Emerald
Simple Call

object A
    B.f[]
end A

object B
    function f[]
end B
Remote Call

object A
    B.f[]
end A

object B
    function f[]
end B
Concurrency and Distribution in the Emerald Object-Oriented Language

Eric Jul

Professor II, University of Oslo
Professor Emeritus, University of Copenhagen
Member, Bell Labs Ireland
One Day Four People Gathered to Do an OO Language with Concurrency and Distribution

<table>
<thead>
<tr>
<th></th>
<th>OS/ OO-runtime-mobility</th>
<th>OO-language design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ph.D. Student</td>
<td>Eric Jul</td>
<td>Norm Hutchinson</td>
</tr>
<tr>
<td>Faculty</td>
<td>Hank Levy</td>
<td>Andrew Black</td>
</tr>
</tbody>
</table>
Main Contributions

- Distribution: Mobile objects (Eric/Hank)
  
  *Any object can move at any time. Full on-the-fly*
  
  - object mobility
  - thread mobility
  - heterogeneous mobility: VAX, SUN3, SPARC, DEC Alpha

- Conformity based type system (Norm/Andrew)
  
  Type system based on conformity principle
  
  Well-defined semantics (e.g., NIL makes sense!)

- Clean OO language (better than successors?)
  
  including uniform object model
History

- Developed in Seattle at the University of Washington 1984-1986
- Emerald is green; Emerald City is Seattle
- Original UW version: native code and virtual machine for VAX for speed
- UBC (University of British Columbia) version: Byte Codes for portability; compiler written in BC Emerald
What does it look like?

• In a nutshell: Java with an Algol-like syntax
• Heavily inspired by
  – Algol/Simula for syntax & semantics
• ”Clean” OO language – ”everything” is an object: data, integers, strings, arrays, classes, types as in Smalltalk
• Language constructs are NOT objects – for compilability and speed
• No pointers: just object & object references
Why?

- Objects in a distributed context
- Smalltalk SLOW – want ~ C performance
- Want strong typing
- Want lightweight objects
- Want full distribution including location concept, failure handling
- Want full, on-the-fly mobility
YOUR Background

• Know Java?
• Experienced Java programmer?
• Other OO languages?
Let’s start with objects

Principle: *Everything is an object!*

How to create an object?

Classic method:

\[ X = \text{new } \text{someclass} \]

*But this requires classes – let’s try Occam’s razor:*
Classless Object Construction

Object constructors:

```
object seqno
    var prev: Integer = 0
    Integer operation getSeqNo[]
    prev <- prev +1
    return prev
end getSeqno
end seqno
```

The above is an executable expression!
Classless Object Construction

Object constructors:

```plaintext
x <- object seqno

var prev: Integer = 0

Integer operation getSeqNo[]

prev <- prev + 1

return prev

end getSeqNo

end seqno
```

The above is an executable expression that is assigned to x.
Object Constructors

• Execution results in a new object
• Execute again – and get yet another object
• *No class!*

*Want classes?*
An Object that is a Class

object seqnoclass
  operation create[]
    return
    object seqno
      var prev: Integer = 0
      Integer operation getSeqNo[]
        prev <- prev + 1
        return prev
      end getSeqno
    end seqno
  end create
end seqnoclass
Classes with Free Variables

object seqnoclass
    operation create[]
        return
    object seqno
        var prev: Integer <- InitSN
        Integer operation getSeqNo[]
            prev <- prev +1
            return prev
        end getSeqno
    end seqno
end create
end seqnoclass
Classes with Parameters

object seqnoclass
    operation createInit[InitSN: Integer]
        return
            object seqno
                var prev: Integer <- InitSN
                Integer operation getSeqNo[]
                    prev <- prev +1
                    return prev
                end getSeqno
            end seqno
        end create
    end seqnoclass
The following turns into the previous double object constructor:

class seqno
  var prev: Integer = 0
  Integer operation getSeqNo[]
  prev <- prev +1
  return prev
end getSeqno
end seqno
Inheritance by Sugaring

const SC <- class seqno

var prev: Integer = 0

Integer operation getSeqNo[]

prev <- prev +1

return prev

end getSeqno

end seqno
Inheritance by Sugaring/Adding

const SC2 <- class seqno2 (SC)
   Integer operation getSeqNo2[]
      prev <- prev + 2
      return prev
   end getSeqNo2
end seqno2
Inheritance by Sugaring/Overwrite

const SC2 <- class seqno2 (SC)
    Integer operation getSeqNo[]
        prev <- prev + 2
        return prev
    end getSeqno
end seqno2
const SC2 <- class seqno2 (SC)
class function getSuper[] ->
  [r: Any]

    r <- SC
end getSuper
end seqno2
Using a class to create an object

Var mySeqNo: type-defined-later
mySeqNo <- SC.create[]

