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
# Semantic Domains

<table>
<thead>
<tr>
<th>Domain name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV</td>
<td>( \text{Int} + \text{Bool} + \text{Fun} + \text{Rec} + {\bot} )</td>
</tr>
<tr>
<td>SV</td>
<td>( \text{Int} + \text{Bool} + \text{Fun} + \text{Rec} )</td>
</tr>
<tr>
<td>DV</td>
<td>\text{Loc}</td>
</tr>
<tr>
<td>Rec</td>
<td>\text{Id} \rightsquigarrow \text{Loc}</td>
</tr>
</tbody>
</table>
Function Application

\[
\Sigma \mid \Delta \models E_1 \rightsquigarrow (f, \Sigma') \quad \Sigma' \mid \Delta \models E_2 \rightsquigarrow (v_2, \Sigma'')
\]

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

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

\[
\Sigma \mid \Delta \vdash E \Rightarrow (f, \Sigma')
\]

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

\[
\begin{align*}
\Sigma | \Delta & \vdash E_1 \rightarrow (f, \Sigma') \\
\Sigma' | \Delta & \vdash E_2 \rightarrow (v, \Sigma'') \\
\hline
\Sigma | \Delta & \vdash E_1.q := E_2 \rightarrow (v, \Sigma''[f(q) \leftarrow v])
\end{align*}
\]
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

\[
\Sigma \mid \Delta \vdash E_1 \mapsto (f, \Sigma')
\]

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

\[
\begin{align*}
\Sigma | \Delta \vdash E_1 &\rightarrow (f, \Sigma') & \Sigma' | \Delta \vdash E_2 &\rightarrow (v_2, \Sigma'') \hfill \\
\hline \\
\Sigma | \Delta \vdash (E_1 \ E_2) &\rightarrow f(l, \Sigma''') \\
\end{align*}
\]

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();
}
```
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;
    }
}
```
Imperative Programming: Some Loose Ends

A Virtual Machine for imPL

Object-Oriented Programming

A Realistic Object System

Meta-Programming

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
Reuse heap for implementing imperative constructs
Translation of Assignment

\[ E \overset{\rightarrow}{\rightarrow} s \]

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

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

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

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

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

\[ E_1 ; E_2 \rightarrow s_1.POP.s_2 \]
Implementation of Sequences

\[ s(pc) = \text{POP} \]

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

\[ (os, pc, e, rs, h) \SEarrow_s (os, pc + 1, e, rs, h') \]
Translation of Loops

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

while \( E_1 \) do \( E_2 \)
\[
\leftrightarrow \\
\left( JOFFR \ |s_2 + 3| \right).s_2.POP. \\
\left( GOTOR \ - \ (|s_1| + 2 + |s_2|) \right).LDCB \ \text{true}
\]
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
let newVehicle = fun ms -> [MaxSpeed: ms] end in
    ...
    let myVehicle = (newVehicle 85) in
    ...
end end
let newCar = fun mp -> let c = (newVehicle 95) in c.MaxPassengers := mp 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)

P1  P2  P3

control

time
Control Flow in Object-oriented Languages (time)

objects

methods

M1  M2  M3

O1  

control

O2

object application

time

method application

self or method application
Execution of Object Application

```
window . move ( 100 , 50 )
```
Late Binding in Object-oriented Languages

- late binding
  - setting "this"
  - method execution
  - object application
  - executing code
  - method application
  - method execution
  - early binding

"this" application
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)
Imperative Programming: Some Loose Ends

A Virtual Machine for imPL

Object-Oriented Programming

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

Meta-Programming
First-Class Properties

$q = \mathbf{E}_1 \mathbf{E}_2$

$\mathbf{E}_1 . \mathbf{E}_2$

$\mathbf{E}_1 \text{ hasproperty } \mathbf{E}_2$
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?

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

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

\[
\Sigma | \Delta \vdash E_1 \triangleright (f, \Sigma') \quad \Sigma' | \Delta \vdash E_2 \triangleright (v, \Sigma'')
\]

\[
\Sigma | \Delta \vdash E_1.q := E_2 \triangleright (v, \Sigma''[f(q) \leftarrow v])
\]
Adding Properties in oPL!

```plaintext
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
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
  end

in
  let lookup = fun object methodname ->
    (lookupInClass object.Class methodname)
  end

in ...
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

```
[Parent:x, M_1, \cdots, M_n]
```
A Syntax for Object Application

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

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, 
    method Makeempty() 
        -> this.Content := []; this 
    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.
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 \((\text{lookup } \text{obj } q)\) to a machine instruction sequence

\[
\begin{align*}
\text{LD} & \; <\text{index of obj}> \\
\text{LDPS} & \; q \\
\text{LOOKUP} & \; <\text{class heap address}> \; <\text{cache}>
\end{align*}
\]
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.
1. Imperative Programming: Some Loose Ends
2. A Virtual Machine for imPL
3. Object-Oriented Programming
4. A Realistic Object System
5. Meta-Programming
Representing oPL Programs in oPL

\((f \ x + 1)\)

can be represented by the oPL program

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

in ... end
```
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