Compilation and Program Analysis (#3): Semantics, Evaluators, from theory to practice.

Laure Gonnord

http://laure.gonnord.org/pro/teaching/capM1.html

Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

sept 2016

Credits

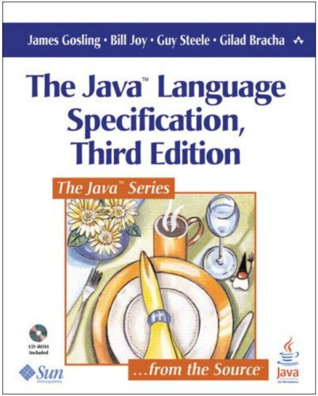
JC Filliâtre (ENS Ulm) / JC Fernandez (Grenoble) /
Nielson-Nielson-Hankin (Book)

Meaning

How to define the meaning of programs in a given language ?

- Informal description most of the time (natural language, ISO, reference book. . .)
- Unprecise, ambiguous.

Informal Semantics



The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right.

It is recommended that code not rely crucially on this specification.

Formal semantics

The formal semantics mathematically characterises the computations done by a given program :

- useful to design tools (compilers, interpreters).
- mandatory to reason about programs.

A bit about syntax

The texts :

```
2*(x+1)
```

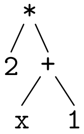
et

```
(2 * ((x) + 1))
```

et

```
2 * (* blablabla *) ( x + 1 )
```

represent the same **abstract syntax tree**.



Example

The grammar of expressions is now :

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{addition} \\
 & | & e \times e & \textit{multiplication} \\
 & | & \ldots &
\end{array}
$$

(avoiding parenthesis, syntactic sugar . . .)

Semantics

On the abstract syntax we will define a/the semantics Different kinds of semantics :

- axiomatic
- dénotational
- by traduction
- operational sémantics (natural, structural)

1. Semantics
   - Different kinds of semantics
   - Operational semantics for mini-while
   - Operational semantics for mini-ml

2. Implementation of evaluators
   - Big picture
   - Old-school way
   - Evaluators with visitors

Axiomatic Semantics (Hoare logic)

(*An axiomatic basis for computer programming*, 1969)

Characterisation by properties on variables, using triples of the form :

$$\{P\}\ i\ \{Q\}$$

"if P is true before the instruction i, then Q is true afterwards"

Example :

$$\{x \geq 0\}\ x := x + 1\ \{x > 0\}$$

Example of generating rule :

$$\{P[x \leftarrow E]\}\ x := E\ \{P(x)\}$$

- See later in the course (proving properties of programs).

Denotational Semantics

Associates to an expression e its mathematical meaning $[\![e]\!]$ that represents its computation.

Example : arithmetic expressions with a unique variable x :

$$e ::= x \mid n \mid e + e \mid e * e \mid \dots$$

Associates to x the value of the expression.

$$\begin{aligned}
[\![x]\!] &= x \mapsto x \\
[\![n]\!] &= x \mapsto n \\
[\![e_1 + e_2]\!] &= x \mapsto [\![e_1]\!](x) + [\![e_2]\!](x) \\
[\![e_1 * e_2]\!] &= x \mapsto [\![e_1]\!](x) \times [\![e_2]\!](x)
\end{aligned}$$

Semantics by translation

(or Strachey denotational semantics)

We can define the semantics of a language by translation into a language whose semantics is already known.

Operational Semantics

Steps of (more or less) elementary computations from the expression to its value. Operates directly on the abstract syntax.

2 kinds :

- "natural" or "*big-steps semantics*"

$$e \xrightarrow{v} v$$

- "by reduction" or "*small-steps semantics*"

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow v$$

1. Semantics
   - Different kinds of semantics
   - Operational semantics for mini-while
   - Operational semantics for mini-ml

2. Implementation of evaluators
   - Big picture
   - Old-school way
   - Evaluators with visitors

mini-While

(abstract) grammar :

$S(Smt)$	$::=$	$x := expr$	assign
	$\mid$	$skip$	do nothing
	$\mid$	$S_1; S_2$	sequence
	$\mid$	$\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2$	test
	$\mid$	$\texttt{while}\ b\ \texttt{do}\ S\ \texttt{done}$	loop

Semantics of expressions

We denote $State = Var \to \mathbf{Z}$. States are denoted by σ.
Substitution is denoted by $\sigma[y \mapsto x]$.

