Lecture 12 — Objects

- Interpreting programs with objects requires a state this is more complicated — something that mimics the combination of stack and heap — to implement *side effects* correctly. In MP7, you will modify your MJ interpreter to handle objects. Today we will discuss the new state structure and write some new SOS rules.
  - Two-level state
  - SOS rules
  - Implementing inheritance

Exercise: Java programs

```
class C { int i;
          int seti (int j) { i=j; return i; }
          int geti () { return i; } }

class D { int f() { x = new C();
                    x.seti(10);
                    return x.geti(); // ?
        }
```

```
class E { int f() { x = new C();
                    y = x;
                    x.seti(10);
                    return y.geti(); // ?
        }
```

Exercise: Java programs (cont.)

```
class C { int i;
          int seti (int j) { i=j; return i; }
          int geti () { return i; } }

class D { int i; C c;
          int seti (int j) { i=j; return i; }
          C setc (C e) { c=e; return c; }
          int geti () { return i; }
          int getc () { return c; } }

class E { int f() { x = new D(); y = new C(); x.setc(y);
                    z = new D(); z.setc(y);
                    x.getc().seti(10);
                    return z.getc().geti(); // ?
        }
```

(Syntactic note: can write these in MiniJava; just make sure all expressions appear in assignment statements.)

Basics of object-oriented programming in Java

- An *object* is a heterogeneous collection of values, together with associated functions.

- The functions associated with an object depend solely on the class of the object. An object created by calling `new C()` contains the values given in `C`'s non-static fields, and the functions defined as methods in `C` (*ignoring inheritance*).

- Methods are called "on" an object, called the "receiver" of the method call: $e.f(e_1, \ldots, e_n)$ — the value of e must be an object, and is the receiver of this call. The method called is the definition of f found in the class of the receiver.

- When executing a method, the receiver can be referred to by the name "`this`". A field `x` of the receiver can be referred as `this.x` or `x`.

Side effects

- *Side effect* = change in state resulting from a method call.
- With side effects, can evaluate the same expression twice and get different results:
  - Is this always true in Java?

    "`y = f(); y = f();`" $\equiv$ "`y = f();`"

- Side effects on the receiver of a method call is a common and essential part of the o-o programming style.
- (Another source of side effects is static variables; MiniJava doesn't have these.)

MP5 MJ has no side effects

THEOREM: **In MP6 version of MiniJava: If $x \notin e$, `x=e;x=e` $\equiv$ `x=e`. That is, for any states σ, σ' and program π,**

$$\texttt{x=e; x=e}, \sigma, \pi \Rightarrow \sigma' \quad \textbf{iff} \quad \texttt{x=e}, \sigma, \pi \Rightarrow \sigma'$$

LEMMA: **Let $Y = \{y_1, \ldots, y_m\}$ be all variables in e, and σ and σ' two states that agree on Y ($\forall y \in Y$, $\sigma(y) = \sigma'(y)$). Then, for any v and π: $e, \sigma, \pi \Downarrow v$ iff $e, \sigma', \pi \Downarrow v$.**

