Operational Semantics of ChocoPy

Lecture 16-17

Lecture Outline

- ChocoPy operational semantics
- Motivation
- Notation
- The rules

Motivation

- We must specify for every ChocoPy expression what happens when it is evaluated
  - This is the "meaning" of an expression

- The definition of a programming language:
  - The tokens $\Rightarrow$ lexical analysis
  - The grammar $\Rightarrow$ syntactic analysis
  - The typing rules $\Rightarrow$ semantic analysis
  - The evaluation rules $\Rightarrow$ code generation and optimization

Evaluation Rules So Far

- So far, we specified the evaluation rules indirectly
  - We specified the compilation of ChocoPy to a stack machine
  - And we specified the evaluation rules of the stack machine

- This is a complete description
- Why isn't it good enough?

Assembly Language Description of Semantics

- Assembly-language descriptions of language implementation have too many irrelevant details
  - Whether to use a stack machine or not
  - Which way the stack grows
  - How integers are represented on a particular machine
  - The particular instruction set of the architecture
- We need a complete but not overly restrictive specification

Programming Language Semantics

- There are many ways to specify programming language semantics
- They are all equivalent but some are more suitable to various tasks than others
- Operational semantics
  - Describes the evaluation of programs on an abstract machine
  - Most useful for specifying implementations
  - This is what we will use for ChocoPy

Other Kinds of Semantics

- Denotational semantics
  - The meaning of a program is expressed as a mathematical object
  - Very elegant but quite complicated
- Axiomatic semantics
  - Useful for checking that programs satisfy certain correctness properties
    - e.g., that the quick sort function sorts an array
  - The foundation of many program verification systems

Introduction to Operational Semantics

- Once, again we introduce a formal notation
  - Using logical rules of inference, just like for typing
- Recall the typing judgment

$$\text{Context} \vdash e : C$$

(in the given Context, expression e has type C)

- We try something similar for evaluation

$$\text{Context} \vdash e : v$$

(in the given Context, expression e evaluates to value v)

Example of Inference Rule for Operational Semantics

- Example:

$$\frac{\text{Context} \vdash e_1 : 5 \qquad \text{Context} \vdash e_2 : 7}{\text{Context} \vdash e_1 + e_2 : 12}$$

- In general the result of evaluating an expression depends on the result of evaluating its subexpressions
- The logical rules specify everything that is needed to evaluate an expression

What Contexts Are Needed?

- Obs.: Contexts are needed to handle variables
- Consider the evaluation of y = x + 1
  - We need to keep track of values of variables
  - We need to allow variables to change their values during the evaluation
- We track variables and their values with:
  - *An* <u>environment</u> : tells us at what address in memory is the value of a variable stored
  - *A* <u>store</u> : tells us what is the contents of a memory location

Variable Environments

- A variable environment is a map from variable names to locations
- Tells in what memory location the value of a variable is stored
- Keeps track of which variables are in scope
- Example:

$$E = [a : l_1, b : l_2]$$

- To lookup a variable a in environment E we write $E(a)$

Stores

- A store maps memory locations to values
- Example:

$$S = [l_1 \to 5, l_2 \to 7]$$

- To lookup the contents of a location l_1 in store S we write $S(l_1)$
- To perform an assignment of 12 to location l_1 we write $S[12/l_1]$
  - This denotes a store S' such that

$$S'(l_1) = 12 \quad \text{and} \quad S'(l) = S(l) \text{ if } l \neq l_1$$

ChocoPy Values

- All values in ChocoPy are objects
  - All objects are instances of some class (the dynamic type of the object)
- To denote a ChocoPy object we use the notation $X(a_1 = l_1, \ldots, a_n = l_n)$ where
  - X is the dynamic type of the object
  - a_i are the attributes and methods (including those inherited)
  - l_i are the locations where the values of attributes and methods are stored

ChocoPy Values (Cont.)

