# (Py-)computability

Richard Roth
copying freely from slides by Cian Dorr

8th December 2022

New York University

# Motivation

Sometimes computer programs run for a long time before they spit out their result.
In other cases, programs never halt, e.g. because they get stuck in a never-ending loop.

Question: Can we write a computer program that determines for an any other computer program whether it will eventually spit out a result?

Today, we will see why such a program cannot be written.
If we don't run out of time, that is...

# Key concepts

# Computable functions and decidable sets

A function $f : \mathbb{S} \rightarrow \mathbb{S}$ from strings to strings is *effectively computable* iff there is a systematic procedure (algorithm) which, given any string $s$ as input, will (given enough time and storage resources) output the value $f(s)$ of the function on that string.

A set of strings is *effectively decidable* iff there is a systematic procedure (algorithm) which, given any string as input, will tell us whether the string is in the set.

- So set $X$ is effectively decidable iff the function that maps every member of $X$ to `Yes` and every other string to `No` is effectively computable.

# Different precisifications

These definitions rely on vague words (like ‘systematic procedure’). There are different ways to make these vague words precise.

- Turing’s precisification: Turing machines
- Church’s precisification: “recursive” functions on the natural numbers
- Any computer programming language with a precisely specified syntax and semantics provides a further precisification. That’ll be our approach: we’ll use a super-simple programming language called Py.

# Py and the Church-Turing Thesis

These precisifications, and many others, can be shown to agree on what functions as effectively computable or not: a function can be computed by a Turing machine iff it can be computed by Church's functions iff it can be computed by a Py-program.

The 'Church-Turing thesis' says that they all agree with the notion of effective computability. That's a philosophical thesis; it's not clear what it would even mean to "prove" it, since it's not like we were handed down some list of axioms involving the informal expression 'effectively computable'. But the provable agreement facts constitute a powerful argument for it!

# Introducing Py

# Py-terms

There are three kinds of *Py-terms*: variables, constants, and simple functional terms.

- A *Py-variable* can be any nonempty string not containing any special characters (spaces, newlines, quotes, `+`, parentheses. . . ) and that isn't on a short list of "reserved words" such as `quote`. (So not just $x$, $y$, $x'$, $z2$ etc.!)
- *Py-constants* are exactly the constants from Str, so `"`$c$`"` for any unicode character $c$, except for a few special ones, and `""`, `new`, `quo`, `lpa`, `rpa`, `com` and such.
- Whenever $u$ and $v$ are Py-variables then `head(`s`)`, `tail(`s`)`, and `(`s` + `t`)` are simple functional Py-terms.

The set of Py-terms clearly has the injective property.

Surprisingly, things like `head("A" + "A")` are officially not Py-terms. Why?

## Let-statements

Py-programs can contain let-statements:

```
firstValue = ""
secondValue = "A"
secondValue = secondValue + secondValue
result = head(secondValue)
```

Each line here is called a *let-statement*. Let-statements are of the form $v$ `=` $t$, where $v$ is a Py-variable and $t$ is a Py-term.

- Think of let-statements as an instruction to set the value of $v$ to the value of $t$.
- Note that at the end of the little program above, the value of `result` is set to `head(secondValue)`, where `secondValue` has previously been set to `"A"+"A"`. That's why we don't need to have terms like `head("A" + "A")`.

# While-statements

Let-statements are one of the two kinds of statement in Py. The other kind are while-statements. When $A$ is any Py-*program* and $t_1$ and $t_2$ are any two Py-terms, the following is a while-statement:

`while` $t_1$ `!=` $t_2$`:`

`    `$A$

Here what goes after the first `new` character is the program $A$, but with every line indented by four extra spaces.

- Think of while-statements as an instruction to check if the value of $t_1$ the same as the value of $t_2$. If they do, skip to the end of the block. If they don't, run the indented program $A$ until they do (possibly infinitely!).

