



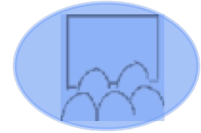
Limits of Computation

10 - Partial evaluation & self-referencing programs
Bernhard Reus



So far ...

- ... we have seen how programs can be encoded as objects to be used as input to other programs.
- Example: self-interpreter to show semi-decidability of Halting Problem.



Question:

Can we write a Java program that prints itself or a While program that returns its own AST (without using its input)?



Other uses of programs-as-data

- **Partial Evaluation** (Kleene's S-m-n Theorem)
 - Use of partial evaluation for Optimisation and Compiler Generation
- **Self-referencing Programs** (Kleene's Recursion Theorem)
 - Use of Recursion Theorem for Recursion Elimination

```
public class Quine
{
    public static void main(String[] args)
    {
        char q = 34; // Quotation mark character
        String[] l = { // Array of source code
            "public class Quine",
            "{",
            "    public static void main(String[] args)",
            "    {",
            "        char q = 34; // Quotation mark character",
            "        String[] l = { // Array of source code",
            "            ",
            "        }",
            "        for(int i = 0; i < 6; i++) // Print opening code",
            "            System.out.println(l[i]);",
            "        for(int i = 0; i < l.length; i++) // Print string array",
            "            System.out.println(l[6] + q + l[i] + q + ',');",
            "        for(int i = 7; i < l.length; i++) // Print this code",
            "            System.out.println(l[i]);",
            "    }",
            "}",
            "}",
        };
        for(int i = 0; i < 6; i++) // Print opening code
            System.out.println(l[i]);
        for(int i = 0; i < l.length; i++) // Print string array
            System.out.println(l[6] + q + l[i] + q + ',');
        for(int i = 7; i < l.length; i++) // Print this code
            System.out.println(l[i]);
    }
}
```

THIS TIME

an interesting Java program: what does it do?

Simple Version of S-m-n Theorem

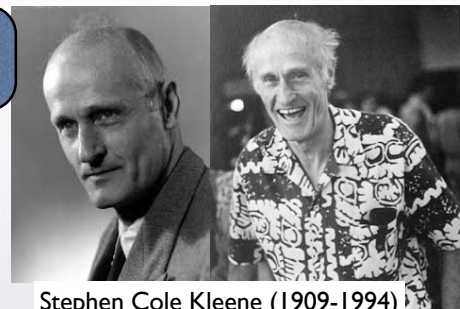
Theorem (S-1-1 Theorem). *For any programming language L with pairing and programs-as-data there exists a program spec such that*

$$\llbracket \llbracket \text{spec} \rrbracket^L(p, s) \rrbracket^L(d) = \llbracket p \rrbracket^L(s, d)$$

Diagram illustrating the S-1-1 Theorem equation with annotations:

- $\llbracket \text{spec} \rrbracket^L(p, s)$: Program with 2 inputs (as pair)
- p : partial (1st) input
- s : remaining (2nd) input
- $\llbracket _ \rrbracket^L$: notation for generic pairing
- (s, d) : both inputs as pair

- It can be generalised to programs with $m+1$ (not just 2) input and providing n concrete values (not just 1) thus the name (Kleene's) S-m-n Theorem.

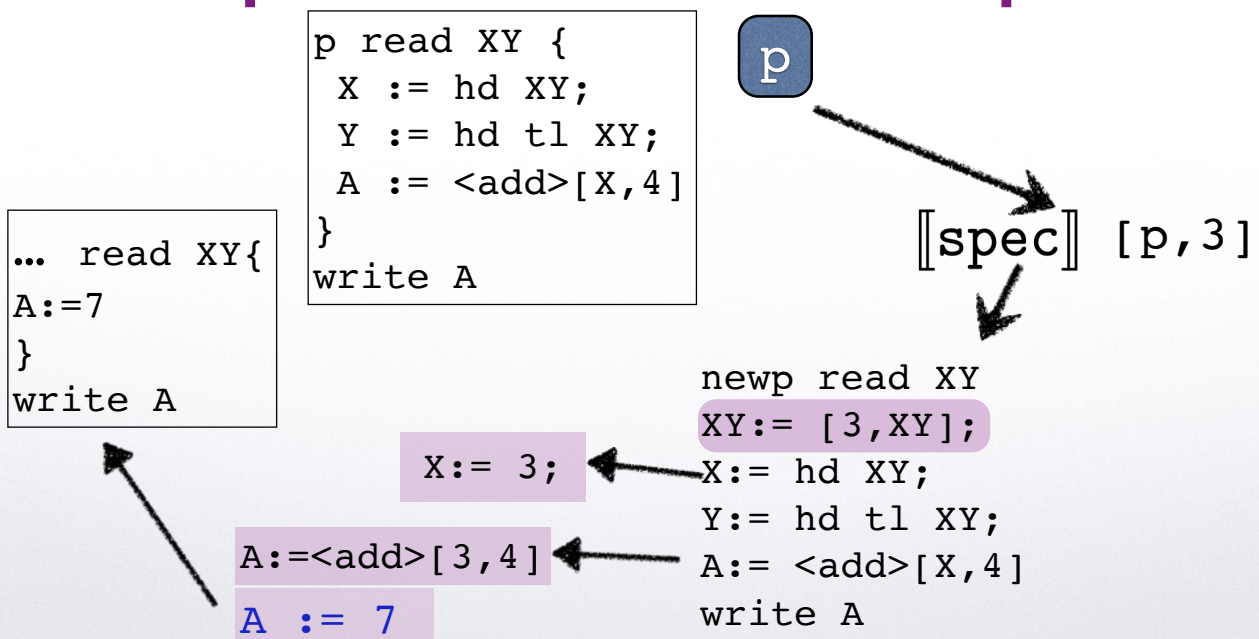


Stephen Cole Kleene (1909-1994)
American mathematician
(also did regular expressions!)

