Mechanized semantics: when machines reason about their languages

Introduction

Xavier Leroy

2019-11-28

Collège de France, chair of software sciences

The semantics of a programming language

Assigning meaning to programs. (Floyd, 1967)

Less ambitiously: giving an answer to the question

"What does this program do, exactly?"

What does this program do, exactly?

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l              ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

(Raymond Cheong, IOCCC 2001)

What does this program do, exactly?

```
#include <stdio.h>
int l;int main(int o,char **O,
int I){char c,*D=O[1];if(o>0){
for(l=0;D[l              ];D[l
++]-=10){D   [l++]-=120;D[l]-=
110;while   (!main(0,O,l))D[l]
+=   20;   putchar((D[l]+1032)
/20   )   ;}putchar(10);}else{
c=o+     (D[I]+82)%10-(I>l/2)*
(D[I-l+I]+72)/10-9;D[I]+=I<0?0
:!(o=main(c/10,O,I-1))*((c+999
)%10-(D[I]+92)%10);}return o;}
```

(Raymond Cheong, IOCCC 2001)

(It computes square roots in arbitrary precision.)

What about this program?

```
#define _ F-->00 || F-OO--;
long F=00,OO=00;
main()F_OO();printf("%1.3f\n", 4.*-F/OO/OO);F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_
            _-_-_-_
}
```

(Brian Westley, IOCCC 1988)

What about this program?

```
#define _ F-->00 || F-OO--;
long F=00,OO=00;
main()F_OO();printf("%1.3f\n", 4.*-F/OO/OO);F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
       _-_-_-_-_-_-_-_-_
            _-_-_-_
}
```

(Brian Westley, IOCCC 1988)

(It computes an approximation of π)

What about this program?

```
#define crBegin static int state=0; switch(state) { case 0:
#define crReturn(x) do { state=__LINE__; return x; \
                         case __LINE__:; } while (0)
#define crFinish }
```

```
int decompressor(void) {
    static int c, len;
    crBegin;
    while (1) {
        c = getchar();
        if (c == EOF) break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--) crReturn(c);
        } else crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}
```

(Simon Tatham, author of PuTTY)

What about this program?

```
#define crBegin static int state=0; switch(state) { case 0:
#define crReturn(x) do { state=__LINE__; return x; \
                         case __LINE__:; } while (0)
#define crFinish }
```

```
int decompressor(void) {
    static int c, len;
    crBegin;
    while (1) {
        c = getchar();
        if (c == EOF) break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--) crReturn(c);
        } else crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}
```

(Simon Tatham, author of PuTTY)

(It's a decompressor for run-length encoding, written as a co-routine)

Three degrees of semantics

Intuitive semantics:
a well-written program in an appropriate programming language tells a good story and should read easily.

Precise semantics:
reference manuals, ISO standards, and other normative texts.

Formal semantics: (these lectures)
describe the behaviors of programs with absolute mathematical precision.

A brief history
of programming languages
and their semantics

Prehistory: machine language

"It's all zeros and ones!"

```
10111000 00000001 00000000 00000000 00000000
10111010 00000010 00000000 00000000 00000000
00111001 11011010 01111111 00000110
00001111 10101111 11000010
01000010 11101011 11110110
11000011
```

(x86 machine code for the factorial function)

Classical Antiquity (1949): assembly languages

A textual representation of machine language, with mnemonics for instructions, labels for program points, and comments for humans to read.

Example: the factorial function in x86 assembly

```
; On entry: argument N in EBX register
; On exit: factorial(N) in EAX register
factorial:
        mov eax, 1      ; initial result = 1
        mov edx, 2      ; index i = 2
L1:     cmp edx, ebx    ; while i <= N ...
        jg L2
        imul eax, edx   ; multiply result by i
        inc edx         ; increment i
        jmp L1          ; end while
L2:     ret             ; end function
```

A very precise semantics!

Expressed as the effect of every instruction on the processor state. No or few ambiguities if the reader is familiar with hardware architecture.

Instruction Set Architecture

Synergistic Processor Unit

Add Word **Required** **v 1.0**

a **rt,ra,rb**

0	0	0	1	1	0	0	0	0	0	0	RB	RA	RT
0	1	2	3	4	5	6	7	8	9	10	11 12 13 14 15 16 17	18 19 20 21 22 23 24	25 26 27 28 29 30 31

For each of four word slots:

- The operand from register RA is added to the operand from register RB.
- The 32-bit result is placed in register RT.
- Overflows and carries are not detected.