# An example

Example:

```
let result = ""
while input != "":
    result = head(input)+result
    input = tail(input)
```

# Defining the set of Py-programs

The *Py-programs* are the smallest set containing the empty string and closed under

$$A \mapsto \begin{array}{l} v\ \texttt{ = }\ t \\ A \end{array}$$

for every Py-variable $v$ and Py-term $t$, and

$$\langle A, B \rangle \mapsto \begin{array}{l} \texttt{while }\ t_1\ \texttt{ != }\ t_2\texttt{:} \\ \text{indent}\, A \\ B \end{array}$$

for any Py-terms $t_1, t_2$. (Here indent $A$ indents each line of $A$, i.e. is the result of inserting four spaces at the beginning of $A$ and immediately after every `new` character in $A$ except the last character in $A$.)

Again, it's pretty clear this has the inductive property.

# Semantics

## Semantics step one: denotations of terms

An assignment function is a function from some *finite* set of Py-variables to strings.

The denotation of a Py-term is a partial function from assignment functions to strings. This is very similar to the recursive definition of the denotation of terms in first-order languages, but not relative to a structure:

$$[\![v]\!]^g = g(v)$$

$$[\![c]\!]^g = \text{the denotation of } c \text{ in } \mathbb{S}$$

$$[\![\texttt{head(}t\texttt{)}]\!]^g = \text{the first character of } [\![t]\!]^g$$

$$[\![\texttt{tail(}t\texttt{)}]\!]^g = \text{the rest of } [\![t]\!]^g$$

$$[\![\texttt{(}t_1\texttt{+}t_2\texttt{)}]\!]^g = [\![t_1]\!]^g \oplus [\![t_1]\!]^g$$

Some Py-terms like `head("")` and `tail("")` don't denote anything on *any* assignment.

# Semantics step two: Denotations of Py-programs

For every Py-program $A$, $[\![A]\!]$ is a set of pairs of assignment functions. $[\![\cdot]\!]$ is the smallest function from Py-programs to sets of pairs of assignment functions satisfying

(i) The trivial program denotes the identity function: $[\![[]]\!] = \{\langle g, h\rangle \mid g = h\}$.

(ii) If $\langle g, h\rangle \in [\![A]\!]$ and $g = g'[v \mapsto [\![t]\!]^{g'}]$ then $\langle g', h\rangle \in \left[\!\!\left[\begin{array}{l} v\ \texttt{=}\ t \\ A \end{array}\right]\!\!\right]$.

(iii) If $\langle g, h\rangle \in [\![B]\!]$ and $[\![t_1]\!]^g = [\![t_2]\!]^g$ then $\langle g, h\rangle \in \left[\!\!\left[\begin{array}{l} \texttt{while}\ t_1\ \texttt{!=}\ t_2\texttt{:} \\ \quad\quad A \\ B \end{array}\right]\!\!\right]$.

(iv) If $\langle g', g\rangle \in [\![A]\!]$ and $[\![t_1]\!]^{g'} \neq [\![t_2]\!]^{g'}$ then

if $\langle g, h\rangle \in \left[\!\!\left[\begin{array}{l} \texttt{while}\ t_1\ \texttt{!=}\ t_2\texttt{:} \\ \quad\quad A \\ B \end{array}\right]\!\!\right]$ then $\langle g', h\rangle \in \left[\!\!\left[\begin{array}{l} \texttt{while}\ t_1\ \texttt{!=}\ t_2\texttt{:} \\ \quad\quad A \\ B \end{array}\right]\!\!\right]$

# Semantics step two: Denotations of Py-programs

**Functionality of $[\![A]\!]$**

For any Py-Program $A$, $[\![A]\!]$ is a partial function from assignments to assignments.

That is, for no Py-Program $A$ and assignment $g$ are there assignments $h, h'$ with $\langle g, h\rangle \in [\![A]\!]$ and $\langle g, h'\rangle \in [\![A]\!]$ and $h \neq h'$.

Given this, we can write $[\![A]\!]^g = h$ instead of $\langle g, h\rangle \in [\![A]\!]$.

