The Emerald Programming Language and its Development

The Emerald Programming Language and its Development

Andrew P. Black
Norman C. Hutchinson
Eric Jul
Henry M. Levy

What is Emerald?

- Object-based programming language
- Static types, parametric polymorphism
- Mobile objects (and mobile processes)
- Efficient implementation
- Designed and implemented 1984-1987

What is Emerald?

- Like Java, except that:
  + Objects are mobile
  + Objects can be persistent
  + Remote invocation has the same semantics as Local Invocation
  + Parameterized types (from the beginning)
  + No inheritance

2007

- Objects have won
  - Object-oriented languages are mainstream
  - Distributed objects are the dominant paradigm for internet computing
    - J2EE, .Net, CORBA, SOAP, ...
  - It wasn't always like this!

1984

- Objects were hot!
  - but people were skeptical about performance

1984

- Objects were hot!
  - but people were skeptical about performance
- Objects were not accepted even for non-distributed computing
  - Simula was “way out there”, C++ wasn’t yet available, Java wouldn’t be born for 15 years
  - Smalltalk was “hip,” but only if you had a Dorado

At the University of Washington

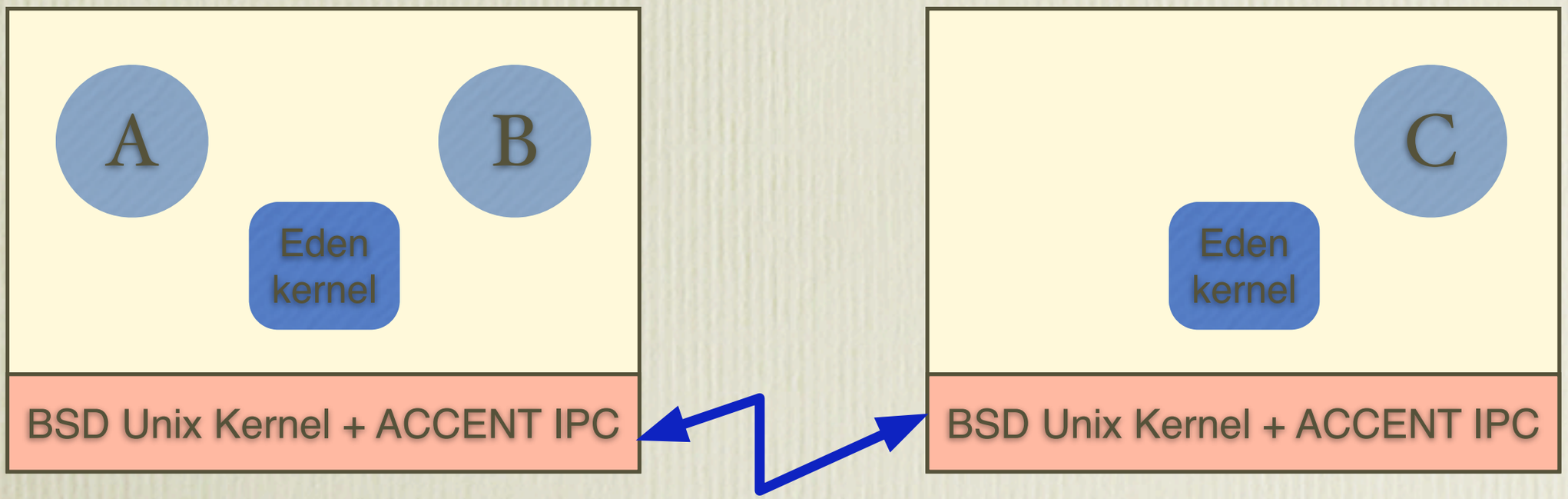
At the University of Washington

- We had just started running *Eden*: the first Object-oriented distributed operating system

