

Software Language Engineering Semantics: Interpreters

Tijs van der Storm



Centrum Wiskunde & Informatica



university of
groningen

Recap

- Grammar -> Parser -> Parse Tree -> ~~AST~~
- Name resolution: recover referential structure
- Checking: find errors not captured by syntax
- Today:
 - semantics
 - interpreters

Formal semantics

- Axiomatic semantics
- Operational semantics
 - small-step
 - big-step (aka “Natural semantics”)
- Denotational semantics

Formal semantics

- Prove things (e.g., determinism, type soundness)
- Simulate & explore (e.g. using Redex)
- Generate interpreter

Big-step

$$\frac{\langle b, \sigma \rangle \Downarrow \mathbf{true} \quad \langle c_0, \sigma \rangle \Downarrow \sigma'}{\langle \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma \rangle \Downarrow \sigma'} \quad (\text{conditionals})$$

$$\frac{\langle b, \sigma \rangle \Downarrow \mathbf{false} \quad \langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle \mathbf{if } b \mathbf{ then } c_0 \mathbf{ else } c_1, \sigma \rangle \Downarrow \sigma'}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \mathbf{false}}{\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \Downarrow \sigma} \quad (\text{while loops})$$

$$\frac{\langle b, \sigma \rangle \Downarrow \mathbf{true} \quad \langle c, \sigma \rangle \Downarrow \sigma'' \quad \langle \mathbf{while } b \mathbf{ do } c, \sigma'' \rangle \Downarrow \sigma'}{\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \Downarrow \sigma'}$$

Small-step

$$\text{[LEFT]} \quad \frac{E_0 \longrightarrow E'_0}{E_0 \times E_1 \longrightarrow E'_0 \times E_1}$$

$$\text{[RIGHT]} \quad \frac{E_1 \longrightarrow E'_1}{E_0 \times E_1 \longrightarrow E_0 \times E'_1}$$

$$\text{[LEFT}_0\text{]} \quad \frac{}{0 \times E_1 \longrightarrow 0}$$

$$\text{[RIGHT}_0\text{]} \quad \frac{}{E_0 \times 0 \longrightarrow 0}$$

$$\text{[MUL]} \quad \frac{}{z_0 \times z_1 \longrightarrow z} \quad z = z_0 z_1$$

$$\text{[COND]} \quad \frac{E_0 \longrightarrow E'_0}{E_0 ? E_1 : E_2 \longrightarrow E'_0 ? E_1 : E_2}$$

$$\text{[COND}_Z\text{]} \quad \frac{}{z ? E_1 : E_2 \longrightarrow E_2} \quad z = 0$$

$$\text{[COND}_{NZ}\text{]} \quad \frac{}{z ? E_1 : E_2 \longrightarrow E_1} \quad z \neq 0$$

In code...

- Big-step: $\text{eval}(\text{Exp}, \text{Env}, \text{Store}) \rightarrow \langle \text{Value}, \text{Store} \rangle$
- Small-step: $\text{step}(\text{Exp}, \text{Env}, \text{Store}) \rightarrow \langle \text{Exp}, \text{Store} \rangle$
- Denotational: $\text{map}(\text{Exp}) \rightarrow (\text{Env} \times \text{Store} \rightarrow \text{Store})$

evalquote[fn;x] = apply[fn;x;NIL]

where

apply[fn;x;a] =
[atom[fn] → [eq[fn;CAR] → caar[x];
eq[fn;CDR] → cdar[x];
eq[fn;CONS] → cons[car[x];cadr[x]];
eq[fn;ATOM] → atom[car[x]];
eq[fn;EQ] → eq[car[x];cadr[x]];
T → apply[eval[fn;a];x;a]];
eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
caddr[fn]];a]]]

eval[e;a] = [atom[e] → cdr[assoc[e;a]];
atom[car[e]] →
[eq[car[e];QUOTE] → cadr[e];
eq[car[e];COND] → evcon[cdr[e];a];
T → apply[car[e];evlis[cdr[e];a];a]];
T → apply[car[e];evlis[cdr[e];a];a]]

pairlis and assoc have been previously defined.

evcon[c;a] = [eval[caar[c];a] → eval[cadar[c];a];
T → evcon[cdr[c];a]]

and

evlis[m;a] = [null[m] → NIL;
T → cons[eval[car[m];a];evlis[cdr[m];a]]]

