Designing and Implementing CLASSES

CS164

Lecture 16-18

CLASSES

- Extending REFS to CLASSES
  - no more lambdas
- Designed to
  - Give a taste of implementation of modern features
    - Abstraction
    - Static typing with subtypes
    - Reuse (inheritance)
    - Memory management
    - And more …
- But many things are left out

A Simple Example

```
class Point extends object {
        x : int;
        y : int;
}
```

- CLASSES programs are sets of class definitions
  - A special class **Main** with a special method **main**
  - No separate notion of lambdas
- class = a collection of attributes and methods
- Instances of a class are objects

CLASSES Objects

```
class Point extends object {
      x : int;
      y : int;
}
```

- The expression "new Point" creates a new object of class Point
- An object can be thought of as a record with a slot for each attribute

object as Record

x	y
0	0

Methods

- A class can also define methods for manipulating the attributes

```
class Point extends object {
     x : int;
     y : int;
     movePoint(newx : int, newy : int): Point =
      { x = newx;
        y = newy;
        self
      };
}
```

- Methods can refer to the current object using *self*

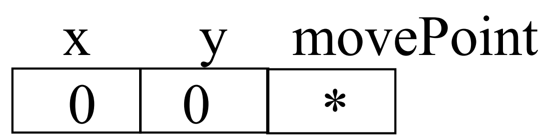
Information Hiding in CLASSES

- Methods are global
- Attributes are local to a class
  - They can only be accessed by the class' s methods
  - self.attr is not allowed
- Example:

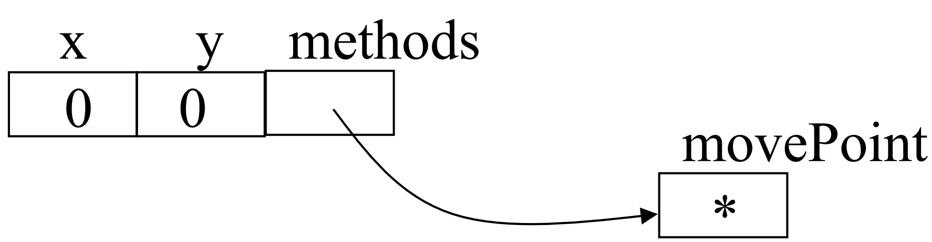
```
class Point extends object {
      x: int;
      x () : int = x;
      setx (newx : int) : int = x = newx;
}
```

Methods

- Each object knows how to access the code of a method
- As if the object contains a slot pointing to the code



- In reality implementations save space by sharing these pointers among instances of the same class



Inheritance

- We can extend points to colored points using subclassing => class hierarchy

```
class ColorPoint extends Point {
   color : int;
   movePoint(newx : int, newy : int): Point = {
     color = 0;
     x = newx;
     y = newy;
     self
   };
}
```

Inheritance

- We can extend points to colored points using subclassing => class hierarchy

```
class ColorPoint extends Point {
   color : int;
   movePoint(newx : int, newy : int): Point = {
      color = 0;
      x = newx;
      y = newy;
      self
   };
}
```

x	y	color	movePoint
0	0	0	*

Redefinition of attributes and methods

- attributes cannot be redefined in a subclass
- methods can be redefined as long as the signature remains the same

CLASSES Types

- Every class is a type
- Base classes:
  - int for integers
  - boolean for boolean values
  - string for strings
  - object root of the class hierarchy
  - int, boolean, string are subtypes of object
- All variables and attributes must be declared (with their type)
  - compiler infers types for expressions

Type tree and type conformance

CLASSES Type Checking

```
x : P;
x = new C;
```

- Is well typed if P is an ancestor of C in the class hierarchy
  - Anywhere an P is expected a C can be used

- Type safety:
  - A well-typed program cannot result in runtime type errors

Method Invocation and Inheritance

- Methods are invoked by dispatch
- Understanding dispatch in the presence of inheritance is a subtle aspect of OO languages

```
p : Point;
p = new ColorPoint;
p.movePoint(1,2);
p[Point].movePoint(1,2);
```

  - p has static type Point
  - p has dynamic type ColorPoint
  - p.movePoint must invoke the ColorPoint version
  - to invoke Point version call p[Point].movePoint(1,2);

