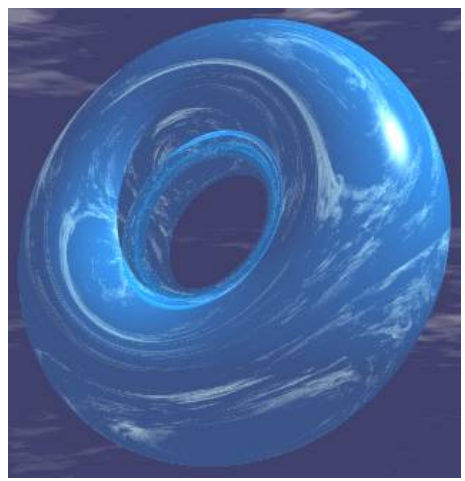
Circularity in Computer Programs

Larry Moss
Indiana University, Bloomington

ESSLLI 2012, Opole



Goals

Up until now, I have

- presented a large number of instances of circularity, some having to do with self-reference, and some with circularly-defined collections
- discussed circularity in sets in connection with the Hypergame Paradox, and also mentioned AFA

On Thursday and Friday, the thread on circularly-defined collections continues with

- category theory basics
- coalgebra

Coming mid-week, this lecture thus is a kind of break.
We'll see

- computer programs that output themselves: a relative of self-reference
- a related key result in the theory of computation, the Recursion Theorem

"It is generally recognized that
the greatest advances in modern computers
came through the notion that
programs could be kept in the same memory with 'data,'
and that programs could operate on other programs,
or on themselves,
as though they were data."

Marvin Minsky
Computation: Finite and Infinite Machines, 1967.

Let L be a computer programming language.
We assume that L is set up so that an input to a program could well be a program.

Theorem

There is a program trade of L such that for all programs p, running trade on p gives the same result as running p on trade.

Let L be a computer programming language.
We assume that L is set up so that an input to a program could well be a program.

Theorem

There is a program trade of L such that for all programs p, running trade on p gives the same result as running p on trade.

For example, if i is a program that computes the identity function on programs, then running trade on i gives trade.

Let L be a computer programming language.
We assume that L is set up so that an input to a program could well be a program.

Theorem

There is a program trade of L such that for all programs p, running trade on p gives the same result as running p on trade.

For example, if I is a program that computes the number of instructions in its input,
then running trade on I gives the number of instructions in trade.

Let L be a computer programming language.
We assume that L is set up so that an input to a program could well be a program.
We need some assumptions on L, but these are pretty weak.

Another goal of the lesson

There is a programming language L
which can be used the same way as a "standard" language
and which has an easy-to-understand semantics,
and: there is a program trade of L such that for all programs p,
running trade on p gives the same result as running p on trade.

Let L be a computer programming language.
We assume that L is set up so that an input to a program could well be a program.
I've asked around and am not aware of anyone fully exhibiting a trade program in any language.

But you'll see one today!

The language is called 1#.
(This is variously read “one hash” or “one sharp”.)

- It is based on a machine model, the text register machine.
- The machine manipulates words, not numbers.
- It us a programming language that has a very clear and simple syntax and semantics.
  It is rather a toy.
- Also, programs in the language are words in the same alphabet as the machine is working with.

I use register machines, a variant of Turing machines.

The machine has registers containing words (i.e., strings) on the two-symbol alphabet $\{1, \#\}$.

Instructions are taken to be fixed words over the same alphabet.

Programs are then just sequences of instructions, run together as strings on $\{1, \#\}$.

Programs will not be readable!

The language 1#: examples of the syntax and semantics

There are five types of instructions, depending on the number of #s at the end:

Instruction	Semantics
1#	Add 1 on the right end of R1
1111#	Add 1 on the right end of R4
111##	Add # on the right end of R3
1111###	Skip forward 4 instructions
11####	Skip backward 2 instructions
1#####	Cases on the leftmost entry in R1

Full set of instructions, along with their semantics

We use superscripts to indicate repetition.
For example, 1^{14} means 11111111111111.

