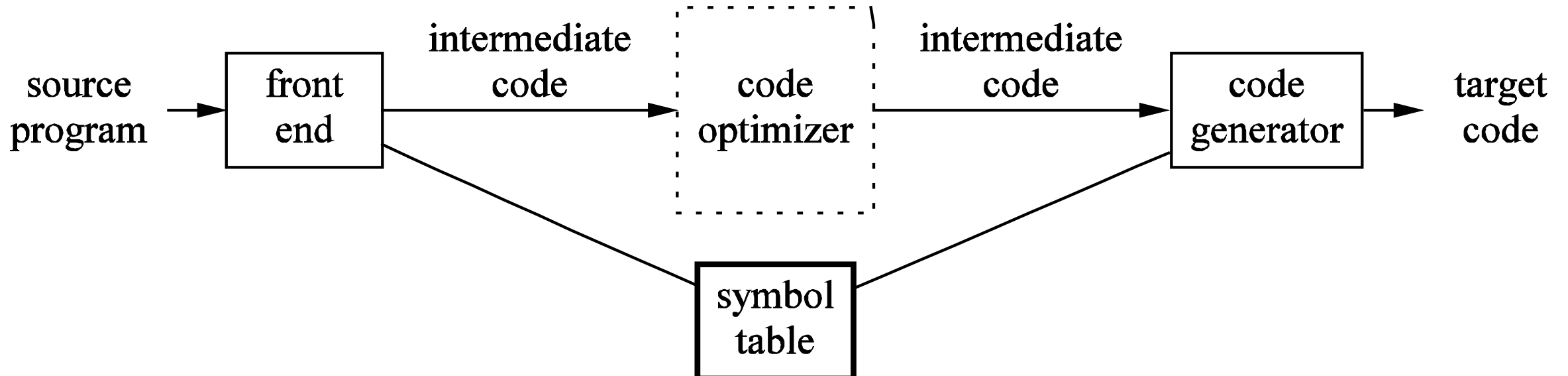


Operational Semantics

- The final phase is the code generator.
- The requirements on a code generator are severe.
- How to generator correct code?



Motivation

- Expressing the meaning of a programming language in natural language is error prone — — **ambiguous**
- Formal semantics gives an unambiguous definition of what a program written in the language should do — — **unambiguous**
 - ✓ Understand the **subtleties** of the language
 - ✓ Offer a **formal reference** and a **correctness definition** for implementers of tools (parsers, compilers, interpreters, debuggers, etc)
 - ✓ **Prove global properties** of any program written in the language, e.g., assertion
 - ✓ **Verify** programs against **formal specifications**
 - ✓ **Prove** two different programs are **equivalent/non-equivalent**
 - ✓ Form a computer readable version of the semantics, **an interpreter can be automatically generated** (full compiler generation is not yet feasible), like K Framework

Formal semantics

```
1 ▼ struct Point {  
2     x: f64,  
3     y: f64,  
4 }  
5  
6 ▼ impl Point {  
7     fn take_ownership(self) {  
8         println!("{}", self.x);  
9     }  
10 }  
11  
12 ▼ fn main() {  
13     let mut p = Point {x:1.0, y:2.0};  
14     p.take_ownership();  
15     p.x = 3.0;  
16     println!("{:?}", p.x);  
17  
18 }  
19
```

Any errors?

Formal semantics

Operational semantics:

- The meaning of a construct is specified by the computation it induces when it is executed on a machine. In particular, it is of interest **how** the effect of a computation is produced.

Denotational semantics:

- Meanings are modelled by mathematical objects that represent the effect of executing the constructs. Thus **only** the effect is of interest, not how it is obtained.

