Operational Semantics, Part 2

Martin Kellogg

Agenda

- Review: basics of operational semantics
- Operational semantics of Cool
- (if time): introduction to static analysis
  - further if time: get into abstract interpretation

Agenda

- **Review: basics of operational semantics**
- Operational semantics of Cool
- (if time): introduction to static analysis
  - further if time: get into abstract interpretation

Review: High-level Idea of Op. Sem.

- An ***operational semantics*** is a precise way of specifying how to evaluate a program.
  - A **formal semantics** tells you what each expression means.

Review: High-level Idea of Op. Sem.

- An *operational semantics* is a precise way of specifying how to evaluate a program.
  - A **formal semantics** tells you what each expression means.
- Meaning depends on **context**: a *variable environment* will map variables to memory locations and a ***store*** will map memory locations to values.

Review: High-level Idea of Op. Sem.

- An *operational semantics* is a precise way of specifying how to evaluate a program.
  - A **formal semantics** tells you what each expression means.
- Meaning depends on **context**: a *variable environment* will map variables to memory locations and a ***store*** will map memory locations to values.
  - **environment**: names -> (abstract) locations
  - **store**: (abstract) locations -> values

Review: High-level Idea of Op. Sem.

- An *operational semantics* is a precise way of specifying how to evaluate a program.
  - A **formal semantics** tells you what each expression means.
- Meaning depends on **context**: a *variable environment* will map variables to memory locations and a ***store*** will map memory locations to values.
  - **environment**: names -> (abstract) locations
  - **store**: (abstract) locations -> values
- We will specify Cool's semantics via **logical rules of inference** that specify how to compute the "next step" in the program

Review: Operational Rules of Cool

- The ***evaluation judgment*** is

  $$\mathbf{so}, E, S \vdash e : v, S'$$

- read as:

Review: Operational Rules of Cool

- The ***evaluation judgment*** is

  $$\textbf{so}, E, S \vdash e : v, S'$$

- read as:
  - Given **so**, the current value of the `self` object;
  - and **E**, the current variable environment;
  - and **S**, the current store;
  - and if the evaluation of **e** ***terminates***, then
  - the returned value is **v**
  - and the new store is **S'**

Review: Operational Semantics for Base Values

$$\frac{}{so, E, S \vdash \texttt{true} : \mathrm{Bool(true)}, S}$$

$$\frac{}{so, E, S \vdash \texttt{false} : \mathrm{Bool(false)}, S}$$

$$\frac{i \text{ is any integer literal}}{so, E, S \vdash \texttt{i} : \mathrm{Int}(i), S}$$

$$\frac{\begin{array}{l} s \text{ is any string literal} \\ n \text{ is the length of } s \end{array}}{so, E, S \vdash \texttt{s} : \mathrm{String}(s, n), S}$$

Review: Operational Semantics for Variables

$$\frac{E(\mathtt{id}) = \mathtt{l}_{\mathtt{id}} \qquad S(\mathtt{l}_{\mathtt{id}}) = v}{so, E, S \vdash \mathtt{id} : v, S}$$

- Note the **double lookup** of variables
  - First from name to location (at compile time)
  - Then from location to value (at run time)

Review: Operational Semantics for Assignments

$$\frac{\text{so}, E, S \vdash e : v, S_1 \qquad E(\texttt{id}) = l_{\texttt{id}} \qquad S_2 = S_1[v/l_{\texttt{id}}]}{\text{so}, E, S \vdash \texttt{id <- } e : v, S_2}$$

- A three-step process:
  - Evaluate the right-hand side to get a value v *and* a new store S_1
  - Fetch the location of the assigned variable
  - The result is the value v and an updated store S_2

Operational Semantics for Conditionals

Operational Semantics for Conditionals

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : Bool(true), S_1 \\ so, E, S_1 \vdash e_2 : v, S_2 \end{array}}{so, E, S \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi} : v, S_2}$$

Operational Semantics for Conditionals

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : Bool(true), S_1 \\ so, E, S_1 \vdash e_2 : v, S_2 \end{array}}{so, E, S \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi} : v, S_2}$$

- The "**threading**" of the store enforces an evaluation sequence:

Operational Semantics for Conditionals

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : \text{Bool(true)}, S_1 \\ so, E, S_1 \vdash e_2 : v, S_2 \end{array}}{so, E, S \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ fi} : v, S_2}$$

- The "**threading**" of the store enforces an evaluation sequence:
  - e_1 must be evaluated first to produce S_1

Operational Semantics for Conditionals

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : Bool(true), S_1 \\ so, E, S_1 \vdash e_2 : v, S_2 \end{array}}{so, E, S \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ fi} : v, S_2}$$

- The "threading" of the store enforces an evaluation sequence:
  - e_1 must be evaluated first to produce S_1
  - then e_2 can be evaluated.

Operational Semantics for Conditionals

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : Bool(true), S_1 \\ so, E, S_1 \vdash e_2 : v, S_2 \end{array}}{so, E, S \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi} : v, S_2}$$

- The "**threading**" of the store enforces an evaluation sequence:
  - e_1 must be evaluated first to produce S_1
  - then e_2 can be evaluated.
- The result of evaluating e_1 is a boolean object
  - the type rules ensure this

Operational Semantics for Conditionals

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : Bool(true), S_1 \\ so, E, S_1 \vdash e_2 : v, S_2 \end{array}}{so, E, S \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \texttt{ fi} : v, S_2}$$

- The "threading" of the store enforces an evaluation sequence:
  - e_1 must be evaluated first to produce S_1
  - then e_2 can be evaluated.
- The result of evaluating e_1 is a boolean object
  - the type rules ensure this
  - there is another, similar, rule for Bool(false)

Operational Semantics for Sequences

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

Operational Semantics for Sequences

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots \\ so, E, S_{n-1} \vdash e_n : v_n, S_n \end{array}}{so, E, S \vdash \{\, e_1 \;;\; \ldots \;;\; e_n \;;\, \} : v_n, S_n}$$

- Again, the threading of the store expresses the intended execution sequence

Operational Semantics for Sequences

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots \\ so, E, S_{n-1} \vdash e_n : v_n, S_n \end{array}}{so, E, S \vdash \{\ e_1\ ;\ \ldots\ ;\ e_n\ ;\ \} : v_n, S_n}$$

- Again, the threading of the store expresses the intended execution sequence
- Only the last value is used

Operational Semantics for Sequences

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots \\ so, E, S_{n-1} \vdash e_n : v_n, S_n \end{array}}{so, E, S \vdash \{\ e_1\ ;\ \ldots\ ;\ e_n\ ;\ \} : v_n, S_n}$$

- Again, the threading of the store expresses the intended execution sequence
- Only the last value is used
- But, all side-effects are collected (how?)

