Virtual Machines

Lecture 5 – Dynamic (or Operational) Semantics and First Assignment

Module 2 -Revised Content

It consists of about 6 lectures and covers the following topics:

- **Design an Abstract Syntax Tree for a CoreJava (a small Object-Oriented) language (1st assignment—7% of the final grade)**
- **Implement an interpreter for CoreJava (2nd assignment—25% of the final grade)**
- **Implement a Lexer and a Parser for CoreJava(3rd assignment—25% of the final grade)**
- **Implement a Type Checker for CoreJava (4th assignment—25% of the final grade)**

- **Revised Evaluation**:
  - **Labs activity (85%):**
    - **Group of 2 or 3 students : same project consisting of 4 assignments (82%)**
    - **Oral presentation of your final graduate project (3%)**
  - **Individual Oral Final exam (open book) (15%)**

Your group assignments, your final graduate project and Individual Oral Final Exam will be in the last two weeks. Each group must reserve one of the following time slots (max 5 groups per time slot):

» **11 May 2016,10-12**

» **11 May 2016, 14-16**

» **18 May 2016, 10-12**

» **18 May 2016, 14-16**

Formal Dynamic (or Operational) Semantics of a Language

Review

Previously in Lecture 4:

- simple interpreter for expression language:
  - abstract syntax tree (AST)
  - small-step, substitution model of evaluation

Today:

- Formal dynamic/operational semantics:
  - small-step, substitution model
  - large-step, environment model

Review: Notation

- The interpreter code we've written is one way of *defining* the syntax and semantics of a language

- Programming language designers have another more compact notation that's *independent* of the implementation language of interpreter.

Review: Abstract syntax

```
e ::= x | i | e+e
    | let x = e1 in e2
```

e, **x**, **i**: *meta-variables* that stand for pieces of syntax

- **e**: expressions
- **x**: program variables
- **i**: integers

::= and **|** are *meta-syntax:* used to describe syntax of language

notation is called ***Backus-Naur Form*** (BNF) from its use by Backus and Naur in their definition of Algol-60

Dynamic/Operational semantics

Defined by a *judgement*:

`e --> e'`

Read as **`e`** takes a single step to **`e'`**

e.g., **`(5+2)+0 --> 7+0`**

Expressions continue to step until they reach a *value*

e.g., **`(5+2)+0 --> 7+0 --> 7`**

Values are a syntactic subset of expressions:

`v ::= i`

Dynamic/Operational semantics

Reflexive, transitive closure of `-->` is written `-->*`

`e -->* e'` read as `e` multisteps to `e'` or `e` evaluates to `e'`

e.g., `(5+2)+0 -->* 7`

this style of definition is called a *small-step semantics:* based on taking single small steps

Dynamic/Operational semantics of expr. lang.

```
e1 + e2 --> e1' + e2
  if e1 --> e1'

v1 + e2 --> v1 + e2'
  if e2 --> e2'

v1 + v2 --> n

  if n is the sum of v1 and v2
```

Dynamic/Operational semantics of expr. lang.

```
let x = e1 in e2 --> let x = e1' in e2
  if e1 --> e1'
```

```
let x = v1 in e2 --> e2{v1/x}
```

read **`e2{v1/x}`** as **`e2`** with **`v1`** substituted for **`x`**
(as we implemented in **`subst`**)

so we call this the substitution model of evaluation

Dynamic/Operational semantics of expr. lang.

```
  if e1 then e2 else e3
  -->   if   e1'     then e2  else e3
         if e1 -->  e1'

if true then e2 else e3 --> e2

if false then e2 else e3 --> e3
```

Dynamic/Operational semantics of expr. lang.

