.

We can implement a Python interpreter in JavaScript.
We can implement a JavaScript interpreter in Python.

We can implement a Turing machine simulator in Ruby.
We can implement Ruby as a Turing machine.

Tag systems

SKI calculus

Game of Life

Ruby

Lisp

Register machines

XSLT

JavaScript

Magic: The Gathering

Partial recursive functions

C

Python

Java

Turing machines

Lambda calculus

Rule 110

C++

Haskell

Universal systems can run **software**.

We don't just want machines, we want **general-purpose** machines.

PROGRAMS ARE DATA

```
>> puts 'hello world'
hello world
=> nil

>> program = "puts 'hello world'"
=> "puts 'hello world'"

>> bytes_in_binary = program.bytes.
                       map { |byte| byte.to_s(2).rjust(8, '0') }
=> ["01110000", "01110101", "01110100", "01110011", "00100000",
    "00100111", "01101000", "01100101", "01101100", "01101100",
    "01101111", "00100000", "01110111", "01101111", "01110010",
    "01101100", "01100100", "00100111"]

>> number = bytes_in_binary.join.to_i(2)
=> 979654384950070652110298049571774002183479 1
```

```
>> number = 97965438495007065211029804957177400218347 91
=> 97965438495007065211029804957177400218347 91

>> bytes_in_binary = number.to_s(2).scan(/.+?(?=.{8}*\z)/)
=> [ "1110000", "01110101", "01110100", "01110011", "00100000",
     "00100111", "01101000", "01100101", "01101100", "01101100",
     "01101111", "00100000", "01110111", "01101111", "01110010",
     "01101100", "01100100", "00100111"]

>> program = bytes_in_binary.map { |string| string.to_i(2).chr }.join
=> "puts 'hello world'"

>> eval program
hello world
=> nil
```

UNIVERSAL SYSTEMS

+

PROGRAMS ARE DATA

=

INFINITE LOOPS

Every universal system can simulate every other universal system, including itself.

More specifically: every universal programming language can implement its own interpreter.

```ruby
def evaluate(program, input)
  # parse program
  # evaluate program on input while capturing output
  # return output
end
```

```
>> evaluate('print $stdin.read.reverse', 'hello world')
=> "dlrow olleh"
```

```ruby
def evaluate(program, input)
  # parse program
  # evaluate program on input while capturing output
  # return output
end

def evaluate_on_itself(program)
  evaluate(program, program)
end
```