Operational Semantics for While (1)

Operational Semantics for While (1)

$$\frac{so, E, S \vdash e_1 : \text{Bool(false)}, S_1}{so, E, S \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : \text{void}, S_1}$$

Operational Semantics for While (1)

$$\frac{so, E, S \vdash e_1 : \text{Bool(false)}, S_1}{so, E, S \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : \text{void}, S_1}$$

- If e_1 evaluates to Bool(false), then the loop terminates immediately

Operational Semantics for While (1)

$$\frac{\text{so}, E, S \vdash e_1 : \text{Bool(false)}, S_1}{\text{so}, E, S \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : \text{void}, S_1}$$

- If e_1 evaluates to Bool(false), then the loop terminates immediately
  - with the side-effects from the evaluation of e_1

Operational Semantics for While (1)

$$\frac{so, E, S \vdash e_1 : \text{Bool(false)}, S_1}{so, E, S \vdash \texttt{while } e_1 \texttt{ loop } e_2 \texttt{ pool} : \text{void}, S_1}$$

- If e_1 evaluates to Bool(false), then the loop terminates immediately
  - with the side-effects from the evaluation of e_1
  - and with the (arbitrary) result of void

Operational Semantics for While (1)

$$\frac{so, E, S \vdash e_1 : \text{Bool(false)}, S_1}{so, E, S \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : \text{void}, S_1}$$

- If e_1 evaluates to Bool(false), then the loop terminates immediately
  - with the side-effects from the evaluation of e_1
  - and with the (arbitrary) result of void
- The type rules ensure that e_1 evaluates to a boolean object

Operational Semantics for While (1)

$$\frac{so, E, S \vdash e_1 : \text{Bool(false)}, S_1}{so, E, S \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : \text{void}, S_1}$$

- If e_1 evaluates to Bool(false), then the loop terminates immediately
  - with the si
  - and with th
- The type rules

In-class exercise: given this rule for a **false** loop guard, what do you think the rule for a **true** loop guard looks like?

- In groups of 2 or 3, write down a rule.
- I will collect these; put your UCIDs/emails on it (mostly graded on completion)

(for those reading later online, this is the answer to the in-class exercise on the previous slide)

Operational Semantics for While (2)

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

(for those reading later online, this is the answer to the in-class exercise on the previous slide)

Operational Semantics for While (2)

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

- Note the sequencing (S -> S_1 -> S_2 -> S_3)

(for those reading later online, this is the answer to the in-class exercise on the previous slide)

Operational Semantics for While (2)

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

- Note the sequencing ($S \rightarrow S_1 \rightarrow S_2 \rightarrow S_3$)
- Note how looping is expressed (**recursively!**)
  - Evaluation of a `while` loop is expressed in terms of evaluating a `while` loop in another state

(for those reading later online, this is the answer to the in-class exercise on the previous slide)

Operational Semantics for While (2)

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

- Note the sequencing (S -> S_1 -> S_2 -> S_3)
- Note how looping is expressed (**recursively!**)
    - Evaluation of a `while` loop is expressed in terms of evaluating a `while` loop in another state
- The result of evaluating e_2 is discarded; only the side-effects are kept

Operational Semantics for Let (1)

Operational Semantics for Let (1)

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ so, ?, ? \vdash e_2 : v_2, S_2 \end{array}}{so, E, S_1 \vdash \texttt{let}\ id : T\ \texttt{<-}\ e_1\ \texttt{in}\ e_2 : v_2, S_2}$$

Operational Semantics for Let (1)

$$\frac{so, E, S \vdash e_1 : v_1, S_1 \qquad so, ?, ? \vdash e_2 : v_2, S_2}{so, E, S_1 \vdash \texttt{let}\ id : T\ \texttt{<-}\ e_1\ \texttt{in}\ e_2 : v_2, S_2}$$

- What is the context in which e_2 should be evaluated?

Operational Semantics for Let (1)

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ so, ?, ? \vdash e_2 : v_2, S_2 \end{array}}{so, E, S_1 \vdash \texttt{let}\ \mathbf{id} : \mathbf{T}\ \texttt{<-}\ e_1\ \texttt{in}\ e_2 : v_2, S_2}$$

- What is the context in which e_2 should be evaluated?
    - Environment should be like **E** but with a new binding of **id** to a fresh location $\texttt{l}_{\texttt{new}}$

Operational Semantics for Let (1)

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ so, ?, ? \vdash e_2 : v_2, S_2 \end{array}}{so, E, S_1 \vdash \texttt{let}\ id : T\ \texttt{<-}\ e_1\ \texttt{in}\ e_2 : v_2, S_2}$$

- What is the context in which e_2 should be evaluated?
  - Environment should be like **E** but with a new binding of **id** to a fresh location $\texttt{l}_{\texttt{new}}$
  - Store like S_1 but with $\texttt{l}_{\texttt{new}}$ mapped to v_1

Operational Semantics for Let (2)

- We write $\mathtt{l}_{\mathtt{new}}$ = $\mathtt{newloc(S)}$ to say that $\mathtt{l}_{\mathtt{new}}$ is a location that is not already used in **S**

Operational Semantics for Let (2)

- We write l_{new} = `newloc(S)` to say that l_{new} is a location that is not already used in **S**
  - Think of `newloc` as the dynamic memory allocation function (or as reserving stack space)

Operational Semantics for Let (2)

- We write $\mathtt{l}_{\mathtt{new}} = \mathtt{newloc}(S)$ to say that $\mathtt{l}_{\mathtt{new}}$ is a location that is not already used in S
  - Think of `newloc` as the dynamic memory allocation function (or as reserving stack space)
- This lets(haha) us write the correct let rule:

$$\frac{\begin{array}{c} so, E, S \vdash e_1 : v_1, S_1 \\ \mathtt{l}_{\mathtt{new}} = \mathtt{newloc}(S_1) \\ so, E[\mathtt{l}_{\mathtt{new}}/id], S_1[v_1/\mathtt{l}_{\mathtt{new}}] \vdash e_2 : v_2, S_2 \end{array}}{so, E, S_1 \vdash \mathtt{let}\ id : T \mathtt{<-}\ e_1\ \mathrm{in}\ e_2 : v_2, S_2}$$

Warning: The Going Gets Tough

- Now we're going to do some **very difficult** rules

Warning: The Going Gets Tough

- Now we're going to do some **very difficult** rules
  - new, dispatch

Warning: The Going Gets Tough

- Now we're going to do some **very difficult** rules
    - `new`, dispatch
- This may initially seem tricky
    - How could that possibly work?
    - What's going on here?

Warning: The Going Gets Tough

- Now we're going to do some **very difficult** rules
  - `new`, dispatch
- This may initially seem tricky
  - How could that possibly work?
  - What's going on here?
- Once you’ve studied them a bit, hopefully you’ll agree they’re actually **quite elegant**
  - But they will probably not seem that way at first

Operational Semantics of `new`

- Consider the expression `new T`
- Its informal semantics are:

Operational Semantics of `new`

- Consider the expression `new T`
- Its informal semantics are:
    - Allocate new locations to hold the values for all attributes of an object of class T
        - Essentially, allocate space for a new object

Operational Semantics of `new`

- Consider the expression `new T`
- Its informal semantics are:
    - Allocate new locations to hold the values for all attributes of an object of class T
        - Essentially, allocate space for a new object
    - Initialize those locations with the ***default values*** of attributes

Operational Semantics of `new`

- Consider the expression `new T`
- Its informal semantics are:
    - Allocate new locations to hold the values for all attributes of an object of class T
        - Essentially, allocate space for a new object
    - Initialize those locations with the ***default values*** of attributes
    - Evaluate the initializers and set the resulting attribute values

