CS 320: Principles of Programming Languages

Mark P Jones, Portland State University

Spring 2019

Week10: Formalizing Programming Language Semantics

What is a programming language?

- A tool for constructing descriptions of how a computer should behave
- A combination of
  - **Syntax**: Specifying how programs are written, presented to, or read by a computer or a human
  - **Semantics**: Specifying what programs "mean": what effects they have when executed; which function they correspond to; etc...

Describing programming languages

- A programming language can be described in different ways:
  - **Informal descriptions** capture intuitions and basic concepts. But natural language is often incomplete (it doesn't cover all cases) and ambiguous (it can be interpreted in multiple ways)
  - **Implementations** (compilers/interpreters) can be used as specifications:
    - Syntax = what the implementation accepts
    - Semantics = what the implementation does
    - But programs are often cluttered with implementation-specific details and depend on the semantics of the implementation language ...
  - Are there other options to consider?

Formalizing programming languages

- A **formal** description aims to define an entity in a precise, unambiguous manner in terms of simpler, well-understood formalisms such as logic, set theory, abstract machines, or some other branch of mathematics
- **Example**: standard formalisms for syntax include:
  - Regular expressions
  - Finite automata
  - Context-free grammars
  - etc...
- **Example**: standard formalisms for static semantics include:
  - Inference rules
  - Attribute grammars
  - etc...

Why should we care?

- A formal description provides a basis for sharing concepts and expectations:
  - **Between a programmer and an implementation**: the programmer should be able to predict which programs an implementation will accept and what they will do
  - **Between multiple implementations**: different implementations of the same language should accept the same programs and produce the same behavior
- How can you write good programs if the meaning of your programs is not well-defined?
- How can you make effective use of a programming language if the language does not have a well-defined semantics?

... continued

- Formalisms can also be used to prove properties about things that programs won't/can't do:
  - **Example**: a well-typed program should not cause a run-time type error
  - **Example**: a program or script downloaded from an untrusted site should not be able to compromise the machine on which it is running
- Formalisms encourage careful thought and reflection, potentially yielding cleaner, simpler, and more consistent designs

Real world applications

- **CompCert** (Leroy et al.): A verified compiler for (almost all of) the ISO C90/ANSI C language, generating efficient code for the PowerPC, ARM, and x86 processors
  - "By ruling out the possibility of compiler-introduced bugs, verified compilers strengthen the guarantees that can be obtained by applying formal methods to source programs."
- **seL4** (Heiser, Klein, et al.): A formally correct operating system kernel.
  - A small, high-performance microkernel; about 8,700 lines of C code; with a proof (around 200K lines) that "seL4 implements its contract: an abstract, mathematical specification of what it is supposed to do."

Formalizing dynamic semantics

- Standard techniques for specifying dynamic semantics (i.e., the run time behavior of programs) include:
  - Denotational semantics: capture behavior by giving a translation/meaning/denotation of each program construct in a precisely-defined mathematical model
  - Operational semantics: describe behavior in terms of an abstract machine that executes programs
  - Axiomatic semantics: characterize behavior in terms of logical propositions and inference rules

Example: Prop

- Ways to provide a semantics for Prop formulas:
  - Denotational semantics: truth tables
  - Operational semantics: reduction rules
  - Axiomatic semantics: equivalences

Example: regular expressions

- Ways to provide a semantics for regular expressions:
  - Denotational semantics: languages as sets of strings
  - Operational semantics: finite automata, matching
  - Axiomatic semantics: equivalences between regular expressions. r+ = rr*, (r | s) = (s | r), r** = r*, ...

Plan for the rest of this lecture

- A brief taste of each of these approaches
- We will only begin to scratch the surface
- These techniques are widely used in programming language research
- Current state of the art: particularly relevant in systems with critical safety or security requirements; challenging to scale them to real-world problems; but some major steps forward in recent years.