- Special cases (classes without attributes)

Int(5)	the integer 5
Bool(true)	the boolean true
String(7, "ChocoPy")	the string "ChocoPy" of length 7

- There is a special value None that is a member of all types
  - No operations can be performed on it
  - Except for the test is
  - Concrete implementations might use null here

Operational Rules of ChocoPy

- The evaluation judgment is

$$E, S \vdash e : v, S', R'$$

read:

  - Given E the local variable environment
  - And S the current store
  - And R the return value
  - If the evaluation of expression or non-expression e terminates then
  - The returned value is v
  - And the new store is S'
  - And the new return value is R'
- v is _ if e is not an expression
- R' is _ if nothing has been returned

Notes

- The “result” of evaluating an expression is a value and a new store
- The “result” of evaluating a non-expression is _ and a new store
- Changes to the store model the side-effects
- The variable environment does not change
- The operational semantics allows for non-terminating evaluations
- We define one rule for each kind of syntactic structure

Operational Semantics for Base Values

$$\frac{}{E, S \vdash \text{True} : \text{Bool}(\text{true}), S, \_}$$

$$\frac{}{E, S \vdash \text{False} : \text{Bool}(\text{false}), S, \_}$$

$$\frac{\text{i is an integer literal}}{E, S \vdash i : \text{Int}(i), S, \_}$$

$$\frac{\begin{array}{c}\text{s is a string literal}\\ \text{n is the length of s}\end{array}}{E, S \vdash s : \text{String}(n,s), S, \_}$$

- No side effects in these cases
  (the store does not change)

Operational Semantics of Variable References

$$\frac{E(id) = l_{id} \quad S(l_{id}) = v}{E, S \vdash id : v, S, \_}$$

- Note the double lookup of variables
  - First from name to location
  - Then from location to value
- The store does not change

Operational Semantics of Assignment Expression

$$\frac{\begin{array}{c} E, S \vdash e : v, S_1, \_ \\ E(id) = l_{id} \\ S_2 = S_1[v/l_{id}] \end{array}}{E, S \vdash id = e : v, S_2, \_}$$

- A three step process
  - Evaluate the right hand side
    - $\Rightarrow$ a value <u>and</u> a new store S_1
  - Fetch the location of the assigned variable
  - The result is the value v and an updated store
- The environment does not change

Operational Semantics of Assignment Statement

$$\frac{\begin{array}{c} E, S \vdash e : v, S_1, \_ \\ E(id) = l_{id} \\ S_2 = S_1[v/l_{id}] \end{array}}{E, S \vdash id = e : \_, S_2, \_}$$

- Value of the statement is _

Operational Semantics of Conditionals

$$\frac{\begin{array}{c} E, S \vdash e_1 : \text{Bool(true)}, S_1, \_ \\ E, S_1 \vdash b_2 : \_, S_2, \_ \end{array}}{E, S \vdash \text{if } e_1\text{: } b_2 \text{ else: } b_3 : \_, S_2, \_}$$

- The "threading" of the store enforces an evaluation sequence
  - e_1 must be evaluated first to produce S_1
  - Then b_2 can be evaluated
- The result of evaluating e_1 is a boolean object
  - The typing rules ensure this
  - There is another, similar, rule for Bool(false)
  - The rules can be extended to handle elif

Operational Semantics of Sequences of Statements

$$\frac{\begin{array}{c} E, S \vdash e_1 : \_, S_1, \_ \\ E, S_1 \vdash e_2 : \_, S_2, \_ \\ \ldots \\ E, S_{n-1} \vdash e_n : \_, S_n, \_ \end{array}}{E, S \vdash e_1 NL \ldots NL \; e_n NL : \_, S_n, \_}$$

- NL stands for newline
- Again the threading of the store expresses the intended evaluation sequence
- But all the side-effects are collected

Operational Semantics of while (I)

$$\frac{E, S \vdash e_1 : \text{Bool(false)}, S_1, \_}{E, S \vdash \text{while } e_1 \text{: } b_2 : \_, S_1, \_}$$

- If e_1 evaluates to Bool(false) then the loop terminates immediately
  - With the side-effects from the evaluation of e_1
  - And with result value _
- The typing rules ensure that e_1 evaluates to a Boolean object