Operational Semantics of `new`

- Consider the expression `new T`
- Its informal semantics are:
  - Allocate new locations to hold the values for all attributes of an object of class T
    - Essentially, allocate space for 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

Default Values

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

Default Values

- For each class **A** there is a default value denoted by D_A
  - D_{Int} = **Int(0)**

Default Values

- For each class **A** there is a default value denoted by D_A
  - D_{Int} = Int(0)
  - D_{Bool} = Bool(0)

Default Values

- For each class **A** there is a default value denoted by D_A
  - D_{Int} = Int(0)
  - D_{Bool} = Bool(0)
  - D_{String} = String(0, “”)

Default Values

- For each class **A** there is a default value denoted by D_A
  - D_{Int} = Int(0)
  - D_{Bool} = Bool(0)
  - D_{String} = String(0, “”)
  - D_A = void   for all other classes **A**

More Notation

More Notation

- For each class **A** we write

$$\text{class}(\mathbf{A}) = (\mathbf{a_1} : \mathbf{T_1} \text{ <- } \mathbf{e_1}, \ldots, \mathbf{a_n} : \mathbf{T_n} \text{ <- } \mathbf{e_n})$$

where

More Notation

- For each class **A** we write

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

where

- a_i are the attributes (including inherited ones)

More Notation

- For each class **A** we write

$$\text{class}(\mathbf{A}) = (\mathbf{a}_1 : \mathbf{T}_1 \text{ <- } \mathbf{e}_1, \ldots, \mathbf{a}_n : \mathbf{T}_n \text{ <- } \mathbf{e}_n)$$

where

- $\mathbf{a}_i$ are the attributes (including inherited ones)
- $\mathbf{T}_i$ are their declared types

More Notation

- For each class **A** we write

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

where

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

More Notation

This is exactly the *class map* from PA2!

- For each class A we write

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

where

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

Operational Semantics for new

Operational Semantics for new

$$\frac{\begin{array}{c} T_0 = \text{if } T = \textbf{SELF\_TYPE} \text{ and } \textbf{so} = X(\ldots) \text{ then } X \text{ else } T \\ \text{class}(T_0) = (a_1 : T_1 \text{ <- } e_1, \ldots, a_n : T_n \text{ <- } e_n) \\ \forall\, i \in [1 \ldots n],\ \mathtt{l}_i = \mathtt{newloc}(S) \\ v = T_0(a_1 = \mathtt{l}_1, \ldots, a_n = \mathtt{l}_n) \\ S_1 = S[D_{T1}/\mathtt{l}_1, \ldots, D_{Tn}/\mathtt{l}_n] \\ E' = [a_1 : \mathtt{l}_1, \ldots, a_n : \mathtt{l}_n] \\ \textbf{so}, E', S_1 \vdash \{a_1 \text{ <- } e_1 ; \ldots ; a_n \text{ <- } e_n ;\} : v_n, S_2 \end{array}}{\textbf{so}, E, S \vdash \mathtt{new}\ T : v, S_2}$$

Operational Semantics for new

$$\frac{\begin{array}{c} T_0 = \text{if } T = \textbf{SELF\_TYPE} \text{ and } \textbf{so} = X(\ldots) \text{ then } X \text{ else } T \\ \text{class}(T_0) = (a_1 : T_1 \text{ <- } e_1, \ldots, a_n : T_n \text{ <- } e_n) \\ \forall\, i \in [1 \ldots n],\ \mathtt{l}_i = \mathtt{newloc(S)} \\ v = T_0(a_1 = \mathtt{l}_1, \ldots, a_n = \mathtt{l}_n) \\ S_1 = S[D_{T1}/\mathtt{l}_1, \ldots, D_{Tn}/\mathtt{l}_n] \\ E' = [a_1 : \mathtt{l}_1, \ldots, a_n : \mathtt{l}_n] \\ \textbf{so}, E', S_1 \vdash \{a_1 \texttt{ <- } e_1 \texttt{;} \ \ldots \ \texttt{;}\ a_n \texttt{ <- } e_n \texttt{;}\} : v_n, S_2 \end{array}}{\textbf{so}, E, S \vdash \mathtt{new\ T} : v, S_2}$$

if the desired type is **SELF_TYPE**, use the **so** object; otherwise use the type named in the expression (**T**)

Operational Semantics for new

$$T_0 = \text{if } T = \textbf{SELF\_TYPE} \text{ and } \textbf{so} = X(\ldots) \text{ then } X \text{ else } T$$

$$\text{class}(T_0) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n)$$

$$\forall\, i \in [1\ldots n],\ l_i = \texttt{newloc}(S)$$

$$v = T_0(a_1 = l_1, \ldots, a_n = l_n)$$

$$S_1 = S[D_{T1}/l_1, \ldots, D_{Tn}/l_n]$$

$$E' = [a_1 : l_1, \ldots, a_n : l_n]$$

$$\frac{\textbf{so}, E', S_1 \vdash \{a_1 \leftarrow e_1 ; \ \ldots \ ; a_n \leftarrow e_n ;\} : v_n, S_2}{\textbf{so}, E, S \vdash \texttt{new } T : v, S_2}$$

fetch the template for the class to instantiate

Operational Semantics for new

$$\dfrac{\begin{array}{c} T_0 = \text{if } T = \textbf{SELF\_TYPE} \text{ and } \textbf{so} = X(\ldots) \text{ then } X \text{ else } T \\ \text{class}(T_0) = (a_1 : T_1 \leftarrow e_1, \ldots, a_n : T_n \leftarrow e_n) \\ \forall\, i \in [1 \ldots n],\ l_i = \texttt{newloc}(S) \\ v = T_0(a_1 = l_1, \ldots, a_n = l_n) \\ S_1 = S[D_{T1}/l_1, \ldots, D_{Tn}/l_n] \\ E' = [a_1 : l_1, \ldots, a_n : l_n] \\ \textbf{so}, E', S_1 \vdash \{a_1 \leftarrow e_1\,;\ \ldots\ ;\, a_n \leftarrow e_n\,;\} : v_n, S_2 \end{array}}{\textbf{so}, E, S \vdash \texttt{new } T : v, S_2}$$

make space for each of its attributes (now, we've *allocated* the new object)

Operational Semantics for new

$$\frac{\begin{array}{c} T_0 = \text{if } T = \textbf{SELF\_TYPE} \text{ and } \textbf{so} = X(\ldots) \text{ then } X \text{ else } T \\ \text{class}(T_0) = (a_1 : T_1 \texttt{<-} e_1, \ldots, a_n : T_n \texttt{<-} e_n) \\ \forall\, i \in [1\ldots n],\ \mathtt{l}_i = \mathtt{newloc}(S) \\ v = T_0(a_1 = \mathtt{l}_1, \ldots, a_n = \mathtt{l}_n) \\ S_1 = S[D_{T1}/\mathtt{l}_1, \ldots, D_{Tn}/\mathtt{l}_n] \\ E' = [a_1 : \mathtt{l}_1, \ldots, a_n : \mathtt{l}_n] \\ so, E', S_1 \vdash \{a_1 \texttt{<-} e_1 ;\ \ldots\ ; a_n \texttt{<-} e_n ;\} : v_n, S_2 \end{array}}{so, E, S \vdash \mathtt{new}\ T : v, S_2}$$

create a new value for the newly-created object; make each attribute point to the appropriate new location (this step is the start of *initialization*)

