Interpretation of Object-Oriented Languages

Éric Tanter
PLEIAD Laboratory
Computer Science Dept (DCC)
University of Chile
etanter@dcc.uchile.cl

Pleiad
Programming Languages and Environments for Intelligent, Adaptable and Distributed systems

Object-oriented programming

object = piece of state + behavior (coherent whole)

- local state: fields
- methods: behavior that has access to fields
  - calling a method <=> message passing

in higher-order procedural languages with state (Scheme)

- a closure is an object
- fields: free variables
- method: only one! apply!

Classes

Reuse of methods across several pieces of state (*)

- class = structure that specifies fields and methods of a bunch of objects
- object = instance of a class

Sharing of implementation / specialization

- inheritance ("A *inherits from* B", "A *extends* B")
- incremental modification of existing class
  - add or change behavior
  - add state

(*) *Prototypes have a different/more general way of addressing this reuse/extension issue*

Dynamic Dispatch

```
(class in-node extends object
  (field left)
  (field right)
  (method init (l r)
    (begin (set! left l)
           (set! right r)))
  (method sum ()
    (+ (send left sum ())
       (send right sum ()))))

(class leaf extends object
  (field value)
  (method init (v) (set! value v))
  (method sum () value))
```

```
(let ((o1 (new in-node
              ((new in-node
                 ((new leaf (3))
                  (new leaf (4))))
               (new leaf (5))))))
  (send o1 sum ()))
```

Methods are Mutually Recursive

```
(class odd-even extends object
  (method even (n)
    (if (= 0 n)
        #t
        (send self odd (- n 1))))
  (method odd (n)
    (if (= 0 n)
        #f
        (send self even (- n 1)))))

(let ((o (new odd-even ())))
  (send o odd (13)))
```


Inheritance

Incremental modification of existing classes

- parent, superclass
- child, subclass: c2 < c1

Single vs. multiple inheritance

Subclass polymorphism

- instances of c2 < c1 can be used anywhere instance of c1 can.

Redefinitions

- fields: shadowing (lexical scoping)
- methods: overriding

Terminology: Host class

Host class of a method

- class in which a method is declared

Host class of an expression

- host class of the method (if any) in which the expression occurs

Superclass of a method or expression

- parent class of the host class

For a given “operation”, potentially many methods / host classes.

Self and Super

Self call

- look for method in the class of *self*
- *self* is a dynamically-scoped variable (passed as implicit parameter)
- this is *dynamic* method dispatch

Super call

- super.n(...), (super n ...)
- super call of *n* in body of method *m* invokes a method *n* of the parent of *m*’s host class
- (can be) != parent of the class of *self*
- this is *static* method dispatch

Classes: A simple Class-Based OO Language

```
ExpVal = Int + Bool + Proc + Listof(ExpVal) + Obj
DenVal = Ref(ExpVal)
```

```
PROGRAM
<prog> ::= <class-decl>* <expr>

DECLARATIONS
<class-decl>  ::= (class <id> extends <id>
                    <field-decl>* <method-decl>*)
<field-decl>  ::= (field <id>)
<method-decl> ::= (method <id> (<id>*) <expr>)

NEW EXPRESSIONS
<expr> ::= (new <id> (<id>*))
<expr> ::= (send <expr> <id> (<expr>*))
<expr> ::= (super <id> (<expr>*))
<expr> ::= self
```


Interpretation Phases

1. Elaboration of classes
   - all class declarations are processed
   - initialize a global class environment (name -> cls)
2. Interpretation of expressions
   - start with the ‘startup expression’ (eg. Main.main())
   - manage environment, environment-passing style
   - deal with new kinds of expressions

Self/Super

Expression evaluated as part of a method operating on some object

Self must be bound in the environment

- pseudo-variable %self
- bound lexically, but different properties

Must also “remember” the superclass of the host class

- pseudo-variable %super

New Expressions

Self

self

- just lookup %self in the environment

Method Call

(send *obj n arg...*)

- evaluate arguments and object expressions
- look in object to get class name
- get the class
- look up the method (find-method: cname mname -> method)
- apply method

Super Call

(super *n arg...*)

- same as method call **except** for method lookup
- get super class of current host class: %super in environment

Object Creation

(new *c arg...*)

- evaluate arguments
- using class name *n*, create new empty object
- call initialize method on new object (ignore result)
- return new initialized object

Representing Objects, Methods, and Classes

Objects

Data structure

- class name
- list of (references to) fields

Creating a new object

- get field list from class
- create new list of fields with refs to illegal values

Methods

Like a normal procedure, except that

- it does not have a saved environment
- keeps track of
  - the names of the fields it “sees”
  - the name of its superclass

Applying a method

- run body in environment where:
  - formal parameters bound to arguments (by-copy semantics)
  - %self bound to current object
  - %super bound to method’s superclass
  - visible field names bound to fields of the current object

Looking up Fields

Field shadowing

- a method “sees” the most recently defined fields *at its level*
- keep position constant if field not redefined
- field list built from left to right
  - in c1: (x y) [0 1 2 3]
  - in c2: (x y y) [0 1 2 3]
  - in c3: (x y y z) [0 1 2 3]
- lookup should use *last* position!
  - dedicated lookup, or
  - shadow field names: in c3: (x y%1 y z)

Classes

Get information for a class from its name

- class environment: define new class, lookup class
- classes are static, top-level entities (globally visible)

Data structure for classes

- superclass name (#f if object)
- field names
- method environment

Class elaboration phase

- start with empty class environment
- initialize with class **object**
- for each declaration, add a new binding in class environment