Recursion over structure of expressions

```
public int eval(Exp exp) {  
    switch (exp) {  
        case nat(int nat): return nat;  
  
        case mul(Exp lhs, Exp rhs): return eval(lhs) * eval(rhs);  
  
        case div(Exp lhs, Exp rhs): return eval(lhs) / eval(rhs);  
  
        case add(Exp lhs, Exp rhs): return eval(lhs) + eval(rhs);  
  
        case min(Exp lhs, Exp rhs): return eval(lhs) - eval(rhs);  
  
        case gt(Exp lhs, Exp rhs): return eval(lhs) > eval(rhs) ? 1 : 0;  
  
        case lt(Exp lhs, Exp rhs): return eval(lhs) < eval(rhs) ? 1 : 0;  
  
        case geq(Exp lhs, Exp rhs): return eval(lhs) >= eval(rhs) ? 1 : 0;  
  
        case leq(Exp lhs, Exp rhs): return eval(lhs) <= eval(rhs) ? 1 : 0;  
  
        case cond(Exp cond, Exp then, Exp otherwise):  
            return eval(cond) != 0 ?  
                eval(then) : eval(otherwise);  
    }  
}
```

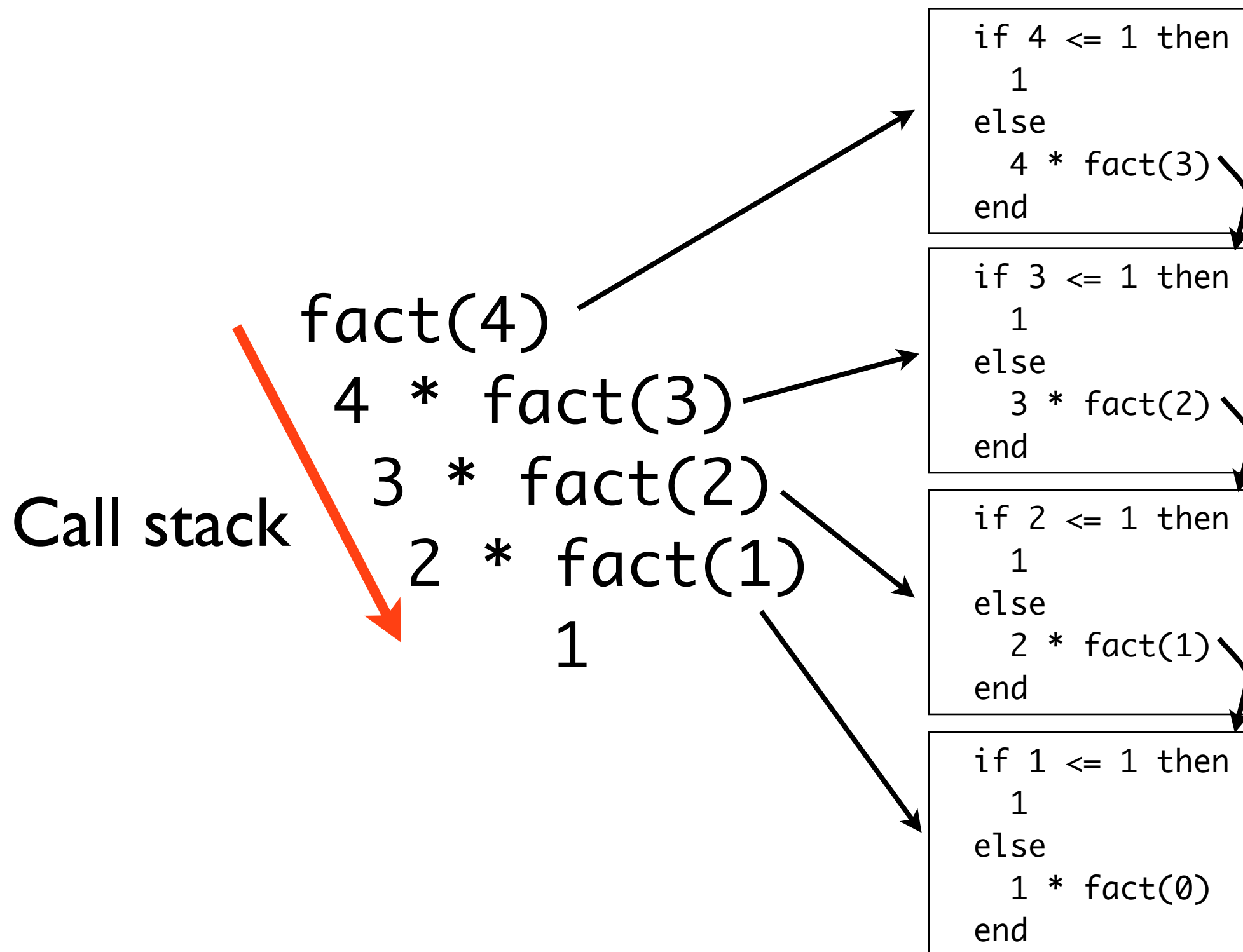
Example

Formal
parameter

```
fact(n) =  
  if n <= 1 then  
    1  
  else  
    n * fact(n-1)  
  end
```