Operational Semantics for new

$$
\frac{\begin{array}{c}
T_0 = \text{if } T = \textbf{SELF\_TYPE} \text{ and } \textbf{so} = X(\ldots) \text{ then } X \text{ else } T \\
\text{class}(T_0) = (a_1 : T_1 \texttt{<-} e_1, \ldots, a_n : T_n \texttt{<-} e_n) \\
\forall\, i \in [1 \ldots n],\ \texttt{l}_i = \texttt{newloc}(S) \\
v = T_0(a_1 = \texttt{l}_1, \ldots, a_n = \texttt{l}_n) \\
S_1 = S[D_{T1}/\texttt{l}_1, \ldots, D_{Tn}/\texttt{l}_n] \\
E' = [a_1 : \texttt{l}_1, \ldots, a_n : \texttt{l}_n] \\
\textbf{so}, E', S_1 \vdash \{a_1 \texttt{<-} e_1 ; \ldots ; a_n \texttt{<-} e_n ;\} : v_n, S_2
\end{array}}{\textbf{so}, E, S \vdash \texttt{new } T : v, S_2}
$$

create a new store with each new attribute location set to the **default value** for its type

Operational Semantics for new

$$\frac{\begin{array}{c} T_0 = \text{if } T = \textbf{SELF\_TYPE} \text{ and } \textbf{so} = X(\ldots) \text{ then } X \text{ else } T \\ class(T_0) = (a_1 : T_1 \texttt{<-} e_1, \ldots, a_n : T_n \texttt{<-} e_n) \\ \forall\, i \in [1 \ldots n],\ \mathtt{l}_i = \mathtt{newloc}(S) \\ v = T_0(a_1 = \mathtt{l}_1, \ldots, a_n = \mathtt{l}_n) \\ S_1 = S[D_{T1}/\mathtt{l}_1, \ldots, D_{Tn}/\mathtt{l}_n] \\ E' = [a_1 : \mathtt{l}_1, \ldots, a_n : \mathtt{l}_n] \\ so, E', S_1 \vdash \{a_1 \texttt{<-} e_1 ;\ \ldots\ ; a_n \texttt{<-} e_n ;\} : v_n, S_2 \end{array}}{so, E, S \vdash \mathtt{new}\ T : v, S_2}$$

create a new environment with only the attributes **in-scope** (in which to evaluate the initializers)

Operational Semantics for new

$$\frac{\begin{array}{c} T_0 = \text{if } T = \textbf{SELF\_TYPE} \text{ and } \textbf{so} = X(\ldots) \text{ then } X \text{ else } T \\ class(T_0) = (a_1 : T_1 \texttt{<-} e_1, \ldots, a_n : T_n \texttt{<-} e_n) \\ \forall\, i \in [1 \ldots n],\ \mathtt{l}_i = \mathtt{newloc}(S) \\ v = T_0(a_1 = \mathtt{l}_1, \ldots, a_n = \mathtt{l}_n) \\ S_1 = S[D_{T1}/\mathtt{l}_1, \ldots, D_{Tn}/\mathtt{l}_n] \\ E' = [a_1 : \mathtt{l}_1, \ldots, a_n : \mathtt{l}_n] \\ \textbf{so}, E', S_1 \vdash \{a_1 \texttt{<-} e_1\ ;\ \ldots\ ;\ a_n \texttt{<-} e_n\ ;\} : v_n, S_2 \end{array}}{\textbf{so}, E, S \vdash \mathtt{new}\ T : v, S_2}$$

evaluate all of the initializers, keeping the side-effects in S_2

Trivia Break: Real World Languages

English is the single-most widely-spoken and only official language in this West African country, which, with over 230 million people, is the most populous country in Africa (and its former capital, Lagos, is one of Africa's largest cities). The country's linguistic diversity is a microcosm of Africa as a whole, with significant numbers of native speakers of languages from the three major African language families: Afroasiatic, Nilo-Saharan and Niger-Congo.

Name the country and any one language of African origin that is spoken there by at least 2 million people.

Trivia Break: Math

This Austrian mathematician moved to New Jersey in a rather circuitous way: after the Anschluss in 1938, the Nazis found him - previously a lecturer at the University of Vienna - fit for conscription. He fled across the Soviet Union, sailed to Japan and then on to San Francisco, and then traveled across the US to take up a position at the Institute for Advanced Study (IAS) in Princeton. Toward the end of his own life, fellow IAS researcher Albert Einstein confided that his "own work no longer meant much, that he came to the Institute merely ... to have the privilege of walking home with [him]". Though his work spanned several areas of mathematics, philosophy, and logic, he is most famous for his Incompleteness Theorem.

Operational Semantics of Method Dispatch

- Consider the expression `e`$_0$`.f(e`$_1$`,...,e`$_n$`)`
- Its informal semantics are:

Operational Semantics of Method Dispatch

- Consider the expression $\mathtt{e_0.f(e_1, ..., e_n)}$
- Its informal semantics are:
  - Evaluate the arguments in order $\mathbf{e_1}$, ... , $\mathbf{e_n}$

Operational Semantics of Method Dispatch

- Consider the expression $\texttt{e}_0\texttt{.f(e}_1\texttt{,...,e}_n\texttt{)}$
- Its informal semantics are:
  - Evaluate the arguments in order $\mathbf{e_1}$, ... , $\mathbf{e_n}$
  - Evaluate $\mathbf{e_0}$ to the target object

Operational Semantics of Method Dispatch

- Consider the expression `e`$_0$`.f(e`$_1$`, . . . , e`$_n$`)`
- Its informal semantics are:
    - Evaluate the arguments in order $\mathbf{e_1}, \ldots, \mathbf{e_n}$
    - Evaluate $\mathbf{e_0}$ to the target object
    - Let **X** be the dynamic type of the target object

Operational Semantics of Method Dispatch

- Consider the expression `e`$_0$`.f(e`$_1$`, ..., e`$_n$`)`
- Its informal semantics are:
  - Evaluate the arguments in order $\mathbf{e_1}, \ldots, \mathbf{e_n}$
  - Evaluate $\mathbf{e_0}$ to the target object
  - Let **X** be the dynamic type of the target object
  - Fetch from **X** the definition of `f` (with *n* args)

Operational Semantics of Method Dispatch

- Consider the expression $\texttt{e}_0\texttt{.f(e}_1\texttt{, . . . , e}_n\texttt{)}$
- Its informal semantics are:
  - Evaluate the arguments in order $\mathbf{e}_1, \dots, \mathbf{e}_n$
  - Evaluate $\mathbf{e}_0$ to the target object
  - 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

Operational Semantics of Method Dispatch

- Consider the expression `e₀.f(e₁,...,eₙ)`
- Its informal semantics are:
  - Evaluate the arguments in order $e_1, \ldots, e_n$
  - Evaluate e_0 to the target object
  - 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

Operational Semantics of Method Dispatch

- Consider the expression `e`$_0$`.f(e`$_1$`,...,e`$_n$`)`
- Its informal semantics are:
  - Evaluate the arguments in order $\mathbf{e_1}, \ldots, \mathbf{e_n}$
  - Evaluate $\mathbf{e_0}$ to the target object
  - 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
  - Set self to the target object and evaluate `f`'s body