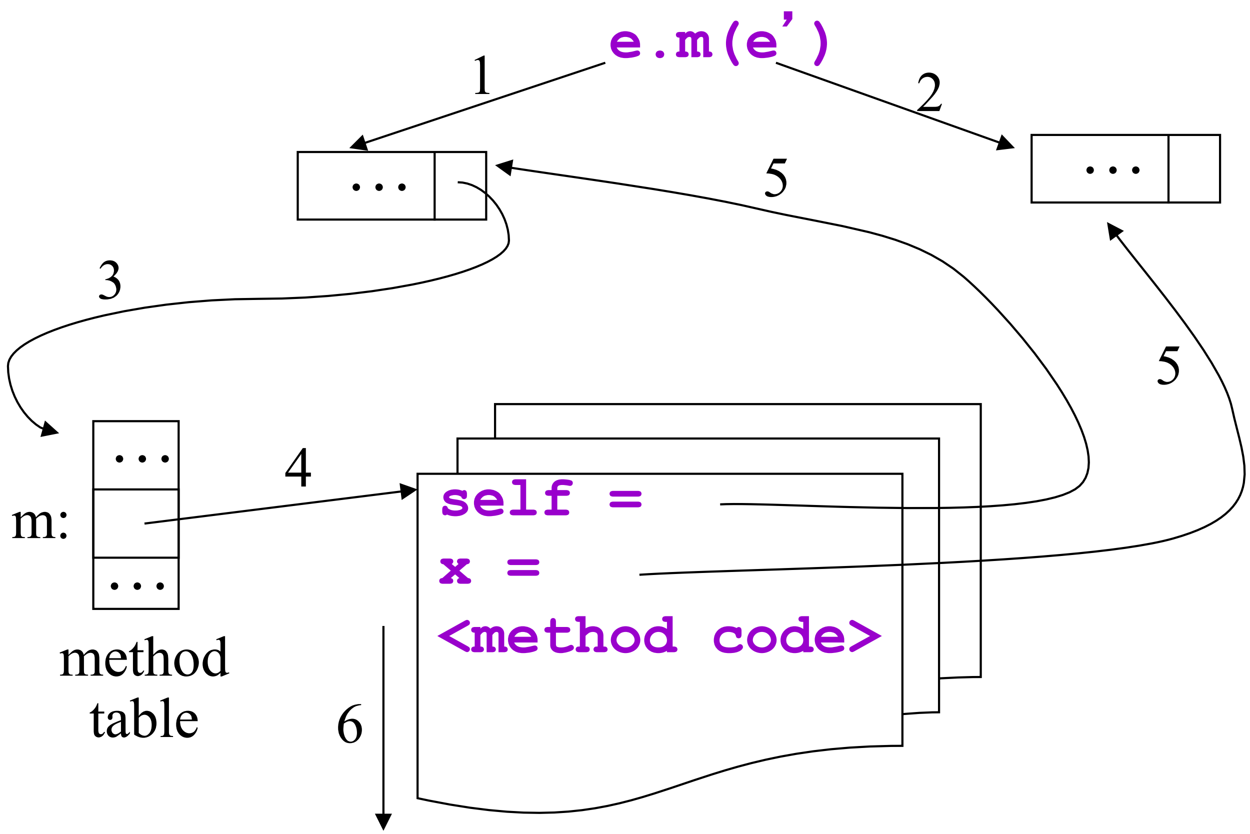
Method Invocation

- Example: invoke two-argument method `m`

```
e.m(e1,e2)
```

Method Invocation

- Example: invoke one-argument method m



1. Eval e
2. Eval. e'
3. Find class of e
4. Find code of m
5. Bind self and x
6. Run method

Default Attribute Initialization

- int: initialized to 0
- boolean: initialized to false
- string: initialized to ""
- T: initialized to null where T is any type other than int, boolean, and string

null

- no explicit keyword
- no “null” type
- `isnull expr` checks if `expr` is null
- An attribute of type other than `int, boolean, and string` is initialized to null
- `while e1 do e2` returns `null` and the return type is `object`

Other Expressions

- Expression language (every expression has a type and a value) like in REFS
  - Type cast statement   if E is x : T then E1 else E2
  - Method call   E.m(E1) or E[T].m(E1)
  - Let…in…   let x : T = E1 in E2
  - Conditionals   if E then E1 else E2
  - Loops: while E do E1
  - Arithmetic, logical operations as in REFS
  - Assignment   x = E
  - Sequence   {E1; … ;En}
  - No fun (lambdas) and no null literal
- Missing features:
  - Arrays, Floating point operations, Interfaces, Exceptions,…

CLASSES Memory Management

- Memory is allocated every time new is invoked

- Memory is deallocated automatically when an object is not reachable anymore
  - Done by the garbage collector (GC)
  - There is a CLASSES GC

Formal Grammar of CLASSES

```
prog: cls*;

cls: 'class' type 'extends' type '{' field* method*
'}';
field: iden ':' type ';';
method: iden '(' paramlist ')' ':' type '=' expr ';';

expr: 'new' type
    | expr '.' iden '(' arglist ')'
    | expr '[' type ']' '.' iden '(' arglist ')'
    | expr ('*' | '/') expr
    | expr ('+' | '–') expr
    | expr ('=='|'!='|'>'|'<'| '>=' | '<=') expr
    | iden '=' expr
    | 'let' iden ':' type '=' expr 'in' expr
    | 'if' expr 'then' expr 'else' expr
    | 'if' expr 'is' iden ':' type 'then' expr 'else'
expr
    | 'while' expr 'do' expr
    | 'print' expr
    | 'isnull' expr
    | 'self'
    | num
    | iden
    | string
    | '{' exprseq '}'
    | '(' expr ')'
    ;
paramlist: param (',' param)*
    | ;
param: iden ':' type;
arglist: expr (',' expr)*
    | ;
exprseq: expr (';' expr)*;

type: ID ;
num: INT;
iden: ID;
string: STRING;

ID  :  ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..
'9'|'_')* ;
INT :  '0'..'9'+ ;
WS  :  (' ' '\t' | ' \r' | ' \n')+ -> skip ;
STRING : '"' (~'"')* '";
```

What Does Semantic Analysis Do?