Actual
parameter

Idea: substitute formals



Lookup table for functions

`alias PEnv = map[str, Func];`

a procedure
environment

maps names to
functions

```
data Prog = prog(list[Func] funcs);  
data Func = func(str name, list[str] formals, Exp body);
```

```
public int eval(Exp exp) {  
    switch (exp) {  
        case nat(int nat): return nat;  
  
        case mul(Exp lhs, Exp rhs): return eval(lhs) * eval(rhs);  
  
        case div(Exp lhs, Exp rhs): return eval(lhs) / eval(rhs);  
  
        case add(Exp lhs, Exp rhs): return eval(lhs) + eval(rhs);  
  
        case min(Exp lhs, Exp rhs): return eval(lhs) - eval(rhs);  
  
        case gt(Exp lhs, Exp rhs): return eval(lhs) > eval(rhs) ? 1 : 0;  
  
        case lt(Exp lhs, Exp rhs): return eval(lhs) < eval(rhs) ? 1 : 0;  
  
        case geq(Exp lhs, Exp rhs): return eval(lhs) >= eval(rhs) ? 1 : 0;  
  
        case leq(Exp lhs, Exp rhs): return eval(lhs) <= eval(rhs) ? 1 : 0;  
  
        case cond(Exp cond, Exp then, Exp otherwise):  
            return eval(cond) != 0 ?  
                eval(then) : eval(otherwise);  
    }  
}
```

```

public int eval(Exp exp, PEnv penv) {
    switch (exp) {
        case nat(int nat): return nat;

        case mul(Exp lhs, Exp rhs): return eval(lhs, penv) * eval(rhs, penv);

        case div(Exp lhs, Exp rhs): return eval(lhs, penv) / eval(rhs, penv);

        case add(Exp lhs, Exp rhs): return eval(lhs, penv) + eval(rhs, penv);

        case min(Exp lhs, Exp rhs): return eval(lhs, penv) - eval(rhs, penv);

        case gt(Exp lhs, Exp rhs): return eval(lhs, penv) > eval(rhs, penv) ? 1 : 0;

        case lt(Exp lhs, Exp rhs): return eval(lhs, penv) < eval(rhs, penv) ? 1 : 0;

        case geq(Exp lhs, Exp rhs): return eval(lhs, penv) >= eval(rhs, penv) ? 1 : 0;

        case leq(Exp lhs, Exp rhs): return eval(lhs, penv) <= eval(rhs, penv) ? 1 : 0;

        case cond(Exp cond, Exp then, Exp otherwise):
            return eval(cond, penv) != 0 ?
                eval(then, penv) : eval(otherwise, penv);
    }
}

```

Evaluating calls

lookup the called
function in the env

```
...  
case call(str name, list[Exp] args): {  
  f = penv[name];  
  b = subst(f.body, f.formals, [ eval(a, penv) | a <- args])  
  return eval(b, penv);  
}  
...
```

eval body where all
variables have been
substituted for values

eval actual args
to perform
substitution

Let bindings

```
(define let
  (macro (bindings . body)
    (define (named-let name bindings body)
      `(let ((,name #f))
         (set! ,name (lambda ,(map first bindings) . ,body))
         (,name . ,(map second bindings))))
      (if (symbol? bindings)
          (named-let bindings (first body) (rest body))
          `((lambda ,(map first bindings) . ,body) .
             ,(map second bindings)))))
```

```
(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))
    ((let tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (lambda (name ...)
                      body1 body2 ...)))
      tag)
      val ...))))
```


Func1 = Func0 + {let}

local
variables

scope of
bindings

```
syntax Exp = let: "let" {Binding ","}* "in" Exp "end";  
syntax Binding = binding: Ident "=" Exp;
```

```
data Exp = let(list[Binding] bindings, Exp exp);  
data Binding = binding(str var, Exp exp);
```