More Notation

- For a class **A** and a method `f` of **A** (possibly inherited) we write:

More Notation

- For a class **A** and a method `f` of **A** (possibly inherited) we write:

$$\text{imp}(\mathbf{A}, \texttt{f}) = (\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{e}_{\mathbf{body}})$$

where:

More Notation

- For a class **A** and a method `f` of **A** (possibly inherited) we write:

$$\text{imp}(\mathbf{A}, f) = (\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{e}_{\mathbf{body}})$$

where:

- $\mathbf{x}_i$ are the names of the formal arguments

More Notation

- For a class $\mathbf{A}$ and a method `f` of $\mathbf{A}$ (possibly inherited) we write:

$$\text{imp}(\mathbf{A}, \texttt{f}) = (\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{e}_{\mathbf{body}})$$

where:

- $\mathbf{x}_i$ are the names of the formal arguments
- $\mathbf{e}_{\mathbf{body}}$ is the body of the method

More Notation

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

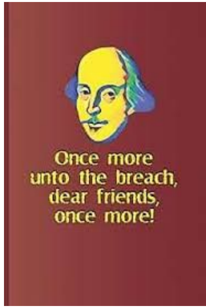
$$\mathrm{imp}(A, f) = (\mathbf{x}_1, \ldots, \mathbf{x}_n, \mathbf{e}_{\mathbf{body}})$$

This is exactly the ***implementation map*** from PA2!

where:

- $\mathbf{x}_i$ are the names of the formal arguments
- $\mathbf{e}_{\mathbf{body}}$ is the body of the method

Operational Semantics for Method Dispatch



Operational Semantics for Method Dispatch

$$\frac{\begin{array}{l}
so, E, S \vdash e_1 : v_1, S_1 \qquad so, E, S_1 \vdash e_2 : v_2, S_2 \\
\ldots \qquad so, E, S_{n-1} \vdash e_n : v_n, S_n \\
so, E, S_n \vdash e_0 : v_0, S_{n+1} \\
v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\
imp(X, f) = (x_1, \ldots, x_n, e_{body}) \\
\forall\, i \in [1 \ldots n],\ l_{xi} = \mathtt{newloc}(S_{n+1}) \\
E' = [x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1, \ldots, a_m : l_m] \\
S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] \\
v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3}
\end{array}}{so, E, S \vdash e_0.\mathtt{f}(e_1, \ldots, e_n) : v, S_{n+3}}$$

Operational Semantics for Method Dispatch

$$
\frac{
\begin{array}{l}
so, E, S \vdash e_1 : v_1, S_1 \qquad so, E, S_1 \vdash e_2 : v_2, S_2 \\
\ldots \qquad\qquad\qquad\qquad\quad so, E, S_{n-1} \vdash e_n : v_n, S_n \\
so, E, S_n \vdash e_0 : v_0, S_{n+1} \\
v_0 = X(a_1 = \mathtt{l}_1, \ldots, a_m = \mathtt{l}_m) \\
\mathrm{imp}(X, \mathtt{f}) = (x_1, \ldots, x_n, e_{body}) \\
\forall\, i \in [1 \ldots n], \mathtt{l}_{xi} = \mathtt{newloc}(S_{n+1}) \\
E' = [x_1 : \mathtt{l}_{x1}, \ldots, x_n : \mathtt{l}_{xn}, a_1 : \mathtt{l}_1, \ldots, a_m : \mathtt{l}_m] \\
S_{n+2} = S_{n+1}[v_1/\mathtt{l}_{x1}, \ldots, v_n/\mathtt{l}_{xn}] \\
v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3}
\end{array}
}{
so, E, S \vdash e_0.\mathtt{f}(e_1, \ldots, e_n) : v, S_{n+3}
}
$$

evaluate all of the arguments

Operational Semantics for Method Dispatch

$$\frac{\begin{array}{l} so, E, S \vdash e_1 : v_1, S_1 \qquad so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots \qquad\qquad\qquad\qquad so, E, S_{n-1} \vdash e_n : v_n, S_n \\ so, E, S_n \vdash e_0 : v_0, S_{n+1} \\ v_0 = X(a_1 = \mathtt{l}_1, \ldots, a_m = \mathtt{l}_m) \\ \text{imp}(X, \mathtt{f}) = (x_1, \ldots, x_n, e_{body}) \\ \forall\, i \in [1 \ldots n],\ \mathtt{l}_{\mathtt{x}i} = \mathtt{newloc}(S_{n+1}) \\ E' = [x_1 : \mathtt{l}_{\mathtt{x}1}, \ldots, x_n : \mathtt{l}_{\mathtt{xn}}, a_1 : \mathtt{l}_1, \ldots, a_m : \mathtt{l}_m] \\ S_{n+2} = S_{n+1}[v_1/\mathtt{l}_{\mathtt{x}1}, \ldots, v_n/\mathtt{l}_{\mathtt{xn}}] \\ v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3} \end{array}}{so, E, S \vdash e_0.\mathtt{f}(e_1, \ldots, e_n) : v, S_{n+3}}$$

evaluate the receiver object (= object on which method is called)

Operational Semantics for Method Dispatch

$$\frac{\begin{array}{ll} so, E, S \vdash e_1 : v_1, S_1 & so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots & so, E, S_{n-1} \vdash e_n : v_n, S_n \\ so, E, S_n \vdash e_0 : v_0, S_{n+1} & \\ v_0 = X(a_1 = l_1, \ldots, a_m = l_m) & \\ imp(X, f) = (x_1, \ldots, x_n, e_{body}) & \\ \forall\, i \in [1 \ldots n],\ l_{xi} = newloc(S_{n+1}) & \\ E' = [x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1, \ldots, a_m : l_m] & \\ S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] & \\ v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3} & \end{array}}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}}$$

find the receiver's type and attributes

Operational Semantics for Method Dispatch

$$\frac{\begin{array}{ll} so, E, S \vdash e_1 : v_1, S_1 & so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots & so, E, S_{n-1} \vdash e_n : v_n, S_n \\ so, E, S_n \vdash e_0 : v_0, S_{n+1} & \\ v_0 = X(a_1 = l_1, \ldots, a_m = l_m) & \\ imp(X, f) = (x_1, \ldots, x_n, e_{body}) & \\ \forall\, i \in [1 \ldots n],\ l_{xi} = \mathtt{newloc}(S_{n+1}) & \\ E' = [x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1, \ldots, a_m : l_m] & \\ S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] & \\ v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3} & \end{array}}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}}$$

find formals and body

Operational Semantics for Method Dispatch

$$\frac{\begin{array}{ll} so, E, S \vdash e_1 : v_1, S_1 & so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots & so, E, S_{n-1} \vdash e_n : v_n, S_n \\ so, E, S_n \vdash e_0 : v_0, S_{n+1} & \\ v_0 = X(a_1 = l_1, \ldots, a_m = l_m) & \\ imp(X, f) = (x_1, \ldots, x_n, e_{body}) & \\ \forall i \in [1 \ldots n], l_{xi} = newloc(S_{n+1}) & \\ E' = [x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1, \ldots, a_m : l_m] & \\ S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] & \\ v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3} & \end{array}}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}}$$

call by reference or by value?