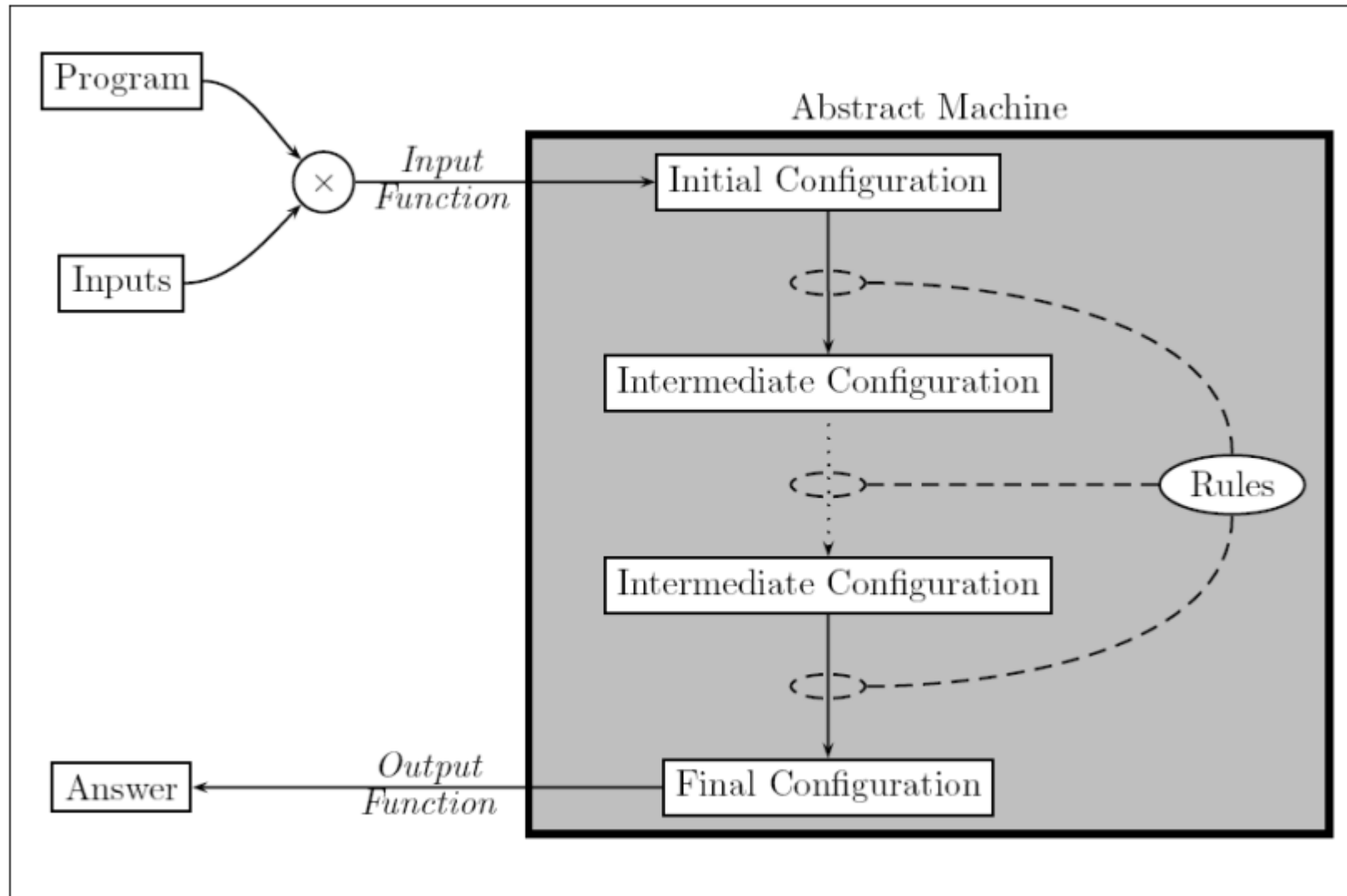
Axiomatic semantics:

- Specific properties of the effect of executing the constructs are expressed as **assertions**. Thus there may be aspects of the executions that are ignored.

Operational Semantics

- Operational semantics defines program executions:
 - ✓ Sequence of steps, formulated as transitions of an **abstract machine/interpreter**
- **Configurations** of the abstract machine include:
 - ✓ **Expression/statements** being evaluated/executed
 - ✓ **States**: abstract description of registers, memory and other data structures involved in computation
- Most useful for specifying implementations
- This is what we will use for Cool

Operational Semantics



Other Kinds of Semantics

Denotational semantics

- The meaning of a program is expressed as a **mathematical object/denotation**
- Very elegant but quite complicated
e.g., Functional languages often translate the language into **domain theory**