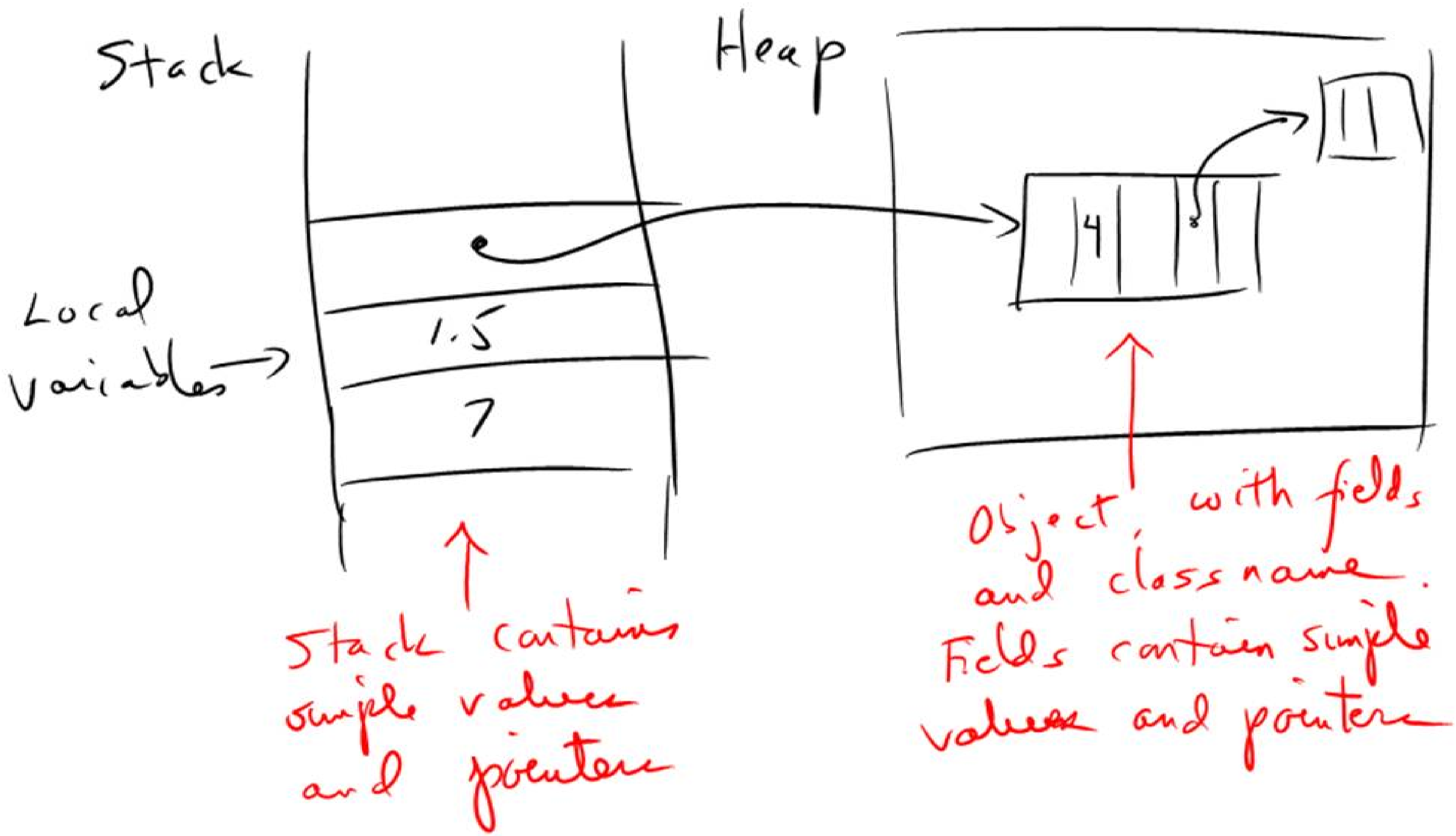
PROOF: **By induction on the structure of the SOS proof of $e, \sigma, \pi \Downarrow v$.**

PROOF OF THEOREM: **If e, σ, $\pi \Downarrow v$, then `x=e`, σ, $\pi \Rightarrow \sigma[v/x]$. The lemma tells us that e, $\sigma[v/x]$, $\pi \Downarrow v$. Therefore, `x = e; x = e`, σ, $\pi \Rightarrow \sigma[v/x]$.**

Stack and heap in Java

- **If Java had no objects and no static variables, it would have no side effects.**
- **Primitive values are placed in stack. When passed to method as argument, value is *copied* to top of stack; original variable (if any) that had that value is not altered by method call.**
- **Objects are placed in the heap; stack frame just contains a pointer. When passed to method as argument, or as the receiver, the *pointer* is copied to top of stack.**
- **Every reference to a field of an object goes through the pointer to get to the object in the heap. The side effect happens because the called method has a pointer to the same memory as the caller has.**

Stack and heap in Java



Two-level state in MJ

- **Thus, variables of object type go through a two-stage lookup: get pointer from stack, then get object from heap. Need to use same idea in interpreter for MJ.**

- **Terminology: Instead of "stack" and "heap," we will say "environment" and "store." "State" is a pair of an environment and a store (called "two-level state").**

- **In MP7, will use two-level state. As in Java:**
  - **Environment contains simple value and pointers.**
  - **Store contains objects.**
  - **Objects contain simple values and pointers.**
  - ***(We will not implement arrays.)***

SOS rules

```
type varname = string
type classname = string
type stackvalue = IntV of int | StringV of string
                | BoolV of bool | NullV | Location of location
and location = int
type environment = (varname * stackvalue) list
type heapvalue = Object of classname * environment
type store = heapvalue list
type state = environment * store
```

- **In SOS rules, we still use σ for a state, but often write (ρ, η) instead, with ρ for environment and η for store.**
- **Consider evaluation of `new C`. Involves putting a new object in the store, so we must change the judgments for evaluation:**

$$e, \sigma, \pi \Downarrow v, \eta$$

or, equivalently, $e, (\rho, \eta), \pi \Downarrow v, \eta$**.**

SOS rules (cont.)

```
type stackvalue = IntV of int | StringV of string
                | BoolV of bool | NullV | Location of location
and location = int
type environment = (varname * stackvalue) list
type heapvalue = Object of classname * environment
type store = heapvalue list
type state = environment * store
```

- **Write an expression of type `state` for a state that contains variables `x` bound to 3 and `y` bound to an object of class `C`; `C` contains fields `a` and `b`, and in `y` these have integer values 4 and 5.**

SOS rules (cont.)

