defining semantics for complex systems part 2: semantics

Pieter Koopman, Rinus Plasmeijer

Radboud University Nijmegen
The Netherlands

syntax & semantics

- the syntax tells what is allowed to write in a (programming) language

  exp = var | num | exp + exp | exp - exp | exp * exp
  var = char alpha*
  num = [-] digit+

  - e.g. x + 32 is allowed
    fac 5 is not allowed
- the semantics tells what valid sentences mean
  - we are not interested in the semantics of invalid sentences (not error, undefined, ..)
  - if we know that x ↦ 10 (x has the value 10), the expression x + 4*8 has value 42
  - fac 5 has no value in this language

different kinds of semantics

- operational semantics:
  how has the value of a sentence to be computed
  - hides details like storage allocation
  - structural operational semantics (small step)
    focus on individual computation steps
  - natural semantics (big step)
    hides more details, computes values in one go
- denotational semantics:
  gives the value of constructs without worrying how it has to be obtained
- algebraic semantics:
  gives algebraic properties of sentences
  - not necessarily complete

semantics for imperative language

- consider the very simple language While

  v a variable
  n a number
  a = v | n | a + a | a - a | a * a
  b = TRUE | FALSE | a = a | a < a | ¬ b | b && b
  S = x := a | skip | S ; S | if b S else S | while b S

  - for instance a statement to compute factorial of 4:

```
x := 4;
y := 1;
while (x>1)
(  y := y*x;
   x := x-1
)
```

the state in semantics

- in order to compute values we need to know the values of variables
- we store values in a function called state:
  state : Variable → Integer
- the state can be updated:
  $[x \mapsto v]$ s is the state that maps variable x to value v and all other variables to the value in s:
  $([x \mapsto v]\ s)\ x = v$
  $([x \mapsto v]\ s)\ y = s\ y$, if $x \neq y$

the semantics of arithmetic expressions

- use Scott brackets, ⟦ and ⟧,

  to indicate a pattern math on syntax elements in an operational semantics

$\mathcal{A}$: a → State → Number
$\mathcal{A}$⟦ n ⟧ s = $\mathcal{N}$⟦ n ⟧ — number
$\mathcal{A}$⟦ v ⟧ s = s v — variable
$\mathcal{A}$⟦ a_1 + a_2 ⟧ s = $\mathcal{A}$⟦ a_1 ⟧ s + $\mathcal{A}$⟦ a_2 ⟧ s
$\mathcal{A}$⟦ a_1 - a_2 ⟧ s = $\mathcal{A}$⟦ a_1 ⟧ s - $\mathcal{A}$⟦ a_2 ⟧ s
$\mathcal{A}$⟦ a_1 * a_2 ⟧ s = $\mathcal{A}$⟦ a_1 ⟧ s × $\mathcal{A}$⟦ a_2 ⟧ s

syntax

mathematical operation

executable operational semantics

- for a functional programmer:

$\mathcal{A}$: a → State → Number
$\mathcal{A}$⟦ n ⟧ s = $\mathcal{N}$⟦ n ⟧
$\mathcal{A}$⟦ v ⟧ s = s v
$\mathcal{A}$⟦ a_1 + a_2 ⟧ s = $\mathcal{A}$⟦ a_1 ⟧ s + $\mathcal{A}$⟦ a_2 ⟧ s
$\mathcal{A}$⟦ a_1 - a_2 ⟧ s = $\mathcal{A}$⟦ a_1 ⟧ s - $\mathcal{A}$⟦ a_2 ⟧ s
$\mathcal{A}$⟦ a_1 * a_2 ⟧ s = $\mathcal{A}$⟦ a_1 ⟧ s × $\mathcal{A}$⟦ a_2 ⟧ s

pattern match: hence a data structure

a function

representation of expressions

the grammar

```
a
  = v
  | n
  | a + a
  | a - a
  | a * a
```

the data type

```
:: AExpr
   = Int Int
   | Var Var
   | (+.) infixl 6 AExpr AExpr
   | (-.) infixl 6 AExpr AExpr
   | (*.) infixl 7 AExpr AExpr
:: Var :== String
```

dot to avoid name conflicts

infix constructor with binding power

semantic functions for arithmetic expressions

Scott brackets

$\mathcal{A} : a \to \text{State} \to \text{Number}$
$\mathcal{A}[\![n]\!]\, s = \mathcal{N}[\![n]\!]$
$\mathcal{A}[\![v]\!]\, s = s\ v$
$\mathcal{A}[\![a_1 + a_2]\!]\, s$
$= \mathcal{A}[\![a_1]\!]\, s + \mathcal{A}[\![a_2]\!]\, s$
$\mathcal{A}[\![a_1 - a_2]\!]\, s$
$= \mathcal{A}[\![a_1]\!]\, s - \mathcal{A}[\![a_2]\!]\, s$
$\mathcal{A}[\![a_1 * a_2]\!]\, s$
$= \mathcal{A}[\![a_1]\!]\, s \times \mathcal{A}[\![a_2]\!]\, s$