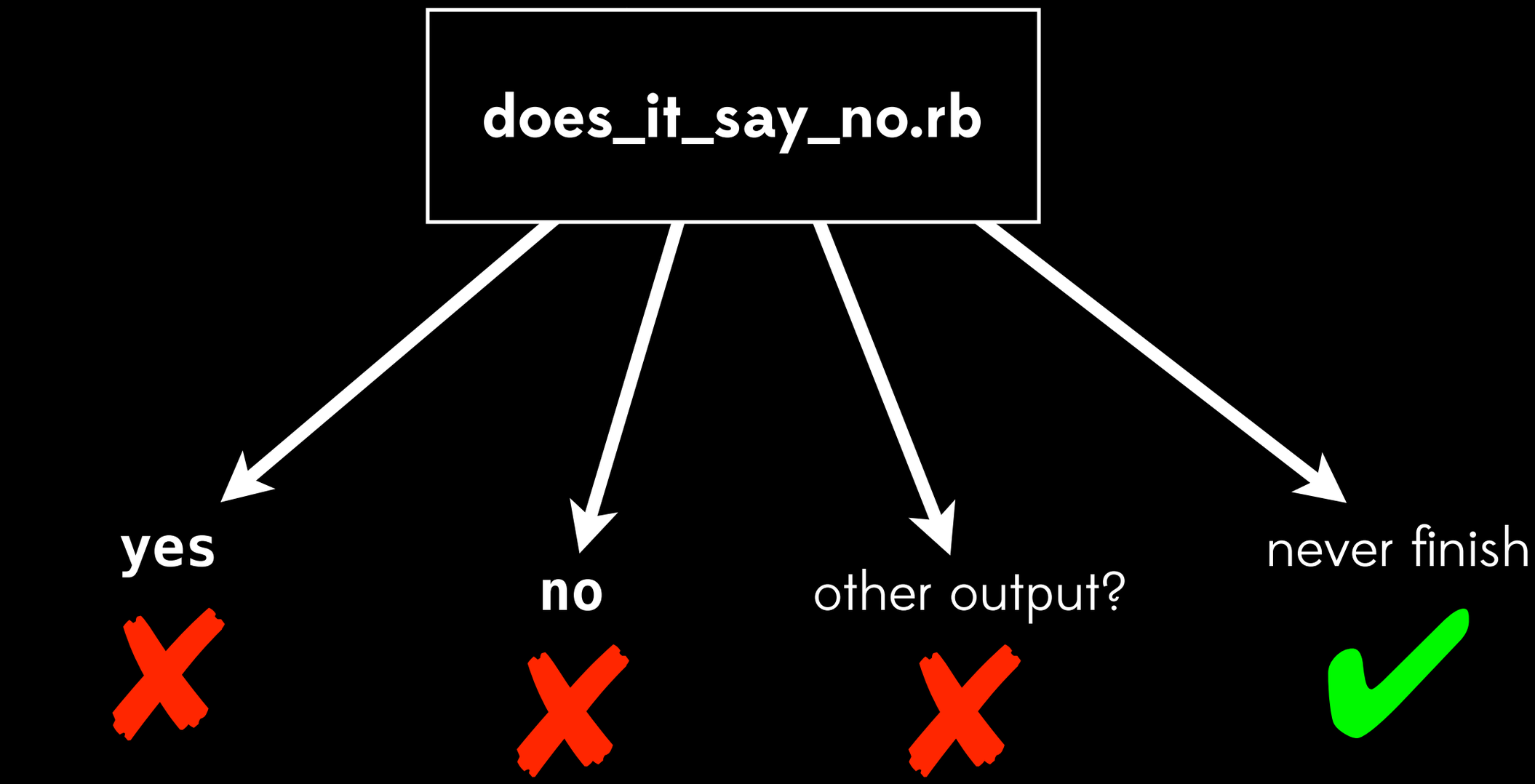
```
>> evaluate_on_itself('print $stdin.read.reverse')
=> "esrever.daer.nidts$ tnirp"
```

```ruby
def evaluate(program, input)
  # parse program
  # evaluate program on input while capturing output
  # return output
end

def evaluate_on_itself(program)
  evaluate(program, program)
end
```

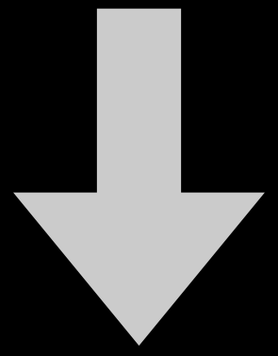
```ruby
program = $stdin.read

if evaluate_on_itself(program) == 'no'
  print 'yes'
else
  print 'no'
end
```

does_it_say_no.rb

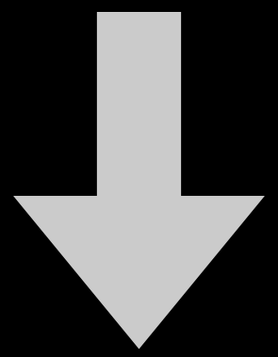
```
$ echo 'print $stdin.read.reverse' | ruby does_it_say_no.rb
no

$ echo 'print "no" if $stdin.read.include?("no")' | ruby does_it_say_no.rb
yes

$ ruby does_it_say_no.rb < does_it_say_no.rb
???
```


does_it_say_no.rb
yes
no
other output?
never finish

Ruby is universal



so we can write **`#evaluate`** in it



so we can construct a special program that loops forever

so here's one

IMPOSSIBLE PROGRAM

Sometimes infinite loops are bad.

