10—Object-Oriented Programming

CS4215: Programming Language Implementation

Martin Henz

March 22, 2011

Generated on Saturday 24 March, 2012, 11:16

Semantic Domains

Domain name	Definition
EV	$\mathbf{Int} + \mathbf{Bool} + \mathbf{Fun} + \mathbf{Rec} + \{\bot\}$
SV	$\mathbf{Int} + \mathbf{Bool} + \mathbf{Fun} + \mathbf{Rec}$
DV	$\mathbf{Loc}$
Rec	$\mathbf{Id} \leadsto \mathbf{Loc}$

Function Application

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (f, \Sigma') \qquad \Sigma' \mid \Delta \Vdash E_2 \rightarrowtail (v_2, \Sigma'')}{\Sigma \mid \Delta \Vdash (E_1 \ E_2) \rightarrowtail f(l, \Sigma''[l \leftarrow v_2])}$$

where l is a new location in Σ''.

Property Access

$$\frac{\Sigma \mid \Delta \Vdash E \rightarrowtail (f, \Sigma')}{\Sigma \mid \Delta \Vdash E.q \rightarrowtail (\Sigma'(f(q)), \Sigma')}$$

Property Assignment

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (f, \Sigma') \qquad \Sigma' \mid \Delta \Vdash E_2 \rightarrowtail (v, \Sigma'')}{\Sigma \mid \Delta \Vdash E_1.q{:=}E_2 \rightarrowtail (v, \Sigma''[f(q) \leftarrow v])}$$

imPL: Passing Records to Functions

```
let a = [Myfield: 0]
in
   (fun b ->
       b.Myfield := 1;
       b := [Myfield: 2];
       b.Myfield := 3
    end
    a);
   a.Myfield
end
```

Alternative: Pass-by-Reference

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (f, \Sigma')}{\Sigma \mid \Delta \Vdash (E_1\ x) \rightarrowtail f(\Delta(x), \Sigma')}$$

Alternative: Pass-by-Copy

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (f, \Sigma') \qquad \Sigma' \mid \Delta \Vdash E_2 \rightarrowtail (v_2, \Sigma'')}{\Sigma \mid \Delta \Vdash (E_1\ E_2) \rightarrowtail f(l, \Sigma''')}$$

if $v_2 \in \mathbf{Rec}$,
where $l, l'_1, \ldots, l'_n$ are new locations in Σ'',
$\{q_1, \ldots, q_n\} = dom(v_2)$,
$$\Sigma''' = \Sigma''[l \leftarrow \{(q_1, l'_1), \ldots, (q_n, l'_n)\}] \\ [l'_1 \leftarrow \Sigma''(v_2(q_1))] \ldots [l'_n \leftarrow \Sigma''(v_2(q_n))]$$

A Naive Interpreter

Idea

Translate denotational semantics to a high-level language, in our case Java.

Example

```
public class Sequence implements Expression {
   public Expression firstPart, secondPart;
   public StoreAndValue eval(Store s, Environment e) {
      StoreAndValue s_and_v_1 = firstPart.eval(s, e);
      return secondPart.eval(s_and_v_1.store,e);
   }
}
```

Naive Implementation of Store

```
public class Store extends Vector<Value> {
   public Store extend(int location, Value value) {
      Store newStore = (Store) clone();
      newStore.add(location,value);
      return newStore;
   }
}
```

Observation

Threading of store

The store is threaded through the execution of the program. At any point in time, only one store is in use. Older stores are inaccessible.

Idea

Use only one store, and physically change it upon assignment.

Realistic Implementation of Store

```
public class Store extends Vector<Value> {
   public Store set(int location, Value value) {
      add(location,value);
      return this;
   }

   public static Store theStore = new Store();
}
```

Naive Implementation of Assignment

```
public StoreAndValue eval(Store s, Environment e) {
     StoreAndValue s_and_v = rightHandSide.eval(s, e);
     s_and_v.store.setElementAt(s_and_v.value,
                                   e.access(leftHandSi
return new StoreAndValue(s_and_v.store,
 s_and_v.value);
}
```

Realistic Implementation of Assignment

```
public class Assignment implements Expression {
   public String leftHandSide, rightHandSide;
   public Value eval(Environment e) {
      Value r = rightHandSide.eval(e);
      Store.theStore.set(e.access(leftHandSide),r);
      return r;
   }
}
```

Idea

Reuse heap for implementing imperative constructs

Translation of Assignment

$$\frac{E \hookrightarrow s}{x \ \texttt{:=}\ E \hookrightarrow s.\texttt{ASSIGNS}\ x}$$