- **Give the SOS rule for `new`:**

  $(\textsc{New})\quad \texttt{new C(), } (\rho, \eta)\texttt{, } \pi \Downarrow$

- **New form of SOS rules reflected in new type of `eval`:**

```
let rec eval (e:exp) ((env,sto) as sigma:state) (prog:program)
       : stackvalue * store =
```

- **and the corresponding clause in `eval` (you can assume any auxiliary functions you think useful):**

```
    | NewId c ->
```

Threading the store

- **The major new thing is that evaluation of expressions can have side effects, in particular, they can cause the store to change. (Evaluation of an expression cannot cause the *environment* to change.)**

- **Therefore, need to take the store from any expression evaluation and pass it along to the next expression evaluation. Can never discard any changes that occur in the store.**

$$(\textsc{Binop-Less})\qquad \frac{e_1,\ (\rho,\ \eta),\ \pi \Downarrow \texttt{IntV}\ (i_1),\ \eta' \qquad e_2,\ (\rho,\ \eta'),\ \pi \Downarrow \texttt{IntV}\ (i_2),\ \eta''}{e_1 \texttt{ < } e_2,\ (\rho,\ \eta),\ \pi \Downarrow \texttt{BoolV}\ (i_1 < i_2),\ \eta''}$$

SOS rules (v. 2) (cont.)

(NOT) $!e$, (ρ, η), $\pi \Downarrow$

(INT-MULT) e_1 * e_2, (ρ, η), $\pi \Downarrow$

SOS rules (v. 2) (cont.)

(VAR) $x, (\rho, \eta), \pi \Downarrow$

(FIELD) $x, (\rho, \eta), \pi \Downarrow$

(METHOD-CALL)

$e_0 \,.\, f\,(e_1, \ldots, e_n)\,,\ (\rho, \eta),\ \sigma,\ \pi \Downarrow v$

SOS rules (v. 2) (cont.)

- **Rules for statements actually don't change — they always passed the state along from one to the next — except for assignment.**

(VarAsgn) x**=**e**,** **(**ρ**,** η**),** $\pi \Rightarrow$

(FieldAsgn) x**=**e**,** **(**ρ**,** η**),** $\pi \Rightarrow$

Inheritance in Java

```
// EXAMPLE 1
class B {                                    class C extends B {
     string f() { return this.g(); }              string g() { return "C"; } }
     string g() { return "B"; } }

x = new B(); y = new C();
x.f(); // ?
y.f(); // ?

// EXAMPLE 2
class B { B aB;                                   class C extends B {
     void r() { aB = this; }                           string g() { return "C"; } }
     string s() { return aB.g(); }
     string g() { return "B"; } }

x = new B(); y = new C(); x.r();  y.r();
x.s(); // ?
y.s(); // ?
```

Inheritance in Java (cont.)

```
// EXAMPLE 3
class B {
     B aB;
     void q(B x) { aB = x; }
     string s() { return aB.g(); }
     string g() { return "B"; } }
```

```
class C extends B {
     string g() { return "C"; } }
```

```
x = new B(); y = new C();
x.q(x); x.s(); // ?
x.q(y); x.s(); // ?
y.q(y); y.s(); // ?
y.q(x); y.s(); // ?
```

Inheritance in Java (cont.)

```
// EXAMPLE 4
class B {
     string f() { return this.g(); }  }
     string g() { return "B"; } }
```

```
class C extends B {
     B b;
     string g() { return "C"; }
     string f() { return b.g(); }
     void h(B y) { b = y; } }
```

```
x = new B(); y = new C();
y.h(y); y.f(); // ?
y.h(x); y.f(); // ?
```

Principles of inheritance in Java and MJ

- **Inheriting fields and methods:**
  - **Fields of all superclasses are included in the object.**
  - **The methods associated with an object include those of its class and all superclasses; if there is more than one method with the same name, the "closest" one is called.**
- **Changes in SOS rules (and in functions in MP 6):**
  - **To evaluate `new C()`, create an object consisting of all fields of `C` and inherited fields; class of the new object is still `C`.**
  - **To call $e_0.f(e_1, \ldots, e_n)$, evaluate e_0 and find its class; look for f in that class, or, if not found, in its superclass, and so on, until a definition of f is found.**

Wrap-up

- **Today we discussed:**
  - **implementing objects using two-level store**
- **We discussed it because:**
  - **Two-level store is necessary to implement objects correctly (i.e. with side effects).**
- **What to do now:**
  - **In MP7, you will modify your MP6 interpreter to use the two-level store, then add object-oriented features like object creation and fields.**