Operational Semantics of while (II)

$$\frac{\begin{array}{c} E, S \vdash e_1 : Bool(true), S_1, \_ \\ E, S_1 \vdash b_2 : \_, S_2, \_ \\ E, S_2 \vdash while\ e_1 : b_2 : \_, S_3, \_ \end{array}}{E, S \vdash while\ e_1 : b_2 : \_, S_3, \_}$$

- Note the sequencing ($S \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$)
- Note how looping is expressed
  - Evaluation of "while …" is expressed in terms of the evaluation of itself in another state

Operational Semantics of return (I)

$$\frac{E, S \vdash e : v, S_1, \_}{E, S \vdash \text{return } e : \_, S_1, v}$$

- Set R to the return value

Operational Semantics of return (I)

$$\frac{}{E, S \vdash \text{return} : \_, S, \text{None}}$$

- Set R to the return value

Propagate return values

- All existing rules for non-expressions are changed to propagate the return value

$$\frac{E, S \vdash e_1 : \text{Bool(true)}, S_1, \_ \qquad E, S_1 \vdash b_2 : \_, S_2, R}{E, S \vdash \text{if } e_1\text{: } b_2 \text{ else: } b_3 : \_, S_2, R}$$

Operational Semantics of Sequences of Statements with Return

$$\frac{\begin{array}{c} E, S \vdash e_1 : \_, S_1, \_ \\ E, S_1 \vdash e_2 : \_, S_2, \_ \\ \ldots \\ E, S_{i-1} \vdash e_i : \_, S_i, R \end{array}}{E, S \vdash e_1 NL \ldots NL\ e_n NL \ : \_, S_i, R}$$

- NL stands for newline
- Assume R is not _
- Skip execution of statements e_k for $k > i$

Operational Semantics of Function Invocation

- Consider the expression $f(e_1,\ldots,e_n)$
- Informal semantics:
  - Evaluate the arguments in order $e_1,\ldots,e_n$
  - Lookup the value of the function:
    - $S(E(f)) = (x_1, \ldots, x_n, y_1=e'_1, \ldots, y_k=e'_k, b_{body}, E_f)$
  - E_f is the environment at the time of defining f
  - $y_1,\ldots,y_k$ are the locally defined variables and nested functions
  - Create n new locations and an environment that maps f's formal arguments to those locations
  - Initialize the locations with the actual arguments
  - Create k new locations and map f's local variables to those locations
  - Initialize the locations with the initializers for those local variables

Operational Semantics of Function Invocation: not quite correct. Why?

$$
\frac{
\begin{array}{l}
S(E(f)) = (x_1, \ldots, x_n, y_1{=}e'_1, \ldots, y_k{=}e'_k, b_{body}, E_f) \\
E, S \vdash e_1 : v_1, S_1, \_ \\
\ldots \\
E, S_{n-1} \vdash e_n : v_n, S_n, \_ \\
l_{xi} = newloc(S_n) \text{ for } i = 1,\ldots,n \\
l_{yi} = newloc(S_n) \text{ for } i = 1,\ldots,k \\
E' = E[l_{x1}/x_1] \ldots [l_{xn}/x_n]\ [l_{y1}/y_1] \ldots [l_{yk}/y_k] \\
E', S_n \vdash e'_1 : v'_1, S_n, \_ \\
\ldots \\
E', S_n \vdash e'_k : v'_k, S_n, \_ \\
S_{n+1} = S_n[v_1/l_{x1},\ldots,v_n/l_{xn}, v'_1/l_{y1},\ldots,v'_k/l_{yk}] \\
E', S_{n+1} \vdash b_{body} : \_, S_{n+2}, R
\end{array}
}{
E, S \vdash f(e_1,\ldots,e_n) : R, S_{n+2}, \_
}
$$

Functions environment: a tricky example

```
def f() :
    x: int = 1
    def g() -> int:
        return x + 1
    def h() :
        x: int = 2
        print(g())
    h()
f()
```