We could remove features from a language until there's no way to cause an infinite loop.

- **No unlimited iteration**

  remove **`while`** loops etc, only allow iteration over finite data structures

- **No lambdas**

  to prevent **`(λx.x x)(λx.x x)`**

- **No recursive method calls**

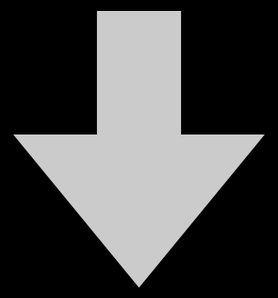
  e.g. only allow a method to call other methods whose names come later in the alphabet

- **No blocking I/O**
- **…**

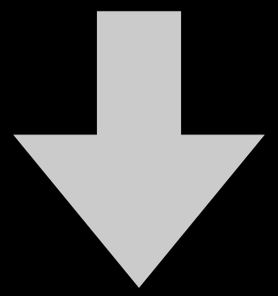
The result is called a **total** programming language.

It must be impossible to write an interpreter for a total language in itself.

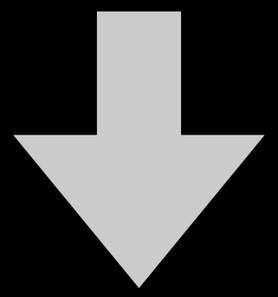
if we could write **`#evaluate`** in a total language



then we could use it to construct a special program
that loops forever



but a total language doesn't let you write programs
that loop forever



so it must be impossible to write **`#evaluate`** in one

(That's weird, because a total language's interpreter always finishes eventually, so it feels like the kind of program we should be able to write.)

We **could** write an interpreter for a total language in a universal language, or in **some other** more powerful total language.

okay but

WHAT ABOUT REALITY?

`#evaluate` is an impossible program for any total language, which means that total languages can't be universal.

Universal systems have impossible programs too.

```
input = $stdin.read
puts input.upcase
```

This program always finishes.*

* assuming **STDIN** is finite & nonblocking

```
input = $stdin.read

while true
  # do nothing
end

puts input.upcase
```

This program always loops forever.

Can we write a program that can
decide this in general?

(This question is called the **halting problem**.)

```ruby
input = $stdin.read
output = ''

n = input.length

until n.zero?
  output = output + '*'
  n = n - 1
end

puts output
```

```ruby
require 'prime'

def primes_less_than(n)
  Prime.each(n - 1).entries
end

def sum_of_two_primes?(n)
  primes = primes_less_than(n)
  primes.any? { |a| primes.any? { |b| a + b == n } }
end

n = 4

while sum_of_two_primes?(n)
  n = n + 2
end

print n
```

```ruby
def halts?(program, input)
  # parse program
  # analyze program
  # return true if program halts on input, false if not
end
```