Aside: Call by Reference vs Call by Value

Aside: Call by Reference vs Call by Value

- Cool uses ***call by reference***
  - when a function is called, only the pointer to each argument is copied
  - modifications to the arguments are reflected at the call-site

Aside: Call by Reference vs Call by Value

- Cool uses ***call by reference***
  - when a function is called, only the pointer to each argument is copied
  - modifications to the arguments are reflected at the call-site
- The alternative is ***call by value***
  - when a function is called, a full copy of each argument is passed to the callee
  - this is fine for e.g., integers, but for objects it gets expensive quickly

Aside: Call by Reference vs Call by Value

- Cool uses ***call by reference***
  - when a function is called, only the pointer to each argument is copied
  - modifications to the arguments are reflected at the call-site
- The alternative is ***call by value***
  - when a function is called, a **full copy** of each argument is passed to the callee
  - this is fine for e.g., integers, but for objects it gets expensive quickly
- Which does C support?

Operational Semantics for Method Dispatch

$$\frac{\begin{array}{ll} so, E, S \vdash e_1 : v_1, S_1 & so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots & so, E, S_{n-1} \vdash e_n : v_n, S_n \\ so, E, S_n \vdash e_0 : v_0, S_{n+1} & \\ v_0 = X(a_1 = \mathtt{l}_1, \ldots, a_m = \mathtt{l}_m) & \\ imp(X, \mathtt{f}) = (x_1, \ldots, x_n, e_{body}) & \\ \forall\, i \in [1 \ldots n],\ \mathtt{l}_{xi} = \mathtt{newloc}(S_{n+1}) & \\ E' = [x_1 : \mathtt{l}_{x1}, \ldots, x_n : \mathtt{l}_{xn}, a_1 : \mathtt{l}_1, \ldots, a_m : \mathtt{l}_m] & \\ S_{n+2} = S_{n+1}[v_1/\mathtt{l}_{x1}, \ldots, v_n/\mathtt{l}_{xn}] & \\ v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3} & \end{array}}{so, E, S \vdash e_0.\mathtt{f}(e_1, \ldots, e_n) : v, S_{n+3}}$$

call by reference, only allocate space for copies of the pointers

Operational Semantics for Method Dispatch

$$\mathrm{so}, E, S \vdash e_1 : v_1, S_1 \qquad \mathrm{so}, E, S_1 \vdash e_2 : v_2, S_2$$
$$\ldots \qquad \mathrm{so}, E, S_{n-1} \vdash e_n : v_n, S_n$$
$$\mathrm{so}, E, S_n \vdash e_0 : v_0, S_{n+1}$$
$$v_0 = X(a_1 = \mathtt{l}_1, \ldots, a_m = \mathtt{l}_m)$$
$$\mathrm{imp}(X, \mathtt{f}) = (x_1, \ldots, x_n, e_{body})$$
$$\forall\, i \in [1 \ldots n],\ \mathtt{l}_{xi} = \mathtt{newloc}(S_{n+1})$$
$$E' = [x_1 : \mathtt{l}_{x1}, \ldots, x_n : \mathtt{l}_{xn}, a_1 : \mathtt{l}_1, \ldots, a_m : \mathtt{l}_m]$$
$$S_{n+2} = S_{n+1}[v_1/\mathtt{l}_{x1}, \ldots, v_n/\mathtt{l}_{xn}]$$
$$v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3}$$
$$\overline{\mathrm{so}, E, S \vdash e_0.\mathtt{f}(e_1, \ldots, e_n) : v, S_{n+3}}$$

new environment, with the formals and the attributes of the receiver in-scope

Operational Semantics for Method Dispatch

so, E, S ⊢ … S_2

…

… , S_n

so, E, S_n

v_0 = X(a_1

imp(X, f) = (x_1, …, x_n, e_{body})

Do you think the **order matters** here? What could go wrong if the formals were after the attributes?

$\forall\, i \in [1 \ldots n]$, $\mathtt{l}_{xi}$ = newloc(S_{n+1})

E' = [x_1 : $\mathtt{l}_{x1}$, … , x_n : $\mathtt{l}_{xn}$, a_1 : $\mathtt{l}_1$, … , a_m : $\mathtt{l}_m$]

new environment, with the formals and the attributes of the receiver in-scope

S_{n+2} = S_{n+1}[v_1/$\mathtt{l}_{x1}$, … , v_n/$\mathtt{l}_{xn}$]

v_0, E', S_{n+2} ⊢ e_{body} : v, S_{n+3}

---

so, E, S ⊢ e_0.f(e_1, …, e_n) : v, S_{n+3}

Operational Semantics for Method Dispatch

$$\frac{\begin{array}{l} so, E, S \vdash e_1 : v_1, S_1 \qquad so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots \qquad so, E, S_{n-1} \vdash e_n : v_n, S_n \\ so, E, S_n \vdash e_0 : v_0, S_{n+1} \\ v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\ imp(X, f) = (x_1, \ldots, x_n, e_{body}) \\ \forall\, i \in [1 \ldots n],\ l_{xi} = \mathtt{newloc}(S_{n+1}) \\ E' = [x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1, \ldots, a_m : l_m] \\ S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] \\ v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3} \end{array}}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}}$$

new store with formals pointing to the actual arguments

Operational Semantics for Method Dispatch

$$\frac{\begin{array}{ll} so, E, S \vdash e_1 : v_1, S_1 & so, E, S_1 \vdash e_2 : v_2, S_2 \\ \ldots & so, E, S_{n-1} \vdash e_n : v_n, S_n \\ so, E, S_n \vdash e_0 : v_0, S_{n+1} & \\ v_0 = X(a_1 = l_1, \ldots, a_m = l_m) & \\ imp(X, f) = (x_1, \ldots, x_n, e_{body}) & \\ \forall\, i \in [1 \ldots n],\ l_{xi} = newloc(S_{n+1}) & \\ E' = [x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1, \ldots, a_m : l_m] & \\ S_{n+2} = S_{n+1}[v_1/l_{x1}, \ldots, v_n/l_{xn}] & \\ v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3} & \end{array}}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) : v, S_{n+3}}$$

finally, evaluate the body

Notes on OpSem for Dispatch

- The body of the method is invoked with:

Notes on OpSem for Dispatch

- The body of the method is invoked with:
    - **E'** mapping formal arguments and self's attributes

Notes on OpSem for Dispatch

- The body of the method is invoked with:
    - **E'** mapping formal arguments and self's attributes
    - $\mathbf{S}_{\mathbf{n+2}}$ like the caller's except with actual arguments bound to the locations allocated for formals

Notes on OpSem for Dispatch

- The body of the method is invoked with:
  - **E'** mapping formal arguments and self's attributes
  - $\mathbf{S}_{\mathbf{n+2}}$ like the caller's except with actual arguments bound to the locations allocated for formals
- The notion of the **activation frame** is implicit
  - New locations are allocated for actual arguments

Notes on OpSem for Dispatch

- The body of the method is invoked with:
  - **E'** mapping formal arguments and self's attributes
  - $\mathbf{S}_{\mathbf{n+2}}$ like the caller's except with actual arguments bound to the locations allocated for formals
- The notion of the **activation frame** is implicit
  - New locations are allocated for actual arguments
- The semantics of **static dispatch** is similar except the implementation of `f` is taken from the specified class

Run-time Errors

Run-time Errors

- The operational semantics **do not** cover all possible cases!