Operational Semantics of Function Invocation

$$\frac{\begin{array}{l} S(E(f)) = (x_1, \ldots, x_n, y_1{=}e'_1, \ldots, y_k{=}e'_k, b_{body}, E_f) \\ E, S \vdash e_1 : v_1, S_1, \_ \\ \ldots \\ E, S_{n-1} \vdash e_n : v_n, S_n, \_ \\ l_{xi} = newloc(S_n) \text{ for } i = 1,\ldots,n \\ l_{yi} = newloc(S_n) \text{ for } i = 1,\ldots,k \\ E' = E_f[l_{x1}/x_1] \ldots [l_{xn}/x_n]\ [l_{y1}/y_1] \ldots [l_{yk}/y_k] \\ E', S_n \vdash e'_1 : v'_1, S_n, \_ \\ \ldots \\ E', S_n \vdash e'_k : v'_k, S_n, \_ \\ S_{n+1} = S_n[v_1/l_{x1}, \ldots, v_n/l_{xn}, v'_1/l_{y1}, \ldots, v'_k/l_{yk}] \\ E', S_{n+1} \vdash b_{body} : \_, S_{n+2}, R \end{array}}{E, S \vdash f(e_1,\ldots,e_n) : R, S_{n+2}, \_}$$

Operational Semantics of Function Invocation : no return in body

$$\frac{\begin{array}{l} S(E(f)) = (x_1, \ldots, x_n, y_1{=}e'_1, \ldots, y_k{=}e'_k, b_{body}, E_f) \\ E, S \vdash e_1 : v_1, S_1, \_ \\ \ldots \\ E, S_{n-1} \vdash e_n : v_n, S_n, \_ \\ l_{xi} = newloc(S_n) \text{ for } i = 1,\ldots,n \\ l_{yi} = newloc(S_n) \text{ for } i = 1,\ldots,k \\ E' = E_f[l_{x1}/x_1] \ldots [l_{xn}/x_n]\ [l_{y1}/y_1] \ldots [l_{yk}/y_k] \\ E', S_n \vdash e'_1 : v'_1, S_n, \_ \\ \ldots \\ E', S_n \vdash e'_k : v'_k, S_n, \_ \\ S_{n+1} = S_n[v_1/l_{x1}, \ldots, v_n/l_{xn}, v'_1/l_{y1}, \ldots, v'_k/l_{yk}] \\ E', S_{n+1} \vdash b_{body} : \_, S_{n+2}, \_ \end{array}}{E, S \vdash f(e_1, \ldots, e_n) : \text{None}, S_{n+2}, \_}$$

More Notation

- For a class A we write

$\text{class}(A) = (a_1 = e_1, \ldots, a_m = e_m)$ where

  - a_i are the attributes and methods (including the inherited ones)
  - e_i are the initializers or method defs

Operational Semantics of Method Dispatch

- Consider the expression $e_0.f(e_1,\ldots,e_n)$
- Informal semantics:
  - Evaluate e_0 to the target object
  - Evaluate the arguments in order $e_1,\ldots,e_n$
  - Let X be the dynamic type of the target object
  - Fetch from X 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
  - Create k new locations and map f's local variables to those locations
  - Initialize the locations with the initializers for those local variables

Operational Semantics of Dispatch

$$\frac{\begin{array}{l} E, S \vdash e_0 : v_0, S_0, \_ \\ v_0 = X(a_1 = l_1, \ldots, f = l_f, \ldots, a_m = l_m) \\ S_0(l_f) = (x_0, x_1, \ldots, x_n, y_1{=}e'_1, \ldots, y_k{=}e'_k, b_{body}, E_f) \\ E, S_0 \vdash e_1 : v_1, S_1, \_ \\ \ldots \\ E, S_{n-1} \vdash e_n : v_n, S_n, \_ \\ l_{xi} = newloc(S_n) \text{ for } i = 0,\ldots,n \\ l_{yi} = newloc(S_n) \text{ for } i = 1,\ldots,k \\ E' = E_f[l_{x0}/x_0][l_{x1}/x_1] \ldots [l_{xn}/x_n]\ [l_{y1}/y_1] \ldots [l_{yk}/y_k] \\ E', S_n \vdash e'_1 : v'_1, S_n, \_ \\ \ldots \\ E', S_n \vdash e'_k : v'_k, S_n, \_ \\ S_{n+1} = S_n[v_0/l_{x0}, v_1/l_{x1}, \ldots, v_n/l_{xn}, v'_1/l_{y1}, \ldots, v'_k/l_{yk}] \\ E', S_{n+1} \vdash b_{body} : \_, S_{n+2}, R \end{array}}{E, S \vdash e_0.f(e_1,\ldots,e_n) : R, S_{n+2}, \_}$$