Axiomatic semantics

- The meaning of a program is described the **logical axioms, e.g., Hoare logic**
- Useful for checking that programs satisfy certain correctness properties using proof systems, e.g., **that the quick sort function sorts an array**
- The foundation of many program verification systems

Operational semantics

$z:=x; x:=y; y:=z$

Configuration: Variable \rightarrow Value, e.g., $[x \rightarrow 5, y \rightarrow 7, z \rightarrow 0]$

$z:=x; x:=y; y:=z, [x \rightarrow 5, y \rightarrow 7, z \rightarrow 0]$

$\Rightarrow x:=y; y:=z, [x \rightarrow 5, y \rightarrow 7, z \rightarrow 5]$

$\Rightarrow y:=z, [x \rightarrow 7, y \rightarrow 7, z \rightarrow 5]$

$\Rightarrow [x \rightarrow 7, y \rightarrow 5, z \rightarrow 5]$

- This explanation gives an abstraction of how the program is executed on a machine.
- It is important to observe that it is indeed an **abstraction**
- We ignore details such as the use of registers and addresses for variables.
- So the operational semantics is rather **independent of machine architectures and implementation strategies**.

Denotational semantics

$z:=x; x:=y; y:=z$

Mathematical object: a function $F: 2^{\text{Configuration}} \rightarrow 2^{\text{Configuration}}$

$F[z:=x] = \lambda c: c[z \rightarrow c(x)] \quad F[x:=y] = \lambda c: c[x \rightarrow c(y)] \quad F[y:=z] = \lambda c: c[y \rightarrow c(z)]$

$F[z:=x; x:=y; y:=z] = F[y:=z] \circ F[x:=y] \circ F[z:=x]$

$F[z:=x; x:=y; y:=z] ([x \rightarrow 5, y \rightarrow 7, z \rightarrow 0])$

$= (F[y:=z] \circ F[x:=y] \circ F[z:=x]) ([x \rightarrow 5, y \rightarrow 7, z \rightarrow 0])$

$= F[y:=z] (F[x:=y] (F[z:=x] ([x \rightarrow 5, y \rightarrow 7, z \rightarrow 0])))$

$= F[y:=z] (F[x:=y] ([x \rightarrow 5, y \rightarrow 7, z \rightarrow 5]))$

$= F[y:=z] ([x \rightarrow 7, y \rightarrow 7, z \rightarrow 5])$

$= [x \rightarrow 7, y \rightarrow 5, z \rightarrow 5]$

- The benefits: **abstracts away from how programs are executed.**
- Amounts to **reasoning about mathematical objects.**
- But, have to establish a firm **mathematical basis for denotational semantics**, and this task turns out **not to be entirely trivial**

Axiomatic semantics

$z:=x; x:=y; y:=z$

{Precondition} P {Postcondition}

$\{x=n \wedge y=m\} z:=x \{z=n \wedge y=m\}$

$\{z=n \wedge y=m\} x:=y \{z=n \wedge x=m\}$

$\{z=n \wedge x=m\} y:=z \{y=n \wedge x=m\}$

$\{x=n \wedge y=m\} z:=x; x:=y \{z=n \wedge x=m\}$

$\{x=n \wedge y=m\} z:=x; x:=y; y:=z \{y=n \wedge x=m\}$

- The axiomatic semantics provides a logical system for proving **partial correctness** properties of individual programs.
- **Partial correctness:** A program is partially correct, with respect to a precondition and a postcondition, if whenever the initial state fulfils the precondition and the program terminates, then the final state is guaranteed to fulfil the postcondition
- **Total correctness:** partial correctness + termination

Operational Semantics

- Small step semantics (structural operational semantics, SOS)
- Big step semantics (natural semantics)
 - ✓ differs from SOS by **hiding** even more execution details.