Clean

```
A :: AExpr State → Int
A (Int i)   s = i
A (Var v) s = s v
A (x +. y) s = A x s + A y s
A (x -. y) s = A x s - A y s
A (x *. y) s = A x s * A y s
```

see Nielson & Nielson 1992
only the syntax is improved

main idea

- semantics ≈ interpreter that focuses on clarity rather than efficiency

evaluating the FPL approach

disadvantages

- less abstract/ mathematical
- harder to reason about
- nontermination is a problem
- semantics inherits from embedding programming language

advantage

- compiler checks proper use of identifiers and types
- we can execute the semantics
  - simulate for validation
  - model based testing of properties
- nontermination always requires separate attention
- the price to be paid is rather small

Boolean expressions

grammar/data type

```
:: BExpr
 = TRUE
 | FALSE
 | (=.) infix 4 AExpr AExpr
 | (<.) infix 4 AExpr AExpr
 | ~. BExpr
 | (&&.) infixr 3 BExpr BExpr
```

semantic function

```
B :: BExpr State → Bool
B TRUE      s = True
B FALSE     s = False
B (x =. y)  s
   = A x s == A y s
B (x <. y)  s
   = A x s < A y s
B (~. exp)  s
   = not (B exp s)
B (x &&. y) s
   = B x s && B y s
```

semantic domains

- in this way the semantics of While inherits the numbers and Booleans of Clean
- if this would be undesirable we can always introduce a new type and associated operators

```
:: TruthVal = TT | FF
```

```
B :: BExpr State → Bool
B TRUE      s = True
B FALSE     s = False
B (x &&. y) s
    = B x env && B y env
..
```

```
B :: BExpr State → TruthVal
B TRUE      s = TT
B FALSE     s = FF
B (x &&. y) s
 | B x env == TT && B y env == TT
    = TT
    = FF
..
```

better: define an instance of && for TruthVal

the state

- (at least) two possibilities
  - data structure, e.g. [(Var,Int)]
    - needs separate lookup and store functions
    - easy to compare states
  - function, :: State :== Var → Int
    - close to the mathematical semantics
    - hard to compare states
- we will use the function approach

```
emptyState :: State
emptyState = λ x → 0

(↦) infix :: Var Int → State → State
(↦) v i = λ env x → if (x==v) i (env x)
```


statements in While

syntax

S
= x **:=** a
| S **;** S
| **skip**
| **if** b S **else** S
| **while** b S

data structure

```
:: Stmt
   = (:=.) infix 2 Var AExpr
   | (:.) infixr 1 Stmt Stmt
   | Skip
   | IF BExpr Stmt Stmt
   | While BExpr Stmt
```


operational semantics of statements

- big step operational semantics:
  - describe how the result must be calculated
  - in one go to the result: a new state

```
ns :: Stmt State → State
ns (v :=. e)   s = (v ↦ A e s) s
ns (s1 :. s2)  s = ns s2 (ns s1 s)
ns Skip        s = s
ns (IF c t e)  s |   B c s   = ns t s
ns (IF c t e)  s | ~(B c s) = ns e s
ns (While c b) s |   B c s   = ns (While c b) (ns b s)
ns (While c b) s | ~(B c s) = s
```

note: alternatives are mutual exclusive, can be placed in any order

how the type system helps us

- suppose we would write

ns :: Stmt State → State
ns (v :=. e) s = (v ↦ e) s
ns (s1 :. s2) s = ns s2 (ns s1 s)
ns Skip s = s
....

