CS 320
Operational semantics: Variables

Marco Gaboardi
CDS 1019
gaboardi@bu.edu

Operational Semantics

- It is usually provided at a level of abstraction that is independent from the machine.
- The detailed characteristics of the particular computer would make actions difficult to describe/understand.
- Different formalism has been developed to describe the operational semantics in a machine-independent way.

We will look into formal rules and derivations.

An example:

```
(add/int(4)::int(5)::S) →(/int(4+5)::S)    (/int(4+5)::S) →0 (/int(4+5)::S)
(push 4;add/int(5)::S) →(add/int(4)::int(5)::S)    (add/int(4)::int(5)::S) →1 (/int(4+5)::S)
(push 5;push 4;add/S) → (push 4;add/int(5)::S)    (push 4;add/int(5)::S) →2 (/int(4+5)::S)
(push 5;push 4;add/S) →3 (/int(9)::S)
```

Boolean expressions

- Let us consider this simple language for Boolean expressions

```
<bexpr> ::= <const> <bop> <bexpr> | <const>
<bop>::= and | or | eq
<const>::= bool | int
```

What are the challenges here?

Operational semantics for basic boolean expressions

$$e \longrightarrow ?$$

Here the expression e is itself a configuration. We already have all the information we need to execute it.

What can $?$ be?

Operational semantics for basic boolean expressions

$$\texttt{e} \longrightarrow \texttt{?}$$

Here the expression `e` is itself a configuration. We already have all the information we need to execute it.

What can `?` be?

$$\texttt{e} \longrightarrow \texttt{e'} \qquad\qquad \texttt{e} \longrightarrow \texttt{err}$$

Rules

$$\frac{v_1\ v_2\ \text{different type}}{v_1\ \texttt{eq}\ v_2 \rightarrow \texttt{err}}$$

$$\frac{v_1\ v_2\ \text{same type}}{v_1\ \texttt{eq}\ v_2 \rightarrow v_1 = v_2}$$

$$\frac{v_1\ v_2\ \text{bool}}{v_1\ \texttt{and}\ v_2 \rightarrow v_1 \wedge v_2}$$

$$\frac{v_1\ v_2\ \text{bool}}{v_1\ \texttt{or}\ v_2 \rightarrow v_1 \vee v_2}$$

$$\frac{e_1 \rightarrow e_1' \qquad e_1' \neq \texttt{err}}{e_1\ \texttt{bop}\ e_2 \rightarrow e_1'\ \texttt{bop}\ e_2}$$

$$\frac{e_2 \rightarrow e_2' \qquad e_2' \neq \texttt{err}}{v_1\ \texttt{bop}\ e_2 \rightarrow v_1\ \texttt{bop}\ e_2'}$$

c here is a configuration, either an expression e or err

$$\frac{}{c \rightarrow^{0} c}$$

$$\frac{c \rightarrow c' \qquad c' \rightarrow^{k} c''}{c \rightarrow^{k+1} c''}$$

$$\frac{v_1\ v_2\ \text{not bool}}{v_1\ \texttt{and}\ v_2 \rightarrow \texttt{err}}$$

$$\frac{v_1\ v_2\ \text{not bool}}{v_1\ \texttt{or}\ v_2 \rightarrow \texttt{err}}$$

$$\frac{e_1 \rightarrow \texttt{err}}{e_1\ \texttt{bop}\ e_2 \rightarrow \texttt{err}}$$

$$\frac{e_2 \rightarrow \texttt{err}}{v_1\ \texttt{bop}\ e_2 \rightarrow \texttt{err}}$$

What can we do to have a more efficient semantics for boolean expressions?

What can we do to have a more efficient semantics for boolean expressions?

What if we know that one of the elements of an or is true or one of the elements of an and is false?

More efficient rules

$$\frac{\mathtt{e}_2 \longrightarrow \mathtt{e}_2'}{\mathtt{v}_1\ \mathtt{bop}\ \mathtt{e}_2 \longrightarrow \mathtt{v}_1\ \mathtt{bop}\ \mathtt{e}_2'}$$

We could change this rule:

$$\frac{}{\mathtt{true}\ \mathtt{or}\ \mathtt{e}_2 \longrightarrow \mathtt{true}}$$

$$\frac{\mathtt{e}_2 \longrightarrow \mathtt{e}_2'}{\mathtt{false}\ \mathtt{or}\ \mathtt{e}_2 \longrightarrow \mathtt{false}\ \mathtt{or}\ \mathtt{e}_2'}$$

Are the two semantics equivalent?

Variables

Variables

- Functional languages use variables as names (where the association name-value is stored in an environment).
  - We can remember the association, or read the value, but we cannot change it.
- Imperative languages are abstractions of von Neumann architecture
  - A variable abstracts the concept of memory location
- Understanding how variables are managed is an important part to understand the semantics of a programming language.