- Checks for static errors:
  1. All identifiers are declared
  2. Types
  3. A class must be defined before it is extended
  4. Classes defined only once
  5. Methods and attributes in a class defined only once
  6. Attributes from super classes cannot be redefined
  7. Methods from super classes can be redefined provided that signature remains same
  8. Reserved identifiers are not misused
  9. Assignment to self must be disallowed
  10. No declared identifier name should be a keyword
  11. Two parameters cannot have the same name in a method
  12. One must not extend T or call new T where T is string, int, or boolean
- The requirements depend on the language

Example: Use Before Definition

```
class Foo extends object {
   . . . let y: Bar in . . .
};

class Bar extends object {
   . . .
};
```

More Scope (Cont.)

- Method and attribute names have complex rules
- A method need not be defined in the class in which it is used, but in some parent class
- Methods may also be redefined (overridden)
- A method parameter can shadow an attribute
  - Types of parameter and attribute could be different
- A attribute, say attr, cannot be accessed using self.attr

class Definitions

- class names can be used before being defined except in extends
- We can't check this property
  - using an Environment
  - or even in one pass
- Solution
  - Pass 1: Gather all class names and their attributes'/methods' types
  - Pass 2: Do the checking
- Semantic analysis requires multiple passes
  - Probably more than two

Type Checking: Notation for Inference Rules

- By tradition inference rules are written

$$\frac{\vdash \text{Hypothesis}_1 \quad \ldots \quad \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$

- CLASSES type rules have hypotheses and conclusions of the form:

$$\vdash e : T$$

- $\vdash$ means "we can prove that . . ."

Two Rules

$$\frac{}{\vdash i : int} \text{ [int]} \quad \text{(i is an integer constant)}$$

Two Rules

$$\frac{}{\vdash i : int} \text{[int]} \quad \text{(i is an integer constant)}$$

$$\frac{\begin{array}{c}\vdash e_1 : int \\ \vdash e_2 : int\end{array}}{\vdash e_1 + e_2 : int} \text{[Add]}$$

Two Rules (Cont.)

- These rules give templates describing how to type integers and + expressions

- By filling in the templates, we can produce complete typings for expressions

- Example: 1+2

Example: 1 + 2

$$\dfrac{\dfrac{}{\vdash 1 : \text{int}} \qquad \dfrac{}{\vdash 2 : \text{int}}}{\vdash 1 + 2 : \text{int}}$$

Soundness

- A type system is <u>sound</u> if
  - Whenever $\vdash e : T$
  - Then e evaluates to a value of type T

- We only want sound rules
  - But some sound rules are better than others:

$$\frac{}{\vdash i : \text{object}} \quad \text{(i is an integer constant)}$$

Type Checking Proofs

- Type checking proves facts $e : T$
  - One type rule is used for each kind of expression

- In the type rule used for a node e:
  - The hypotheses are the proofs of types of e's subexpressions
  - The conclusion is the proof of type of e

Rules for Constants

$$\frac{}{\vdash s : \text{string}} \text{[string]}$$

(s is a string constant)

Rule for New

new T produces an object of type T

$$\frac{}{\vdash \text{new } T : T}$$ [New], where T is not int, string, or boolean

Two More Rules

$$\frac{\vdash e_1 : \text{int} \qquad \vdash e_2 : \text{int}}{\vdash e_1 > e_2 : \text{boolean}} \quad \text{[GT]}$$

$$\frac{\vdash e_1 : \text{boolean} \qquad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ do } e_2 : \text{object}} \quad \text{[Loop]}$$

A Problem

- What is the type of a variable reference?

$$\frac{}{\vdash x : ?} \quad [\text{Var}] \qquad \text{(x is an identifier)}$$

A Problem

- What is the type of a variable reference?

$$\frac{}{\vdash x : ?} \quad [\text{Var}] \qquad (\text{x is an identifier})$$

- This rules does not have enough information to give a type.
  - We need a hypothesis of the form "*we are in the scope of a declaration of x with type T*")

A Solution: Put more information in the rules!