Operational Semantics of Dispatch: no explicit return in body

$$\frac{\begin{array}{l} E, S \vdash e_0 : v_0, S_0, \_ \\ v_0 = X(a_1 = l_1, \ldots, f = l_f, \ldots, a_m = l_m) \\ S_0(l_f) = (x_0, x_1, \ldots, x_n, y_1 = e'_1, \ldots, y_k = e'_k, b_{body}, E_f) \\ E, S_0 \vdash e_1 : v_1, S_1, \_ \\ \ldots \\ E, S_{n-1} \vdash e_n : v_n, S_n, \_ \\ l_{xi} = newloc(S_n) \text{ for } i = 0,\ldots,n \\ l_{yi} = newloc(S_n) \text{ for } i = 1,\ldots,k \\ E' = E_f[l_{x0}/x_0][l_{x1}/x_1] \ldots [l_{xn}/x_n]\ [l_{y1}/y_1] \ldots [l_{yk}/y_k] \\ E', S_n \vdash e'_1 : v'_1, S_n, \_ \\ \ldots \\ E', S_n \vdash e'_k : v'_k, S_n, \_ \\ S_{n+1} = S_n[v_0/l_{x0}, v_1/l_{x1}, \ldots, v_n/l_{xn}, v'_1/l_{y1}, \ldots, v'_k/l_{yk}] \\ E', S_{n+1} \vdash b_{body} : \_, S_{n+2}, \_ \end{array}}{E, S \vdash e_0.f(e_1,\ldots,e_n) : None, S_{n+2}, \_}$$

Operational Semantics of Dispatch and Invocation. Notes.

- The body of a method/function is invoked with
  - E mapping formal arguments and local variables
  - S like the caller's except with actual arguments (and initializers) bound to the locations allocated for formals (and for local variables)
- The notion of the activation frame is implicit
  - New locations are allocated for actual arguments and local variables

Operational Semantics of function/method definitions: no global declarations

$y_1 = e'_1, \ldots, y_k = e'_k$ be the local variable and function definitions in f

$v = (x_1, \ldots, x_n, y_1=e'_1, \ldots, y_k=e'_k, b_{body}, E)$

$$\frac{}{E, S \vdash \text{def } f(x_1{:}T_1, \ldots, x_n{:}T_n) \text{ -> } T_0 : b_{body} : v, S, \_}$$

Operational Semantics of function/method definitions

Need an environment for global variables

$g_1,\ldots,g_L$ be the variables declared as global in f

$y_1 = e'_1, \ldots, y_k = e'_k$ be the local variable and function definitions in f

$E_f = E[???]$

$$\frac{v = (x_1, \ldots, x_n, y_1=e'_1, \ldots, y_k=e'_k, b_{body}, E_f)}{E, S \vdash \text{def } f(x_1{:}T_1,\ldots, x_n{:}T_n) \text{ -> } T_0 : b_{body} : v, S, \_}$$

Operational Rules of ChocoPy

- The evaluation judgment is

$$G, E, S \vdash e : v, S', R'$$

read:

  - Given G the global environment (similar to the variable environment)
  - And E the current variable environment
  - And S the current store
  - If the evaluation of expression or non-expression e terminates then
  - The returned value is v
  - And the new store is S' x
  - And the new return value is R'
- v is _ if e is not an expression
- R' is _ if nothing has been returned