Denotational semantics

Denotational semantics

- Denotational semantics describes the behavior of programs using functions that associate abstract syntax fragments with values ("denotations") in some associated semantic domain

Denotational semantics for Prop

```
eval                  :: Prop -> Env -> Bool

eval (AND p q) env = eval p env && eval q env

eval (OR p q)  env = eval p env || eval q env

eval (NOT p)   env = not (eval p env)

eval TRUE      env = True

eval FALSE     env = False

eval (VAR v)   env
   = case lookup v env of
       Just b  -> b
       Nothing -> error ("No defn for " ++ v)
```


Denotational semantics for regexps

- Every regular expression describes a regular language

$L(\varepsilon) = \{\text{""}\}$

$L(c) = \{\text{"c"}\}$

$L(r_1 \mid r_2) = L(r_1) \cup L(r_2)$

$L(r_1 r_2) = L(r_1)\, L(r_2)$ — $XY = \{ xy \mid x \in X, y \in Y \}$

$L(r*) = L(r)^*$ — $X^* = \{\text{""}\} \cup \{ xy \mid x \in X,\ y \in X^* \}$

$L((r)) = L(r)$

- L(r) is the language "denoted" by r
- This function is an interpreter, mapping a regular expression (syntax) to a set of strings (semantics)

Denotational semantics for grammars

- "We say that a context-free grammar G = (T, N, P, S) **generates** the language that contains all strings in T* that can be derived from S" [Week 2, Slide 61]
- The grammar G denotes the set { s ∈ T* | S derives s }
- This is a denotational semantics that maps syntax (for grammars) to semantics (sets of strings)

Denotational semantics for I(nteger)Exprs

```
data IExpr = IntLit Int
           | Plus IExpr IExpr
           | Minus IExpr IExpr
```

What does one of these expressions denote?

E⟦_⟧ :: IExpr → Int
E⟦n⟧ = N⟦n⟧
E⟦l+r⟧ = E⟦l⟧ + E⟦r⟧
E⟦l−r⟧ = E⟦l⟧ - E⟦r⟧

N⟦_⟧ maps numeric literals to the corresponding integer values

The ⟦...⟧ parentheses used here are sometimes called "Oxford Brackets". The items inside look like concrete syntax … but they represent fragments of abstract syntax and may contain variables that are placeholders for other other AST fragments.

Denotational semantics for IExprs w/ Vars

```
data IExpr = Var String
           | IntLit Int
           | Plus IExpr IExpr
           | Minus IExpr IExpr
```

How do we account for the introduction of variables?

Environments: env ∈ Env = (Var → Int)

E⟦_⟧ :: IExpr → Env → Int
E⟦v⟧ = \env → env(v)
E⟦n⟧ = \env → N⟦n⟧
E⟦l+r⟧ = \env → E⟦l⟧env + E⟦r⟧env
E⟦l−r⟧ = \env → E⟦l⟧env - E⟦r⟧env

Evaluating expressions (in Java)

```
abstract class IExpr { ...
  abstract int eval(Env env);
}
class Var extends IExpr { ...
  int eval(Env env) { return env.lookup(name); }
}
class Int extends IExpr { ...
  int eval(Env env) { return num; }
}
class Plus extends IExpr { ...
  int eval(Env env) { return l.eval(env) + r.eval(env); }
}
class Minus extends IExpr { ...
  int eval(Env env) { return l.eval(env) - r.eval(env); }
}
```


Using denotational semantics

- Denotational semantics can be used to validate laws/ equivalences between program fragments
  - Example: The law `l` = `r` between two expressions is valid if E⟦`l`⟧ = E⟦`r`⟧
- Denotational techniques are widely used in programming language research
- Technical challenge: finding a suitable set of values to model the language of interest
- Proper treatment of real programming languages (e.g., to deal with issues of computability and nontermination) requires sophisticated mathematics (e.g., "domain theory") that is beyond the scope of this course