At the University of Washington

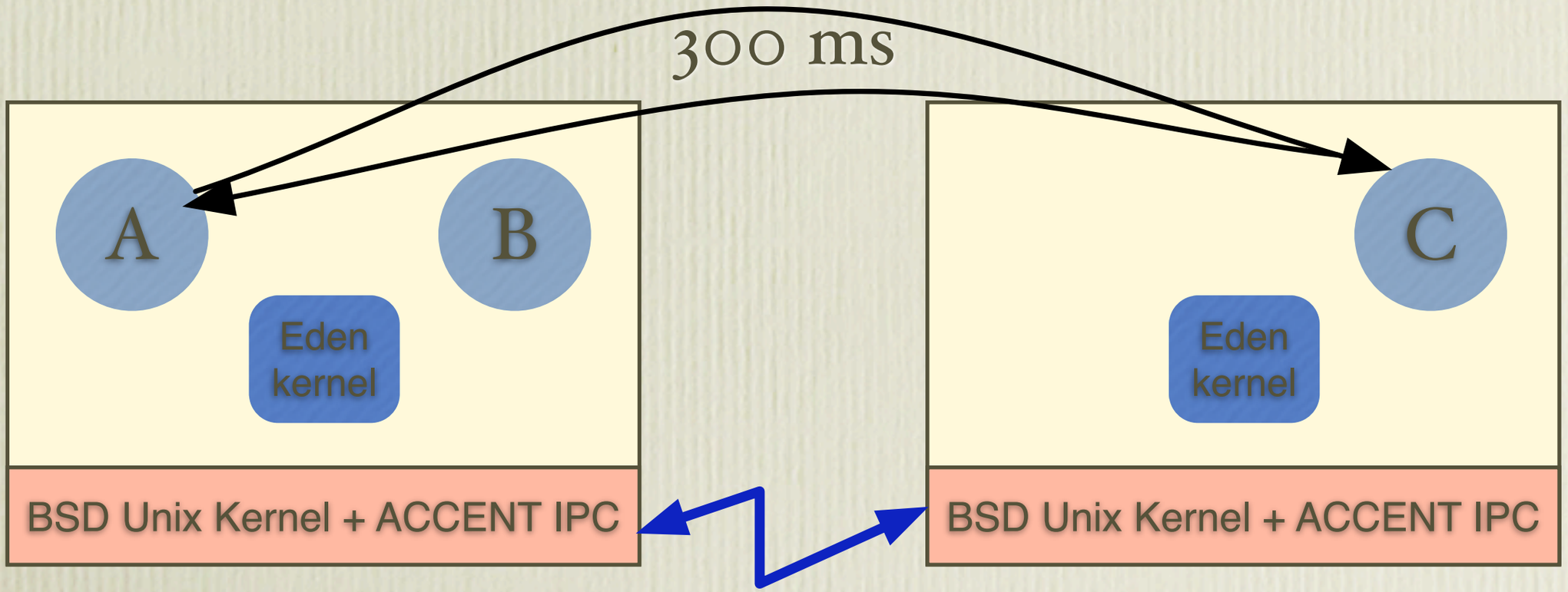
At the University of Washington

- We had just started running *Eden*: the first Object-oriented distributed operating system
  - it worked!
  - performance was unimpressive



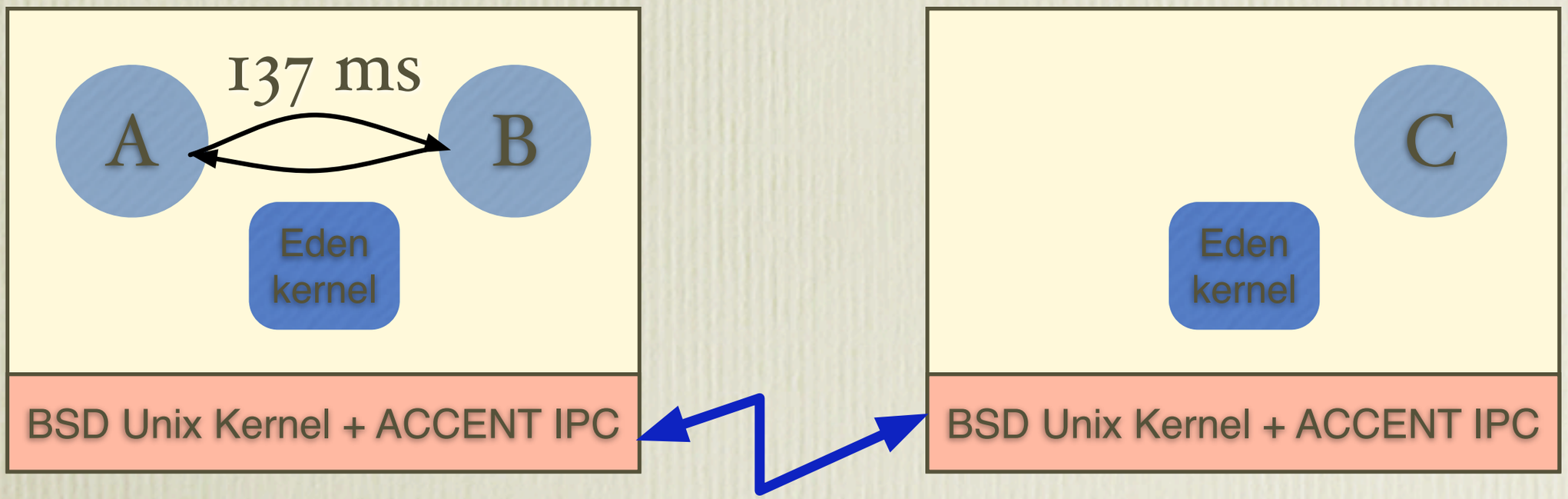
At the University of Washington

- We had just started running *Eden*: the first Object-oriented distributed operating system
  - it worked!
  - performance was unimpressive