Values and variables do not single step:

```
v  -/->
x  -/->
```

But they do multistep:

```
v  -->* v
x  -->* x
```

because multistep includes 0 steps
(i.e., it is the *reflexive* transitive closure of `-->`)

- values don't step because they're done computing
- variables don't step because they're an error: we should never reach a variable; it should have already been substituted away

We need Static Semantics

Suppose we add Booleans, conjunction, and **`if`** expressions to language:

e ::= ...| b | e1 && e2| if e1 then e2 else e3
v ::= ..|b

Now we can have some non-sensical expressions like:

5 + false
If 5 then true else 0

These non-sensical expressions must be ruled out at compile time.
It can be done using a Static Semantic (a Type Checker) that we will discuss in the near future

Interpreter for ext. expr. lang.

See **`interp3.ml`** in code attached for this lecture

ANOTHER FORMAL DYNAMIC SEMANTICS

Dynamic semantics

Two different models of evaluation:

- **Small-step substitution model:** substitute value for variable in body of `let` expression
  - And in body of function, since `let x= e1 in e2` behaves the same as `(fun x -> e2) e1`
  - What we've done so far; good mental model for evaluation
  - Not really what OCaml does
- **Big-step environment model:** keep a data structure around that binds variables to values
  - What we'll do now; also a good mental model
  - Much closer to what OCaml really does

Syntax

```
e ::= x|i | b
     |e1+e2 | e1 && e2
     |let x=e1 in e2
     |if e1 then e2 else e3
v ::= i | b
```

New evaluation judgement

- *Big-step semantics*: we model just the reduction from the original expression to the final value
- Suppose `e-->` `e'` `... --> v`
- We'll abstrat that fact to **e** `==> v`
  - forget about all the intermediate expressions
  - new notation means **e** *evaluates (down) to* **v**, equiv. **e** *takes a big step to* **v**
  - textbooks use down arrows: **e** $\Downarrow$ **v**
- **Goal:** `e ==> v` if and only if `e -->* v`

Values

- Values are already done:
  - Evaluation rule: `v ==> v`

- Constants are values
  - `42` is a value, so `42 ==> 42`
  - `true` is a value, so `true ==> true`

Operator evaluation

```
e1    +e2 ==>v
if e1 ==>    i1
and  e2 ==> i2
and v is the result of the primitive
  operation i1 + i2
```

```
eg.
true && false ==> false
1 + 2 ==> 3
1 + (2 + 3) ==> 6
```

Variables

- What does a variable name evaluate to?

  `x  ==> ???`

- Trick question: we don't have enough information to answer it
- Need to know what value variable was ***bound*** to
  - e.g., **`let x = 2 in x +1`**
  - It evaluates to **`3`**, but we reach a point where we need to know binding of **`x`**

Until now, we've never needed this, because we always **substituted** before we ever get to a variable name

Variables

OCaml doesn't actually do substitution

```
(fun  x   ->  42)  0
```

waste of runtime resources to do substitution inside 42

Instead, OCaml lazily substitutes by maintaining *dynamic environment*

Dynamic environment

- Dictionary of bindings of all current variables
- Changes throughout evaluation:
  - No bindings at $:

```
$ let x = 42 in let y = false in e
```

  - One binding **[x=42]** at $:

```
let x = 42 in
$ let y = false in e
```

  - Two bindings **[x=42,y=false]** at $:

```
let x = 42 in let y = false in
$ e
```

Variable evaluation

To evaluate `x` in environment **`env`** **Look up** value **`v`** of **`x`** in **`env`** **Return `v`**

Type checking guarantees that variable is bound, so we can’ t ever fail to find a binding in dynamic environment

Evaluation judgement

Extended notation:

`<env, e> ==> v`

Meaning: in dynamic environment **`env`**, expression **`e`** takes a big step to value **`v`**

`<env, e>` is called a *machine configuration*

Variable evaluation

```
<env, x> ==> v
  if    =
  v     env(x)
```

`env(x)`:

- meaning: the value to which **`env`** binds **`x`**
- think of it as looking up **`x`** in dictionary **`env`**

Redo: evaluation with environment

```
<env, v> ==> v


<env, e1 + e2> ==> v

 if <env,  e1> ==> i1
 and    <env,e2> ==>i2
 and          v    is   the result of
     primitive operation i1+i2
```