Arithmetical expressions: shape of expressions

- Let us consider this simple language for expressions

```
<expr> ::= <term> <addop> <expr> | <term>
<addop>::= add | sub
<term> ::= var | val
```

What are the challenges here?

What is the form of a program?

```
<term>(add|sub)<term>(add|sub)<term>…<term>
```

Where is the stack here?

Arithmetical expressions: shape of expressions

- Let us consider this simple language for expressions

```
<expr> ::= <term> <addop> <expr> | <term>
<addop>::= add | sub
<term> ::= var | val
```

What are the challenges here?

What is the value of a var?

```
X add 5 add 6
```

Where is the value defined?

Operational semantics for arithmetical expressions

$$(e/m) \longrightarrow (e/m)$$

Here (e/m) is a configuration where e is an expression and m is a memory. We call these pairs configurations because we think in terms of an "abstract machine".

We can think about a memory as a set of (unique) assignments of variables to values:

$$m = ((x_1=v_1), (x_2=v_2) \ldots, (x_n=v_n))$$

Arithmetical expressions with vars: shape of expressions

- Let us consider this simple language for expressions

```
<expr> ::= <term> <addop> <expr> | <term>
<addop>::= add | sub
<term> ::= var | val
```

- What is the potential shape of an expression?

  - `v` Value
  - `x` Variable
  - `(n|x)  add  e` Expression + Constant or Variable

This is recursive

An example: addition of values

$$\frac{(\mathtt{e/m}) \longrightarrow (\mathtt{e_1/m})}{(\mathtt{v_1\ add\ e/m}) \longrightarrow (\mathtt{v_1\ add\ e_1/m})}$$

We can use the fact that e is recursive and hypothetical reasoning.

An example: fetching the value of a variable

$$(\mathtt{x/m}) \longrightarrow \text{??}\ (\mathtt{fetch(x,m)/m})$$

How can we fetch the value of x from the memory?

We can introduce a function `fetch` taking a memory a variable and returning the value of the variable in the memory.

$$\boxed{\mathtt{fetch}(\mathtt{x}_2, ((\mathtt{x}_1=\mathtt{v}_1), (\mathtt{x}_2=\mathtt{v}_2)\ldots, (\mathtt{x}_n=\mathtt{v}_n)) = \mathtt{v}_2}$$

An example: fetching the value of a variable

The variable can appear in an expression.
On the right:

`(v add x/m)` $\longrightarrow$ `(v add fetch(x,m)/m)`

Or on the left:

`(x add e/m)` $\longrightarrow$ `(fetch(x,m) add e/m)`

Summary of the rules:

$$(\mathtt{x\ add\ e/m}) \longrightarrow (\mathtt{fetch(x,m)\ add\ e\ /m})$$

$$(\mathtt{v\ add\ x/m}) \longrightarrow (\mathtt{v\ add\ fetch(x,m)/m})$$

$$(\mathtt{v_1\ add\ v_2/m}) \longrightarrow (\mathtt{v_1+v_2/m})$$

$$\frac{(\mathtt{e/m}) \longrightarrow (\mathtt{e_1/m})}{(\mathtt{v\ add\ e/m}) \longrightarrow (\mathtt{v\ add\ e_1/m})}$$

We need similar rules for sub.

Let in a Functional Language

Let expression

- Let us consider this simple language for expressions

```
<expr> ::= let var= <expr> in <expr> |
           <term> <addop> <expr> | <term>
<addop>::= add
<term> ::= var | val
```

- What is the semantics of a let expression?

Another Example

```
let x = 2 in
let y = x + x in
y * x
```

substitute
2 for x

-->
```
let y = 2 + 2 in
y * 2
```

-->
```
let y = 4       in
y * 2
```

substitute
4 for y

-->
```
4 * 2
```

-->
```
8
```

Moral: Let operates by *substituting* computed values for variables

Rules for let

$$\frac{(e_1/m) \;\longrightarrow\; (e_3/m)}{(\texttt{let } x\texttt{=}e_1 \texttt{ in } e_2/m) \;\longrightarrow\; (\texttt{??let } x\texttt{=}e_3 \texttt{ in } e_2/m)}$$

We can use the fact that e_1 is recursive and hypothetical reasoning.

Recording the value of a variable

`(let x=v in e/m) → ??(e/m@(x=v))`

If variables are just names for values, where shell we store the name-value association?

This is the role of the environment, storing name-value associations.

$$((x_1=v_1),(x_2=v_2)\ldots,(x_n=v_n))$$

Where we use the symbol @ to extend the environment with a new name-value association.