this models lazy evaluation. It requires a state of type: Var → AExpr

- what is wrong with this?
- the type system says:
  Type error [exprSem.icl,67,ns]:"argument 2 of |->" cannot unify types: Int AExpr
- we should have written:

ns (v :=. e) s = (v ↦ A e s) s

mathematical notation of semantic natural (big step) semantics

Scott brackets

$\mathcal{NS}$ ⟦ S_1 ; S_2 ⟧ e
= $\mathcal{NS}$ ⟦ S_2 ⟧ ($\mathcal{NS}$ ⟦ S_1 ⟧ e)
or
$\mathcal{NS}$ ⟦ S_1 ; S_2 ⟧
= $\mathcal{NS}$ ⟦ S_2 ⟧ . $\mathcal{NS}$ ⟦ S_1 ⟧

using Currying and function composition

horizontals bars

if the premises above the bar holds, the conclusion below it can be derived

$$\frac{\langle S_1, e\rangle \rightarrow e_1 \quad \langle S_2, e_1\rangle \rightarrow e_3}{\langle S_1;S_2,\ e\rangle \rightarrow e_3}$$

these things do not have an order

using
$\langle S,e\rangle \rightarrow e_1 \equiv \mathcal{NS}[\![S]\!]\, e = e_1$

a small step operation semantics structural operational semantics

- one step a time

:: Config = Final State | Inter Stmt State

sos1 :: Stmt State -> Config
sos1 (v :=. e) s = Final ((v |-> A e s) s)
sos1 Skip s = Final s
sos1 (x :. y) s
= **case** sos1 x s **of**
Final t = Inter y t
Inter z t = Inter (z :. y) t
sos1 (IF c t e) s | B c s = Inter t s
sos1 (IF c t e) s | ~(B c s) = Inter e s
sos1 (While c b) s = Inter (IF c (b :. While c b) Skip) s

really different

structural operational semantics 2

- trace obtained by applying sos1 until a final state

sosTrace :: Config -> [Config]
sosTrace c=:(Final _) = [c]
sosTrace c=:(Inter ss s) = [c: sosTrace (sos1 ss s)]

- big step by selecting the last state of this trace

sos :: Stmt State -> State
sos s env = env1
where (Final env1) = last (sosTrace (Inter s env))

denotational semantics

- we are interested in the final state, not how it is obtained

```
ds :: Stmt State -> State
ds (v :=. a)    s = (v |-> A a s) s
ds Skip         s = s
ds (s1 :. s2)   s = ds s2 (ds s1 s)
ds (IF c t e)   s = if (B c s) (ds t s) (ds e s)
ds (While c stmt) s = fix f s
where f g s = if (B c s) (g (ds stmt s)) s

fix :: (a -> a) -> a
fix f = f (fix f)
```


main differences of the various semantics

- handling of the while-statement:

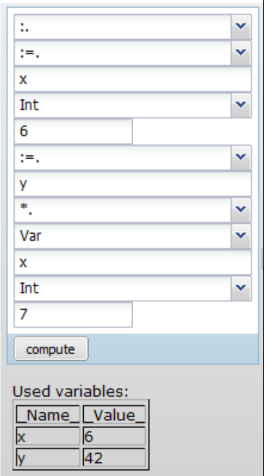
```
ns :: Stmt State -> State
ns (While c b) s |   B c s   = ns (While c b) (ns b s)
ns (While c b) s | ~(B c s) = s

sos1 :: Stmt State -> Config
sos1 (While c b) s = Inter (IF c (b :. While c b) Skip) s

ds :: Stmt State -> State
ds (While c stmt) s = fix f s
where f g s = if (B c s) (g (ds stmt s)) s
```


simulation

- iData makes a syntax directed editor for statements
- any of the semantics can execute this program
- we scan the program for used variables and display their value
- useful for small experiments!
- demo



testing properties

- Clean as its own model-based test tool: Gast
- we use this to test properties of the semantics
  - thousands of tests in a second
  - easy to repeat after each change
- this improves the confidence in the correctness
- if we have gathered enough confidence we can give a mathematical prove of these properties
  - even with a prove assistant this is usually much work

some properties

```
propFac :: (Stmt State -> State) -> Bool
propFac sem = sem facStmt emptyState "y" == 24
```

∀ sem

∀ sem ∈ { ns, ds, sos }

```
propFacAll :: Property
propFacAll = propFac For [ns, ds, sos ]
```

∀ statement

```
prop :: Stmt -> Bool
prop s = eqState (ns s empty) (ds s empty) (allvars s)
```

checks equality for given variables

generate only terminating statements

wrap up: main idea

- semantics ≈ interpreter that focuses on clarity rather than efficiency

lessons learned

- semantics assigns meaning to languages
  - natural semantics: shows how the value is computed in big steps
  - structural operational semantics: small steps
  - denotational semantics: concentrate on the value
- with very little effort this can be expressed in a modern functional programming language
- advantages:
  - checks use of identifiers and types
  - simulate language for validation
  - model based testing of properties
- warning: there is much more in semantics

exercise

- purpose: get acquired with iTasks and this style of semantics
- see http://www.cs.ru.nl/~pieter/cefp09/
  - exercise as pdf
  - Clean files for parts 5 and 6.