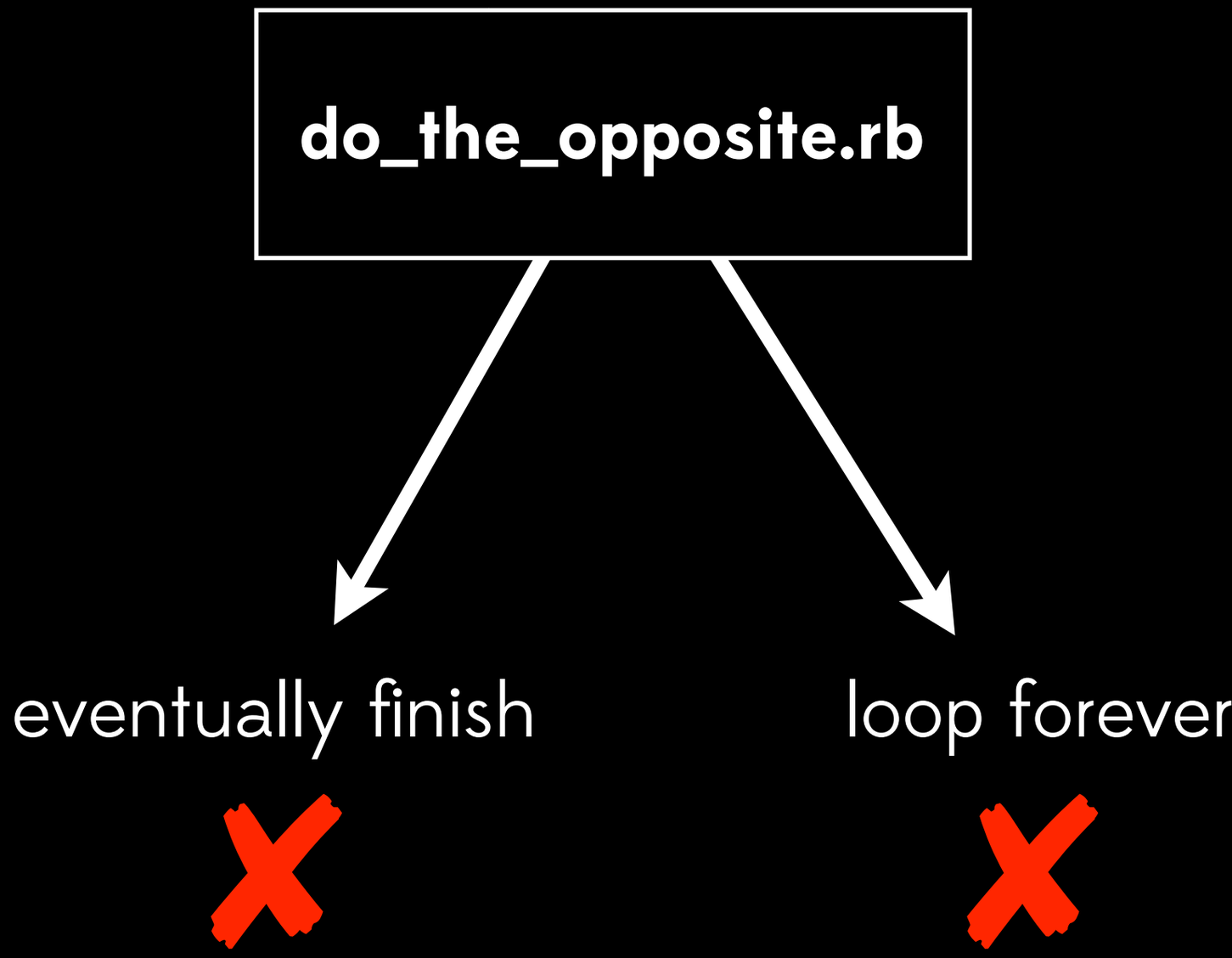
```
>> halts?('print $stdin.read', 'hello world')
=> true
```

```
>> halts?('while true do end', 'hello world')
=> false
```

```ruby
def halts?(program, input)
  # parse program
  # analyze program
  # return true if program halts on input, false if not
end

def halts_on_itself?(program)
  halts?(program, program)
end

program = $stdin.read

if halts_on_itself?(program)
  while true
    # do nothing
  end
end
```

do_the_opposite.rb

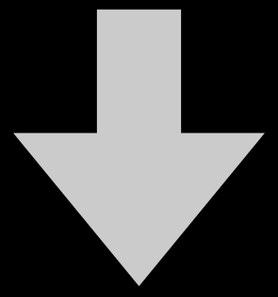
```
$ ruby do_the_opposite.rb < do_the_opposite.rb
```


do_the_opposite.rb
eventually finish
loop forever

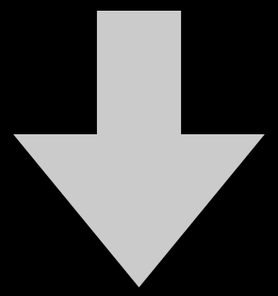
Every real program must either loop forever or not, but whichever happens, **`#halts?`** will be wrong about it.

do_the_opposite.rb forces **`#halts?`** to give the wrong answer.