Most rules remain unchanged: just prepend G to the context

Operational Semantics of Variable References

$$\frac{\begin{array}{c} E(id) = l_{id} \\ S(l_{id}) = v \end{array}}{G, E, S \vdash id : v, S, \_}$$

- Slight change to the rule

Operational Semantics of top-level program P

$y_1 = e'_1, \ldots, y_k = e'_k$ be the variable and function definitions in the top-level program P

$$\frac{\begin{array}{l} l_{yi} = \text{newloc(emptystore) for } i = 1,\ldots,k \\ E' = G = \text{emptyenv } [l_{y1}/y_1] \ldots [l_{yk}/y_k] \\ G, E', \text{emptystore} \vdash e'_1 : v'_1, \text{emptystore}, \_ \\ \ldots \\ G, E', \text{emptystore} \vdash e'_k : v'_k, \text{emptystore}, \_ \\ S' = \text{emptystore } [v'_1/l_{y1}] \ldots [v'_k/l_{yk}] \\ G, E', S' \vdash P : \_, S'', \_ \end{array}}{G, \text{emptyenv}, \text{emptystore} \vdash P : \_, S'', \_}$$

Operational Semantics of function/method definitions

$g_1,\ldots,g_L$ be the variables declared as global in f

$y_1 = e'_1, \ldots, y_k = e'_k$ be the local variable and function definitions in f

$E_f = E[G(g_1)/g_1]\ldots[G(g_L)/g_L]$

$$\frac{v = (x_1, \ldots, x_n, y_1 = e'_1, \ldots, y_k = e'_k, b_{body}, E_f)}{G, E, S \vdash \text{def } f(x_1 : T_1, \ldots, x_n : T_n) \text{ -> } T_0 : b_{body} : v, S, \_}$$

Operational Semantics of new object

- Consider the expression T()
- Informal semantics
  - Allocate new locations to hold the values for all attributes of an object of class T
    - Essentially, allocate a new object
  - Initialize those locations with the default values of attributes
  - Evaluate the initializers and set the resulting attribute values
  - Dispatch the __init__ method on the newly allocated object
  - Return the object

Operational Semantics of new object: T()

$$\frac{\begin{array}{l} class(T) = (a_1 = e_1, \ldots, a_m = e_m) \\ l_i = newloc(S) \text{ for } i = 1,\ldots,m \\ v = T(a_1 = l_1, \ldots, a_m = l_m) \\ G, G, S \vdash e_1 : v_1, S, \_ \\ \ldots \\ G, G, S \vdash e_m : v_m, S, \_ \\ S_1 = S[v_1/l_1, \ldots, v_m/l_m] \\ G, E, S_1 \vdash v.\_\_init\_\_() : None, S_2, \_ \end{array}}{G, E, S \vdash T() : v, S_2, \_}$$

Runtime Errors

Operational rules do not cover all cases

Consider for example the rule for dispatch:

$$\frac{\begin{array}{l} G, E, S \vdash e_0 : v_0, S_0, \_ \\ v_0 = X(a_1 = l_1, \ldots, f = l_f, \ldots, a_m = l_m) \\ S_0(l_f) = (x_0, x_1, \ldots, x_n, y_1{=}e'_1, \ldots, y_k{=}e'_k, b_{body}, E_f) \\ \ldots \end{array}}{G, E, S \vdash e_0.f(e_1,\ldots,e_n) : R, S_{n+2}, \_}$$

What happens if $S_0(l_f)$ is not defined or v_0 does not contain f?

Cannot happen in a well-typed program (Type safety theorem)

Runtime Errors (Cont.)

- There are some runtime errors that the type checker does not try to prevent
  - Division by zero
  - Index out of bounds (for string or list)
  - Operation on None (method dispatch, attribute read/write, list select/update)
  - Out of memory
- In such case the execution must abort gracefully
  - With an error message, not with segfault

Conclusions

- Operational rules are very precise
  - Nothing that matters is left unspecified
- Operational rules contain a lot of details
  - But not too many details
  - Read them carefully
- Most languages do not have a well specified operational semantics
- When portability is important an operational semantics becomes essential
  - But not always using the notation we used for ChocoPy