- A *type environment* gives types for *free* variables
  - A type environment is a mapping from ObjectIdentifiers to Types
  - A variable is free in an expression if:
    - The expression contains an occurrence of the variable that refers to a declaration outside the expression

  - E.g. in the expression "x", the variable "x" is free
  - E.g. in "let x : int in x + y" only "y" is free
  - E.g. in "x + let x : int in x + y" both "x" and "y" are free

Type Environments

Let O be a function from ObjectIdentifiers to Types

The sentence $O \vdash e : T$

is read: Under the assumption that variables in the current scope have the types given by O, it is provable that the expression e has the type T

Modified Rules

The type environment is added to the earlier rules:

$$\frac{}{O \vdash i : \text{int}} \quad [\text{int}] \qquad \text{(i is an integer)}$$

$$\frac{\begin{array}{c} O \vdash e_1 : \text{int} \\ O \vdash e_2 : \text{int} \end{array}}{O \vdash e_1 + e_2 : \text{int}} \quad [\text{Add}]$$

New Rules

And we can write new rules:

$$\frac{(O(x) = T)}{O \vdash x : T} \quad [\text{Var}]$$

Let

Consider let :

$$\frac{O \vdash e_0 : T_0 \qquad [x=T_0]O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 = e_0 \text{ in } e_1 : T_1} \quad \text{[Let]}$$

This rule is weak. Why?

$[x=T_0]O$ means "O modified to map x to T_0 and behaving as O on all other arguments":

$$[x=T_0]O\ (x) = T_0$$

$$[x=T_0]O\ (y) = O(y)$$

Let

- Consider the example:

```
class C extends P { … }
…
let x : P = new C in …
 …
```

- The previous let rule does not allow this code
  - We say that the rule is too weak

Subtyping

- Define a relation $X \leq Y$ on classes to say that:
  - An object of type X could be used when one of type Y is acceptable, or equivalently
  - X conforms with Y
  - In CLASSES this means that X is a subclass of Y
- Define a relation $\leq$ on classes

  $X \leq X$

  $X \leq Y$ if X extends from Y

  $X \leq Z$ if $X \leq Y$ and $Y \leq Z$

Let (Again)

$$\frac{\begin{array}{c} O \vdash e_0 : T \\ T \leq T_0 \\ [x=T_0]O \vdash e_1 : T_1 \end{array}}{O \vdash \text{let } x : T_0 = e_0 \text{ in } e_1 : T_1} \text{[Let]}$$

- Both rules for let are sound
- But more programs type check with the latter

Let with Subtyping. Notes.

- There is a tension between
  - Flexible rules that do not constrain programming
  - Restrictive rules that ensure safety of execution

Expressiveness of Static Type Systems

- A static type system enables a compiler to detect many common programming errors
- The cost is that some correct programs are disallowed
  - Some argue for dynamic type checking instead
  - Others argue for more expressive static type checking

- But more expressive type systems are also more complex

Dynamic And Static Types

- The <u>dynamic type</u> of an object is the class *C* that is used in the "new *C*" expression that creates the object
  - A run-time notion
  - Even languages that are not statically typed have the notion of dynamic type
- The <u>static type</u> of an expression is a notation that captures all possible dynamic types the expression could take
  - A compile-time notion

Dynamic and Static Types. (Cont.)

- In early type systems the set of static types correspond directly with the dynamic types
- Soundness theorem: for all expressions E

  dynamic_type(E) = static_type(E)

  (in **all** executions, E evaluates to values of the type inferred by the compiler)

- This gets more complicated in advanced type systems

Dynamic and Static Types in CLASSES

```
class A { … }
class B extends A {…}
class Main {
  A x = new A;
  …
  x = new B;
  …
}
```

x has static type A

Here, x's value has dynamic type A

Here, x's value has dynamic type B

- A variable of static type A can hold values of static type B, if $B \leq A$

Dynamic and Static Types

Soundness theorem for the CLASSES type system:

$$\forall E. \quad \text{dynamic\_type}(E) \leq \text{static\_type}(E)$$

Dynamic and Static Types

Soundness theorem for the CLASSES type system:

$$\forall E.\quad \text{dynamic\_type}(E) \leq \text{static\_type}(E)$$

Why is this Ok?

- For E, compiler uses static_type(E) (call it *C*)
- All operations that can be used on an object of type *C* can also be used on an object of type $C' \leq C$
  - Such as fetching the value of an attribute
  - Or invoking a method on the object
- Subclasses can <u>only add</u> attributes or methods
- Methods can be redefined but with same type !

Let. Examples.

- Consider the following CLASSES class definitions

```
class A { a() : int = 0 ; }
class B extends A { b() : int = 1; }
```

- An instance of B has methods "a" and "b"
- An instance of A has method "a"
  - A type error occurs if we try to invoke method "b" on an instance of A

Example of Wrong Let Rule (1)

- Now consider a hypothetical let rule:

$$\frac{O \vdash e_0 : T \qquad T \leq T_0 \qquad O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 = e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?

Example of Wrong Let Rule (1)

- Now consider a hypothetical let rule:

$$\frac{O \vdash e_0 : T \qquad T \leq T_0 \qquad O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 = e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?
- The following good program does not typecheck

  let x : int = 0 in x + 1

- And some bad programs do typecheck

  foo(x : B) : int = let x : A =new A in x.b();

Example of Wrong Let Rule (2)

- Now consider another hypothetical let rule:

$$\frac{O \vdash e_0 : T \qquad T_0 \leq T \qquad [x{=}T_0]\, O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 = e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?