$[\![\cdot]\!]$ is *not* a total function. For example,

$$\left[\!\!\left[\begin{array}{l}\texttt{while "A" != "B":}\\ \quad\quad [\,]\\ [\,]\end{array}\right]\!\!\right] = \varnothing.$$

# Semantics step two: Denotations of Py-programs

*Proof.* By Induction on $A$. Base case: If $A = []$ then only $\langle g, g \rangle \in [\![A]\!]$.

Induction step: (i) Suppose that if $\langle g, h \rangle \in [\![A]\!]$ and $\langle g, h' \rangle \in [\![A]\!]$ then $h = h'$. Then if $\langle g', h \rangle \in \left[\!\!\left[\begin{array}{l} v\ \texttt{=}\ t \\ A \end{array}\right]\!\!\right]$ and $\langle g', h' \rangle \in \left[\!\!\left[\begin{array}{l} v\ \texttt{=}\ t \\ A \end{array}\right]\!\!\right]$ then $\langle g, h \rangle \in [\![A]\!]$ and $\langle g, h' \rangle \in [\![A]\!]$ for $g = g'[v \mapsto [\![t]\!]^{g'}]$, and so by the IH $h = h'$.

(ii) Suppose that if $\langle g, h \rangle \in [\![P]\!]$ and $\langle g, h' \rangle \in [\![P]\!]$ then $h = h'$ for all $g, h, h'$ and $P \in \{A, B\}$. Let $\langle g, h \rangle \in \left[\!\!\left[\begin{array}{l} \texttt{while}\ t_1\ \texttt{!=}\ t_2\texttt{:} \\ \quad\quad A \\ B \end{array}\right]\!\!\right]$ and $\langle g, h' \rangle \in \left[\!\!\left[\begin{array}{l} \texttt{while}\ t_1\ \texttt{!=}\ t_2\texttt{:} \\ \quad\quad A \\ B \end{array}\right]\!\!\right]$.

Then there is a sequence of assignments $g_0, \ldots g_n$ (unique by IH) with $g_0 = g$ and $[\![t_1]\!]^{g_n} = [\![t_2]\!]^{g_n}$ and $\langle g_i, g_{i+1} \rangle \in [\![A]\!]$ for all $i < n$. (We explicitly allow $n = 0$, in which case $[\![t_1]\!]^{g} = [\![t_2]\!]^{g}$.) Then $\langle g_n, h \rangle \in [\![B]\!]$ and $\langle g_n, h' \rangle \in [\![B]\!]$ and so $h = h'$ by the IH.

# Computability and decidability

# Py-computability

A partial function $f$ is *Py-computable* iff there is some Py-program $A$ such that $[\![A]\!]^{[\texttt{input} \mapsto d]}(\texttt{result}) = f(d)$ for all $d$ in the domain of $f$, and $[\![A]\!]^{[\texttt{input} \mapsto d]}$ is undefined otherwise.

A set of strings $Y$ is *Py-decidable* iff the function that maps every member of $Y$ to `True` and every other string to `False` is Py-computable.

A set of strings $Y$ is *Py-semidecidable* iff some partial function whose domain includes $Y$ and that maps all and only the members of $Y$ to `True` is Py-computable.

# The Church-Turing Thesis (Py version)

"Church-Turing thesis": a set of strings is decidable iff it is Py-decidable.

Another version of the Church-Turing thesis: a function is computable iff it is Py-computable.

# Programming a Py-interpreter in Py

# What is a Py-interpreter?

In Py, one can write a *universal* Py-program, also called a Py-interpreter: a Py-program that takes a Py-program $A$ and a starting string $s$ as input, and outputs the result of running $P$ on $s$ if defined and doesn't halt otherwise.