Extending an environment

What happens if `m` already contains `u`?

Suppose that we have

```
m=((x=1),(z=5),(y=3))
```

Then, if we extend m with the new pair (u=4), in symbols

```
m@(u=4)
```

We get:

```
m@(u=4)=((x=1),(z=5),(y=3),(u=4))
```

Extending an environment

Suppose that we have

```
m=((u=1),(z=5),(y=3))
```

Then, if we want to add the new pair (u=4), we need to generate a new variable, say w and replace it everywhere:

```
(let u=v in e/m) →(rename u to w in e/m@(w=v))
```

Question: how can we generate a new variable name?

Summary of the rules:

(F) $(\mathtt{x/m}) \longrightarrow (\mathtt{fetch(x,m)/m})$

(B) $(\mathtt{x\ add\ e/m}) \longrightarrow (\mathtt{fetch(x,m)\ add\ e/m})$

(C) $(\mathtt{v_1\ add\ v_2/m}) \longrightarrow (\mathtt{v_1+v_2/m})$

(D) $$\frac{(\mathtt{e/m}) \longrightarrow (\mathtt{e_1/m})}{(\mathtt{v\ add\ e/m}) \longrightarrow (\mathtt{v\ add\ e_1/m})}$$

(T) $$\frac{(\mathtt{e_1/m}) \longrightarrow (\mathtt{e_3/m})}{(\mathtt{let\ x=e_1\ in\ e_2/m}) \longrightarrow (\mathtt{let\ x=e_3\ in\ e_2/m})}$$

(L) $(\mathtt{let\ x=v\ in\ e/m}) \longrightarrow (\mathtt{e/m@(x=v)})$

$$\frac{}{\mathtt{e} \rightarrow^{0} \mathtt{e}} \qquad \frac{\mathtt{e} \rightarrow \mathtt{e'} \qquad \mathtt{e'} \rightarrow^{k} \mathtt{e''}}{\mathtt{e} \rightarrow^{k+1} \mathtt{e''}}$$

Example:

Let us call m= `(y=5,z=7)` `we have:`

`(let k=(1 add y)in(z add k)/m)` $\longrightarrow$5`(13/m@(k=6))`

Update in a Imperative Language

Assignment

- Let us consider this simple language for expressions

```
<prog>   ::= <assgn> | <assgn> ; <prog>
<assgn> ::= var := <expr>
<expr>   ::= <term> <addop> <expr> | <term>
<addop> ::= add
<term>   ::= var | val
```

- What is the semantics of an assignment?

Operational semantics for programs with assignments

$$(\texttt{prog/m}) \rightarrow (\texttt{prog/m})$$

Here `(prog/m)` is a configuration where prog is a program and m is a memory.

We can think about a memory as a set of (unique) assignments of variables to values:

$$\texttt{m} = ((x_1=v_1),(x_2=v_2)\ldots,(x_n=v_n))$$

The memory is the environment where the variable of an expression are defined.

Rules for assignment

$$\frac{(\mathtt{e}_1/\mathtt{m}) \;\longrightarrow\; (\mathtt{e}_2/\mathtt{m})}{(\mathtt{x{:=}e}_1\mathtt{;prog/m}) \;\longrightarrow\; (\mathtt{??\,x{:=}e}_2\mathtt{;prog/m})}$$

We can use hypothetical reasoning.

An example: recording the value of a variable

`(x:=v;prog/m)` $\longrightarrow$ `(prog/update(x,v,m))`

??

Where we use the function `update(x,v,m)` to update the value of the variable `x` in `m` to `v`.

Updating an environment

What happens if `m` does not contain `x`?

Suppose that we have

`m=((x=1),(z=5),(y=3))`

Then, if we update m with the following command

`update(x,4,m)`

We get:

`update(x,4,m)=((x=4),(z=5),(y=3))`

Initializing a variable

Suppose that we have

```
m=((u=1),(z=5),(y=3))
```

What shall we do if we have the following?

```
update(x,4,m)
```

Basically there are two strategies:

1- we create a new pair:

```
update(x,4,m)=((u=1),(z=5),(y=3),(x=4))
```

2- we give an error because the variable has not been initialized – how can we fix this?

Example:

Let us call m= `(x=3,z=6,u=0)` `we have:`

`(z:=x add 5 ; u:=u add z;p/m)` $\rightarrow$7 `(p/(x=3,z=8,u=8))`

Mutable vs Immutable Variables

- When we consider variables as names we are working with immutable variables (e.g. the part of OCaml we studied)
- When we consider variables as memory locations we are working with mutable variables (e.g. Python, c, etc.)
- Understanding how variables are managed is an important part to understand the semantics of a programming language.