Example of Wrong Let Rule (2)

- Now consider another hypothetical let rule:

$$\frac{O \vdash e_0 : T \qquad T_0 \leq T \qquad [x=T_0]\, O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 = e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?
- The following bad program is well typed

  let x : B = new A in x.b()

- Why is this program bad?

Example of Wrong Let Rule (3)

- Now consider another hypothetical let rule:

$$\frac{O \vdash e_0 : T \qquad T \leq T_0 \qquad [x=T]O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 = e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?

Example of Wrong Let Rule (3)

- Now consider another hypothetical let rule:

$$\frac{O \vdash e_0 : T \qquad T \leq T_0 \qquad [x=T]O \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 = e_0 \text{ in } e_1 : T_1}$$

- How is it different from the correct rule?
- The following good program is not well typed

  let x : A = new B in {… x = new A; x.a(); }
- Why is this program not well typed?

Comments

- The typing rules use very concise notation
- They are very carefully constructed
- Virtually any change in a rule either:
  - Makes the type system unsound
    (bad programs are accepted as well typed)
  - Or, makes the type system less usable
    (good programs are rejected)

- But some good programs will be rejected anyway
  - The notion of a good program is undecidable

Assignment

More uses of subtyping:

$$\frac{\begin{array}{c} O(id) = T_0 \\ O \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O \vdash id = e_1 : T_1} \quad \text{[Assign]}$$

If-Then-Else

- Consider:

  if e_0 then e_1 else e_2

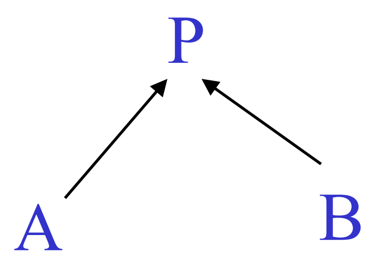
- The result can be either e_1 or e_2

- The dynamic type is either e_1's or e_2's type

- The best we can do statically is the smallest supertype larger than the type of e_1 and e_2

If-Then-Else example

- Consider the class hierarchy



- … and the expression

  if … then new A else new B

- Its type should allow for the dynamic type to be both A or B
  - Smallest supertype is P

Least Upper Bounds

- lub(X,Y), the least upper bound of X and Y, is Z if
  - $X \le Z \wedge Y \le Z$

    Z is an upper bound
  - $X \le Z' \wedge Y \le Z' \Rightarrow Z \le Z'$

    Z is least among upper bounds
- In CLASSES, the least upper bound of two types is their least common ancestor in the inheritance tree

If-Then-Else Revisited

$$\frac{\begin{array}{c} O \vdash e_0 : \text{boolean} \\ O \vdash e_1 : T_1 \\ O \vdash e_2 : T_2 \end{array}}{O \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \text{lub}(T_1, T_2)} \quad \text{[If-Then-Else]}$$

if-is-then-else

- The rule for if-is-then-else expressions takes a lub over all branches

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ [x_1{=}T_1]O \vdash e_1 : T_1' \\ O \vdash e_2 : T_2' \end{array}}{O \vdash \text{if } e_0 \text{ is } x_1{:}T_1 \text{ then } e_1 \text{ else } e_2 : \text{lub}(T_1', T_2')} \quad \text{[if-is]}$$

Next

- Type checking method dispatch

Method Dispatch

- There is a problem with type checking method calls:

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O \vdash e_1 : T_1 \\ \ldots \\ O \vdash e_n : T_n \end{array}}{O \vdash e_0.f(e_1,\ldots,e_n) : ?} \quad \text{[Dispatch]}$$

- We need information about the formal parameters and return type of f

Notes on Dispatch

- In CLASSES, method and object identifiers live in different name spaces
  - A method foo and an object foo can coexist in the same scope
- In the type rules, this is reflected by a separate mapping M for method signatures

$$M(C,f) = (T_1, \ldots T_n, T_{n+1})$$

means in class C there is a method f

$$f(x_1:T_1, \ldots, x_n:T_n):\ T_{n+1}$$

An Extended Typing Judgment

- Now we have two environments O and M

- The form of the typing judgment is

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

read as: "with the assumption that the object identifiers have types as given by O and the method identifiers have signatures as given by M, the expression e has type T"

The Method Environment

- The method environment must be added to all rules
- In most cases, M is passed down but not actually used
  - Example of a rule that does not use M:

$$\frac{O, M \vdash e_1 : T_1 \qquad O, M \vdash e_2 : T_2}{O, M \vdash e_1 + e_2 : \text{int}} \quad \text{[Add]}$$

  - Only the dispatch rules use M

The Dispatch Rule Revisited

$$\frac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \ldots \\ O, M \vdash e_n : T_n \\ M(T_0, f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad (\text{for } 1 \leq i \leq n) \end{array}}{O, M \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}'} \text{[Dispatch]}$$

