Operational Semantics

Semantics Overview

We must specify for every C expression what happens when it is evaluated. (this is not exactly C, but close).

- This is the "meaning" of an expression

The definition of a programming language:

- The tokens --> lexical analysis
- The grammar --> syntactic analysis
- The typing rules --> semantic analysis
- The evaluation rules --> code generation and optimization

We have specified evaluation rules indirectly

- The compilation of a C program to a stack machine
- The evaluation rules of the stack machine

This is a complete description

- Why isn't it good enough?
  - This is an overspecification. We need the exact specification as that would allow more freedom in code generation and optimization. Overspecification restricts these choices

Assembly-language descriptions of language implementations have irrelevant detail

- Whether to use a stack machine or not
- Which way the stack grows
- How integers are represented
- The particular instruction set of the architecture

We need a complete description that is not overly restrictive. We need a high-level way of describing the behaviour of languages.

Another approach: reference implementations. Again, there will be artifacts of the particular way it was implemented that you didn't mean to be a part of the language (over-specification)

Many ways to specify semantics

- All equally precise/powerful in their specification (do not under/over-specify)
- Some more suitable to various tasks than others

Operational semantics

- Describes program evaluation via execution rules
  - on an abstract machine
- Most useful for specifying implementations
- Two types: big-step and small-step
  - Big-step semantics specify the value of the full expression in terms of its constituent expressions. Below, we will use big-step semantics for our language.
  - Small-step semantics specify how a single step (small step) is taken in the evaluation of a program. These are also called reduction semantics. For example, see Norrish's paper on small-step semantics for the C programming language: An abstract dynamic semantics for C

Denotational semantics

- Program's meaning is a mathematical function
  - A mathematical function that maps input to output
  - Elegant approach
  - Adds complexity through functions that we don't really need to consider for the purposes of describing an implementation

Axiomatic semantics

- Program behaviour described via logical formulae
  - If execution begins in state satisfying X, then it ends in state satisfying Y
  - X, Y formulas in logic
- Foundation of many program verification systems. e.g., static analysis systems to verify properties or discover bugs

Operational semantics

Notation: Logical rules of inference, as in type checking

Recall the typing judgement

```
Context |- e:C
```

In the given *context*, expression `e` has type `C`.

We use something similar for evaluation

```
Context |- e:v
```

In the given *context* (the meaning of context is different from the context for typing judgements), expression `e` evaluates to value `v`.

Example

```
Context |- e1: 5
Context |- e2 : 7
-----------------
Context |- e1+e2 : 12
```

The `Context` does not do much in this rule. In general, `Context` may provide mappings from variables to values for the variables used in `e1` and `e2`.

Example

```
y <-- x + 1
```

We track variables and their values with:

- An *environment*: *where* in memory a variable is
- A *store*: *what* is in the memory

For C/C++: environment maps *variables* to *memory locations*
store maps *memory locations* to *values*

A variable environment maps variables to locations

- Keeps track of which variables are in scope
- Tells us where those variables are

```
E = [a:l1, b:l2]
```

Store maps memory locations to values

```
S = [l1 --> 5, l2 --> 7]
```

`S' = S[12/l1]` defines a store S' such that

```
S'(l1) = 12
S'(l) = S(l) if l != l1
```

C++ values are objects

- All objects are instances of some type or class

`X(a1=l1, ..., an=ln)` is a C++ object where

- `X` is the class of the object
- `ai` are the attributes (including inherited ones)
- `li` is the location where the value of `ai` is stored

Notice that the specification is deliberately keeping things as abstract as possible (avoiding over-specification). e.g., the layout of the object is not specified.

There are constant values in C/C++ that do not have associated memory locations; they only have values., e.g., `(int)5`, `(bool)true`, (char const *)"constant string", \ldots

There is a special value `void`:

- No operations can be performed on it
- Concrete implementations might use any representation of `void` value

The evaluation judgement is:

```
s0, E,S |- e:v, S'
```

- Given `s0` the current (this/self) object
- Given `E` the current variable environment
- Given `S` the current store
- If the evaluation of `e` terminates then
  - The value of `e` is `v`
  - And the new store is `S'`

Notice: the current self object `s0` and the current environment `E` does not change by evaluating an expression. These are invariant under evaluation. However, the contents of the memory may change.