Operational semantics

Reduction and normalization

- In Prop:

```
OR (AND (NOT a) b) (AND c d)
⟹ OR (AND (NOT TRUE) b) (AND c d)
⟹ OR (AND FALSE b) (AND c d)
⟹ OR (AND FALSE FALSE) (AND c d)
⟹ OR FALSE (AND c d)
⟹ OR FALSE (AND FALSE d)
⟹ OR FALSE (AND FALSE TRUE)
⟹ OR FALSE FALSE
⟹ FALSE
```

- This describes semantics using the reduction of expressions to normal forms where no more reductions are possible

Operational semantics

- Operational semantics describes the meaning of programs in terms of the execution of program fragments and their effect on program "states"
- Generalization the ideas of evaluation by reduction:

```
product [1,2,3,4]
  ⟹ 1 * product [2,3,4]
  ⟹ 1 * 2 * product [3,4]
  ⟹ 1 * 2 * 3 * product [4]
  ⟹ 1 * 2 * 3 * 4 * product []
  ⟹ 1 * 2 * 3 * 4 * 1
  ⟹ 1 * 2 * 3 * 4
  ⟹ 1 * 2 * 12
  ⟹ 1 * 24
  ⟹ 24
```

```
product [] = 1
product (n:ns)
  = n * product ns
```


Evaluation of expressions ("small step")

- We will describe the evaluation of expressions using "judgements" of the form M, e ⟶ M', e' where:
  - M is the initial memory (an environment mapping variables to values)
  - M' is the final memory
  - e is the expression to be evaluated
  - e' is a (partially) evaluated version of e
- This form of semantics allows expressions with side-effects
- General form of rules: $\dfrac{\text{Hypothesis}_1 \quad \ldots \quad \text{Hypothesis}_n}{\text{Conclusion}}$

Examples

$$\frac{M_1, e_1 \longrightarrow M_2, e_2 \quad M_2, e_2 \longrightarrow M_3, e_3}{M_1, e_1 \longrightarrow M_3, e_3}$$

$$\frac{M_1, e \longrightarrow M_2, e'}{M_1, e + f \longrightarrow M_2, e' + f} \qquad \frac{M_1, f \longrightarrow M_2, f'}{M_1, n + f \longrightarrow M_2, n + f'}$$

$$\frac{\text{val } n + \text{val } m = \text{val } t}{M, n + m \longrightarrow M, t}$$

Here, val is a function that maps each numeric literal to the corresponding numeric value

$$\frac{M(v) = n}{M, v{+}{+} \longrightarrow \{ v \mapsto n+1 \} \oplus M, n} \qquad \frac{M(v) = n}{M, {+}{+}v \longrightarrow \{ v \mapsto n+1 \} \oplus M, n+1}$$

Using operational semantics

- An operational semantics gives meaning to program fragments in terms of an abstract/idealized interpreter
- As such, an operational semantics can more easily capture subtleties about how long a computation takes to run, how much memory it uses, etc. than other approaches
- Operational semantics is popular in applications using automated proof assistants because of the opportunities it provides for mechanized evaluation/execution
- Operational semantics is also useful for proving general properties of programming languages
  - Example: if e has type t, and M, e ⟶ M, e' then e' has type t

Axiomatic semantics

Axiomatic semantics

- Axiomatic semantics describes the behavior of programs in terms of logical formulas about program states
- One approach: "Hoare logic" (named after Tony Hoare)
- A Hoare triple {P} s {Q} comprises
  - A precondition, P, that describes the state that the machine should be in before the computation starts
  - A statement, s, to describe the program that we will run
  - A postcondition, Q, that describes the state of the machine after the program is finished, assuming that it terminates
- Example:

```
{x≤12 && even(x)} x = x + 1; {x≤13 && odd(x)}
```


