10—Object-Oriented Programming

CS4215: Programming Language Implementation

Martin Henz

March 22, 2011

Generated on Saturday 24 March, 2012, 11:16
1. Imperative Programming: Some Loose Ends
   - Denotational Semantics
   - Alternatives: Pass-by-reference and pass-by-copy
   - A Realistic Interpreter for imPL

2. A Virtual Machine for imPL

3. Object-Oriented Programming

4. A Realistic Object System

5. Meta-Programming

CS4215: Programming Language Implementation 10—Object-Oriented Programming
## Semantic Domains

<table>
<thead>
<tr>
<th>Domain name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV</td>
<td>Int + Bool + Fun + Rec + {⊥}</td>
</tr>
<tr>
<td>SV</td>
<td>Int + Bool + Fun + Rec</td>
</tr>
<tr>
<td>DV</td>
<td>Loc</td>
</tr>
<tr>
<td>Rec</td>
<td>Id ⇝ Loc</td>
</tr>
</tbody>
</table>
Function Application

\[
\Sigma | \Delta \models E_1 \mapsto (f, \Sigma') \quad \Sigma' | \Delta \models E_2 \mapsto (v_2, \Sigma'')
\]

\[
\Sigma | \Delta \models (E_1 \ E_2) \mapsto f(l, \Sigma''[l \leftarrow v_2])
\]

where \( l \) is a new location in \( \Sigma'' \).
Property Access

\[
\Sigma \mid \Delta \vdash E \rightarrow (f, \Sigma') \\
\hline
\Sigma \mid \Delta \vdash E.q \rightarrow (\Sigma'(f(q)), \Sigma')
\]
Property Assignment

\[
\Sigma \mid \Delta \models E_1 \mapsto (f, \Sigma') \quad \Sigma' \mid \Delta \models E_2 \mapsto (v, \Sigma'')
\]

\[
\Sigma \mid \Delta \models E_1.q := E_2 \mapsto (v, \Sigma''[f(q) \leftarrow v])
\]
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

\[
\Sigma \mid \Delta \vdash E_1 \hookrightarrow (f, \Sigma') \\
\Sigma \mid \Delta \vdash (E_1 \ x) \hookrightarrow f(\Delta(x), \Sigma')
\]
Alternative: Pass-by-Copy

\[ \Sigma | \Delta \models E_1 \mapsto (f, \Sigma') \quad \Sigma' | \Delta \models E_2 \mapsto (v_2, \Sigma'') \]

\[ \Sigma | \Delta \models (E_1 \ E_2) \mapsto f(l, \Sigma''') \]

if \( v_2 \in \text{Rec}, \)
where \( l, l'_1, \ldots, l'_n \) are new locations in \( \Sigma'' \),
\[ \{q_1, \ldots, q_n\} = \text{dom}(v_2), \]
\[ \Sigma''' = \Sigma''[l \leftarrow \{(q_1, l'_1), \ldots, (q_n, l'_n)\}] \]
\[ [l'_1 \leftarrow \Sigma''(v_2(q_1))] \ldots [l'_n \leftarrow \Sigma''(v_2(q_n))] \]
A Naive Interpreter

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

Example

```java
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

```java
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

```java
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

```java
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(leftHandSide));
    return new StoreAndValue(s_and_v.store, s_and_v.value);
}
```
Realistic Implementation of Assignment

```java
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;
    }
}
```
1. Imperative Programming: Some Loose Ends

2. A Virtual Machine for imPL
   - Idea
   - Assignment
   - Sequences
   - Loops

3. Object-Oriented Programming

4. A Realistic Object System

5. Meta-Programming
Idea

Reuse heap for implementing imperative constructs
Translation of Assignment

\[ E \rightarrow s \]

\[ x := E \rightarrow s.\text{ASSIGNS} \ x \]
Execution of Assignment

\[ s(pc) = \text{ASSIGNS } x \]

\[ (os, pc, e, rs, h) \xrightarrow{s} (os, pc + 1, e, rs, update(e, x, v, h')) \]

where \( (v, h') = \text{pop}(os, h) \)
Translation of Sequences

\[ E_1 \leftarrow s_1 \quad E_2 \leftarrow s_2 \]

\[ E_1; E_2 \leftarrow s_1.Pop.s_2 \]
Implementation of Sequences

\[
s(pc) = \text{POP} \\
\text{where } (v, h') = \text{pop}(os, h) \\
(os, pc, e, rs, h) \xrightarrow{s} (os, pc + 1, e, rs, h')
\]
Translation of Loops

\[
\begin{align*}
E_1 & \rightarrow s_1 \\
E_2 & \rightarrow s_2 \\
\text{while } E_1 \text{ do } E_2 & \rightarrow \\
& \quad s_1.(\text{JOFR} \mid s_2 + 3).s_2.\text{POP}. \\
& \quad (\text{GOTOR} - (|s_1| + 2 + |s_2|)).l\text{DCB} \text{ true}
\end{align*}
\]
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

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

```plaintext
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)

objects

methods

O1
M1 M2 M3
control

O2
M1 M2 M3
object application

self or method application

time
Execution of Object Application

```
window . move ( 100 , 50 )
```
Early Binding in Procedural Languages

- Executing code
- Procedure application
- Procedure execution
- Early binding
Late Binding in Object-oriented Languages

- setting "this"
- method execution
- object application
- executing code
- method application
- method execution
- early binding
- "this" application
- late binding
A Simple Object System in imPL

```ml
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)
1. Imperative Programming: Some Loose Ends

2. A Virtual Machine for imPL

3. Object-Oriented Programming

4. A Realistic Object System
   - First-class Properties
   - Late Binding in imPL
   - Inheritance
   - Object-oriented Syntax for oPL
   - Implementation of oPL

5. Meta-Programming
First-Class Properties

\[ q \quad E_1 \quad E_2 \]

\[ E_1 . E_2 \]

\[ E_1 \quad E_2 \]

\[ E_1 \text{ hasproperty } E_2 \]
Example

```ocaml
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:

\[
\begin{align*}
\Sigma | \Delta \vdash E_1 \rightarrow (f, \Sigma') & \quad \Sigma' | \Delta \vdash E_2 \rightarrow (v, \Sigma'') \\
\hline 
\Sigma | \Delta \vdash E_1 . q := E_2 \rightarrow (v, \Sigma''[f(q) \leftarrow v])
\end{align*}
\]
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
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
in
  let lookup = fun object methodname ->
    (lookupInClass object.Class methodname)
  end
in ...
end
end
Object-oriented Syntax for oPL

\[
\text{method } q(x_1 \cdots x_n) \rightarrow E \text{ end}
\]

is abbreviation for the association

\[
q : \text{fun this } x_1 \cdots x_n \rightarrow E \text{ end}
\]
A Syntax for Classes

class $M_1 \cdots M_n$ end

stands for

$[M_1, \cdots, M_n]$
A Syntax for Classes

class extends $x \ M_1 \cdots M_n \ end$

stands for

$[\text{Parent}: x, \ M_1, \cdots, M_n]$
A Syntax for Object Application

\[ E \cdot q(E_1 \cdots E_n) \]

stands for

\[ \text{let } \text{obj} = E \text{ in } ((\text{lookup } \text{obj } q) \text{ obj } E_1 \cdots E_n) \text{ end} \]
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 \((\text{lookup obj } q)\) 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.
Imperative Programming: Some Loose Ends
A Virtual Machine for imPL
Object-Oriented Programming
A Realistic Object System
Meta-Programming
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*