Also notice: there is a qualification which says that if the evaluation of `e` terminates. Read as "if `e` terminates..."

"Result" of evaluation is a value and a new store

- New store models the side-effects

Some things don't change

- The variable environment
- The operational semantics allows for non-terminating evaluations

Operational Semantics for C

Start with simple operational-semantic rules and work our way up to more complex rules.

```
------------------------------     [bool-true]
s0,E,S |- true: bool(true), S
```

```
-------------------------------   [bool-false]
s0,E,S |- false: bool(false), S

i is an integer literal
----------------------   [int]
s0,E,S |- i: int(i), S

s is a string literal
---------------------------------  [string-lit]
s0,E,S |- s: (char const *)(s), S

E(id) = lid
S(lid) = v
----------------  [id]
s0,E,S |- id:v,S

-------------------- [this]
s0,E,S|- this: s0, S

s0,E,S |- e:v, S1
E(id) = lid
S2 = S[v/lid]
--------------------------- [assignment]
s0,E,S |- id <-- e:v, S2
```

(e.g., `x <-- 1 + 1`)

```
s0,E,S |- e1:v1, S1
s0,E,S1 |- e2:v2, S2
--------------------------  [add]
s0,E,S |- e1+e2 : v1+v2, S2
```

(notice that the store used while evaluating `e2` includes the side-effects of evaluating `e1`). These stores also dictate the order of evaluation of the expressions --- e1 needs to be evaluated first to get S1 which is needed by evaluation of `e2`

```
s0,E,S |- e1:v1, S1
s0,E,S1 |- e2:v2, S2
---------------------- [stmt]
s0,E,S |- e1;e2 : v2,S2

s0,E,S |- e:v,S1
---------------------- [stmt-block]
s0,E,S |- {e} : v,S1
```

Example: Consider the expression

```
{ X <-- 7 + 5; 4; }
```

Let's say that initially, `s0=s0`, `E=x:l`, `l<-0`. Let's evaluate this expression in this context (start-state/environment/store).

```
   ....
   .....

-----------------------------------------   -------------------------------
s0,[X:l],[l<-0] |- X<-7+5:12,[l<-12}    s0,[x:l],[l:..] |- 4:4,[l:...]
-------------------------------------------------------------------------
s0,[X:l],[l<-0] |- {X <- 7+5; 4 } : 4, [l:12]
```

More evaluation rules

```
s0,E,S |- e1:bool(true),S1
s0,E,S1 |- e2:v,S2
------------------------------------  [ite-true]
s0,E,S |- if e1 then e2 else e3 : v,S2
```

```
s0,E,S |- e1:bool(false),S1
s0,E,S1 |- e3:v,S3
------------------------------------  [ite-false]
s0,E,S |- if e1 then e2 else e3 : v,S3

s0,E,S |- e1 : bool(false), S1
---------------------------------- [while-false]
s0,E,S |- while (e1) {e2} : void, S1

s0,E,S |- e1 : bool(true), S1
s0,E,S1 |- e2 : v, S2
s0,E,S2 |- while e1 {e2} : void, S3
---------------------------------- [while-true]
s0,E,S |- while (e1) {e2} : void, S3
```

Declarations

Partial rule

```
s0,E,S |- e1 : v1, S1
s0,?,? |- e2: v2, S2
-------------------------------------------
s0,E,S |- { Decl(id:T <-- e1); e2 } v2, S2
```

In what context should `e2` be evaluated?

- Environment like `E` but with a new binding of `id` to a fresh location `lnew`.
- Store like `S1` but with `lnew` mapped to `v1`.

We write `lnew = newloc(S)` to say that `lnew` is a location not already used in `S`.

- `newloc` is like the memory allocation function
- Notice that the spec does not say anything about stack, etc. Just that some memory location is allocated that is not already used.