SOS

Configuration: a function Variable \rightarrow Value

- $z:=x; x:=y; y:=z, [x \rightarrow 5, y \rightarrow 7, z \rightarrow 0]$
- $\Rightarrow x:=y; y:=z, [x \rightarrow 5, y \rightarrow 7, z \rightarrow 5]$
- $\Rightarrow y:=z, [x \rightarrow 7, y \rightarrow 7, z \rightarrow 5]$
- $\Rightarrow [x \rightarrow 7, y \rightarrow 5, z \rightarrow 5]$

Natural semantics is represented by the derivation tree

$$\begin{array}{c}
 \frac{\langle z:=x, s_0 \rangle \rightarrow s_1 \quad \langle x:=y, s_1 \rangle \rightarrow s_2}{\langle z:=x; x:=y, s_0 \rangle \rightarrow s_2} \quad \langle y:=z, s_2 \rangle \rightarrow s_3 \\
 \hline
 \langle z:=x; x:=y; y:=z, s_0 \rangle \rightarrow s_3
 \end{array}$$

| | | |
|-------|-----|---|
| s_0 | $=$ | $[x \mapsto 5, y \mapsto 7, z \mapsto 0]$ |
| s_1 | $=$ | $[x \mapsto 5, y \mapsto 7, z \mapsto 5]$ |
| s_2 | $=$ | $[x \mapsto 7, y \mapsto 7, z \mapsto 5]$ |
| s_3 | $=$ | $[x \mapsto 7, y \mapsto 5, z \mapsto 5]$ |

$$\langle z:=x; x:=y; y:=z, s_0 \rangle \rightarrow s_3$$

hidden the explanation above of how it was actually obtained¹¹

Operational Semantics for COOL

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

$O, M, C \vdash e : T$

- Under **Context** (O, M, C) , e has type T

- We try something similar for evaluation

Context $\vdash e : v$

- Under **Context**, e evaluates to the value v

Example of Inference Rule for Operational Semantics

$$\frac{\begin{array}{l} \text{Context} \vdash e_1 : 2 \\ \text{Context} \vdash e_2 : 3 \end{array}}{\text{Context} \vdash e_1 + e_2 : 5} \quad \text{What Contexts Are Needed?}$$

- 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

Contexts

Contexts are needed to handle variables

$x = 1; y = x + 2; x = 3$

- 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 **environment E**: tells us at what address in memory is the value of a variable stored
- A **store S**: tells us what is the contents of a memory location

Variable Environments

- A variable environment E 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:

$$E = [x : l_1, y : l_2]$$

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

Stores

- A store S maps memory locations to values

Example:

$$S = [l_1 \rightarrow 2, l_2 \rightarrow 3]$$

- To lookup the contents of a location l_1 in store S we write $S(l_1)$
- To perform an assignment of 5 to location l_1 , we write $S[5/l_1]$
 - This denotes a new store S' such that

$$S'(l_1) = 5$$

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

Cool Values

- All values in Cool are objects
 - ✓ All objects are instances of some class (the dynamic type of the object)
- To denote a Cool object we use the notation

$$v = X(a_1 = l_1, \dots, a_n = l_n)$$

where

- ✓ X is the dynamic type of the object
- ✓ a_i 's are the attributes (including those inherited)
- ✓ l_i are the locations where the values of attributes are stored
- ✓ The value v is a member of class X containing the attributes a_1, \dots, a_n whose locations are l_1, \dots, l_n .

Cool Values (Cont.)

- Special cases (classes without attributes)
 - ✓ `Int(5)` the integer 5
 - ✓ `Bool(true)` the boolean true
 - ✓ `String(4, "Cool")` the string "Cool" of length 4
- There is a special value `void` that is a member of all types
 - ✓ No operations can be performed on it
 - ✓ Except for the test `isvoid`
 - ✓ Concrete implementations might use NULL here

Operational Rules of Cool

- The evaluation judgment is

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

read:

- ✓ Given so the current value of the **self** object
- ✓ E the current variable **environment**
- ✓ S the current **store**
- ✓ If the evaluation of e terminates then e evaluates to v , and resulting the new store is S'

Notes

- The “result” of evaluating an expression is a **value** and a **new store**
- Changes to the store model the side-effects
- The variable environment does not change, nor does the value of **self**
- **self** is just the object to which the identifier **self** refers if **self** appears in the expression.
- We do not place **self** in the environment and store?
- Because **self** is not a variable—it cannot be assigned to
- The operational semantics allows for nonterminating evaluations
- We define one rule for each kind of expression