Classes ARE merely objects!
Types

Types are abstract descriptions of the operations required of an object (think: Java Interfaces – they are close to types in Emerald).

Collection of operation signatures.
Simple Type Example

type SeqNoSource
    Integer getSeqNo[]
end SeqNoSource

Think Java interface
Using a class to create an object

Var mySeqNo: SeqNoSource
mySeqNo <- SC.create[]
What is conformity?

Conformity object-to-type
and type-to-type

BankAccount conforms to
DepositOnlyBankAccount because it support all the require operations – and the parameters also conform
Conformity informally

An object is said to conform to a type, if

• It has the operations specified by the type
• For each operation in the type:
  – The number of parameters is the same in the object as in the type
  – Each input parameter of the object conforms to the corresponding param of the type
  – Each output parameter of the type conforms to the corresponding param of the object (contra variant)
Conformity between types

Conformity is a mathematical relationship
If T is to conform to S:
1. T must have all the operations required by S
2. For each operation in T the corresponding operation in S:
   • in-parameters must conform
   • out-parameters must conform in opposite order

Contravariance: not in Simula nor Eiffel
necessary to make semantic sense of programs
Conformity details

- Conformity is implicit
- No ”implements” as in Java
- Operation names important
- Parameter names do not matter, just their type
- Arity matters: foo(char) different from foo(char, float)
Conformity more formally

• Don’t listen to me: Talk to Andrew Black!
• An object can conform to many different types
• An object has a ”best-fitting” type: the ”largest” of the types that the object conforms to. Essentially just collect all its methods
• Conformity defined between types
Lattice of types

- Types form a lattice
- Top is `type Any` `end Any`
- Bottom is `Noone` (it has ALL operations”)
- NIL conforms to `Noone`
- NIL can thus be assigned to *any* variable!
  (Read ”Much Ado About NIL.”)
Const SC <- object seqnoclass
operation create[] -> [r: SeqNoSource]
return
  object seqno
  var prev: Integer = 0
operation getSeqNo[] -> [s:int]
  prev <- prev +1
  s <- prev
end getSeqno
end seqno
end create
end seqnoclass
Concurrenty

object A
  process
    ... do something
  end process
end A
Initialization

object A
    initially
    ... initialize object
end initially
process
    ... do something
end process
end A
Distribution

- Sea of objects (draw)
- Sea is divided into disjunct parts called Nodes
- An object is on one and only one Node at a time
- Each node is represented by a `Node` object
Location Primitive

- **Locate X** returns the node where X is (was!)
- **Note that the object may already have moved to another node (actually any number of moves)**
Mobility Primitive

move X to Y
Mobility Primitive

Basic primitive is \textit{move X to Y}
The object X is moved where Y is.
More formally: The object denoted by the expression X is move to the node where the object denoted by expression Y was!
If the move cannot be done, it is \textit{ignored}.
NOTHING is guaranteed – nothing may happen.
Strong Move: Fix

Basic primitive is $\text{fix } X \text{ at } Y$

The object $X$ is moved where $Y$ is & stays there.

More formally: The object denoted by the expression $X$ is move to the node where the object denoted by expression $Y$ was!

Either the move happens – or it fails.