Execution of Assignment

$$\frac{s(pc) = \texttt{ASSIGNS}\ x}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc+1, e, rs, update(e, x, v, h'))}$$

where $(v, h') = pop(os, h)$

Translation of Sequences

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1 \, ; E_2 \hookrightarrow s_1.\mathtt{POP}.s_2}$$

Implementation of Sequences

$$\frac{s(pc) = \texttt{POP}}{(os, pc, e, rs, h) \rightrightarrows_s (os, pc+1, e, rs, h')} \text{where } (v, h') = pop(os, h)$$

Translation of Loops

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{\begin{array}{c}\texttt{while } E_1 \texttt{ do } E_2 \\ \hookrightarrow \\ s_1.(\texttt{JOFR } |s_2 + 3|).s_2.\texttt{POP}. \\ (\texttt{GOTOR } -(|s_1| + 2 + |s_2|)).LDCB \texttt{ true}\end{array}}$$

1. Imperative Programming: Some Loose Ends
2. A Virtual Machine for imPL
3. Object-Oriented Programming
   - Knowledge Representation View of Objects
   - Software Development View
   - Object-oriented programming in imPL
4. A Realistic Object System
5. Meta-Programming

Knowledge Representation View of Objects

- Aggregation
- Classification
- Specialization

Aggregation

```
let myVehicle = [MaxSpeed: 85] in ... end
```

Classification

```
let newVehicle = fun ms -> [MaxSpeed: ms] end
in
   ...
   let myVehicle = (newVehicle 85) in
      ...
   end
end
```

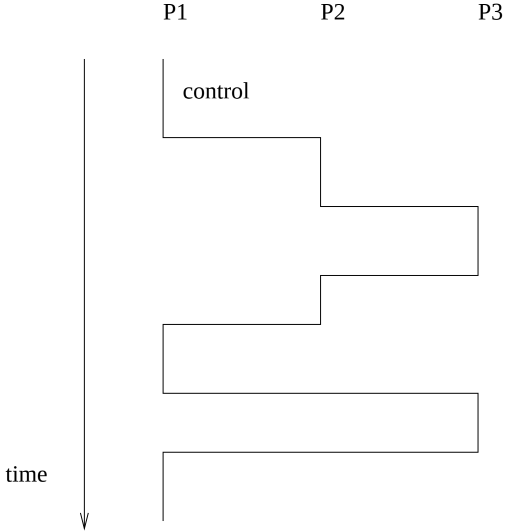
Specialization

```
let newCar = fun mp -> let c = (newVehicle 95)
                       in c.MaxPassengers := mp
                       end
             end
in
   ...
   let myCar  = (newCar 5) in
      ...
   end
end
```

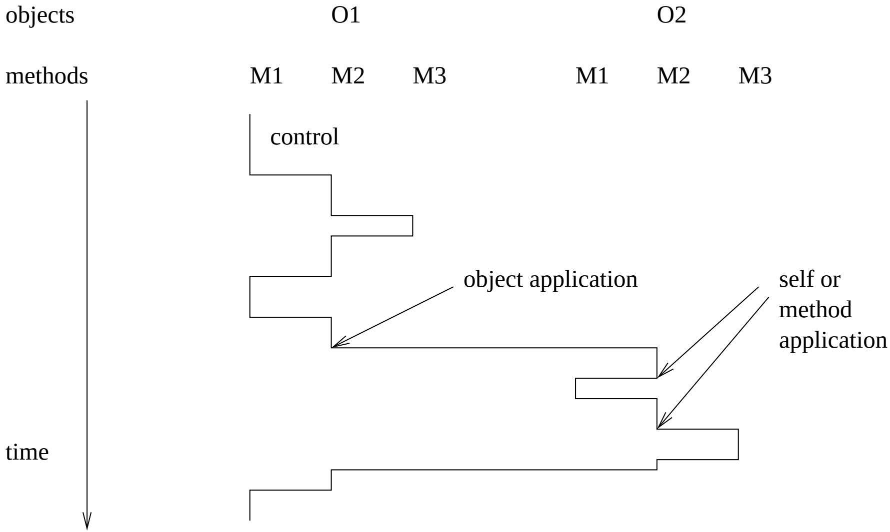
Specialization

Usually, the concept of inheritance (class extension) achieves specialization in object-oriented languages.