Sample inference rules

$$\frac{\{P\}\ s\ \{Q\} \qquad \{Q\}\ t\ \{R\}}{\{P\}\ s; t\ \{R\}} \qquad \frac{}{\{[e/x]P\}\ x = e;\ \{P\}}$$

$$\frac{\{P \,\&\&\, b\}\ s\ \{Q\} \qquad \{P \,\&\&\, \neg b\}\ t\ \{Q\}}{\{P\}\ \texttt{if (b) s else t}\ \{Q\}}$$

$$\frac{P \Rightarrow P' \qquad \{P'\}\ s\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ s\ \{Q\}} \qquad \frac{\{P \,\&\&\, b\}\ s\ \{P\}}{\{P\}\ \texttt{while (b) s}\ \{P \,\&\&\, \neg b\}}$$

Inference rules and annotated programs

$$\frac{\{P\}\ s\ \{Q\} \qquad \{Q\}\ t\ \{R\}}{\{P\}\ s; t\ \{R\}}$$

```
{ P }
s
{ Q }
t
{ R }
```

$$\frac{P \Rightarrow P' \qquad \{P'\}\ s\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ s\ \{Q\}}$$

```
{ P  }    P ⇒ P'
{ P' }
s
{ Q' }    Q' ⇒ Q
{ Q  }
```

Inference rules and annotated programs

$$\frac{}{\{[e/x]P\}\ x = e;\ \{P\}}$$

```
{[e/x]P}
x = e;
{P}
```

Examples:

```
{ x ≥ 0.5 }{ (2x-1) ≥ 0 } x = 2x-1;   { x ≥ 0 }
 { even(x)} { odd(x+1) } x = x + 1;  { odd(x) }
           { odd(f(y)) } x = f(y);   { odd(x) }
```


Inference rules and annotated programs

$$\frac{\{P\ \&\&\ b\}\ s\ \{Q\} \qquad \{P\ \&\&\ \neg b\}\ t\ \{Q\}}{\{P\}\ \ \texttt{if}\ (b)\ s\ \texttt{else}\ t\ \ \{Q\}}$$

```
{ P }
if (b) {
   { P && b }
   s
   { Q }
} else {
   { P && ¬b }
   t
   { Q }
}
{Q}
```


Inference rules and annotated programs

$$\frac{\{P\ \&\&\ b\}\ s\ \{P\}}{\{P\}\ \ \texttt{while}\ (b)\ s\ \ \{P\ \&\&\ \neg b\}}$$

```
{ P }
while (b) {
   { P && b }
   s
   { P }
}
{ P && ¬b }
```

A formula P that satisfies this property is often referred to as a **loop invariant**

- P is true at the start of the loop
- P is preserved by the body of the loop
- So P must be true if/when the loop terminates

Example

```
while (n>0) {
  m = m + 1;
  n = n - 1;
}
```


Example

```
{ n=N && m=M && n≥0 }
while (n>0) {
  m = m + 1;
  n = n - 1;
}

{ m=N+M }
```

precondition

postcondition

Example

```
{ n=N && m=M && n≥0 }
{ n+m=N+M && n≥0 }
while (n>0) {
  m = m + 1;
  n = n - 1;
}

{ m=N+M }
```

precondition

loop invariant

postcondition

Example

precondition

loop invariant

```
{ n=N && m=M && n≥0 }
{ n+m=N+M && n≥0 }
while (n>0) {
  { n+m=N+M && n≥0 && n>0 }
  m = m + 1;

  n = n - 1;

}


{ m=N+M }
```

postcondition

Example

precondition

loop invariant

```
{ n=N && m=M && n≥0 }
{ n+m=N+M && n≥0 }
while (n>0) {
  { n+m=N+M && n≥0 && n>0 }
  m = m + 1;
  { n+m=N+M+1 && n≥0 && n>0 }
  n = n - 1;

}


{ m=N+M }
```

postcondition

Example

precondition

loop invariant

```
{ n=N && m=M && n≥0 }
{ n+m=N+M && n≥0 }
while (n>0) {
  { n+m=N+M && n≥0 && n>0 }
  m = m + 1;
  { n+m=N+M+1 && n≥0 && n>0 }
  n = n - 1;
  { n+m=N+M && n≥0 }
}


{ m=N+M }
```