The Renaissance (1957): Fortran

Arithmetic expressions that look like familiar mathematical formulas:

```
D  = SQRT(B*B - 4*A*C)
X1 = (-B + D) / (2*A)
X2 = (-B - D) / (2*A)
```

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

One command for structured control: the counted loop

```
      DO 10 I=1,N
      ...
   10 CONTINUE
```

(Plus `GO TO` and `IF` as in assembly.)

Syntax and semantics are less clear

Lexical conventions are hard to read and prone to errors:

`DO10I=1,20`	loop for `I` from 1 to 20
`DO10I=1.20`	assigning 1.20 to the variable `DO10I`

Precedence and associativity of operators:

`A + B * C`	means	`A + (B * C)`	but not	`(A + B) * C`
`A - B - C`	means	`(A - B) - C`	but not	`A - (B - C)`

The compiler can "associate" `A + B + C` as `(A + B) + C` or as `A + (B + C)` or as `(A + C) + B`. In floating-point, the three interpretations compute different values.

The Enlightenment (1960): Algol

Arithmetic expressions + structured control (with keywords that tell a story: `begin...end`, `if...then...else`, `for...do`, etc).

Procedures and functions to support code reuse:

```
procedure quadratic(x1, x2, a, b, c);
     value a, b, c; real a, b, c, x1, x2;
begin
  real d;
  d := sqrt(b * b - 4 * a * c);
  x1 := (-b + d) / (2 * a);
  x2 := (-b - d) / (2 * a)
end;
```

Which semantics for function calls?

Algol 60 offers two semantics for passing arguments to functions, the two semantics that looked most natural at the time:

- call by value for parameters marked `value`
  ($\approx$ Lisp, C, C++, Java, Caml, $\ldots$)
  ($\approx$ call-by-value λ-calculus)
- copy rule for parameters not marked `value`
  (substituting the argument expression for the function parameter)
  ($\approx$ Lisp macros)
  ($\approx$ call-by-name λ-calculus)

Copy rule + assignments = an explosive mix!

Greatness of the copy rule

A very general function for summation:

```
real procedure Sum(k, l, u, ak)
      value l, u;  integer k, l, u;  real ak;
   begin
      real s;
      s := 0;
      for k := l step 1 until u do
         s := s + ak;
      Sum := s
   end;
```

Sum of squares: `Sum(i, 1, n, i*i)`

Sum of matrix `A`: `Sum(i, 1, m, Sum(j, 1, n, A[i,j]))`

Misery of the copy rule

```
procedure swap(a, b)
    integer a, b;
    begin
        integer temp;
        temp := a;
        a := b;
        b := temp;
    end;
```

This procedure can fail to swap its arguments!

For instance, `swap(i, A[i])` expands to

`temp := i; i := A[i]; A[i] := temp`.

The Enlightenment (1958): Lisp

The first of the functional programming languages:

- Structured around expressions and recursive functions.
- Minimalistic, unambiguous syntax (S-expressions).
- Semantics that is intended to be mathematical from day one: explicit connections with recursive function theory.

  (J. McCarthy, *Towards a Mathematical Science of Computation*, IFIP Congress 1962.)

The semantics of functions turns out to be delicate…

Scope of a variable binding

```
(let  ((x 1))                     ; first binding of x
(flet ((f (y) (+ x y)))           ; function f uses x
(let  ((x "foo"))                 ; second binding of x
   (f 0))))                       ; call to f
```

What is the value of `x` in the body of `f` when we evaluate `f 0`?

- Static ("lexical") scoping: the value of `x` when `f` was defined, that is, `1`. That's what the λ-calculus predicts.
- Dynamic scoping: the value of `x` at the time of the call, that is, `"foo"`. This is what the first Lisp implementations did, but is considered an historical mistake.

Summary

Around 1965, several hundred programming languages already exist. (P. J. Landin, *The next 700 programming languages*, 1966.)

It is known how to formalize their syntax, using grammatical frameworks such as Backus-Naur form (BNF).

The need to formalize their semantics is growing:
the higher-level languages become, the more surprising their (intuitive or precise) semantics become!

A brief history of formal semantics

Three styles of formal semantics

Operational semantics

Formally describe the steps of executing the program.

E.g. by successive reductions (rewrites) of (syntactic) terms.