Example

```
fact(n) =  
  let  
    x = n  
  in  
    if x <= 1 then  
      x  
    else  
      x * fact(x-1)  
    end  
  end
```

Shadowing

```
fact(n) =  
  let
```

NB: not an
assignment!

```
    n = n + 1
```

formal param *n*
is shadowed by
let-bound *n*

```
  in
```

```
    if n ≤ 1 then
```

```
      n
```

```
    else
```

```
      n * fact(n-1)
```

```
    end
```

```
  end
```

Substitution?

```
fact(4) =  
  let  
    n = 4 + 1  
  in  
    if 4 <= 1 then  
      4  
    else  
      4 * fact(4-1)  
    end  
  end
```

Wrong

Different approach: environments

```
alias Env = map[str, int];
```



variables

values

```

public int eval(Exp exp, PEnv penv) {
    switch (exp) {
        case nat(int nat): return nat;

        case mul(Exp lhs, Exp rhs): return eval(lhs, penv) * eval(rhs, penv);

        case div(Exp lhs, Exp rhs): return eval(lhs, penv) / eval(rhs, penv);

        case add(Exp lhs, Exp rhs): return eval(lhs, penv) + eval(rhs, penv);

        case min(Exp lhs, Exp rhs): return eval(lhs, penv) - eval(rhs, penv);

        case gt(Exp lhs, Exp rhs): return eval(lhs, penv) > eval(rhs, penv) ? 1 : 0;

        case lt(Exp lhs, Exp rhs): return eval(lhs, penv) < eval(rhs, penv) ? 1 : 0;

        case geq(Exp lhs, Exp rhs): return eval(lhs, penv) >= eval(rhs, penv) ? 1 : 0;

        case leq(Exp lhs, Exp rhs): return eval(lhs, penv) <= eval(rhs, penv) ? 1 : 0;

        case cond(Exp cond, Exp then, Exp otherwise):
            return eval(cond, penv) != 0 ?
                eval(then, penv) : eval(otherwise, penv);
    }
}

```

```

public int eval(Exp exp, Env env, PEnv penv) {
    switch (exp) {
        case nat(int nat): return nat;

        case mul(Exp lhs, Exp rhs): return eval(lhs, env, penv) * eval(rhs, env, penv);

        case div(Exp lhs, Exp rhs): return eval(lhs, env, penv) / eval(rhs, env, penv);

        case add(Exp lhs, Exp rhs): return eval(lhs, env, penv) + eval(rhs, env, penv);

        case min(Exp lhs, Exp rhs): return eval(lhs, env, penv) - eval(rhs, env, penv);

        case gt(Exp lhs, Exp rhs): return eval(lhs, env, penv) > eval(rhs, env, penv) ? 1 : 0;

        case lt(Exp lhs, Exp rhs): return eval(lhs, env, penv) < eval(rhs, env, penv) ? 1 : 0;

        case geq(Exp lhs, Exp rhs): return eval(lhs, env, penv) >= eval(rhs, env, penv) ? 1 : 0;


        case leq(Exp lhs, Exp rhs): return eval(lhs, env, penv) <= eval(rhs, env, penv) ? 1 : 0;

        case cond(Exp cond, Exp then, Exp otherwise):
            return eval(cond, env, penv) != 0 ?
                eval(then, env, penv) : eval(otherwise, env, penv);
    }
}

```

Evaluating variables

```
case var(str name):  
  return env[name];
```



lookup *name*
in *env*

Evaluating calls

```
...  
case call(str name, list[Exp] args): {  
  f = penv[name];  
  env = bind(f.formals, [ eval(a, env, penv) | a <- args ]);  
  return eval(f.body, env, penv);  
}  
...
```

create a **new**
environment by binding
actuals to formals

evaluate the body of *f*
in the new env

Evaluating let

```
...  
case let(list[Binding] bindings, Exp exp): {  
  env += ( b.var : eval(b.exp, env, penv) | b <- bindings );  
  return eval(exp, env, penv);  
}  
...
```

update the current
environment
(shadowing)

eval exp in the
updated env

Evaluating with *env*

Call stack

fact(4):

4*fact(3)

3*fact(2)

2*fact(1)

1

Output

= 24

= 6

= 2

= 1

Env

(“n”: 4)

(“n”: 3)

(“n”: 2)

(“n”: 1)

```
fact(n) =  
  if n <= 1 then  
    1  
  else  
    n * fact(n-1)  
  end
```

Next up

- State machines live coding
- Outlook to QL exercise