Now arithmetic expressions : $\mathcal{A} \to (State \to \mathbf{Z})$ (in each state an integer value) : **complete the slide !**

$$\mathcal{A}[n]\sigma = \mathcal{N}(n)$$
$$\mathcal{A}[x]\sigma = \sigma(x)$$

Semantics of boolean expressions

$\mathcal{B} \to (State \to \mathbf{Z})$ **complete the slide !**

Natural semantics (big step) for mini-while 1/2

Statements : $Stm \to (State \to State)$

$$(x := a, \sigma) \to \sigma[x \mapsto \mathcal{A}[a]\sigma]$$

$$(\texttt{skip}, \sigma) \to \sigma$$

$$\frac{(S_1, \sigma) \to \sigma' \ (S_2, \sigma') \to \sigma''}{\big((S_1; S_2), \sigma\big) \to \sigma''}$$

Natural semantics (big step) for mini-while 2/2

$$\text{if } \mathcal{B}[b]\sigma = tt : \frac{(S_1, \sigma) \rightarrow \sigma'}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \rightarrow \sigma'}$$

$$\text{if } \mathcal{B}[b]\sigma = \mathit{ff} : \frac{(S_2, \sigma) \rightarrow \sigma'}{(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2, \sigma) \rightarrow \sigma'}$$

$$\text{if } \mathcal{B}[b]\sigma = tt : \frac{(S, \sigma) \rightarrow \sigma', (\texttt{while } b \texttt{ do } S, \sigma') \rightarrow \sigma''}{(\texttt{while } b \texttt{ do } S, \sigma) \rightarrow \sigma''}$$

$$\text{if } \mathcal{B}[b]\sigma = \mathit{ff} : (\texttt{while } b \texttt{ do } S, \sigma) \rightarrow \sigma$$

Example

Compute the semantics (leaves are axioms, nodes are rules) of :

- $x := 2; \texttt{while}\ x > 0\ \texttt{do}\ x := x - 1\ \texttt{done}$
- $x := 2; \texttt{while}\ x > 0\ \texttt{do}\ x := x + 1\ \texttt{done}$

Determinism

Theorem

For all S, for all $\sigma, \sigma', \sigma''$:

- If $(S, \sigma) \rightarrow \sigma'$ and $(S, \sigma) \rightarrow \sigma''$ then $\sigma' = \sigma''$.
- If $(S, \sigma) \rightarrow \sigma'$, there is no infinite derivation.

Proof by induction on the structure of the derivation tree.

Structural Op. Semantics (small steps) for mini-while "SOS" 1/2

$$(x := a, \sigma) \Rightarrow \sigma[x \mapsto \mathcal{A}[a]\sigma]$$

$$(\texttt{skip}, \sigma) \Rightarrow \sigma$$

$$\frac{(S_1, \sigma) \Rightarrow \sigma'}{((S_1; S_2), \sigma) \Rightarrow (S_2, \sigma')} \quad \frac{(S_1, \sigma) \Rightarrow (S_1', \sigma')}{((S_1; S_2), \sigma) \Rightarrow (S_1'; S_2, \sigma')}$$

$$\mathcal{B}[b]\sigma = \mathit{ff} : (\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma) \Rightarrow (S_2, \sigma)$$

$$\mathcal{B}[b]\sigma = \mathit{tt} : (\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, \sigma) \Rightarrow (S_1, \sigma)$$

Structural Op. Semantics (small steps) for mini-while "SOS" 2/2

$$(\texttt{while}\ b\ \texttt{do}\ S, \sigma) \Rightarrow$$

$$(\texttt{if}\ b\ \texttt{then}\ (S; \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip}), \sigma)$$

Example

Compute the semantics (leaves are axioms, nodes are rules) of :

- $x := 2; \texttt{while}\ x > 0\ \texttt{do}\ x := x - 1\ \texttt{done}$
- $x := 2; \texttt{while}\ x > 0\ \texttt{do}\ x := x + 1\ \texttt{done}$

Comparison : divergence

A program diverge in state σ iff :

- NAT : no successor to (S, σ).
- SOS : infinite sequence begining with (S, σ).

Comparison : equivalence of programs

S_1 and S_2 are semantically equivalent iff :

- NAT : $\forall \sigma, \sigma', (S_1, \sigma) \rightarrow \sigma'$ iff $(S_2, \sigma) \rightarrow \sigma'$
- SOS : $\forall \sigma$:
  - for all config (blocking or not) : $(S_1, \sigma) \Rightarrow^* \gamma$ iff $(S_2, \sigma) \Rightarrow^* \gamma$
  - there exists an infinite sequence from (S_1, σ) iff same for (S_2, σ)

Semantic functions

$$\mathcal{S}_{NS}[S]\sigma = \begin{cases} \sigma' & \text{If } (S,\sigma) \rightarrow \sigma' \\ undef & \text{else} \end{cases}$$

$$\mathcal{S}_{SOS}[S]\sigma = \begin{cases} \sigma' & \text{If } (S,\sigma) \Rightarrow^* \sigma' \\ undef & \text{else} \end{cases}$$

Theorem

$$\mathcal{S}_{NS} = \mathcal{S}_{SOS}$$

Equivalence of semantics 1/2

Proposition

If $(S, \sigma) \rightarrow \sigma'$ then $(S, \sigma) \Rightarrow \sigma'$.