To do this, we need to "represent" whole assignment-functions as single strings. And we'll need to define two key functions for manipulating these representations:

1. `getValue(assignment, variableName)`
2. `updateAssignment(oldAssignment, variableName, newValue)`

# Coding assignment functions as strings

The first thing we need to do this is a pair of computable functions *code* and *decode* such for any string $s$, $\mathit{encode}(s)$ is a comma-free string, and $\mathit{decode}(\mathit{code}(s)) = s$. (There's nothing special here about comma, we could use any character as the separator.) Then we can represent an assignment function

$$[v_1 \mapsto s_1, v_2 \mapsto s_2, \ldots, v_n \mapsto s_n]$$

using the string

$$\text{code}(v_1{:}s_1)\text{,}\text{code}(v_2{:}s_2)\text{,}\cdots\text{,}\text{code}(v_n{:}s_n)\text{,}$$

# Encoding and decoding a string

Encoding and decoding can be done in all sorts of ways. For example, we could use the standard label of a string as its code. Or we could encode by going through a string, replacing every comma with `!c` and every `!` with `!!`, and decode by doing the reverse.

## Getting the value of a variable

To get `assignment`'s value for `variableName`, we go through `assignment` from the left, setting `thisVariable` to the left-most remaining variable and `thisValue` to its value, until `thisVariable` is `variableName`. Then we output `thisValue`.

```
def getValue(assignment, variableName):
    thisVariable  = ""
    while thisVariable != variableName:
        codedFirstEntry = upToFirstComma(assignment)
        firstEntry = decode(codedFirstEntry)
        thisVariable = upToFirstColon(firstEntry)
        thisValue = everythingAfterFirstColon(firstEntry)
        assignment = everythingAfterFirstComma(assignment)
    return thisValue
```

# Updating an assignment

Since we get values of variables by reading the value from the left, we can add update variable assignments by adding new values on the left, and allow junky old values of variables to accumulate to the right of the new values rather than bothering to overwrite them:

```
def updateAssignment(assignment, variableName, newValue):
    newEntry = variableName + ":" newValue
    codedNewEntry = encode(newEntry)
    newAassignment = codedNewEntry + "," assignment
    return newAssignment
```

# Evaluating Py-terms

Next we write a program that gets the denotation of a Py-term on a given (string representation of an) assignment-function.

```
def evaluateTerm(term, g):
 kind = kindOfTerm(term)
 if kind == "variable":
 result = getValue(g, term)
 elif kind == "constant":
 result = getStringFromCons(term)
 elif kind == "head":
 x = innerTermOfHead(term)
 result = head(getValue(g, x))
 elif kind == "tail":
 x = innerTermOfTail(term)
 result = tail(getValue(g, x))
 elif kind == "join":
 x = firstTermOfJoin(term)
 y = secondTermOfJoin(term)
 result = getValue(g, x) + getValue(g, y)
 return result
```

# Our little Py-Interpreter

```
def run(program, g):
    while program != "":
        kind = kindOfProgram(program)
        if kind == "let":
            variable = variableInLetStatement(program)
            term = termInLetStatement(program)
            value = evaluateTerm(term, g)
            g = updateAssignment(g, variable, value)
            program = remainderAfterLetStatement(program)
        elif kind == "while":
            a = firstTermInWhileStatement(program)
            b = secondTermInWhileStatement(program)
            block = blockInWhileStatement(program)
            value1 = evaluateTerm(a, g)
            value2 = evaluateTerm(b, g)
            if value1 != value2:
                 program = block + program
            else:
                 program = remainderAfterWhileStatement(program)
     return g
```

# Our little Py-Interpreter, Part I

Let's break this up into two pieces. We go through our program from the beginning, interpreting one statement at a time until only the empty program is left.

For let-statements, we update `g` for the variable in the let-statement to the denotation of the term in the let-statement.

```
def run(program, g):
    while program != "":
        kind = kindOfProgram(program)
            if kind == "let":
                variable = variableInLetStatement(program)
                term = termInLetStatement(program)
                value = evaluateTerm(term, g)
                g = updateAssignment(g, variable, value)
                program = remainderAfterLetStatement(program)
```

## Our little Py-Interpreter, Part II

For while-statements, we copy the indented part to the beginning of `program` and interpret, until the variables in the while-statement have the same denotation on `g`.

```
            elif kind == "while":
                a = firstTermInWhileStatement(program)
                b = secondTermInWhileStatement(program)
                block = blockInWhileStatement(program)
                value1 = evaluateTerm(a, g)
                value2 = evaluateTerm(b, g)
                if value1 != value2:
                     program = block + program
                else:
                       program = remainderAfterWhileStatement(program)
     return g
```

# Upshots

The partial function $f$ that takes a Py-program $A$ and a string $s$ that codes an assignment, such that $\mathrm{Decode}(f(A,s)) = [\![A]\!]^{(\mathrm{Unzip}(s))}$, is computable.