At the University of Washington

- We had just started running *Eden*: the first Object-oriented distributed operating system
  - it worked!
  - performance was unimpressive



The People

Andrew Black

Norm Hutchinson

Eric Jul

Henry (Hank) Levy

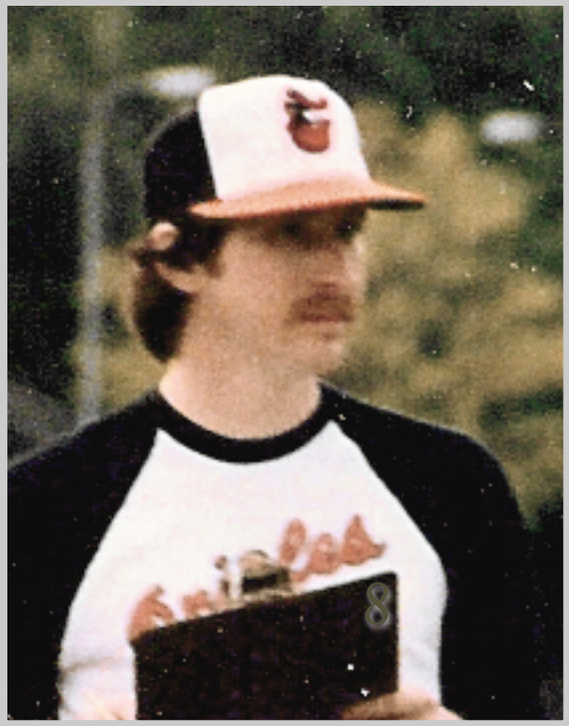
The People

Andrew Black

Norm Hutchinson

Eric Jul

Henry (Hank) Levy



Fall 1983: The Coffee Shop

- Hank asked Eric and Norm for a cup of coffee
- He also asked:
  - What's wrong with Eden?
  - What would you do differently?
  - Why don't you do it?

Getting to Oz

Hank Levy, Norm Hutchinson, Eric Jul
April 27, 1984

For the past several months, the three of us have been discussing the possibility of building a new object-based system. Carl Binding frequently attended our meetings, and occasionally Guy Almes, Andrew Black, or Alan Borning sat in also. This system, called Oz, is an outgrowth of our experience with the Eden system. In this memo we try to capture the background that led to our current thinking on Oz. This memo is not a specification but a brief summary of the issues discussed in our meetings.

Starting in winter term, 1984, we began to examine some of the strengths and weaknesses of Eden. Several of the senior Eden graduate students had been experimenting with improving Eden's performance. Although they were able to significantly decrease Eden invocation costs, performance was still far from acceptable. Certainly some of the performance problem was due to Eden's current invocation semantics, some was due to implementation of invocation, and some was due to the fact that Eden is built on top of the Unix system.

In addition to performance problems, Eden suffered from the lack of a clean interface. That is, the Eden programmer needs to know about Eden, about EPL (Eden Programming Language -- an preprocessor-implemented extension to Concurrent Euclid), and about Unix to build Eden applications. Also, there was at that time no Eden user interface. Users built Eden applications with the standard Unix command system.

This combination of issues led us to consider building a better integrated system from scratch. Performance was at the top of our priorities. To date, object systems have a reputation of being slow and we don't think this is inherently due to their support for objects. We want to build a distributed object-based system with performance comparable to good message passing systems. To do this, we would have to build a low-level, bare-machine kernel and compiler support. In addition, we would like our system to have an object-based user interface as well as an object-based programming interface. Thus, users should be able to create and manipulate objects from a display.

Our first discussions concentrated on low-level kernel issues. In the Eden system, there are two types of processes and two levels of scheduling. Applications written in EPL contain multiple lightweight processes that coexist within a single Unix address space. These processes are scheduled by a kernel running within that address space. This kernel gains control through special checks compiled into the application. At the next level, multiple address spaces (Unix processes) are scheduled by the Unix system.