Lemma for Proposition

If $(S_1, \sigma) \Rightarrow^k \sigma'$ then $((S_1; S_2), \sigma) \Rightarrow^k (S_2, \sigma')$

Demo : structural induction on S (derivation tree).

Equivalence of semantics 2/2

Proposition

If $(S, \sigma) \Rightarrow^k \sigma'$ then $(S, \sigma) \rightarrow \sigma'$.

Lemma for Proposition

If $(S_1; S_2, \sigma) \Rightarrow^k \sigma''$ then there exists σ', k_1 such that $(S_1, \sigma) \Rightarrow^{k_1} \sigma'$ and $(S_2, \sigma) \Rightarrow^{k-k_1} \sigma''$

Demo : induction on k.

Expressing parallelim

SOS can express interleaving, NAT cannot :

$$\frac{(S_1,\sigma)\rightarrow\sigma',\big(S_2,\sigma'\big)\rightarrow\sigma'')}{(S_1||S_2,\sigma)\rightarrow\sigma''}\quad\frac{(S_2,\sigma)\rightarrow\sigma',\big(S_1,\sigma'\big)\rightarrow\sigma''}{(S_1||S_2,\sigma)\rightarrow\sigma''}$$

$$\frac{(S_1,\sigma)\Rightarrow(S_1',\sigma')}{\big((S_1||S_2),\sigma\big)\Rightarrow(S_1'||S_2,\sigma')}\quad\frac{(S_2,\sigma)\Rightarrow(S_2',\sigma')}{\big((S_1||S_2),\sigma\big)\Rightarrow(S_1||S_2',\sigma')}$$

1. Semantics
   - Different kinds of semantics
   - Operational semantics for mini-while
   - Operational semantics for mini-ml

2. Implementation of evaluators
   - Big picture
   - Old-school way
   - Evaluators with visitors

Mini-ML

Same game for *mini-ML* :

e	$::=$	x	identificator
	$\mid$	c	constant (1, 2, ..., *true*, ...)
	$\mid$	op	primitive ($+$, $\times$, *fst*, ...)
	$\mid$	$\texttt{fun}\ x \rightarrow e$	function
	$\mid$	$e\ e$	application
	$\mid$	(e, e)	pair
	$\mid$	$\texttt{let}\ x = e\ \texttt{in}\ e$	local binding

Examples

$$\texttt{let}\ \textit{compose} = \texttt{fun}\ f \rightarrow \texttt{fun}\ g \rightarrow \texttt{fun}\ x \rightarrow f\ (g\ x)\ \texttt{in}$$
$$\texttt{let}\ plus = \texttt{fun}\ x \rightarrow \texttt{fun}\ y \rightarrow +\ (x, y)\ \texttt{in}$$
$$\textit{compose}\ (\textit{plus}\ 2)\ (\textit{plus}\ 4)\ 36$$

$$\texttt{let}\ \textit{distr\_pair} = \texttt{fun}\ f \rightarrow \texttt{fun}\ p \rightarrow (f\ (\textit{fst}\ p), f\ (\textit{snd}\ p))\ \texttt{in}$$
$$\texttt{let}\ p = \textit{distr\_pair}\ (\texttt{fun}\ x \rightarrow x)\ (40, 2)\ \texttt{in}$$
$$+\ (\textit{fst}\ p, \textit{snd}\ p)$$

Big steps operational semantics for miniML

We want to define the following relation :

$$e \xrightarrow{v} v$$

Abstract syntax for values :

$$\begin{array}{rcll} v & ::= & c & \text{constant} \\ & | & op & \text{primitive} \\ & | & \texttt{fun}\ x \rightarrow e & \text{function} \\ & | & (v, v) & \text{pair} \end{array}$$