Static Dispatch

- Static dispatch is a variation on normal dispatch

- The method is found in the class explicitly named by the programmer

- The inferred type of the dispatch expression must conform to the specified type

Static Dispatch (Cont.)

$$\frac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \ldots \\ O, M \vdash e_n : T_n \\ T_0 \leq T \\ M(T, f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad (\text{for } 1 \leq i \leq n) \end{array}}{O, M \vdash e_0[T].f(e_1, \ldots, e_n) : T_{n+1}'} \quad \text{[StaticDispatch]}$$

Initial Type Environment: Attributes

- Let $O_C(x) = T$ for all attributes $x{:}T$ in class C
  - O_C represents the class-wide scope

Methods

$$\frac{\begin{array}{c} M(C, f) = (T_1,\ldots,T_n, T_0) \\ [self=C][x_1=T_1]\ldots[x_n=T_n]O_c,M \vdash e : T_0 \end{array}}{O_c,M \vdash f(x_1:T_1,\ldots, x_n:T_n) : T_0 = e}$$

[Method]

Type Systems

- The rules in these lecture were CLASSES-specific
  - Other languages have very different rules
- General themes
  - Type rules are defined on the structure of expressions
  - Types of variables are modeled by an environment
- Types are a play between flexibility and safety

Code Generation for Object-Oriented Languages

Formal Grammar of CLASSES

```
prog: cls*;

cls: 'class' type 'extends' type '{' field* method*
'}';
field: iden ':' type ';';
method: iden '(' paramlist ')' ':' type '=' expr ';';

expr: 'new' type
    | expr '.' iden '(' arglist ')'
    | expr '[' type ']' '.' iden '(' arglist ')'
    | expr ('*' | '/') expr
    | expr ('+' | '-') expr
    | expr ('=='|'!='|'>'|'<'| '>=' | '<=') expr
    | iden '=' expr
    | 'let' iden ':' type '=' expr 'in' expr
    | 'if' expr 'then' expr 'else' expr
    | 'if' expr 'is' iden ':' type 'then' expr 'else'
expr
    | 'while' expr 'do' expr
    | 'print' expr
    | 'isnull' expr
    | 'self'
    | num
    | iden
    | string
    | '{' exprseq '}'
    | '(' expr ')'
    ;
paramlist: param (',' param)*
    | ;
param: iden ':' type;
arglist: expr (',' expr)*
    | ;
exprseq: expr (';' expr)*;

type: ID ;
num: INT;
iden: ID;
string: STRING;

ID  :  ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..
'9'|'_')* ;
INT :  '0'..'9'+ ;
WS  :  (' ' '\t' | ' \r' | ' \n')+ -> skip ;
STRING : '"' (~'"')* '"';
```

Object Layout

- OO implementation = Stuff from codegen lecture for TYPEDREFS + More stuff

- OO Slogan: If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected

- This means that code in class A works unmodified for an object of class B

Two Issues

- How are objects represented in memory?
- How is dynamic dispatch implemented?

Object Layout (Cont.)

An object is like a struct in C. The reference

foo.field

is an index into a foo struct at an offset corresponding to field

Objects in CLASSES are implemented similarly

- Objects are laid out in contiguous memory
- Each attribute stored at a fixed offset in object
- When a method is invoked, the object is self and the fields are the object's attributes

CLASSES Object Layout

- The first 3 words of CLASSES objects contain header information:

Class Tag
Object Size
Dispatch Ptr
Attribute 1
Attribute 2
. . .

CLASSES Object Layout (Cont.)

- Class tag is an integer
  - Identifies class of the object
- Object size is an integer
  - Size of the object
- Dispatch ptr is a pointer to a table of methods
  - More later
- Attributes in subsequent slots
- Lay out in contiguous memory
  - an array of Value objects

Value class in LLJ: to denote values of any type

```
class Value {
   public int i;
   public boolean b;
   public Lambda l;
   public String s;
   public Value[] a;
}
```

If *C/C++* is the target language, one can use *C*'s union.

A list of extra valid instructions in LLJ

In the following, $f could be i, s, b, l and $r or $r<i> could one of a0, t1, t2, t3, t4, $l could be any valid label, $fun is the name of any class that extends Lambda, $c is some integer literal, $lit is some integer, string, or boolean literal

- **$r.a=new Value[$c];**
- **$r1.$f=$r2.a[$c].$f;**
- **$r1.a[$c].$f=$r2.$f;**

Object Layout Example

```
class A {
    a: int;
    d: int;
    f(): int = a = a + d ;
};

class  B extends A {
    b: int;
    f(): int = a; // Override
    g(): int = a = a - b ;
};

class  C extends A {
    c: int;
    h(): int = a = a * c ;
};
```

Object Layout (Cont.)