Instruction	Intended meaning
$1^k\#$	Add 1 on the right end of Rk
$1^l\#\#$	Add # on the right end of Rl
$1^p\#\#\#$	Skip forward p instructions
$1^q\#\#\#\#$	Skip backward q instructions
$1^r\#\#\#\#\#$	Cases on the leftmost entry in Rr

Here is how a case statement like 1##### works:

- If R1 is empty, go to the next instruction.
- If R1 begins with a 1, delete it, and go to the second instruction after the case statement.
- If R1 begins with a #, delete it, and go to the third instruction.

Programs are sequences of instructions.

(For the interpreter and the slides, spaces and line breaks do not count.)

The language is uniquely parsable, and regular.

The semantics of a program is pretty much obvious.

A crash course in a programming language

- Write a program to output “Hello World”.
- Write a program to implement a numerical function defined by *recursion*,
  We’ll skip this today.
- Write a program which outputs itself.
- Write a program p which runs other programs in the same language:

  running p on (a program) q and other inputs
  gives the same output as running q on those inputs

- Write a program which trades places with its input:

  running p on q gives the same output as
  running q on p

We don't have letters of the Roman alphabet in 1#,
so we can't directly write "Hello World".

But we can do something similar.

First, let's convert it to Morse code.

We don't have letters of the Roman alphabet in 1#,
so we can't directly write "Hello World".

But we can do something similar.

First, let's convert it to Morse code.
In Morse code, it's

..... – ... – .. – – – . – – – – – – . – .. – .. – ..

Let's agree to use 1 for . and # for –.

So we take "Hello World" to be

111111#111#11###1#####1#11#11#11

We want a program *p* which,
when run with all registers empty,
gives

111111#111#11###1#####1#11#11#11

in register 1.

The program is

1#1#1#1#1#1#1##1#1#1#1##1#1#
1##1##1##1#1##1##1##1##1##
1#1##1#1#1##1#1#1##1#1#

When $i \neq j$, $\mathsf{move}_{i,j}$ moves the contents of Register i onto the end of Register j, emptying Register i.
Here is $\mathsf{move}_{2,1}$:

```
11#####111111###111###1##1111####1#111111####
```

`11#####`	Cases on R2
`111111###`	(when R2 was empty), go forward 6: we're done!
`111###`	(when R2 began with 1), go forward 3
`1##`	(when R2 began with #), add # to R1
`1111####`	Go backward 4 (to the top)
`1#`	Add 1 to R1
`111111####`	Go backward 6 (to the top)

We represent numbers in backwards binary, using # for 0.

n	$bb(n)$
0	#
1	1
2	#1
3	11
4	##1
5	1#1
6	#11
7	111
8	###1
9	1##1

n	$bb(n)$
10	#1#1
11	11#1
12	##11
13	1#11
14	#111
15	1#11
16	####1
17	1###1
18	#1##1
19	11##1

Then we can look up programs which compute familiar operations, such as addition and multiplication.

The language is Turing complete.

The notation $\varphi_p(x) = y$

Notice that programs are different than the functions they compute!

Suppose that when p is a program,
and we run
p with the word x in R1 and all other registers empty,
then the register machine eventually halts with y in R1
and all other registers empty.

If this happens, then we say that $\varphi_p(x) = y$.
(Better notation: $\varphi_p(x) \simeq y$.)

In all other cases (if p isn't a program, or if it is but does not halt with x in R1, or if it halts but not all of the registers besides R1 are empty), then we say that $\varphi_p(x)$ is undefined.

For example,

$$\begin{array}{lcl} \varphi_{1\#}(\#\#11) & = & \#\#111 \\ \varphi_{1\#\#1\#\#}(\#11) & = & \#11\#\# \\ \varphi_1(x) & & \text{is undefined for all } x \end{array}$$

$\varphi_p(\,) = x$ MEANS:

p is a program, and running p with all registers empty
eventually leads to
x in R1 and all other registers empty,
and the program ends one line after the bottom.

For example,

$$\begin{array}{lcl} \varphi_{1\#\#}() & = & \# \\ \varphi_1(\,) & & \text{is undefined} \\ \varphi_{1\#\#\#1\#\#\#\#}(\,) & & \text{is also undefined} \\ & & \text{the run loops forever} \end{array}$$

We also could talk about functions of more than one argument:

$$\varphi_{mult}(11, \#\#1) \quad = \quad \#\#11$$

- Write a program, say self, that outputs itself.
  Find a program self so that $\varphi_{\mathsf{self}}(\,) = \mathsf{self}$.