Natural semantics of mini-ML

$$\frac{}{c \xrightarrow{v} c} \qquad \frac{}{op \xrightarrow{v} op} \qquad \frac{}{(\texttt{fun}\ x \to e) \xrightarrow{v} (\texttt{fun}\ x \to e)}$$

$$\frac{e_1 \xrightarrow{v} v_1 \quad e_2 \xrightarrow{v} v_2}{(e_1, e_2) \xrightarrow{v} (v_1, v_2)} \qquad \frac{e_1 \xrightarrow{v} v_1 \quad e_2[x \leftarrow v_1] \xrightarrow{v} v}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \xrightarrow{v} v}$$

$$\frac{e_1 \xrightarrow{v} (\texttt{fun}\ x \to e) \quad e_2 \xrightarrow{v} v_2 \quad e[x \leftarrow v_2] \xrightarrow{v} v}{e_1\ e_2 \xrightarrow{v} v}$$

- **call by value !**

Primitive semantics for mini-ML

$$\frac{e_1 \xrightarrow{v} + \quad e_2 \xrightarrow{v} (n_1, n_2) \quad n = n_1 + n_2}{e_1\ e_2 \xrightarrow{v} n}$$

$$\frac{e_1 \xrightarrow{v} fst \quad e_2 \xrightarrow{v} (v_1, v_2)}{e_1\ e_2 \xrightarrow{v} v_1}$$

Derivation example

$$
\dfrac{
  \dfrac{
    + \xrightarrow{v} + \quad \dfrac{20 \xrightarrow{v} 20 \quad 1 \xrightarrow{v} 1}{(20,1) \xrightarrow{v} (20,1)}
  }{
    +(20,1) \xrightarrow{v} 21
  }
  \quad
  \dfrac{
    \texttt{fun}\ldots \xrightarrow{v} \quad 21 \xrightarrow{v} 21 \quad \dfrac{\vdots}{+(21,21) \xrightarrow{v} 42}
  }{
    (\texttt{fun}\ y \rightarrow +(y,y))\ 21 \xrightarrow{v} 42
  }
}{
  \texttt{let}\ x = +(20,1)\ \texttt{in}\ (\texttt{fun}\ y \rightarrow +(y,y))\ x \xrightarrow{v} 42
}
$$

1. Semantics

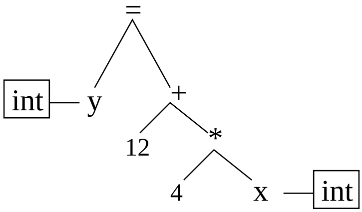
2. Implementation of evaluators
   - Big picture
   - Old-school way
   - Evaluators with visitors

1. Semantics
   - Different kinds of semantics
   - Operational semantics for mini-while
   - Operational semantics for mini-ml
2. Implementation of evaluators
   - Big picture
   - Old-school way
   - Evaluators with visitors

Steps

- Construct the derivation tree wrt the abstract syntax (AST).
- Evaluate the tree wrt the (operational/contructive) semantics.

Abstract Syntax Tree



- AST : memory representation of a program ;
- Node : a language construct ;
- Sub-nodes : parameters of the construct ;
- Leaves : usually constants or variables.

Separation of concerns

- The semantics of the program could be defined in the semantic actions (of the grammar). Usually though :
  - Syntax analyzer only produces the AST ;
  - The rest of the compiler directly **works with this AST**.
- Why ?
  - Manipulating a tree (AST) is easy (recursive style) ;
  - Separate language syntax from language semantics ;
  - During later compiler phases, we can assume that the AST is **syntactically correct** $\Rightarrow$ simplifies the rest of the compilation.

Running example : semantics for numerical expressions

$$
\begin{array}{rcll}
e & ::= & c & \textit{constant} \\
 & | & x & \textit{variable} \\
 & | & e + e & \textit{add} \\
 & | & e \times e & \textit{mult} \\
 & | & \ldots &
\end{array}
$$

1. Semantics
   - Different kinds of semantics
   - Operational semantics for mini-while
   - Operational semantics for mini-ml
2. Implementation of evaluators
   - Big picture
   - Old-school way
   - Evaluators with visitors

Explicit construction of the AST

- Declare a type for the abstract syntax.
- Construct instances of these types during parsing (trees).
- Evaluate with tree traversal.

Example in OCaml 1/3

Types for the abstract syntax :

```
type binop = Add | Mul | ...

type expr_e =
  | Cte of int
  | Var of string
  | Bin of binop * expression * expression
  | ...
```

Example in OCaml 2/3

Pattern matching in parsing rules :

```
%type<Mysyntax.expr_e>  expr

  expr:
  INT                              {Cte (Int64.of_string $1)}
  | LPAREN expr RPAREN             { $2 }
  | expr PLUS expr                 { Bin(Add, $1, $3) }
  | var                            { Var ($1) }
```

Example in OCaml 3/3

Tree traversal with pattern matching (for expression eval) :

```
let rec eval sigma = function
  | Cte(i) -> i
  | Bin(bop,e1,e2) -> let num1= eval sigma e1
                      and num2 = eval sigma e2 in ....
  | Var(s) -> Hashtbl.find sigma s
```

- we need σ, the environnement (implemented with a map).