Our first decision was that our system would provide both lightweight processes that share might contain processes, be shared, and move from node to node. Small objects are used within other objects to implement simple abstractions. They are not shared or moved. They

paces would add protection, . Within a single address r within Oz would schedule fs to schedule lightweight ould remove the overhead

hing between them involves re important, we wanted to or less. This requires that ler support.

nology to increase system ckage invocation parameters at the receiving end. The cking and produce efficient e the use of objects invoked hat use. For example, if the ct B, it could produce inline

nostly from the kernel level. invocation, and scheduling. sues were not at the heart of the fundamental issue. Our d Toto) and the interesting ent much time discussing the object based in the sense of

plications on Eden. In fact, e ease with which distributed nt objects. However, the eir use to certain classes of PL and use its type facilities. gs that are essentially objects. e large but not in the small. most objects are local to the pay the cost?

uage that supports both large lboxes, etc.) and small objects *gle* semantics. Large objects

the same way, however the sage. In this way, we use ith the needs of the object. ine code, through procedure

al area network. Since we're concept of object location s should be easily expressible programming that we wish to rocess migration but have no he unit of movement and an includes statements that (1) particular site, and (3) fix an

nt invocation. Rather, we say not. That is, location is a ocate and move other objects. on system usage information. cal and remote objects.

invocation parameter passing. the system and by the options process or not). In general, mutable objects may be passed pass a reference to the integer is *call by move*, in which the bject should be moved to the e principal facility for object

must allow the programmer to hat arise due to the distributed up and down, links break, etc. ailures. Program bugs, on the by zero, are not handled in Oz. these conditions.

language that supports:
are supported by a single

thin an address space and

remely simple. We are ble shortly.

performance problem was due to Eden's current invocation semantics, some was due to implementation of invocation, and some was due to the fact that Eden is built on top of the Unix system.

In addition to performance problems, Eden suffered from the lack of a clean interface. That is, the Eden programmer needs to know about Eden, about EPL (Eden Programming Language -- an preprocessor-implemented extension to Concurrent Euclid), and about Unix to build Eden applications. Also, there was at that time no Eden user interface. Users built Eden applications with the standard Unix command system.

This combination of issues led us to consider building a better integrated system from scratch. Performance was at the top of our priorities. To date, object systems have a reputation of being slow and we don't think this is inherently due to their support for objects. We want to build a distributed object-based system with performance comparable to good message passing systems. To do this, we would have to build a low-level, bare-machine kernel and compiler support. In addition, we would like our system to have an object-based user interface as well as an object-based programming interface. Thus, users should be able to create and manipulate objects from a display.

Our first discussions concentrated on low-level kernel issues. In the Eden system, there are two types of processes and two levels of scheduling. Applications written in EPL contain multiple lightweight processes that coexist within a single Unix address space. These processes are scheduled by a kernel running within that address space. This kernel gains control through special checks compiled into the application. At the next level, multiple address spaces (Unix processes) are scheduled by the Unix system.

Our first decision was that our system would provide both lightweight processes that share

Thus, a major goal of Oz is exploitation of compiler technology to increase system performance. In Eden, EPL generates calls to routines that package invocation parameters into a standard interchange format that must be type-checked at the receiving end. The compiler for Oz would instead perform compile-time type checking and produce efficient code for invocation. In additoin, the Oz compiler would examine the use of objects invoked from a particular object and would generate code according to that use. For example, if the compiler discovered that object A's use was strictly local to object B, it could produce inline code within B for the manipulation of A's representation.

It is interesting that up to this point our approach had been mostly from the kernel level. We had discussed address spaces, sharing, local and remote invocation, and scheduling. However, we began to realize more and more that the kernel issues were not at the heart of the project. Eventually, we all agreed that language design was the fundamental issue. Our kernel is just a run-time system for the Oz language (called Toto) and the interesting questions were the semantics supported by Toto. We also had spent much time discussing the ways in which Eden is and is not object based. That is, Eden is object based in the sense of Hydra but not in the sense of CLU or Smalltalk.

One thinks in terms of objects when designing distributed applications on Eden. In fact, we all agreed that Eden had been a success in demonstrating the ease with which distributed applications could be programmed using location-independent objects. However, the implementation of Eden objects as large entities restricts their use to certain classes of resources. For smaller data abstractions, one must drop into EPL and use its type facilities. These type facilities require different abstractions to define things that are essentially objects. In other words, Eden supports object-based programming in the large but not in the small. This is particularly troublesome if one believes, as we do, that most objects are local to the application and will never be distributed. Why then should they pay the cost?