Operational Semantics for Base Values

$so, E, S \vdash \text{true} : \text{Bool}(\text{true}), S$

$so, E, S \vdash \text{false} : \text{Bool}(\text{false}), S$

2 is an integer literal

“abc” is a string literal
3 is the length of s

$so, E, S \vdash 2 : \text{Int}(2), S$

$so, E, S \vdash s : \text{String}(3, \text{“abc”}), S$

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

Operational Semantics of Variable References

$$E(x) = l_x$$

$$S(l_x) = v$$

$$so, E, S \vdash x: v, S$$

Note the double lookup of variables

- First from name to location
- Then from location to value

The store does not change

A special case:

$$so, E, S \vdash \text{self}: so, S$$

Operational Semantics of Assignment

$$\frac{\begin{array}{c} \text{so, E, S} \mid - e: v, S_1 \\ E(x) = l_x \\ S_2 = S_1[v/l_x] \end{array}}{\text{so, E, S} \mid - x \leftarrow e: v, S_2}$$

A three step process

- Evaluate the right hand side e
 \Rightarrow a value v and a new store S_1
- Fetch the location l_x of the assigned variable x
- The result is the value v and an updated store S_2

Operational Semantics of Conditionals

$$\frac{\text{so, } E, S \vdash e_1 : \text{Bool}(\text{true}), S_1 \quad \text{so, } E, S_1 \vdash e_2 : v, S_2}{\text{so, } E, S \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : v, S_2}$$
$$\frac{\text{so, } E, S \vdash e_1 : \text{Bool}(\text{false}), S_1 \quad \text{so, } E, S_1 \vdash e_3 : v, S_2}{\text{so, } E, S \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : v, S_2}$$

- The “threading” of the store enforces an evaluation sequence
 - e_1 must be evaluated first to produce S_1
 - Then e_2 or e_3 can be evaluated
- The result of evaluating e_1 is a boolean object
 - The typing rules ensure this

Operational Semantics of Sequences

$$\frac{\begin{array}{l} \text{so, E, } S \vdash e_1 : v_1, S_1 \\ \text{so, E, } S_1 \vdash e_2 : v_2, S_2 \end{array}}{\text{so, E, } S \vdash e_1; e_2 : v_2, S_2}$$

- Only the last value is used
- But all the side-effects are collected in stores

Operational Semantics of while (I)

$$\frac{\text{so, E, S} \vdash e_1 : \text{Bool}(\text{false}), S_1}{\text{so, E, S} \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{void}, S_1}$$
$$\frac{\begin{array}{l} \text{so, E, S} \vdash e_1 : \text{Bool}(\text{true}), S_1 \\ \text{so, E, } S_1 \vdash e_2 : v, S_2 \\ \text{so, E, } S_2 \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{void}, S_3 \end{array}}{\text{so, E, S} \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : \text{void}, S_3}$$

- 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 **void**
 - The typing rules ensure that e_1 evaluates to a Bool object
- Otherwise
 - 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**
 - The result **v** of e_2 is discarded, only the side-effect is preserved

Operational Semantics of let Expressions (I)

$$\frac{\begin{array}{l} \text{so, } E, S \vdash e_1 : v_1, S_1 \\ \text{so, } ?, ? \vdash e_2 : v_2, S_2 \end{array}}{\text{so, } E, S \vdash \text{let } x : T \leftarrow e_1 \text{ in } e_2 : v_2, S_2}$$
$$\frac{\begin{array}{l} \text{so, } E, S \vdash e_1 : v_1, S_1 \\ l_{\text{new}} = \text{newloc}(S_1) \\ \text{so, } E[l_{\text{new}}/x], S_1[v_1/l_{\text{new}}] \vdash e_2 : v_2, S_2 \end{array}}{\text{so, } E, S \vdash \text{let } x : T \leftarrow e_1 \text{ in } e_2 : v_2, S_2}$$

- What is the context in which e_2 must be evaluated?
 - Environment like E but with a new binding of x to a fresh location l_{new}
 - Store like S_1 but with l_{new} mapped to v_1
- $l_{\text{new}} = \text{newloc}(S)$: l_{new} is a location that is not already used in S
 - Think of newloc as the dynamic memory allocation function

Default Values

For each class A there is a default value denoted by D_A

- $D_{\text{int}} = \text{Int}(0)$
- $D_{\text{bool}} = \text{Bool}(\text{false})$
- $D_{\text{string}} = \text{String}(0, "")$
- $D_A = \text{void}$ (for another class A)