See the evaluator order, we made a choice !

Example in Java 1/3

AST definition in Java : one class per language construct.

```
public class APlus extends AExpr {
    AExpr e1,e2;

    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }

}
public class AMinus extends AExpr { ...
```

Example in Java 2/3

The parser builds an AST instance using AST classes defined previously.

ArithExprASTParser.g4

```
parser grammar ArithExprASTParser ;
options {tokenVocab=ArithExprASTLexer;}

prog returns [ AExpr e ] : expr EOF { $e=$expr.e; } ;

// We create an AExpr instead of computing a value
expr returns [ AExpr e ] :
  LPAR x=expr RPAR { $e=$x.e; }
| INT { $e=new AInt($INT.int); }
| e1=expr PLUS e2=expr { $e=new APlus($e1.e,$e2.e); }
| e1=expr MINUS e2=expr { $e=new AMinus($e1.e,$e2.e); }
;
```

Example in Java 3/3

Evaluation is an eval function per class :

AExpr.java

```
public abstract class AExpr {
    abstract int eval(); // need to provide semantics
}
```

APlus.java

```
public class APlus extends AExpr {
    AExpr e1,e2;
    public APlus (AExpr e1,AExpr e2) { this.e1=e1; this.e2=e2; }
    // semantics below
    int eval() { return (e1.eval()+e2.eval()); }
}
```

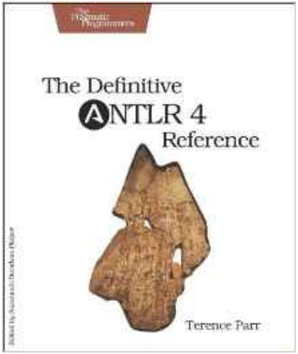
1. Semantics
   - Different kinds of semantics
   - Operational semantics for mini-while
   - Operational semantics for mini-ml
2. Implementation of evaluators
   - Big picture
   - Old-school way
   - **Evaluators with visitors**

Principle - OO programming

The visitor design pattern is a way of separating an algorithm from an object structure on which it operates.[...] In essence, the visitor allows one to add new virtual functions to a family of classes without modifying the classes themselves ; instead, one creates a visitor class that implements all of the appropriate specializations of the virtual function.

`https://en.wikipedia.org/wiki/Visitor_pattern`

Application



Designing evaluators / tree traversal in ANTLR-Python

- The ANTLR compiler generates a Visitor class.
- We override this class to traverse the parsed instance.

Example with ANTLR/Python 1/3

AritParser.g4

```
expr:
        expr mdop expr      #multiplicationExpr
    | expr pmop expr      #additiveExpr
    | atom                          #atomExpr
    ;

atom
    :   INT                 #int
    |   ID                  #id
    |   '(' expr ')'        #parens
```

- compilation with `-Dlanguage=Python2 -visitor`

Example with ANTLR/Python 2/3 -generated file

```
class AritVisitor(ParseTreeVisitor):
...
    # Visit a parse tree produced by AritParser#
        multiplicationExpr.
    def visitMultiplicationExpr(self, ctx):
        return self.visitChildren(ctx)


    # Visit a parse tree produced by AritParser#atomExpr.
    def visitAtomExpr(self, ctx):
        return self.visitChildren(ctx)


..
```

Example with ANTLR/Python 3/3

Visitor class overriding to write the evaluator :

MyAritVisitor.py

```
class MyAritVisitor(AritVisitor):
        # Visit a parse tree produced by AritParser#int.
    def visitInt(self, ctx):
        value = int(ctx.getText());
        return value;

    def visitMultiplicationExpr(self, ctx):
        leftval = self.visit(ctx.expr(0))
        rightval =  self.visit(ctx.expr(1))
        myop = self.visit(ctx.mdop())
        if ( myop == '*'):
            return leftval*rightval
        else:
            return leftval/rightval
```

From grammars to evaluators

- Operational semantics give recursive rules that can be implemented using different programming pattern.
- All the accompagning artefacts (here, σ) have to be implemented as (external) data structures.

Labs :

- Mini project (Lab3) : An evaluator for a Patchwork language.
- Lab4 : evaluator for mini-while.