- Attributes a and d are inherited by classes B and C
- All methods in all classes refer to a
- For A methods to work correctly in A, B, and C objects, attribute a must be in the same "place" in each object

Subclasses

Observation: Given a layout for class A, a layout for subclass B can be defined by extending the layout of A with additional slots for the additional attributes of B

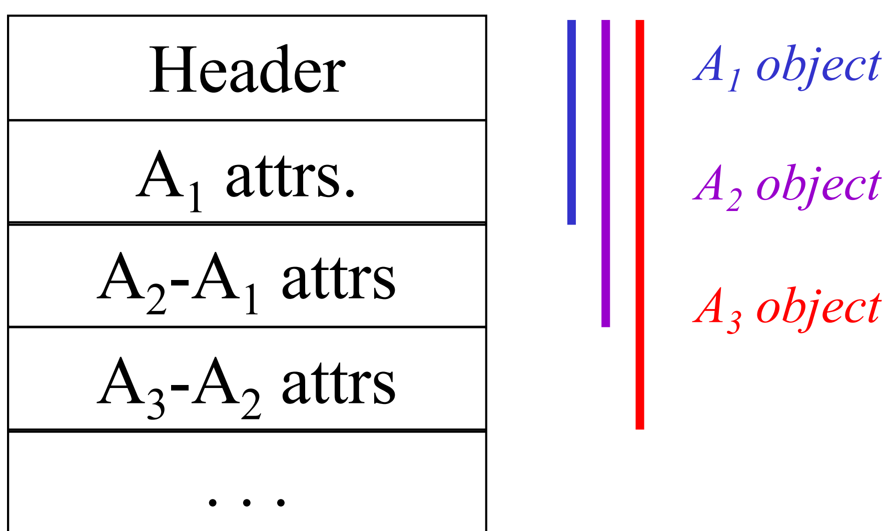
Leaves the layout of A unchanged
(B is an extension)

Layout Picture

Offset \ Class	A	B	C
0	Atag	Btag	Ctag
1	5	6	6
2	*	*	*
3	a	a	a
4	d	d	d
5		b	c

Subclasses (Cont.)

- The offset for an attribute is the same in a class and all of its subclasses
  - Any method for an A_1 can be used on a subclass A_2
- Consider layout for $A_n \leq \ldots \leq A_3 \leq A_2 \leq A_1$



Dynamic Dispatch

- Consider again our example

```
class A {
    a: int;
    d: int;
    f(): int = a = a + d ;
};

class B extends A {
    b: int;
    f(): int = a; // Override
    g(): int = a = a - b ;
};

class C extends A {
    c: int;
    h(): int = a = a * c ;
};
```

Dynamic Dispatch Example

- e.g()
  - g refers to method in B if e is a B
- e.f()
  - f refers to method in A if e is an A or C (inherited in the case of C)
  - f refers to method in B for a B object

- The implementation of methods and dynamic dispatch strongly resembles the implementation of attributes

Dispatch Tables

- Every class has a fixed set of methods (including inherited methods)

- A *dispatch table* indexes these methods
  - An array of method entry points
  - A method f lives at a fixed offset in the dispatch table for a class **and all of its subclasses**

Dispatch Table Example

class / Offset	A	B	C
0	fA	fB	fA
1		g	h

- The dispatch table for class A has only 1 method
- The tables for B and C extend the table for A with more methods
- Because methods can be overridden, the method for f is not the same in every class, but is always at the same offset

Using Dispatch Tables

- The dispatch pointer in an object of class X points to the dispatch table for class X

- Every method f of class X is assigned an offset O_f in the dispatch table at compile time

Using Dispatch Tables (Cont.)

- Every method must know what object is "self"
  - "self" is passed as the first argument to all methods
- To implement a dynamic dispatch e.f() we
  - Evaluate e, obtaining an object x
  - Find D by reading the dispatch-table field of x
  - Call $D[Offset_f](x)$
    - D is the dispatch table for x
    - In the call, self is bound to x

Operational Semantic of CLASSES

Operational Rules of CLASSES

- The evaluation judgment is

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

read:

  - Given so the current value of the self object
  - And E the current variable environment
  - And S the current store
  - If the evaluation of e terminates then
  - The returned value is v
  - And 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"
- The operational semantics allows for non-terminating evaluations
- We define one rule for each kind of expression

CLASSES Values

- Primitive values

Int(5)	the integer 5
Bool(true)	the boolean true
String("hello")	the string "hello"
$X(a_1=l_1, \ldots, a_n=l_n)$	object of type X
Null	

Operational Semantics for Base Values

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

$$\frac{\text{s is a string literal}}{\text{so, E, S} \vdash \text{s : String(s), S}}$$

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

Operational Semantics of Variable References

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

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

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

Operational Semantics of Assignment

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

- A three step process
  - Evaluate the right hand side
    - $\Rightarrow$ a value and 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 Conditionals

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

- The “threading” of the store enforces an evaluation sequence
  - e_1 must be evaluated first to produce S_1
  - Then e_2 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)

Operational Semantics of 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 evaluation sequence
- Only the last value is used
- But all the side-effects are collected