Hence too: the partial function that takes a 1-program $A$ and a string $s$ and spits back $[\![A]\!]^{[\texttt{input} \mapsto s]}(\texttt{result})$ is also Py-computable, by the following program:

```
g = ""
updateAssignment(g,"input",input2)
g = run(input1, g)
result = getValue(g, "result")
```

# The Halting Problem

# The self-halting problem

Let $H$ be the set of 1-programs (programs of one free variable) that halt when given themselves as input (i.e. $A \in H$ iff $[\![A]\!]^{[\texttt{input} \mapsto A]}$ is defined).

Suppose that $H$ was decidable, i.e. that there's some program *HaltsOnSelf* that, when given a program $A$ as input, sets `result` to `T` if $A \in H$ and `F` otherwise. Then consider the following program:

*HaltsOnSelf*

```
while result != "F":
    
```

When its input is a Py-program $A$, this will halt iff $A$ doesn't halt when given itself as input. But does it halt when given itself as input? It does if it doesn't and it doesn't if it does: contradiction. So there can be no program like *HaltsOnSelf*.

# $B$ halts on input $C$

Corollary: the set of ordered pairs $\langle B, C\rangle$ such that $B$ is a 1-program that halts when given $C$ as input is not decidable. For suppose that it was decided by some 2-program *HaltsOn*. Then we could turn this into a 1-program that decides $H$, as follows:

```
input1 = input
input2 = input
```

*HaltsOn*

Analogies: This is just like...

- given that the set of 1-formulae true of themselves is not definable in $\mathbb{S}$, the set of all pairs of 1-formulae $\langle P, Q\rangle$ such that $P$ is true of $Q$ is not definable in $\mathbb{S}$.
- given that the set of all 1-formulae $T$-provable of themselves is not representable in some consistent, negation-complete $T$, the set of all pairs of 1-formulae $\langle P, Q\rangle$ such that $P$ is $T$-provable of $Q$ is not reprensetable in $T$.

# The set of 0-programs that halt

Harder corollary: the set of all 0-programs that halt is not decidable.

To derive this, we need to show that these two functions are Py-computable:

1. The "labelling" function that, given a string, spits back a closed Py-term whose denotation is that string.
2. The "input-set" function that, given a 1-program $A$ and a closed Py-term $t$, spits back the following 0-program:

   `input = ` $t$
   $A$

# Computing substitution and labelling

We can re-use the standard label function from the language of strings except that when $c$ is the three-character constant for character $a$, $\langle(a : s)\rangle$ should be `(`$c$`+`$\langle s\rangle$`)`, instead of $\oplus$`(`$c$`,`$\langle s\rangle$`)`.

Meanwhile the input-set function is obviously computed by the following program:

```
result = "input = " + input2 + new + input1
```

## Putting it together

So, we can put these things together to get a 2-program Apply that, when given as input a 1-program $A$ and a string $s$, spits back the 0-program "$A\langle s\rangle$", i.e.

`input = ` $\langle s\rangle$
$A$

Note that this 0-program gives the same output on the empty assignment that $A$ gives on the assignment [`input` $\mapsto s$].

And we can also make a 1-program SelfApply that, when given as input a 1-program $A$, spits back the 0-program $A\langle A\rangle$, i.e.

`input = ` $\langle A\rangle$
$A$

Note that this 0-program halts iff $A$ halts given itself as input.

# And finally...

So, if we had a 1-program *Halts* that decides the set of 0-programs that halt, we could turn it into a 1-program that decides the set of 1-programs that halt given themselves as input, as follows:

```
input = SelfApply(input)
```

*Halts*

Hence there can be no such program as *Halts*.