Goals for Emerald

1. To implement a high-performance *distributed* object system
2. To demonstrate high-performance objects
3. To simplify distributed programming
4. To exploit information hiding
5. To accommodate failure
6. To support object location explicitly
7. To minimize the size of the language, implementing everything possible as objects

How We Worked

- Self-organizing team, with personal responsibilities

  It will be Norm's job to see that local invocations execute as fast as local procedure calls, and Eric's to make sure that remote invocations go faster than Eden's

- Incremental Implementation
  - First compiler was for mini-language, produced macro-calls, run thru m4 then assembled and linked with "kernel"
  - added features, switched to VAX assembly with JSR to kernel procedures

Compiler–Runtime integration

- All local objects and runtime kernel shared an address space
  - shared knowledge of all basic data structures
  - compiled code did *not* call kernel procedure to interpret an object structure: it reached in and twiddled the bits directly.
- Compiler and runtime must agree on the data layout:


NO SMOKING
Generic OD
tag
dataPtr
location
flags
CTOD
tag
CTOID
CTPtr
location
flags
SSOD
tag
SSOID
SSPtr
location
flags
processODP
Process OD
tag
processOID
location
flags
ownerOID
OD's
tag
Object OID
data Ptr
location
flags
process ODP
invokelist
ATOD
tag
ATOID
flags
Act Record
local vars
regs
invokelist
params
results
Stack Segment
tag
SSOID
ProcessOID
size
first
ready Q link
Register Save
Stack of AR's
Abcon
tag
ATOID
CTOID
CTPtr
AT Ptr
S Templates
bytesize
array of STemplateEntry
Template Entry
code offset
data offset
code offset
length
Date:
Feb 5th, 1986
Oz Project

- Put compiler writer and kernel implementor in the same office
- Define all the basic data structures in a single file
  - in VAX assembler for the compiler
  - in C for the runtime kernel
- Single file ingeniously written so it could be treated as C with assembly language comments, or as assembly language with C comments.

Early eXtreme Programming

- We were implementing many of the practices that would later become part of XP:
  - team room
  - informative workspace
  - incremental development
  - constant testing
  - short (daily) release cycles
  - leave optimization until last
  - acceptance tests run often (daily)
  - defer building functionality until it's needed

Technical Innovations

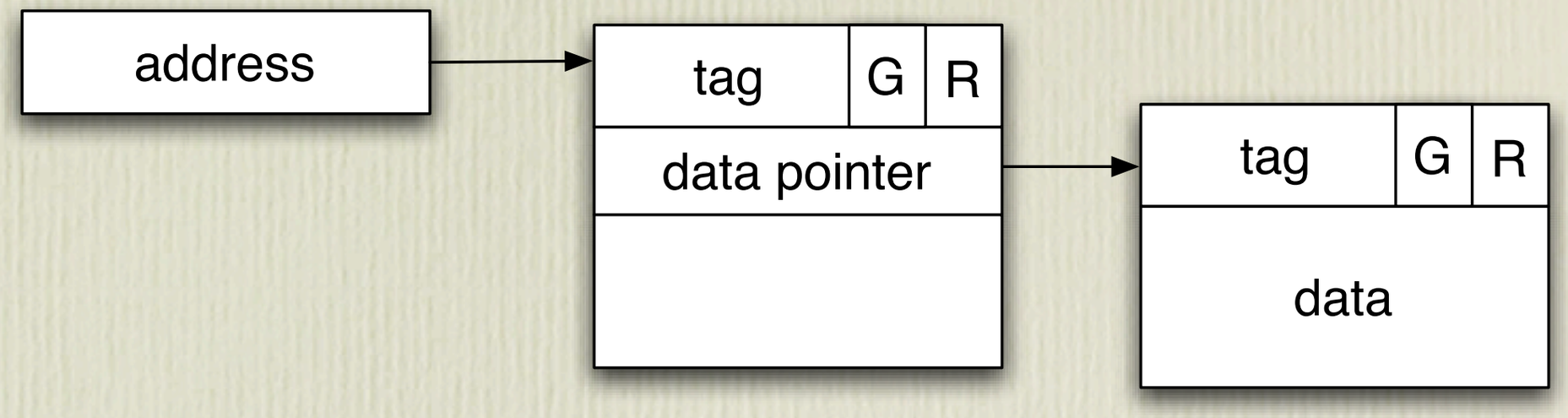
- Object constructors
- Conformity-based typing
- Contravariance
- AbCons
- Separating *locatics* from *semantics*
- Call-by-move & -visit
- Overloading by arity
- F-bounded polymorphism
- type:type
- attachment
- Separation of type and implementation
- Any and None
- location primitives
- unavailability handlers