For a class A we write

$\text{class}(A) = (a_1:T_1 \leftarrow e_1, \dots, a_n:T_n \leftarrow e_n)$ // class mapping

where

- a_i are the attributes (including the inherited ones)
- T_i are their declared types
- e_i are the initializers

Operational Semantics of new

- Consider the expression `new 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
 - ✓ Return the newly allocated object
- Observation: `new SELF_TYPE` allocates an object with the same dynamic type as `self`

Operational Semantics of new

allocate the object

initialize it

$$\begin{array}{l}
 \left\{ \begin{array}{l}
 T_0 = \text{if } T == \text{SELF_TYPE and so} = X(\dots) \text{ then } X \text{ else } T \\
 \text{class}(T_0) = (a_1 : T_1 \leftarrow e_1, \dots, a_n : T_n \leftarrow e_n) \\
 l_i = \text{newloc}(S) \text{ for } i = 1, \dots, n
 \end{array} \right. \\
 \left\{ \begin{array}{l}
 v = T_0(a_1 = l_1, \dots, a_n = l_n) \\
 E' = [a_1 : l_1, \dots, a_n : l_n] \\
 S_1 = S[D_{T_1}/l_1, \dots, D_{T_n}/l_n] \\
 v, E', S_1 \vdash \{ a_1 \leftarrow e_1; \dots; a_n \leftarrow e_n; \} : v_n, S_2
 \end{array} \right.
 \end{array}$$

$$\text{so, } E, S \vdash \text{new } T : v, S_2$$

Only the attributes are in scope (same as in typing)

Operational Semantics of Method Dispatch

- Consider the expression $e_0.f(e_1, \dots, e_n)$
- Informal semantics:
 1. Evaluate the arguments in order e_1, \dots, e_n
 2. Evaluate e_0 to the target object
 3. Let X be the dynamic type of the target object from e_0
 4. Fetch from X the definition of f (with n args.)
 5. Create n new locations and an environment that maps f 's formal arguments to those locations
 6. Initialize the locations with the actual arguments
 7. Set $self$ to the target object and evaluate f 's body

Operational Semantics of Method Dispatch

$$\begin{array}{l} \text{so, } E, S \vdash e_1 : v_1, S_1 \\ \text{so, } E, S_1 \vdash e_2 : v_2, S_2 \\ \dots \\ \text{so, } E, S_{n-1} \vdash e_n : v_n, S_n \end{array}$$

Evaluate the arguments in order

$$\text{so, } E, S_n \vdash e_0 : v_0, S_{n+1} \quad \leftarrow \text{Evaluate } e_0 \text{ to the target object}$$

$$v_0 = X(a_1 = l_1, \dots, a_m = l_m)$$

$$\text{impl}(X, f) = (x_1, \dots, x_n, e_{\text{body}})$$

$$l_{x_i} = \text{newloc}(S_{n+1}) \text{ for } i = 1, \dots, n$$

$$E' = [x_1 : l_{x_1}, \dots, x_n : l_{x_n}, a_1 : l_1, \dots, a_m : l_m]$$

Create n new locations and an environment that maps f's formal arguments to those locations

$$S_{n+2} = S_{n+1}[v_1/l_{x_1}, \dots, v_n/l_{x_n}]$$

Initialize the locations with the actual arguments

$$v_0, E', S_{n+2} \vdash e_{\text{body}} : v, S_{n+3}$$

evaluate f's body

$$\text{so, } E, S \vdash e_0.f(e_1, \dots, e_n) : v, S_{n+3}$$

For a class A and a method f of A (possibly inherited) we write:
(implementation mapping)

$\text{impl}(A, f) = (x_1, \dots, x_n, e_{\text{body}})$ where

- x_i are the names of the formal arguments

- e_{body} is the body of the method

Operational Semantics of Static Method Dispatch

so, E, S $\vdash e_1 : v_1, S_1$
 so, E, S₁ $\vdash e_2 : v_2, S_2$

...