S-m-n describes Specialiser

- The resulting program spec is a specialiser (that takes a program as input and returns a new program).
- The concrete code for a simple spec is relatively straightforward and will be given in notes / as program on Canvas site.
- The partial input provides opportunities for *optimisations* in the code of program p .

Optimisation Example



“Efficient program
specialisation
=
partial evaluation”



Self-referencing Programs

- Can we write programs that refer to their own source or Abstract syntax tree?
- Can we give them a well-defined semantics?
- For instance, a program that prints its own source or returns its own AST?
- Try it, it's not that simple!! What is the problem here?



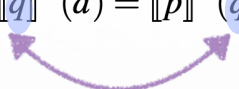
Acceptable Prog. Language

A *programming language* as discussed before, defined by syntax, data type, and semantics, having also programs-as-data and pairing, is called “*acceptable*” iff

- it has a universal program
- it has specialisers (S-m-n theorem holds)
- all functions computed by Turing machines or While-programs are also computed by one of its programs.

Recursion Theorem

Theorem (Kleene's Recursion Theorem). Let L be an acceptable programming language. For any L -program p , there is a L -program q such that for all input $d \in L$ -data we have

$$\llbracket q \rrbracket^L(d) = \llbracket p \rrbracket^L(q, d)$$


- q is the self-referential program
- defined by program p that has extra parameter q
- before we prove it, look at example usage

Eliminate Recursion

Factorial
recursively:
 $0! = 1$
 $n! = n \cdot (n-1)!$

factorial2 has
argument $[q, n]$ so
extra argument q

```
fac2 read qn {  
  q := hd qn;  
  n := hd tl qn;  
  if n {  
    A := <u> [q, tl n];  
    Res := <mult> [n, A]  
  }  
  else {  
    Res := 1  
  }  
}  
write Res
```

u is self-interpret

Recursion Theorem gives us (existence of) `facRef1` such that

$$\llbracket \text{facRef1} \rrbracket^{\text{WHILE}}(d) = \llbracket \text{fac2} \rrbracket^{\text{WHILE}}[\text{facRef1}, d]$$



1st Impact of Theorem

- Any acceptable programming language is already “closed under recursion”. [Neil Jones]
- Usage in interpreted languages: needs a new self-interpreter called for each recursive call, leading to stacks of self-interpreters running each other,
- leading to exponential runtime,
- thus built-in “reflection” mechanisms to avoid this problem.



2nd Impact of Theorem

- Reflection in compiled languages is difficult as there is no source code or AST.
- So no reflection in C or C++.
- But in Java as there is a *ByteCode* interpreter. The **Reflection** library allows to introspect classes/methods.
- This allows one to circumvent static type checking to work with “unknown” classes
- Frameworks like *Spring* heavily use reflection.

“Plug-in” technology
needs reflection!



Kleene's Recursion Theorem semantically validates the principle of reflection in programming



Proof of Recursion Theorem

- Main technique: again self-application!
- We can apply a program to itself and still get a program (S-I-I Theorem i.e. specialiser)

- and can plug that into given program p :

$$f(r, d) = \llbracket p \rrbracket^L (\llbracket \text{spec} \rrbracket^L (r, r), d) \quad (10.1)$$

- f is L -computable by a program p^{self}
 $\llbracket p^{\text{self}} \rrbracket^L = f \quad (10.2)$

Proof of Recursion Theorem

- Now we consider another self-application:

$$q = \llbracket \text{spec} \rrbracket^L (p^{\text{self}}, p^{\text{self}}) \quad (10.3)$$

- which returns an L-program which is our desired program q ;
- it remains to show that for this program q :

$$\llbracket q \rrbracket^L (d) = \llbracket p \rrbracket^L (q, d)$$

Proof of Recursion Theorem

We compute the required equation just by using the definitions we made:

$$\begin{aligned} \llbracket q \rrbracket^L (d) &= \llbracket \llbracket \text{spec} \rrbracket^L (p^{\text{self}}, p^{\text{self}}) \rrbracket^L (d) && \text{use Eq. 10.3, the definition of } q \\ &= \llbracket p^{\text{self}} \rrbracket^L (p^{\text{self}}, d) && \text{use S-1-1 Thm. 10.1 with } p \text{ and } s = p^{\text{self}} \\ &= f(p^{\text{self}}, d) && \text{use Eq 10.2, definition } p^{\text{self}} \\ &= \llbracket p \rrbracket^L (\llbracket \text{spec} \rrbracket^L (p^{\text{self}}, p^{\text{self}}), d) && \text{use Eq. 10.1, definition of } f \\ &= \llbracket p \rrbracket^L (q, d) && \text{use Eq. 10.3, definition } q \text{ backwards} \end{aligned}$$



END

© 2008-25. Bernhard Reus, University of Sussex

Next time:
Why non-computable problems remain
non-computable when using different
notions of “effective procedure”.