loop invariant

postcondition

Example

precondition

loop invariant

```
{ n=N && m=M && n≥0 }
{ n+m=N+M && n≥0 }
while (n>0) {
  { n+m=N+M && n≥0 && n>0 }
  m = m + 1;
  { n+m=N+M+1 && n≥0 && n>0 }
  n = n - 1;
  { n+m=N+M && n≥0 }
}
{ n+m=N+M && n≥0 && ¬(n>0) }

{ m=N+M }
```

loop invariant

postcondition

Example

precondition

loop invariant

```
{ n=N && m=M && n≥0 }
{ n+m=N+M && n≥0 }
while (n>0) {
  { n+m=N+M && n≥0 && n>0 }
  m = m + 1;
  { n+m=N+M+1 && n≥0 && n>0 }
  n = n - 1;
  { n+m=N+M && n≥0 }
}
{ n+m=N+M && n≥0 && ¬(n>0) }
{ n+m=N+M && n=0 }
{ m=N+M }
```

loop invariant

postcondition

List reverse

precondition?

```
r = [];

while (nonEmpty(l)) {

  r = cons(head(l), r);
  l = tail(l);

}
```

loop invariant?

operators on lists:

```
cons(1,[2,3]) = [1,2,3]
head([1,2,3]) = 1
tail([1,2,3]) = [2,3]
```

postcondition?

List reverse

precondition

loop invariant!

```
{ l = xs }
r = [];
{ reverse(xs) = reverse(l) ++ r }
while (nonEmpty(l)) {
  { reverse(xs) = reverse(l) ++ r && l/=[] }
  r = cons(head(l), r);
  l = tail(l);
  { reverse(xs)= reverse(l) ++ r }
}
{ reverse(xs) = reverse(l) ++ r && l=[] }
{ r = reverse(xs) }
```

postcondition

Insertion sort

precondition?

```
r = [];
```

loop invariant?

```
while (nonEmpty(l)) {
  r = insert(head(l), r);
  l = tail(l);


}
```

postcondition?

Insertion sort

precondition

loop invariant!

```
{ l = xs }
r = [];
{ r is sorted &&
    (l ++ r) contains the same elements as xs }
while (nonEmpty(l)) {
  r = insert(head(l), r);
  l = tail(l);
  { r is sorted &&
    (l ++ r) contains the same elements as xs }
}
{ r is sorted &&
    (l ++ r) contains the same elements as xs }
{ r is sorted &&
    r contains same elements as xs }
```

postcondition

Using axiomatic semantics

- The main application for axiomatic semantics is in proving correctness of programs/algorithms
- Some common features of programming languages are notoriously difficult to describe using axiomatic semantics:
  - functions, procedures, ...
  - pointers, aliasing, ...
  - exceptions, ...
- Significant progress has been made in these areas recently
- Practical use of axiomatic semantics is supported by automated proof assistants/theorem provers and verification condition generators

Overall picture (approximate)

```
{ n=N && m=M && n≥0 }
while (n>0) {
  m = m + 1;
  n = n - 1;
}
{ m=N+M }
```

add loop invariants

```
{ n=N && m=M && n≥0 }
while (n>0) {
  { n+m=N+M && n≥0 && n>0 }
  m = m + 1;
  n = n - 1;
}
{ m=N+M }
```

verification conditions (logical formula)

automatic theorem prover ✓

proof assistant ✓

Curious? Try out https://rise4fun.com/Dafny/

Summary

- Formal descriptions of programming languages provide a basis:
  - for establishing the correctness of programming language implementations
  - for reasoning about equivalences between program fragments
  - for proving general properties about programming languages
- Denotational, operational, and axiomatic techniques can all be used to meet this need
- Filling in the details requires some advanced mathematics
- The "price" may be high, but so is the potential "payoff"
- (Intrigued? Maybe further study awaits!)