Example: simplifying arithmetic expressions

$$(1+2) \times (3+4) \quad \to \quad 3 \times (3+4) \quad \to \quad 3 \times 7 \quad \to \quad 21$$

Example: the λ-calculus and its β-reduction

$$(\lambda x.\, M)\, N \quad \to \quad M\{x \leftarrow N\}$$

Operational semantics

Denotational semantics

To each syntactic element of the program, associate a mathematical object that captures its meaning — its *denotation*.

Examples of denotations:

Syntactic element	Denotation
Expression without variables	Its value (a number)
Expression with variables	Function variable values $\mapsto$ expression value
Command without loops	Function variable values “before” $\mapsto$ variable values “after”

Three styles of formal semantics

Operational semantics

Denotational semantics

Axiomatic semantics

Describe the semantics of a program fragment by the logical assertions (*preconditions*, *postconditions*, *invariants*) that it satisfies.

First operational semantics

Peter J. Landin, *The mechanical evaluation of expressions*, The Computer Journal 6(4), 1964.

An "applicative" language based on the λ-calculus

($\approx$ Lisp with static scoping)

Execution model: an *abstract machine* called SECD.

Peter J. Landin, *Correspondence between ALGOL 60 and Church's Lambda-notation*, Comm. ACM 8(2), 8(3), 1965.

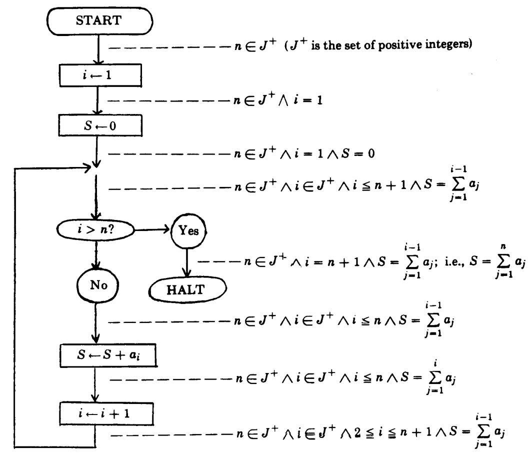
Outline of a translation from Algol 60 to his applicative language + mutable data + continuations ($\approx$ Scheme).

Failed to convince: too complex, not mathematical enough.

Birth of axiomatic semantics

Robert Floyd, *Assigning meaning to programs*, 1967

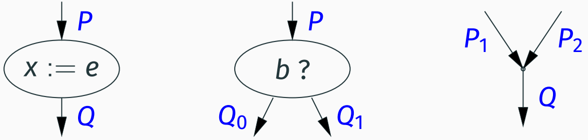
Rediscovers an idea by Turing (1949): to prove a program, it suffices to annotate its flowchart with logical assertion, and to check the consistency of these assertions.



Birth of axiomatic semantics

Robert Floyd, *Assigning meaning to programs*, 1967

Formalizes the logical rules that connect preconditions P and postconditions Q of every node of a flowchart:



$$P \Rightarrow Q\{x \leftarrow e\} \qquad P \land \neg b \Rightarrow Q_0 \qquad P_1 \lor P_2 \Rightarrow Q$$
$$P \land b \Rightarrow Q_1$$

Observes that these rules suffice to define the semantics of any flowchart with mathematical precision.

1969–1980: the golden age of axiomatic semantics

As a tool for program proof: Hoare logic (1969), weakest preconditions calculus (Dijkstra, 1975).

As a development methodology by successive refinements (Wirth, 1971), guarded commands (Dijkstra, 1975).

As a guide to design structured programming languages:

- single-exit commands; no `break`, no `return` (Pascal)
- pure functions vs. procedures with effects (preliminary Ada)

Naive denotational semantics

Christopher Strachey, *Towards a formal semantics*, 1964, 1966.

This text and other notes by Strachey introduce the style of semantics where functions associate a denotation to each syntactic construct.

Expressions: $\mathcal{E} : expr \to env \to val$

$$\mathcal{E}\ x = \lambda e.\, e(x)$$

$$\mathcal{E}\ (a_1\ \texttt{+}\ a_2) = \lambda e.\, \mathcal{E}\ a_1\ e + \mathcal{E}\ a_2\ e$$

Commands: $\mathcal{C} : cmd \to env \to env$

$$\mathcal{C}\ \texttt{skip} = \lambda e.\, e$$

$$\mathcal{C}\ (x\ \texttt{:=}\ a) = \lambda e.\, e\{x \leftarrow \mathcal{E}\ a\ e\}$$

$$\mathcal{C}\ (c_1\texttt{;}\ c_2) = \mathcal{C}\ c_2 \circ \mathcal{C}\ c_1$$

Naive denotatnoial semantics

"*The approach was deliberately informal and, as subsequent events proved, gravely lacking in rigour.*" (Strachey, as quoted by Scott)