Let expressions

To evaluate `let x = e1 in e2` in environment **`env`**

Evaluate the binding expression **`e1`** to a value **`v1`** in environment **`env`**

```
<env,    e1> ==> v1
```

Extend the environment to bind **`x`** to **`v1`**

```
env' =    env[x->v1]
```
new notation

Evaluate the body expression **`e2`** to a value **`v2`** in extended environment **`env'`**

```
<env',    e2> ==> v2
```

Return `v2`

Let expression evaluation rule

```
<env, let x=e1 in e2> ==> v2
  if <env, e1> ==>     v1
  and <env[x->v1], e2> ==> v2
```

Example (let `[]` be the empty environment)

: **`<[],let x = 42 in x> ==> 42`**

Because...

- **<[], 42> ==> 42**
- and **<[][x->42], x> ==> 42**
  - Because **[x=42](x)=42**

Group Project – First Assignment

7% of the final grade

First Assignment

Design in Ocaml the Abstract Syntax Tree for a small object-oriented language called CoreJava.

The syntax of CoreJava is described in the following.

Syntax of CoreJava

A CoreJava program P consists of a list of class declarations as follows:

P ::= def+

def+ ::= def |def1; def2;...;defn

The program contains at least one class declaration.

The last class declaration must contain a method called “main” which is the starting execution point for the CoreJava program.

Syntax of CoreJava

A class declaration def consists of the following:

def ::=   class cn1 extends cn2 { field* # meth*}

where cn1 is the current class name and cn2 is the name of the parent class

The body of the class consists of a list of fields declarations followed by a list of methods declarations. The fields list is separated by "#" from the methods list. The fields list is either empty or field+ ::= field1;field2;...;fieldn. The methods list is either empty or meth+ ::= meth1;meth2;...;methn.

Syntax of CoreJava

A field declaration consists of the following:

field ::= typ fn

where typ is the type of the field and fn is the name of the field

A type typ can be either a primitive type, or a class name or the bottom (the type of null).

typ ::= prim | cn | _|_

prim ::= int | float | bool | void

Syntax of CoreJava

A method declaration consists of the following:

meth ::= typ1 mn((typ pn)*) {e}

where typ1 is the type of the method result, mn is the method name, (typ pn)* is the list of method parameters.

The list of method parameters can be either empty or (typ pn)+ ::= typ1 pn1, typ2 pn2,...,typn pnn

The body of the method is given by the expression e.

Syntax of CoreJava

An expression is defined as follows:

e ::= null (null value)

| kint (a constant value)

|kfloat

| true | false

| () (a value of type void)

| v (a variable name)

| v.f (a field f of an object denoted by v)

Syntax of CoreJava

An expression is defined as follows:

e ::= ...

| v=e (variable assignment)

|v.f=e (object field assignment)

| { (typ v) e1} (local variable declaration such that v has type typ and its scope is the expression e1)

| e1;e2 (a sequence such that e2 is executed only after e1 is executed)

| if v then {e1} else {e2} (conditional where the condition is a variable)

Syntax of CoreJava

An expression is defined as follows:

e ::= ...

| e1 opint e2 (arithmetic expressions for ints where opint ::= +|-|*|/)

|e1 opfloat e2 (arithmetic expressions for floats where opfloat ::= +.|-.|*.|/.)

| e1 && e2 (logical expressions)

| e1 || e2

| !e (negation)

Syntax of CoreJava

An expression is defined as follows:

e ::= ...

| new cn(v*)   (creation of an instance object of the class cn. The fields of the new object are initialized by the list of variables v*. The fields are intialized in the order defined in the declaration of class cn)

|v.mn(v*)   (method call where v is the method receiver and the list of variables v* are the current arguments initializing the method parameters)

| while v {e}   (loop where the condition is a variable)

Operational Semantics of CoreJava

We'll discuss it Next Lecture ...

when you will also get the second assignment, the interpreter implementation