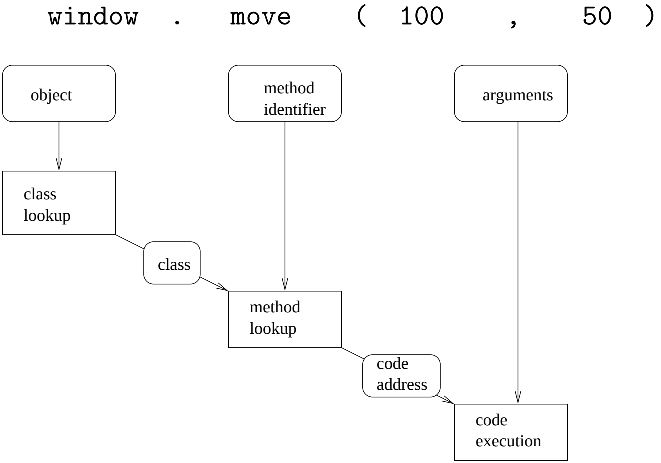
Control Flow in Procedural Languages (time)



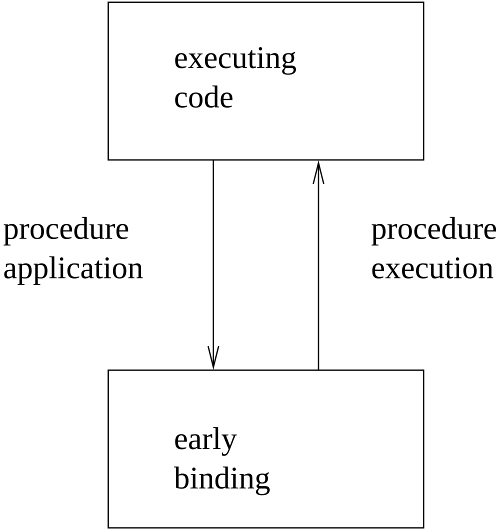
Control Flow in Object-oriented Languages (time)



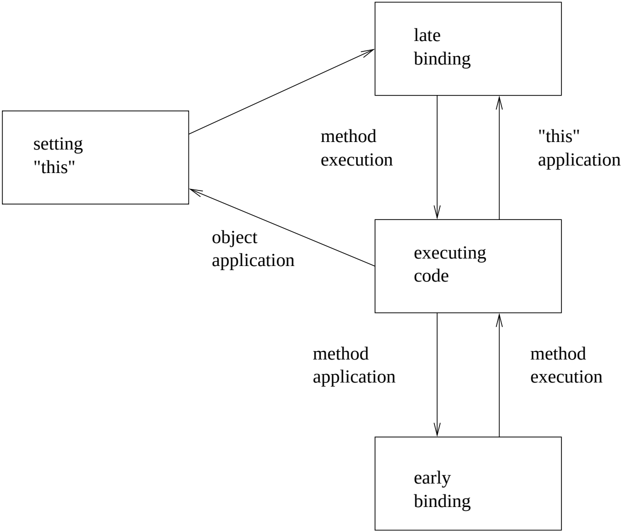
Execution of Object Application



Early Binding in Procedural Languages



Late Binding in Object-oriented Languages



A Simple Object System in imPL

```
let stack =
   [Push: fun this x ->
            this.Content := x :: this.Content end,
    Pop:  fun this   ->
            let top = this.Content.First in
               this.content := this.Content.Second;
               top
            end,
    Makeempty:
          fun this   -> this.Content := []; this end
   ]
in ... end
```

Creating an Object

```
let mystack = (stack.Makeempty []) in ... end
```

Operating on Objects

```
(stack.Push mystack 1);
(stack.Push mystack 2);
(stack.Pop mystack) + (stack.Pop mystack)
```

First-Class Properties

$$\frac{}{q} \qquad\qquad \frac{E_1 \qquad E_2}{E_1\,.\,E_2}$$

$$\frac{E_1 \qquad E_2}{E_1 \;\texttt{hasproperty}\; E_2}$$

Example

```
let access = fun r p -> r.p end
    r = [A: 1, B: 2]
in  (access r B) * 2
end
```

Adding Properties in imPL?

```
let r = [A: 1, B: 2]
in  r . C := 3
end
```

will fail in imPL, due to the semantics of record property assignment:

$$\frac{\Sigma \mid \Delta \Vdash E_1 \rightarrowtail (f, \Sigma') \qquad \Sigma' \mid \Delta \Vdash E_2 \rightarrowtail (v, \Sigma'')}{\Sigma \mid \Delta \Vdash E_1.q\texttt{:=}E_2 \rightarrowtail (v, \Sigma''[f(q) \leftarrow v])}$$

Adding Properties in oPL!

```
let r = [A: 1, B: 2]
in  r . C := 3
end
```

will succeed in oPL, adding the property `C` to the record.