Emerald's Object Model

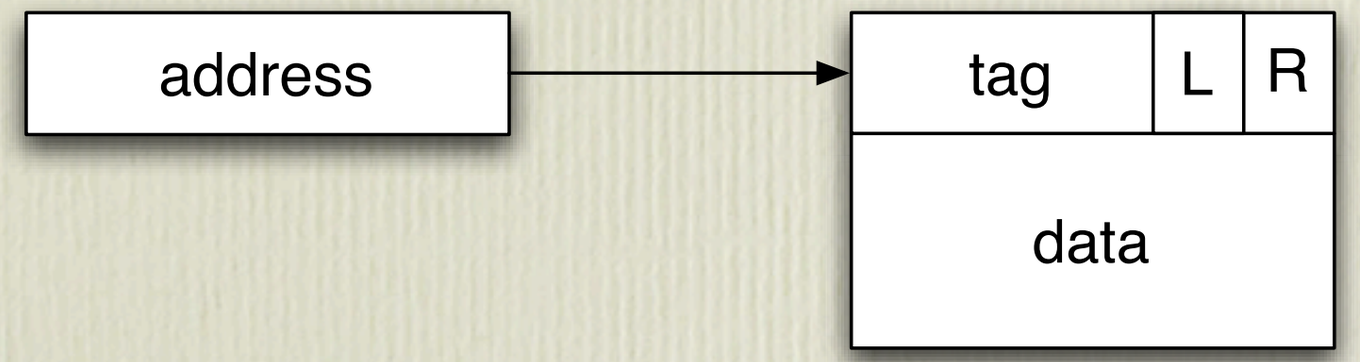
- Strong encapsulation
  - one type, many implementations
  - clients can't tell the difference
  - both end-user programers and compiler-writers benefit
- Immutable objects as well as Mutable objects
- Object constructors replace classes
- Objects are not fragmented
- The unit of mobility
  - an object and all of the objects **attached** to it