Operational Semantics of while (I)

$$\frac{so, E, S \vdash e_1 : \text{Bool(false)}, S_1}{so, E, S \vdash \text{while } e_1 \text{ do } e_2 : \text{Null}, S_1}$$

- 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 Null
- The typing rules ensure that e_1 evaluates to a boolean object

Operational Semantics of while (II)

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

- 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
- The result of evaluating e_2 is discarded
  - Only the side-effect is preserved

Operational Semantics of let Expressions (I)

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

- What is the context in which e_2 must be evaluated?
  - Environment like E but with a new binding of id to a fresh location l_{new}
  - Store like S_1 but with l_{new} mapped to v_1

Operational Semantics of let Expressions (II)

- 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
- The operational rule for let:

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

Default Values

- For each class A there is a default value denoted by D_A
  - D_{int} = Int(0)
  - D_{bool} = Bool(false)
  - D_{string} = String(“”)
  - D_A = Null (for another class A)

More Notation

- For a class A we write

  $\text{class}(A) = (a_1 : T_1, \ldots, a_n : T_n)$ where
  - a_i are the attributes (including the inherited ones)
  - T_i are their declared types

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
  - Return the newly allocated object

Operational Semantics of new

- Rule for new T

$$\frac{\begin{array}{l} \text{class}(T) = (a_1 : T_1, \ldots, a_n : T_n) \\ l_i = \text{newloc}(S) \text{ for } i = 1, \ldots, n \\ v = T\,(a_1 = l_1, \ldots, a_n = l_n) \\ S_1 = S[D_{T1}/l_1, \ldots, D_{Tn}/l_n] \end{array}}{so, E, S \vdash \text{new } T : v, S_1}$$

Operational Semantics of new. Notes.

- The first three lines allocate the object
- The last line initializes it

More Notation

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

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

  - x_i are the names of the formal arguments
  - e_{body} is the body of the method

Operational Semantics of Dispatch

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

Operational Semantics of Dispatch. Notes.

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

Operational Semantics of Static Dispatch

$$
\frac{
\begin{array}{l}
so, E, S \vdash e_0 \ : v_0, S' \\
so, E, S' \vdash e_1 \ : v_1 , S_1 \\
so, E, S_1 \vdash e_2 \ : v_2 , S_2 \\
\ldots \\
so, E, S_{n-1} \vdash e_n \ : v_n , S_n \\
v_0 = X(a_1 = l_1,\ldots, a_m = l_m) \\
impl(T, f) = (x_1,\ldots, x_n, e_{body}) \\
l_{xi} = newloc(S_n) \text{ for } i = 1,\ldots,n \\
E' = [x_1 : l_{x1}, \ldots, x_n : l_{xn}, a_1 : l_1,\ldots,a_m : l_m] \\
S_{n+1} = [l_{x1}=v_1,\ldots,l_{xn}=v_n]S_n \\
v_0 , E' , S_{n+1} \vdash e_{body} : v, S_{n+2}
\end{array}
}{
so, E, S \vdash e_0[T].f(e_1,\ldots,e_n) \ : v, S_{n+2}
}
$$

Runtime Errors

Operational rules do not cover all cases

Consider for example the rule for dispatch:

$$\frac{\begin{array}{l} \ldots \\ so, E, S_n \vdash e_0 \ : v_0, S_{n+1} \\ v_0 = X(a_1 = l_1, \ldots, a_m = l_m) \\ impl(X, f) = (x_1, \ldots, x_n, e_{body}) \\ \ldots \end{array}}{so, E, S \vdash e_0.f(e_1, \ldots, e_n) \ : v, S_{n+3}}$$

What happens if $impl(X, f)$ is not defined?

- 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 null
  - Division by zero
- In such case the execution must abort gracefully
  - With an error message

isnull

$$\frac{so, E, S \vdash e : Null, S_1}{so, E, S \vdash isnull\ e : Bool(true), S_1}$$

$$\frac{so, E, S \vdash e : X(\ldots), S_1}{so, E, S \vdash isnull\ e : Bool(false), S_1}$$

X could be any class including int, object, string, boolean

Operation semantics of if-is-then-else

$$\frac{\begin{array}{l} so, E, S \vdash e_1 : v_1, S_1 \\ v_1 = X(\ldots) \\ X \text{ is not subtype of } T \\ so, E, S_1 \vdash e_3 : v, S_2 \end{array}}{so, E, S \vdash \text{if } e_1 \text{ is id: } T \text{ then } e_2 \text{ else } e_3 : v, S_2}$$

$$\frac{\begin{array}{l} so, E, S \vdash e_1 : v_1, S_1 \\ v_1 = X(\ldots) \\ X \text{ is subtype of } T \\ l_{new} = newloc(S_1) \\ so, [id=l_{new}]E, [l_{new}=v_1]S_1 \vdash e_2 : v, S_2 \end{array}}{so, E, S \vdash \text{if } e_1 \text{ is id: } T \text{ then } e_2 \text{ else } e_3 : v, S_2}$$

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 CLASSES