- Write a program, say u, that can run any program p in 1#.
  Find a program u so that $\varphi_u(p) = \varphi_p(\,)$ for all words p.

- Write a program, say trade, such that running trade on p does the same thing as running p on trade.
  Find trade so that $\varphi_{\mathsf{trade}}(p) = \varphi_p(\mathsf{trade})$ for all words p.

- Write a program, say self, that outputs itself.
  Find a program self so that $\varphi_{\mathsf{self}}(\,) = \mathsf{self}$.
  This has been done explicitly for every programming language: see the "quine" page.

- Write a program, say u, that can run any program p in 1#.
  Find a program u so that $\varphi_u(p) = \varphi_p(\,)$ for all p.
  I have no idea of how many people ever do this in full for any language.

- Write a program, say trade, such that running trade on p does the same thing as running p on trade.
  Find trade so that $\varphi_{\mathsf{trade}}(p) = \varphi_p(\mathsf{trade})$.
  Again, I have no hard information.

A program write which takes an input word X and gives a program that will write X

$\varphi_{\text{write}}(1\#\#11) = 1\#1\#\#1\#\#1\#1\#.$
As an explicit program, write is a simple loop:

```
1#####111111111###11111###11#11##11##111111####
11#11##11111111####1#####111111###111###1##
1111####1#111111####
```

The important property is that

$$\varphi_{\varphi_{\text{write}}(x)}(\,) \quad = \quad x.$$

Let diag be a program with the property that for all p,

$$\varphi_{\mathsf{diag}}(p) \quad = \quad \varphi_{\mathsf{write}}(p) + p.$$

Then for all p,

$$\varphi_{\varphi_{\mathsf{diag}}(p)}(\,) \quad = \quad \varphi_{\varphi_{\mathsf{write}}(p)+p}(\,) \quad = \quad \varphi_p(p)$$

Let diag be a program with the property that for all p,

$$\varphi_{\mathsf{diag}}(p) \quad = \quad \varphi_{\mathsf{write}}(p) + p.$$

Then for all p,

$$\varphi_{\varphi_{\mathsf{diag}}(p)}(\,) \quad = \quad \varphi_{\varphi_{\mathsf{write}}(p)+p}(\,) \quad = \quad \varphi_p(p)$$

Quick exercise

What is

$$\varphi_{\mathsf{diag}}(1\#) \; ??$$

What happens if we run *that* word with all registers empty?

Let diag be a program with the property that for all p,

$$\varphi_{\mathsf{diag}}(p) \quad = \quad \varphi_{\mathsf{write}}(p) + p.$$

Then for all p,

$$\varphi_{\varphi_{\mathsf{diag}}(p)}( ) \quad = \quad \varphi_{\varphi_{\mathsf{write}}(p)+p}( ) \quad = \quad \varphi_p(p)$$

Another quickie

Write a program x so that for all y,

$$\varphi_x(y) \quad = \quad \# + \varphi_{\mathsf{diag}}(y)$$

Let diag be a program with the property that for all p,

$$\varphi_{\text{diag}}(p) \quad = \quad \varphi_{\text{write}}(p) + p.$$

Then for all p,

$$\varphi_{\varphi_{\text{diag}}(p)}(\,) \quad = \quad \varphi_{\varphi_{\text{write}}(p)+p}(\,) \quad = \quad \varphi_p(p)$$

Another quickie

Write a program x so that for all y,

$$\varphi_x(y) \quad = \quad \# + \varphi_{\text{diag}}(y)$$

Answer

$$\text{diag} + \text{move}_{1,2} + 1\#\# + \text{move}_{2,1}$$

Let diag be a program with the property that for all p,

$$\varphi_{\text{diag}}(p) \quad = \quad \varphi_{\text{write}}(p) + p.$$

Then for all p,

$$\varphi_{\varphi_{\text{diag}}(p)}(\,) \quad = \quad \varphi_{\varphi_{\text{write}}(p)+p}(\,) \quad = \quad \varphi_p(p)$$

Let self be the result of running diag on diag itself.
That is,

$$\text{self} \quad = \quad \varphi_{\text{diag}}(\text{diag}).$$

Then

$$\begin{aligned} \varphi_{\text{self}}(\,) \quad &= \quad \varphi_{\varphi_{\text{diag}}(\text{diag})}(\,) \\ &= \quad \varphi_{\text{diag}}(\text{diag}) \\ &= \quad \text{self} \end{aligned}$$

That is, self outputs itself.