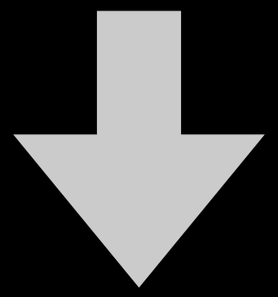
if we could write **`#halts?`**



then we could use it to construct a special program
that forces **`#halts?`** to give the wrong answer



but a correct implementation of **`#halts?`**
would always give the right answer



so it must be impossible to write **`#halts?`**

okay but

WHO CARES?

We never actually want to ask a computer whether a program will loop forever.

But we often want to ask computers other questions about programs.

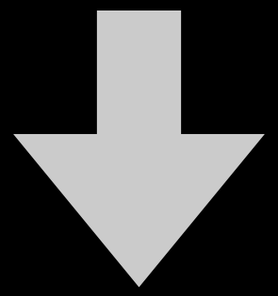
```ruby
def prints_hello_world?(program, input)
  # parse program
  # analyze program
  # return true if program prints "hello world", false if not
end
```

```
>> prints_hello_world?('print $stdin.read.reverse', 'dlrow olleh')
=> true
```

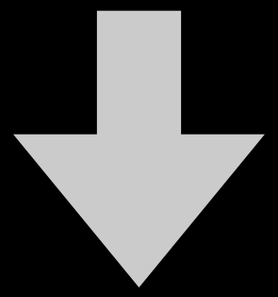
```
>> prints_hello_world?('print $stdin.read.upcase', 'dlrow olleh')
=> false
```

```ruby
def prints_hello_world?(program, input)
  # parse program
  # analyze program
  # return true if program prints "hello world", false if not
end

def halts?(program, input)
  hello_world_program = %Q{
    program = #{program.inspect}
    input = $stdin.read
    evaluate(program, input)
    print 'hello world'
  }

  prints_hello_world?(hello_world_program, input)
end
```

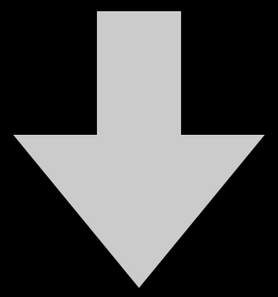
if we could write **`#prints_hello_world?`**



then we could use it to construct a correct implementation of **`#halts?`**



but it's impossible to correctly implement **`#halts?`**



so it must be impossible to write
`#prints_hello_world?`

Not only can we not ask
“does this program halt?”,
we also can’t ask
“does this program do
what I want it to do?”.

This is Rice's theorem:

Any interesting property of program behavior is undecidable.

WHY DOES THIS HAPPEN?

We can't look into the future and predict
what a program will do.

The only way to find out for sure is to run it.

But when we run a program, we don't know
how long we have to wait for it to finish.
(Some programs never will.)

Any system with enough power to be self-referential can't correctly answer every question about itself.

We need to step outside the self-referential system and use a different, more powerful system to answer questions about it.

But there **is** no more powerful system to upgrade to.

HOW CAN WE COPE?

- Ask undecidable questions, but give up if an answer can't be found in a reasonable time.
- Ask several small questions whose answers provide evidence for the answer to a larger question.
- Ask decidable questions by being conservative.
- Approximate a program by converting it into something simpler, then ask questions about the approximation.

From Simple Machines to Impossible Programs

Understanding Computation

O'REILLY®

Tom Stuart

http://computationbook.com/

QCON

(50% off ebook, 40% off print)

THE END

@tomstuart / tom@codon.com