"First-class" and Adding Properties

```
let addProperty = fun r p v -> r.p := v end
in  (addProperty [A:1,B:2] C 3)
end
```

Late Binding in imPL

```
let new = fun theClass -> [Class: theClass] end
in ...
   let mystack = (new stack) in
      (stack.Makeempty mystack)
   end
end
```

Method Lookup

```
let lookup = fun object methodname ->
                object.Class.methodname
             end
in ...
end
```

Method Lookup

```
( (lookup mystack Push) mystack 1 )
```

"This" Application

```
let stack =
  [Push:fun this x -> this.Content := x::this.Content end,
   Pop: fun this   -> let top = this.Content.First
                      in  this.Content
                             := this.Content.Second;
                          top
                      end
        end,
   Makeempty: fun this   -> this.Content := []; this end,
   Pushtwice: fun this x -> ((lookup this Push) this x);
                            ((lookup this Push) this x)
              end]
in ... end
```

Inheritance

```
...
let stackWithTop =
    [Parent: stack,
     Top: fun this -> this.Content.First]
in ...
end
```

Inheritance

```
let lookupInClass =
   recfun lookupInClass theClass methodname ->
      if theClass hasproperty methodname
      then theClass.methodname
      else (lookupInClass theClass.Parent methodname)
      end
   end
in
   let lookup = fun object methodname ->
                 (lookupInClass object.Class methodname)
                end
   in ...
   end
end
```

Object-oriented Syntax for oPL

$$\texttt{method}\ q(x_1 \cdots x_n)\ \texttt{->}\ E\ \texttt{end}$$

is abbreviation for the association

$$q\ \texttt{:}\ \texttt{fun this}\ x_1 \cdots x_n\ \texttt{->}\ E\ \texttt{end}$$

A Syntax for Classes

$$\texttt{class}\ M_1\ \cdots M_n\ \texttt{end}$$

stands for

$$\texttt{[}\ M_1,\ \cdots, M_n\texttt{]}$$

A Syntax for Classes

$$\texttt{class extends } x\ M_1\ \cdots M_n\ \texttt{end}$$

stands for

$$[\texttt{Parent:}x,\ M_1,\ \cdots, M_n]$$

A Syntax for Object Application

$$E\,.\,q(E_1 \cdots E_n)$$

stands for

$$\texttt{let obj = } E \texttt{ in ((lookup obj } q\texttt{) obj } E_1 \cdots E_n\texttt{) end}$$

Putting it all together

```
let stack =
 class method Push(x)->this.Content:=x::this.Content end,
       method Pop()  -> let top = this.Content.First in
                            this.Content
                               := this.Content.Second;
                            top
                         end
       end,
       method Makeempty()
                       -> this.Content := []; this
       end
 end
in ...end
```

Putting it all together

```
...
let stackWithTop =
    class extends stack
          method Top() -> this.Content.First end
    end
in
   let myStackWithTop = (new stackWithTop)
   in  myStackWithTop.Makeempty();
       myStackWithTop.Push(1);
       myStackWithTop.Push(2);
       myStackWithTop.Pop() + myStackWithTop.Pop()
   end
end
```

Implementation of oPL

- Efficient implementation of records described in the previous chapter is not possible for oPL (properties are first-class values in oPL).
- Virtual machine-based implementation of oPL needs to revert to representing records using hashtables.
- Existing languages such as Java avoid hashing for object fields through a type system.

Implementing Late Binding

- Target function can only be determined at runtime.
- Function `lookup` needs to follow the ancestor line of the class of the given object until a class is found that contains a method under the given property
- Process of method lookup can become bottleneck in the implementation of object-oriented languages.

Optimizing Late Binding

Observation

The rationale behind this optimization is that the actual argument objects may change frequently between invocations of `lookup`, whereas the classes of those argument objects change much less frequently. This rationale has been confirmed by statistics on real-world programs [Hoelzle:91].

Idea

Save the class and the result of the lookup for future use.

Inline Caching

Translate (`lookup obj` q) to a machine instruction sequence

```
LD <index of obj>
LDPS q
LOOKUP <class heap address> <cache>
```

Inline Caching

```
LD <index of obj>
LDPS q
LOOKUP <class heap address> <cache>
```

- `LOOKUP` has two extra parameters.
- After lookup, it stores the class and the function.
- Subsequent lookups compare the class with the class of the current argument. If the same, no need for lookup; just use the cached value!
- Technique is called *inline caching* because the machine code is used to store the lookup result.

Representing oPL Programs in oPL

```
(f x + 1)
```

can be represented by the oPL program

```
let fragment =
    (new application).
        Init((new identifier).Init(F),
             (new binaryPrimitiveOperator).
               Init(Plus,
                    (new identifier).Init(X),
                    (new integerConstant).Init(1)
                   ))
in  ... end
```

Meta-Programming in oPL

We can then implement programs in oPL that manipulate data structures that represent oPL program (*meta-programming*). Examples:

- compiler in oPL from oPL to an imPL VM language
- interpreter for oPL in oPL itself, a so-called *meta-circular interpreter*