One model, three implementations

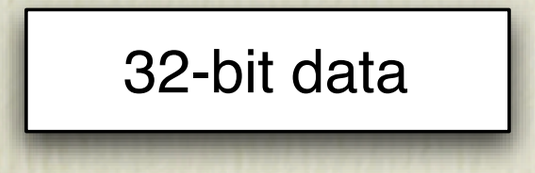
global



local

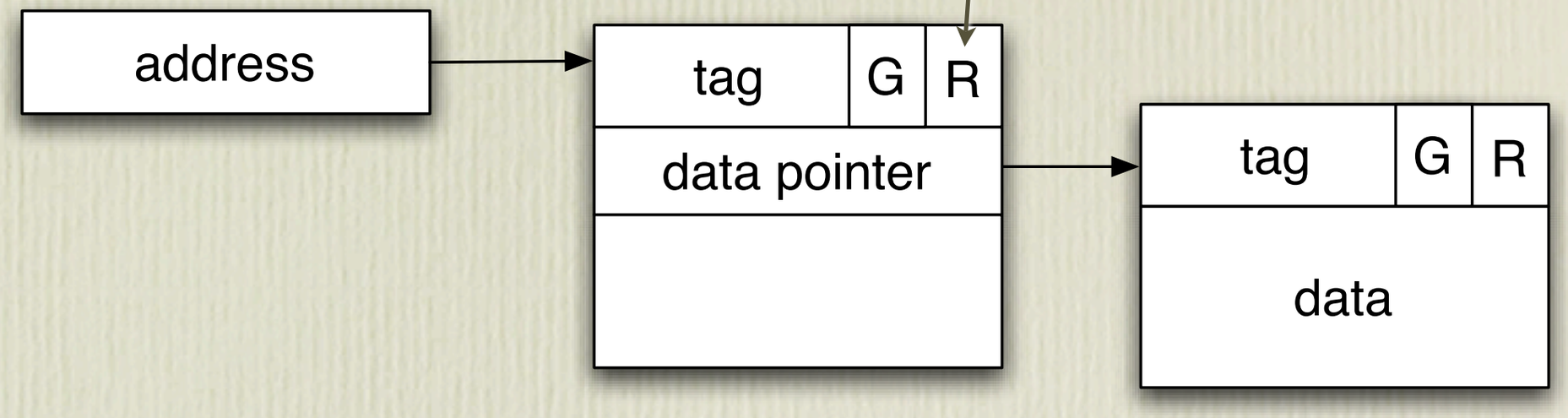


direct

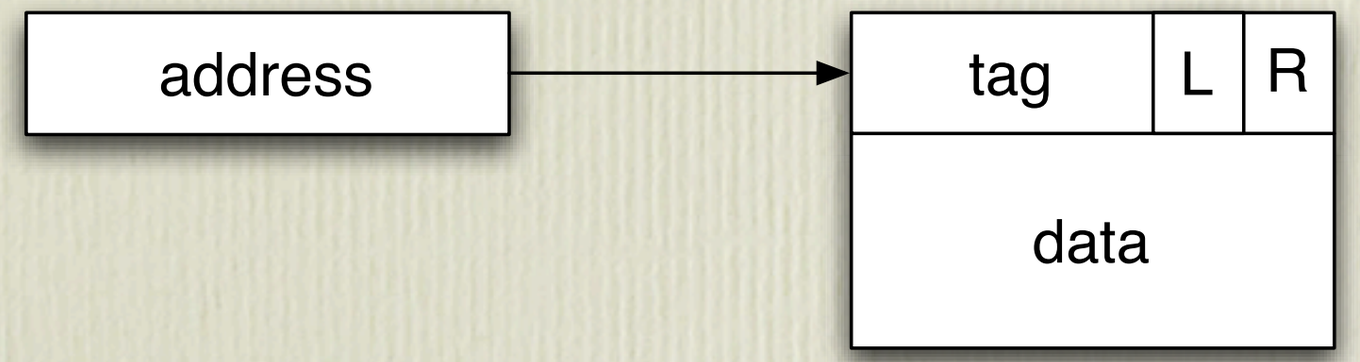


resident/non-resident

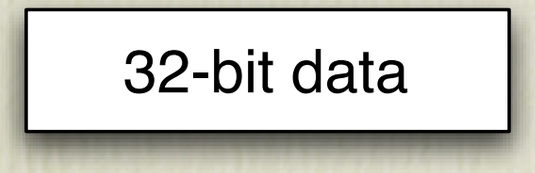
global



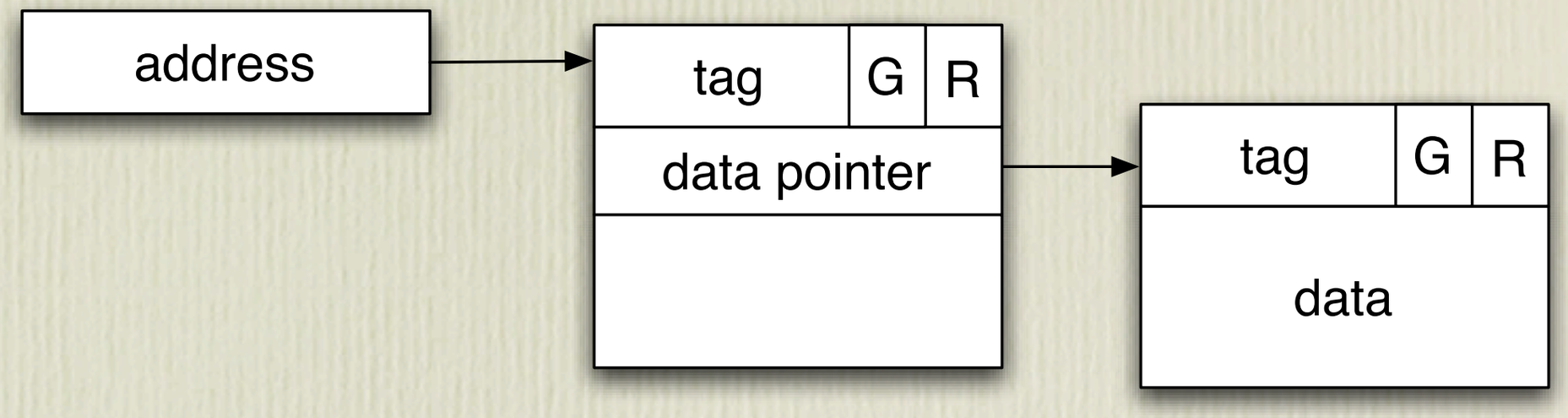
local



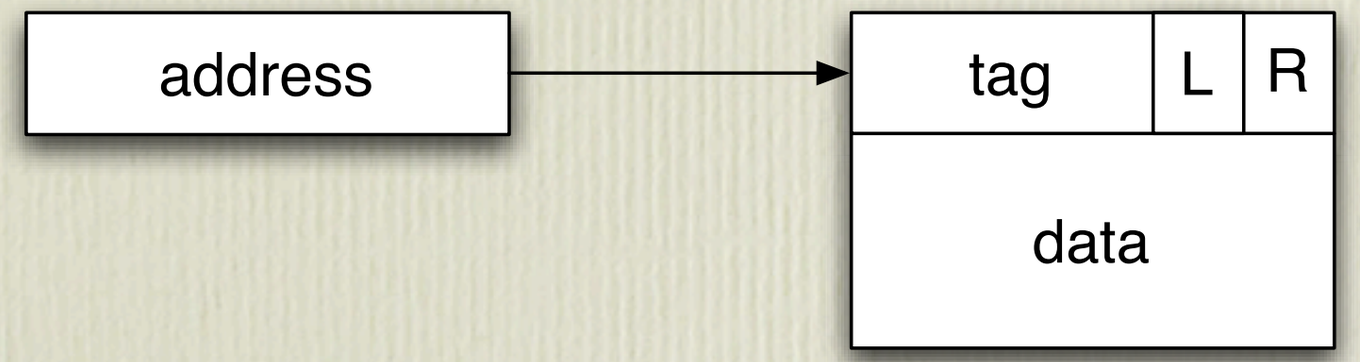
direct



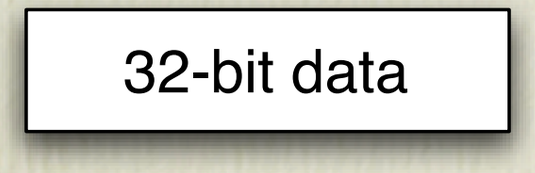