Circularity in the equations for loops and for recursive functions:

$$\mathcal{C}\,(\mathtt{while}\;b\;\mathtt{do}\;c) = \lambda e.\begin{cases} e & \text{if } \mathcal{B}\;b\;e = \mathtt{false} \\ \mathcal{C}\,(\mathtt{while}\;b\;\mathtt{do}\;c)\;(\mathcal{C}\;c\;e) & \text{if } \mathcal{B}\;b\;e = \mathtt{true} \end{cases}$$

Ill-defined sets of denotations:
if D is the set of denotations of pure lambda-terms,
we would like to interpret $\lambda x.M$ as a function $D \to D$,
but $D \approx D \to D$ is impossible (wrong cardinality).

Domain theory

Dana Scott, *Outline of a mathematical theory of computation*, 1970

Dana Scott, *Data types as lattices*, 1975.

Partially-ordered sets, from the least defined element ($\bot$) to more defined elements, equipped with a topological structure (limits, continuous functions).

Fit the needs of denotational semantics:

- Semantics of general loops and general recursion as least fixed points (smallest solutions to an equation).
- Precise reasoning about divergence (non-termination).
- Construction of "circular" domains such as $D_\infty \approx D_\infty \to_{cont} D_\infty$.

1975–1990: the golden age of denotational semantics

Extending the “Scott-Strachey approach” to almost all features of known programming languages.
(Including non-structured control, via continuations.)

The semantic formalism most widely used at the *Principles of Programming Languages* conference until around 1990.

Formalization of a few real-world programming languages, including sequential Ada (V. Donzeau-Gouge, J. Storbank Petersen).

The return of operational semantics

Gordon Plotkin, *Call-by-name, call-by-value and the lambda-calculus*, 1975
Robin Milner, *A calculus of communicating systems*, 1980
Gordon Plotkin, *A structural approach to operational semantics*, 1981
Gilles Kahn, *Natural semantics*, STACS, 1987
Matthias Felleisen, Daniel Friedman, *Control operators, the SECD-machine, and the λ-calculus*, 1987

Generalizing the lambda-calculus approach (sequences of reductions) to many other languages (Plotkin, Felleisen)

Using systems of inference rules for operational semantics (Kahn).

Labeled Transition Systems as the first satisfactory semantics for process calculi (Milner).

1990–2010: the golden age of operational semantics

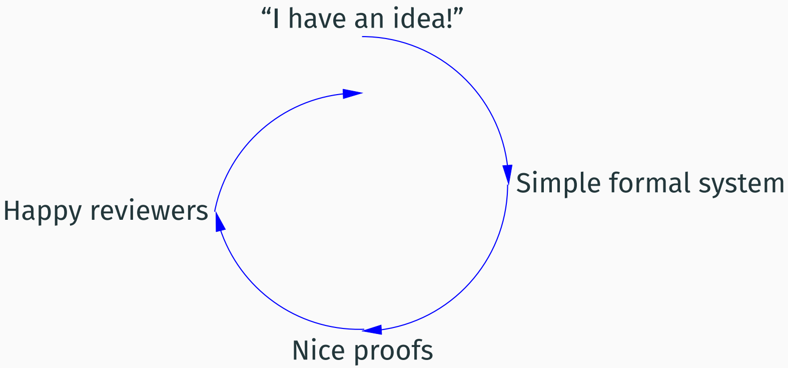
Widely used approach in programming languages research, dominant among POPL papers.

Used to formalize real-world languages:

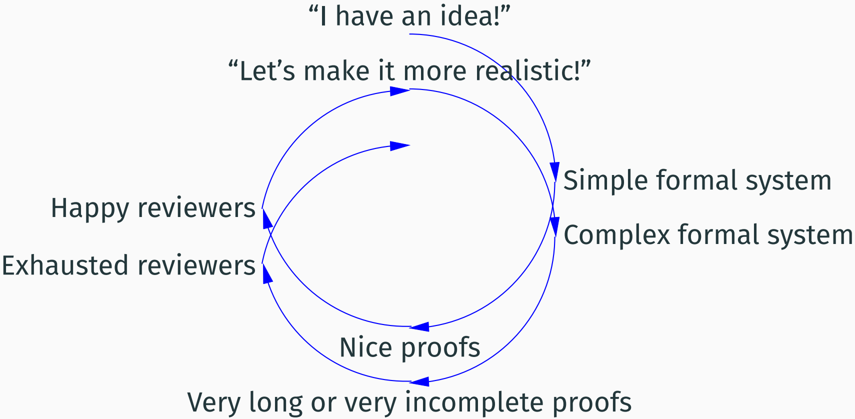
- On paper: *The Definition of Standard ML* (Milner, Tofte, Harper, 1990, 1997).
- On machine: Java (Klein & Nipkow), C (Norrish, Leroy, Krebbers), Javascript (Gardner et al), etc.

Mechanized semantics

A vicious circle



A vicious circle



When proofs become worthless

Proofs written by computer scientists are boring:
they read as if the author is programming the reader.

(John C. Mitchell)

The proofs of the remaining 18 cases are similar and make extensive use of the hypothesis that [. . .]

(anonymous author)

Proof assistants

Computer implementations of mathematical logics.

Provide a specification language (a “mathematical vernacular”) to write definitions and state theorems.

Provide means to build proofs, automatically or in interaction with the user.

Check that the proofs are sound and exhaustive.

Examples: ACL2, Agda, Coq, HOL, Isabelle, Lean, PVS.

The definition of prime numbers:

```
Definition divides (n m: nat) : Prop :=
  exists k, m = k * n.

Definition prime (n: nat) : Prop :=
  n > 1 /\ forall i, divides i n -> i = 1 \/ i = n.
```

There is no largest prime number:

```
Theorem Euclid:
  ~ exists N, forall p, prime p -> p <= N.
Proof.
  ...
Qed.
```

Mechanizing semantics with a proof assistant

Semantics for realistic languages are "big" formal systems (many cases) but "shallow" formal systems (few base concepts).

Proof assistants are very effective at

- handling this "shallow" complexity;
- finding basic mistakes (missing cases, type errors);
- checking the correctness of proofs;
- analyzing the impact of language evolutions;
- making certain definitions executable (for testing).

Course outline

Mechanized semantics

This course is an introduction to the formal semantics of programming languages and to their uses for building and validating programming tools and verification tools:

- type systems;
- program logics;
- static analyzers;
- compilers.

Unified presentation using two “toy” languages:
mostly IMP (imperative), a bit of STLC (functional).

All definitions, properties and proofs are mechanized using the Coq proof assistant.

Do I need to know Coq to take this course?

No, not required to understand the definitions and the main results. (Often stated twice, first in usual mathematics, then in Coq.)

Yes, if you wish to replay and modify the proofs, and to do the exercises.

Course material

Videos and slides on the Collège de France website.

Commented Coq sources on Github:
`https://github.com/xavierleroy/cdf-mech-sem`

Course outline

28/11 Of expressions and commands: the semantics of an imperative language

05/12 Lecture postponed to 06/02

12/12 *Traduttore, traditore*: formal verification of a compiler

19/12 Advanced compilation: optimizations, static analyses, and their verification

09/01 Logics to reason about programs

16/01 Abstract art: static analysis by abstract interpretation

30/01 Eternity is long: divergence, domain theory, coinductive approaches

06/02 Of functions and types: the semantics of a functional language

13/02 Coq in Coq? Mechanizing the logic of a proof assistant

Seminar program

05/12	Seminar postponed to 13/02
12/12	Lambda, the ultimate teaching assistant (Agda version) Philip Wadler (U. Edinburgh)
19/12	L'arithmétique des ordinateurs et sa formalisation Sylvie Boldo (Inria)
09/01	Sémantique formelle de JavaScript Alan Schmitt (Inria)
16/01	Logique de séparation en Coq : théorie et pratique Arthur Charguéraud (Inria)
30/01	Interpréteurs abstraits mécanisés David Pichardie (ENS Rennes)
06/02	Understanding and evolving the Rust language Derek Dreyer (MPI SWS)
13/02	What's in a name? Représenter les variables et leurs liaisons Xavier Leroy

References

References

An introduction to programming language semantics:

- H. R. Nielson and F. Nielson, *Semantics with Applications: an appetizer*, Springer, 2007.

To learn Coq:

- Pierce et al, *Software foundations, vol 1: Logical foundations*, `https://softwarefoundations.cis.upenn.edu/`
- Bertot and Castéran, *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*, Springer-Verlag, 2004.