Strong guarantees; potentially expensive
object Boss
process
  var w: Worker
  var n: Node
  n <- ...find usable node
  move self to n
  w <- Worker.create()
end process
end Boss

class Worker
process
  do work ...
end process
end Worker
Mobility Example
Stationary Boss

object Boss
  var w: Worker
  var n: Node
  n <- ...find usable node
  w <- Worker.create[ ]
  move w to n
  w.StartWork[ ]
end Boss

class Worker
  op StartWork
    slave <- object slave
    process
      work ... work
    end process
    end slave
  end StartWork
end Worker
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>locate X</td>
<td>returns (one of) the object X’s locations</td>
</tr>
<tr>
<td>move X to Y</td>
<td>move the object X to the node where Y is (or rather was)</td>
</tr>
<tr>
<td>fix X at Y</td>
<td>as move but disregard subsequent moves</td>
</tr>
<tr>
<td>refix X at Y</td>
<td>as fix but for fixed objects</td>
</tr>
<tr>
<td>unfix X</td>
<td>allow normal moves</td>
</tr>
</tbody>
</table>
Why two different moves?

- Fast efficient – mobility hint
- Slow but sure for when location is part of the semantics of the application.
Performance

- Local calls are typically 1,000 – 10,000 times faster than remote calls
- Co-locate frequently communicating objects
Call-by-move

var B: some object

object X

... operation F[arg:T]

loop

X.F[move B]

... arg.g[...]

exit after many loops

end loop

end X
Call-by-visit

var B: some object
object X

... operation F[arg:T]

loop

arg.g[...]

exit after

many loops

end loop

end X
How Many Calls of B?

Given a normal PC environment, say 2 GHz CPU, 100 Mbit/s Ethernet, how many calls of a small (say 100 bytes) argument B before breakeven?

- 1
- 10
- 100
- 1,000
- 10,000
- 100,000
- 1,000,000
Where is 17?

IF every object is on exactly one node, where is the integer object 17?

I hope it is not far away!

It doesn’t change—why not a copy everywhere?!?
Immutable Objects

- Immutable objects cannot change state
- Consider: The integer 17
- Immutable objects are omnipresent
- User-defined immutable objects: for example complex numbers
- Types must be immutable to allow static type checking
Return-by-move

When an operation creates a result object and knows it is for the caller’s use only, it can choose to return the parameter *by move*.

Return-by-move is not necessary – but increases efficiency – *why??*
object Killroy
    process
        var myNode <- locate self
        var up:
            array.of[Nodes]
            up <-
            myNode.getNodes[]
        foreach n in up
            move self to n
        end foreach
    end process
end Killroy

• Object moves itself to all available nodes
• On the original MicroVAX (1987) implementation: 20 moves/second!
• Note: the thread (called a process in Emerald) moves along
Conclusion

Emerald has

• concurrency with Hoare monitors
• fully integrated distribution facilities
• has full on-the-fly mobility
• a novel attachment language feature

Many novel implementation techniques (more talks to come!)
Problem:
move an object but its *internal* data structure does *not* move along!

Classic example:
A tree
class TreeClass
    var left, right: TreeClass
    var data: ...
end TreeClass
Attached Tree

class TreeClass
  attached var left, right: TreeClass
  var data: ...
end TreeClass
Attachment: can it be decided automatically?

Tree example

Mail message

TreeNode

To

From

left, right

Subject

Body
Attachment costs

Attachment has NO run-time cost!
Just a bit in the DESCRIPTOR for an object.
One bit for each variable.

Better: compiler sorts by attached bit – then merely two integers, e.g.,
  5 attached variables
  4 non-attached variables
Dynamic Attachment

\[
\text{var } X: \ldots \leftarrow \text{something} \\
\text{attached var } aX: \ldots
\]

\[
\ldots
\]

Join:
\[
aX \leftarrow X
\]

Leave:
\[
aX \leftarrow \text{NIL}
\]
Immutable Objects

- Immutable objects cannot change state
- Examples: The integer 17
- User-defined immutable objects: for example complex numbers
- Immutable objects are omnipresent
- Types must be immutable to allow static type checking
Types are Immutable Objects

Example: arrays

```javascript
var ai: Array.of[Integer]

ai <- Array.of[Integer].create[]

var aai:
    Array.of[Array.of[Integer]]
```
Let’s look at the implementation of Array

(Switch to code…)
Conclusion

Emerald is

- clean OO language
- fully integrated distribution facilities
- has full on-the-fly mobility
- a well-defined type system

Many novel implementation techniques (more talks to come!)
Web Site

Emerald:
http://www.emeraldprogramminglanguage.org/

Source code available on Sourceforge.

For REAL distribution, use Planetlab:
http://www.planet-lab.org