Complete rule

```
s0,E,S |- e1 : v1, S1
lnew = newloc(S1)
s0,E[lnew/id],S1[v1/lnew] |- e2: v2, S2
-------------------------------------------
s0,E,S |- { Decl(id:T <-- e1); e2 } v2, S2
```

Allocation of a new object

Informal semantics of `new T`

- Allocate locations to hold all attributes of an object of class `T`
  - Essentially, allocate a new object
- Set attributes with their default values
- Evaluate the initializers and set the resulting attribute values
- Return the newly allocated object

Let's assume that for each type `A`, there is a default value `D_A`. In reality, C/C++ have a notion of uninitialized variables, and associated undefined behaviour.

- `D_A = void` for any `A` (could have had default values for some types, e.g., `D_int = int(0)`).

For a class `A`, we write

```
class(A) = (a1:T1 <-- e1, a2:T2 <-- e2, ..., an <-- en}
```

where

- `ai` are the attributes (including the inherited ones)
- `Ti` are the attributes' declared types
- `ei` are the initializers
- attributes are listed in the *greatest-ancestor-first* order
    - If C(c1,c2) < B(b1,b2) < A(a1,a2), then

      ```
      class(C) = (a1:.., a2:..., b1:.., b2:.., c1:.., c2:..)
      ```

```
class(T) = (a1:T1 <- e1, ..., an:Tn <- en)
l1 = newloc(S) for i = 1,..,n
v = T(a1=l1,..an=ln)
S1=S[DT1/l1,...,DTn/ln]
E'=[a1:l1, ..., an:ln]
v,E',S1 |- { a1 <-- e1; ... an <-- en; } : vn, S2
-----------------------------------------------------------
s0,E,S |- new T : v,S2
```

Notice that `E'` has nothing to do with `E`, `E'` is due to a completely different scope. Also notice that the evaluation of `e1`, ..., `en` follows the same order as in `class T` (greatest ancestore first). Also notice that the initializers are evaluated with a new value of the self object (`v`). Also notice that `vn` is ignored.

Summarizing:

- The first three steps allocate the object
- The remaining steps initialize it
    - By evaluating a sequence of assignments
- State in which the initializers are evaluated
    - `this` is the current object
    - Only the attributes are in scope (same as in typing)
    - Initial values of attributes are the defaults

Dispatch

Informal semantics of `e0.f(e1,...,en)`

- Evaluate the arguments in order `e1,..., en`
- Let `e0` be the target object
- Let `X` be the dynamic type of the target object
- Fetch from `f` the definition of `f` (with `n` args).
- Create `n` new locations and an environment that maps `f`'s formal arguments to those locations
- Initialize the locations with the actual arguments
- Set `self` to the target object and evaluate `f`'s body

For a class `A` and a method `f` of `A` (possibly inherited):

```
impl(A, f) = (x1, ..., xn, ebody)
```

where

- `xi` are the names of the formal arguments
- `ebody` is the body of the method

Complete rule

```
s0,E,S |- e1 : v1,S1
s0,E,S1 |- e2 : v2,S2
...
s0,E,S(n-1) |- en : vn,Sn
s0,E,Sn |- e0 : v0,S(n+1)
v0 = X(a1=l1, ..., am=lm)
impl(X,f) = (x1,..,xn, ebody)
```

```
lxi = newloc(Sn+1) for i=1..n
E' = [a1:l1, ..., am:lm][x1/lx1,...,xn/lxn]
S(n+2) = S(n+1)[v1/lx1,...,vn/lxn]
v0,E',S(n+2) |- ebody:v,S(n+3)
------------------------------------------- [dispatch]
s0,E,S |- e0.f(e1,...,en) : v, S(n+3)
```

We are interested in the class-tag of `v0` (`X`) and its attributes. Notice that this is the dynamic type of the object v0. (This provides dynamic dispatch).

What are the names in scope of the method `f`: all attributes and all the formal arguments. `E'` assigns locations to all these names.

The construction of `E'` is interesting because it two square brackets: this is because it is possible that a formal parameter may have the same name as an attribute name, and we want to capture this overriding semantics.

The function body is evaluated in an updated store, where the locations of formal arguments are replaced by expression values (call-by-value). Also the self object is v0. Notice that E' has nothing to do with `E` (static scoping). Dynamic scoping may have E' that is dependent on E.

We did not delete the locations `lxi` after finishing the execution of `ebody`. Is that a problem?