Run-time Errors

- The operational semantics **do not** cover all possible cases!
- Consider for example this bit from the dispatch rule:

$$\ldots$$

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

$$v_0 = X(a_1 = \mathtt{l}_1, \ldots, a_m = \mathtt{l}_m)$$

$$imp(X, \mathtt{f}) = (x_1, \ldots, x_n, e_{body})$$

$$\ldots$$

- What happens if imp(X, $\mathtt{f}$) is not defined?

Run-time Errors

- The operational semantics **do not** cover all possible cases!
- Consider for example this bit from the dispatch rule:

$$\ldots$$

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

$$v_0 = X(a_1 = \mathtt{l}_1, \ldots, a_m = \mathtt{l}_m)$$

$$imp(X, \mathtt{f}) = (x_1, \ldots, x_n, e_{body})$$

$$\ldots$$

- What happens if $imp(X, \mathtt{f})$ is not defined?
  - It **cannot be**! Type safety theorem guarantees it :)

Run-time Errors

- There are some run-time errors that the typechecker **does not try** to prevent (but it could - we'll get to it in a few minutes)

Run-time Errors

- There are some run-time errors that the typechecker **does not try** to prevent (but it could - we'll get to it in a few minutes)
  - dispatching on void
  - division by zero
  - substring out of range
  - heap overflow

Run-time Errors

- There are some run-time errors that the typechecker **does not try** to prevent (but it could - we'll get to it in a few minutes)
  - dispatching on void
  - division by zero
  - substring out of range
  - heap overflow
- In such cases the execution must **abort gracefully**

Run-time Errors

- There are some run-time errors that the typechecker **does not try** to prevent (but it could - we'll get to it in a few minutes)
  - dispatching on void
  - division by zero
  - substring out of range
  - heap overflow
- In such cases the execution must **abort gracefully**
  - i.e., with an error message rather than a segfault

Run-time Errors

- There are some run-time errors that the typechecker **does not try** to prevent (but it could - we'll get to it in a few minutes)
  - dispatching on void
  - division by zero
  - substring out of range
  - heap overflow
- In such cases the execution must **abort gracefully**
  - i.e., with an error message rather than a segfault
  - implication: you must generate code in PA3 that **checks for run-time errors**!

Summary of Operational Semantics

Summary of Operational Semantics

- Operational rules are **very precise**
  - Nothing is left unspecified

Summary of Operational Semantics

- Operational rules are **very precise**
  - Nothing is left unspecified
- Operational rules contain a lot of details
  - Read them **carefully**!

Summary of Operational Semantics

- Operational rules are **very precise**
  - Nothing is left unspecified
- Operational rules contain a lot of details
  - Read them **carefully**!
- Most languages **do not** have a well-specified operational semantics :(

Summary of Operational Semantics

- Operational rules are **very precise**
  - Nothing is left unspecified
- Operational rules contain a lot of details
  - Read them **carefully**!
- Most languages **do not** have a well-specified operational semantics :(
- When **portability** is important, an operational semantics is essential
  - But typically not using the exact notation we used for Cool

Agenda

- Review: basics of operational semantics
- Operational semantics of Cool
- **(if time): introduction to static analysis**
    - further if time: get into abstract interpretation

Motivation: many defects are hard to test for

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)
- We want to learn about "**all possible runs**" of the program for particular properties

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)
- We want to learn about "**all possible runs**" of the program for particular properties
  - Without actually running the program!

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it’s hard to find them via testing
- Executing or dynamically analyzing all paths concretely to find such defects is **not feasible** (cf. exhaustive testing is infeasible)
- We want to learn about “**all possible runs**” of the program for particular properties
  - Without actually running the program!
  - Bonus: we don't need test cases!

Motivation: many defects are hard to test for

- Many interesting defects are on **uncommon** or **difficult-to-exercise** execution paths
  - So it's hard to find them via testing
- Executing or dynamically analyz[...] such defects is **not feasible** (cf.[...]
- We want to learn about "**all pos**[...] particular properties
  - Without actually running t[...]
  - Bonus: we don't need test ca[...]

This is especially true for certain kinds of hard-to-test-for defects that might not be apparent even if you do exercise them, such as **resource leaks**

What does static analysis do well?

What does static analysis do well?

- Defects that result from inconsistently following **simple**, mechanical design **rules**

What does static analysis do well?

- Defects that result from inconsistently following **simple**, mechanical design **rules**
  - Security: buffer overruns, input validation
  - Memory safety: null pointers, initialized data
  - Resource leaks: memory, OS resources
  - API Protocols: device drivers, GUI frameworks
  - Exceptions: arithmetic, library, user-defined
  - Encapsulation: internal data, private functions
  - Data races: two threads, one variable

What does static analysis do well?

- Defects that result from inconsiste[nt] mechanical design **rules**
  - Security: buffer overruns, input
  - Memory safety: null pointers, in
  - Resource leaks: memory, OS resources
  - API Protocols: device drivers, GUI frameworks
  - Exceptions: arithmetic, library, user-defined
  - Encapsulation: internal data, private functions
  - Data races: two threads, one variable

There are **rules** for doing each of these things **correctly**, and a static analysis can automate those rules.

What is a static analysis?

What is a static analysis?

Definition: ***static analysis*** is the systematic examination of an abstraction of program state space

What is a static analysis?

Definition: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program

What is a static analysis?

Definition: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program
    - in contrast to a **dynamic analysis**, such as testing, which does execute the program

What is a static analysis?

Definition: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program
  - in contrast to a **dynamic analysis**, such as testing, which does execute the program
- an **abstraction**, in this context, is a **selective representation** of the program that is simpler to analyze

What is a static analysis?

Definition: ***static analysis*** is the systematic examination of an abstraction of program state space

- static analysis **does not execute** the program
    - in contrast to a **dynamic analysis**, such as testing, which does execute the program
- an **abstraction**, in this context, is a **selective representation** of the program that is simpler to analyze
    - **key idea**: the abstraction will have fewer states to explore
        - hopefully, many fewer!

What is a static analysis?

Definition: *static analysis* is the
abstraction of program state s

We have already encountered one kind of static analysis in this class: **type systems**. Type systems aren't special - they are just a very common static analysis.

- static analysis does not exe
  - in contrast to a dynamic analysis, such as testing, which does execute the program
- an **abstraction**, in this context, is a selective representation of the program that is simpler to analyze
  - **key idea**: the abstraction will have fewer states to explore
    - hopefully, many fewer!

Alternative: Dynamic Analysis

Alternative: Dynamic Analysis

- Execute program (over some inputs)
  - The compiler provides the semantics

Alternative: Dynamic Analysis

- Execute program (over some inputs)
  - The compiler provides the semantics
- Observe executions
  - Requires instrumentation infrastructure

Alternative: Dynamic Analysis

- Execute program (over some inputs)
  - The compiler provides the semantics
- Observe executions
  - Requires instrumentation infrastructure
- Analyze results

Alternative: Dynamic Analysis

- Execute program (over some inputs)
  - The **compiler provides the semantics**
- Observe executions
  - Requires instrumentation
- Analyze results

This means that we don't need an **external model** of what the computer does!
(Since your compiler faithfully implements the OpSem, right?)