It might be useful to expose the device behind self by rendering it into English.

We are interested in "programs" of English (that is, sequences of instructions).
We want instructions of a very simple form, including instructions to print various characters, and instructions which accept one or more sequences of words as arguments, and so on.

We'll allow a small amount of quotation.

Perhaps the most direct example of a self-replicating program would be print me.
But this is not directly available in 1#.

Instead of print me, we want to use print the instructions to print what you see before it.
Here "what you see" and "it" refer to the input, and "before" means "to the left of".

For example, applying the instruction

print the instructions to print what you see before it

to

abc

would give

print "a" print "b" print "c" abc

For another example, applying the instruction

print the instructions to print what you see before it

to

print me

would give

print “p” print “r” print “i” print “n” print “t”
print “ ” print “m” print “e” print me

The idea in language, continued

Applying the instruction

print the instructions to print what you see before it

to

print the instructions to print what you see before it

would give

print “p” print “r” print “i” print “n” print “t” print “ ” print “t” print “h” print “e” print “ ” print “i” print “n” print “s” print “t” print “r” print “u” print “c” print “t” print “i” print “o” print “n” print “s” print “ ” print “t” print “o” print “ ” print “p” print “r” print “i” print “n” print “t” print “ ” print “w” print “h” print “a” print “t” print “ ” print “y” print “o” print “u” print “ ” print “s” print “e” print “e” print “ ” print “b” print “e” print “f” print “o” print “r” print “e” print “ ” print “i” print “t” print the instructions to print what you see before it

If we carry out these instructions, we get the long program back!
This is the English version of self.

Exercise

Find a program p such that

$$\varphi_p(\,) = p + \#$$

In words, running p with all registers empty puts p itself in R1 followed by #.

Hint: use diag and the program 1## which puts # in R1.

Also, remember the key point about diag:

$$\varphi_{\varphi_{\mathrm{diag}}(p)}(\,) \quad = \quad \varphi_p(p)$$

The program p that you want is of the form $\varphi_{\mathrm{diag}}(x)$.
You only need to find x.

Exercise

Find a program p such that
running p with all registers empty puts p itself in R1
and # in R2.

Exercise

Find a program p such that

$$\varphi_p(\,) = \# + p$$

In words, running p with all registers empty puts p itself in R1 preceded by #.

Hint: use diag, 1##, and the move programs.

Getting p so that $\varphi_p(\,) = \# + p$

We expect p to be $\varphi_{\mathsf{diag}}(x)$ for some x.
We know that $\varphi_{\varphi_{\mathsf{diag}}(x)}(\,) = \varphi_x(x)$, and so we want

$$\varphi_x(x) \quad = \quad \# + p \quad = \quad \# + \varphi_{\mathsf{diag}}(x).$$

So all we have to do is to write a program x so that for all y,

$$\varphi_x(y) \quad = \quad \# + \varphi_{\mathsf{diag}}(y).$$

We have seen this x: $\mathsf{diag} + \mathsf{move}_{1,2} + 1\#\# + \mathsf{move}_{2,1}$.

Summary:

$$\varphi_{\mathsf{diag}}(\mathsf{diag} + \mathsf{move}_{1,2} + 1\#\# + \mathsf{move}_{2,1}).$$

You can find more problems in the 1# web text.

Some are quite hard:

- Find two different programs p and q so that

$$\varphi_p(\,) = q \quad \text{and} \quad \varphi_q(\,) = p.$$

- Find a program p so that $\varphi_p(\,) = p$, but p only uses R1.