so, E, S_{n-1} $\vdash e_n : v_n, S_n$

so, E, S_n $\vdash e_0 : v_0, S_{n+1}$

$v_0 = X(a_1 = l_1, \dots, a_m = l_m)$

$\text{impl}(X, f) = (x_1, \dots, x_n, e_{\text{body}})$

$l_{xi} = \text{newloc}(S_{n+1})$ for $i = 1, \dots, n$

$E' = [x_1 : l_{x1}, \dots, x_n : l_{xn}, a_1 : l_1, \dots, a_m : l_m]$

$S_{n+2} = S_{n+1}[v_1/l_{x1}, \dots, v_n/l_{xn}]$

$v_0, E', S_{n+2} \vdash e_{\text{body}} : v, S_{n+3}$

so, E, S $\vdash e_0.f(e_1, \dots, e_n) : v, S_{n+3}$

so, E, S $\vdash e_1 : v_1, S_1$
 so, E, S₁ $\vdash e_2 : v_2, S_2$

...

so, E, S_{n-1} $\vdash e_n : v_n, S_n$

so, E, S_n $\vdash e_0 : v_0, S_{n+1}$

$v_0 = X(a_1 = l_1, \dots, a_m = l_m)$

$\text{impl}(T, f) = (x_1, \dots, x_n, e_{\text{body}})$

$l_{xi} = \text{newloc}(S_{n+1})$ for $i = 1, \dots, n$

$E' = [x_1 : l_{x1}, \dots, x_n : l_{xn}, a_1 : l_1, \dots, a_m : l_m]$

$S_{n+2} = S_{n+1}[v_1/l_{x1}, \dots, v_n/l_{xn}]$

$v_0, E', S_{n+2} \vdash e_{\text{body}} : v, S_{n+3}$

so, E, S $\vdash e_0@T.f(e_1, \dots, e_n) : v, S_{n+3}$

Runtime Errors

$$\begin{array}{l} \text{so, } E, S \mid - e_1 : v_1, S_1 \\ \text{so, } E, S_1 \mid - e_2 : v_2, S_2 \end{array}$$

$$\begin{array}{l} \dots \\ \text{so, } E, S_{n-1} \mid - e_n : v_n, S_n \\ \text{so, } E, S_n \mid - e_0 : v_0, S_{n+1} \end{array}$$

$$v_0 = X(a_1 = l_1, \dots, a_m = l_m)$$

$$\text{impl}(X, f) = \text{not defined?}$$

$$l_{xi} = \text{newloc}(S_{n+1}) \text{ for } i = 1, \dots, n$$

$$E' = [x_1 : l_{x1}, \dots, x_n : l_{xn}, a_1 : l_1, \dots, a_m : l_m]$$

$$S_{n+2} = S_{n+1}[v_1/l_{x1}, \dots, v_n/l_{xn}]$$

$$v_0, E', S_{n+2} \mid - e_{\text{body}} : v, S_{n+3}$$

$$\text{so, } E, S \mid - e_0.f(e_1, \dots, e_n) : v, S_{n+3}$$

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
 - A dispatch on void
 - Division by zero
 - Substring out of range
 - Heap overflow
- In such case the execution must abort gracefully
 - With an error message, not with segfault

Conclusion

- Operational rules are very precise
 - Nothing that matters is left unspecified
- Operational rules contain a lot of details
 - But not too many details, no stack or heap
 - 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 Cool

Reading

1. An Executable **Formal Semantics of C** with Applications ,Chucky Ellison and Grigore Rosu, POPL'12, ACM, pp 533-544. 2012,
<http://fsl.cs.illinois.edu/FSL/papers/2011/ellison-rosu-2011-tr/ellison-rosu-2011-tr-public.pdf>
2. K-Java: A Complete **Semantics of Java**, Denis Bogdanas and Grigore Rosu, POPL'15, ACM, pp 445-456. 2015,
<http://fsl.cs.illinois.edu/FSL/papers/2015/bogdanas-rosu-2015-popl/bogdanas-rosu-2015-popl-public.pdf>
3. KJS: A Complete **Formal Semantics of JavaScript** , Daejun Park and Andrei Stefanescu and Grigore Rosu, PLDI'15, ACM, pp 346-356. 2015,
<http://fsl.cs.illinois.edu/FSL/papers/2015/park-stefanescu-rosu-2015-pldi/park-stefanescu-rosu-2015-pldi-public.pdf>