Dynamic Analysis Properties

Dynamic Analysis Properties

- Can be as **fast** as execution (over a test suite, and allowing for data collection)
  - Example: aliasing

Dynamic Analysis Properties

- Can be as **fast** as execution (over a test suite, and allowing for data collection)
    - Example: aliasing
- **Precise**: no abstraction or approximation

Dynamic Analysis Properties

- Can be as **fast** as execution (over a test suite, and allowing for data collection)
  - Example: aliasing
- **Precise**: no abstraction or approximation
- **Unsound**: results may not generalize to future executions
  - Describes execution environment or test suite

Static Analysis Properties

Static Analysis Properties

- **Slow** to analyze large models of state, so use abstraction

Static Analysis Properties

- **Slow** to analyze large models of state, so use abstraction
- **Conservative**: account for abstracted-away state

Static Analysis Properties

- **Slow** to analyze large models of state, so use abstraction
- **Conservative**: account for abstracted-away state
- **Sound**: (weak) properties are guaranteed to be true
    - Some static analyses are not sound, but static analyses ***can*** be made sound

Static vs Dynamic Analyses

Dynamic analyses:

Static analyses:

Static vs Dynamic Analyses

Dynamic analyses:

- Concrete execution
    - slow if exhaustive

Static analyses:

- Abstract domain
    - slow if precise

Static vs Dynamic Analyses

Dynamic analyses:

- Concrete execution
  - slow if exhaustive
- Precise
  - no approximation

Static analyses:

- Abstract domain
  - slow if precise
- Conservative
  - due to abstraction

Static vs Dynamic Analyses

Dynamic analyses:

- Concrete execution
  - slow if exhaustive
- Precise
  - no approximation
- Unsound
  - does not generalize

Static analyses:

- Abstract domain
  - slow if precise
- Conservative
  - due to abstraction
- **Sound**
  - due to conservatism

Analogous Analyses

- **Any** analysis problem can be solved with **either** a static or a dynamic analysis

Analogous Analyses

- **Any** analysis problem can be solved with **either** a static or a dynamic analysis
  - e.g., consider **type safety**: no memory corruption or operations on wrong types of values
    - Static type-checking (e.g., Java, Cool)
    - Dynamic type-checking (e.g., Python)

Analogous Analyses

- **Any** analysis problem can be solved with **either** a static or a dynamic analysis
  - e.g., consider **type safety**: no memory corruption or operations on wrong types of values
    - Static type-checking (e.g., Java, Cool)
    - Dynamic type-checking (e.g., Python)
- This insight gives us a kind of "***PL incompleteness theorem***": either you can know something precisely about one execution (via dynamic analysis) or imprecisely about every execution (via static analysis)

Static vs Dynamic Analyses

Dynamic analyses:

- Concrete execution
  - slow if exhaustive
- Precise
  - no approximation
- Unsound
  - does not generalize

Static analyses:

- Abstract domain
  - slow if precise
- Conservative
  - due to abstraction
- Sound
  - due to conservatism

Sound Dynamic Analysis?

Sound Dynamic Analysis?

- Observe **every possible execution**!

Sound Dynamic Analysis?

- Observe **every possible execution**!
- Problem: **infinite** number of executions

Sound Dynamic Analysis?

- Observe **every possible execution**!
- Problem: **infinite** number of executions
- Solution: test case selection and generation
    - **Efficiency tweaks** to an algorithm that works perfectly in theory but exhausts resources in practice

Precise Static Analysis?

Precise Static Analysis?

- Reason over **full program state**!

Precise Static Analysis?

- Reason over **full program state**!
- Problem: **infinite** number of executions

Precise Static Analysis?

- Reason over **full program state**!
- Problem: **infinite** number of executions
- Solution: data or execution abstraction
  - **Efficiency tweaks** to an algorithm that works perfectly in theory but exhausts resources in practice

Different Subsets

- Dynamic analysis focuses on a **subset of executions**

Different Subsets

- Dynamic analysis focuses on a **subset of executions**
    - i.e., the executions in the test suite, the executions that random input produces, etc.

Different Subsets

- Dynamic analysis focuses on a **subset of executions**
  - i.e., the executions in the test suite, the executions that random input produces, etc.
  - typically **optimistic** about other executions
    - i.e., assume that they will be bug-free

Different Subsets

- Dynamic analysis focuses on a **subset of executions**
  - i.e., the executions in the test suite, the executions that random input produces, etc.
  - typically **optimistic** about other executions
    - i.e., assume that they will be bug-free
- Static analysis focuses on a **subset of data structures**

Different Subsets

- Dynamic analysis focuses on a **subset of executions**
  - i.e., the executions in the test suite, the executions that random input produces, etc.
  - typically **optimistic** about other executions
    - i.e., assume that they will be bug-free
- Static analysis focuses on a **subset of data structures**
  - more precise for data or control described by the abstraction

Different Subsets

- Dynamic analysis focuses on a **subset of executions**
  - i.e., the executions in the test suite, the executions that random input produces, etc.
  - typically **optimistic** about other executions
    - i.e., assume that they will be bug-free
- Static analysis focuses on a **subset of data structures**
  - more precise for data or control described by the abstraction
  - typically **conservative** / **pessimistic** elsewhere
    - i.e., assume that unmodeled state is unsafe

Next Time On...

- Proper introduction to **one formalism** for static analysis
    - ***abstract interpretation*** (which I will call "AI" constantly to upset you and Sam Altman)

Next Time On…

- Proper introduction to **one formalism** for static analysis
    - ***abstract interpretation*** (which I will call "AI" constantly to upset you and Sam Altman)
- Many other formalisms exists

Next Time On...

- Proper introduction to **one formalism** for static analysis
  - ***abstract interpretation*** (which I will call "AI" constantly to upset you and Sam Altman)
- Many other formalisms exists
  - including **type systems** (which we've already discussed)

Next Time On...

- Proper introduction to **one formalism** for static analysis
    - ***abstract interpretation*** (which I will call "AI" constantly to upset you and Sam Altman)
- Many other formalisms exists
    - including **type systems** (which we've already discussed)
    - formally, abstract interpretation is expressive enough that you can describe *any* static analysis using it

Next Time On...

- Proper introduction to **one formalism** for static analysis
  - ***abstract interpretation*** (which I will call "AI" constantly to upset you and Sam Altman)
- Many other formalisms exists
  - including **type systems** (which we've already discussed)
  - formally, abstract interpretation is expressive enough that you can describe *any* static analysis using it
    - that said, you probably don't want to
    - ask me more about Patrick Cousot's work in OH

Course Announcements

- **PA2c2** due today
  - if you haven't started yet, you almost certainly won't finish in time (come talk to me about it)
- I'll hold **two short OH today** for those who want to see a test case before PA2c2:
  - right after class (11:25-11:55am)
  - 4:30-5pm
- **PA2 (full)** is due next Monday (one week from today!)

Agenda

- Review: basics of operational semantics
- Operational semantics of Cool
- (if time): introduction to static analysis
    - **further if time: get into abstract interpretation**