Many of the problems on self-writing programs may be solved "uniformly", using our next result.

The Recursion Theorem

For every program p of two inputs, there is a program q^* such that

$$\varphi_{q^*}(r) \quad = \quad \varphi_p(q^*, r)$$

for all r.

The Recursion Theorem

For every program p of two inputs, there is a program q^* such that

$$\varphi_{q^*}(r) \quad = \quad \varphi_p(q^*, r)$$

for all r.

Sketch of the proof

Define q_1 in terms of p by
$\mathsf{diag}+\mathsf{move}_{1,2}+\varphi_{\mathsf{write}}(\mathsf{move}_{1,4})+\mathsf{move}_{2,1}+\varphi_{\mathsf{write}}(\mathsf{move}_{4,2}+p)$.
Let $q^* = \varphi_{q_1}(q_1)$.
Check that this choice of q^* works.

Using Kleene's Recursion Theorem to get trade

The Recursion Theorem

For every program p of two inputs, there is a program q^* such that

$$\varphi_{q^*}(r) \quad = \quad \varphi_p(q^*, r)$$

for all r.

We want to apply Kleene's Theorem, using a program p such that $\varphi_p(x, y) = \varphi_y(x)$.

Then let trade be the program q^* from the theorem.

For all y, $\varphi_{\text{trade}}(y) = \varphi_p(\text{trade}, y) = \varphi_y(\text{trade})$.

How can we get p?

There is a program u with the property that
running u with a word x in R1
does the same thing as running x with all registers empty:

$$\varphi_u(x) \quad = \quad \varphi_x(\,).$$

u simulates the action of a register machine inside itself.

It takes a lot of work to write u, and this is a
team project in my classes.

A program p such that $\varphi_p(x, y) = \varphi_y(x)$

We take p to be

$$\text{move}_{2,3} + \text{write} + \text{move}_{3,1} + u.$$

To see that this works, we have a table:

	at start	after $\text{move}_{2,3}$	after write	after $\text{move}_{3,1}$	after u
R1	x	x	$\varphi_{\text{write}}(x)$	$\varphi_{\text{write}}(x) + y$	$\varphi_y(x)$
R2	y				
R3		y	y		

At the end, we use the computation

$$\varphi_u(\varphi_{\text{write}}(x) + y) \quad = \quad \varphi_{\varphi_{\text{write}}(x)+y}(\,) \quad = \quad \varphi_y(x)$$

We have p:

$$\mathsf{move}_{2,3} + \mathsf{write} + \mathsf{move}_{3,1} + u.$$

Following the proof of the Recursion Theorem, we let

$$q_1 = \mathsf{diag} + \mathsf{move}_{1,2} + \varphi_{\mathsf{write}}(\mathsf{move}_{1,4}) + \mathsf{move}_{2,1} + \varphi_{\mathsf{write}}(\mathsf{move}_{4,2} + p).$$

Then we apply q_1 to itself to get trade.

Without further ado, we can have a look at trade.

It has 13,035 instructions.

$\varphi_{\text{trade}}(1\#\#\#) = \text{trade}$.
The computation takes 9,124,755,237 steps.

Without further ado, we can have a look at trade.

It has 13,035 instructions.

Let clear_1 be a program to clear out register 1:

$$1\#\#\#\#\#111\#\#\#11\#\#\#\#111\#\#\#\#$$

$\varphi_{\mathsf{trade}}(\mathsf{clear}_1 + 1\#) = 1$.
The computation takes around 63 billion steps.

Has anything been learned?

I think this is the first explicit trade program in any language. Usually, the existence of trade (and all the other programs in this talk) would be argued by hand-waving.

My sense is that

- The explicit program is of little value to the research community.

  I doubt that it could be published.
- On the other hand, a 100% concrete and treatment of the entire subject should be of pedagogic value.

What we learned today: further comments

- Self-reference in computer programs:
  how it can be done.

- Although programs are usually not written using these tricks, in principle all uses of recursion can be cast as applications of the Recursion Theorem.
  So the topic is of theoretical importance.

- At least one computer scientist, Neil Jones,
  has worked on versions of 1# that are more efficient;
  he believes that this could be valuable.