global



local

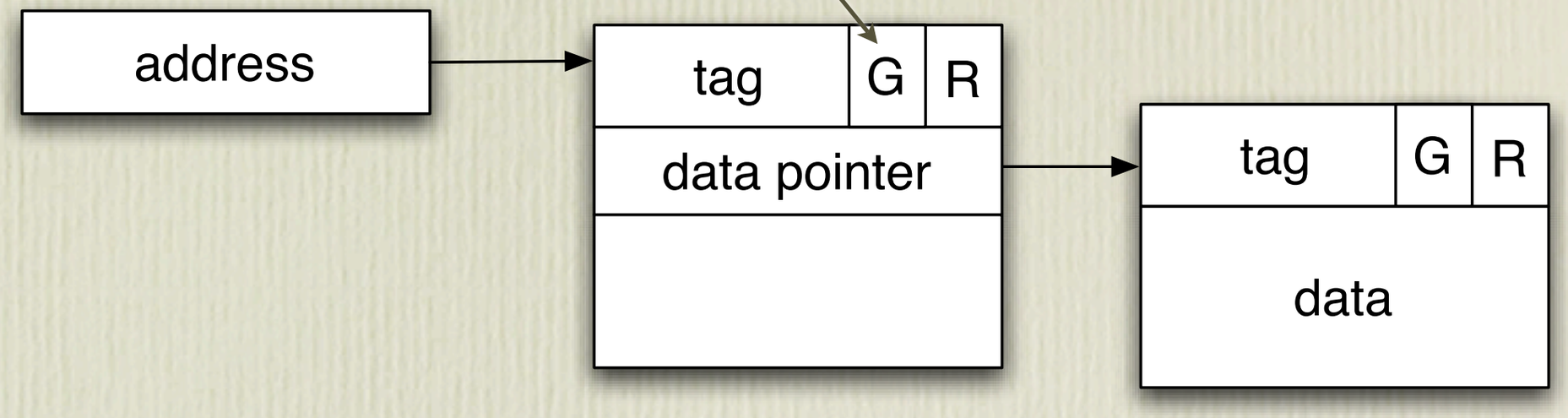


direct

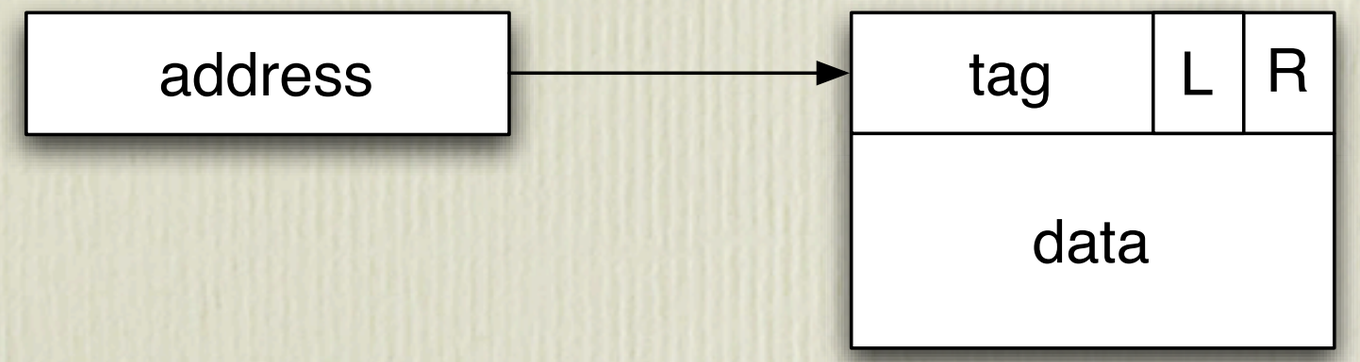


local/global

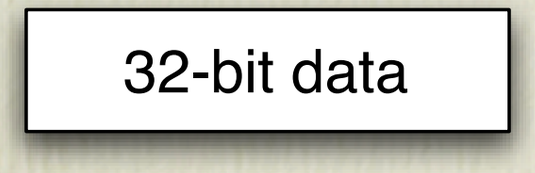
global



local



direct



Object Constructors

- a statement that, when executed, creates an object
  - "magic", like **new**
- Syntax:

object *name*
    private state declarations
    operation declarations
end *name*

Example from Emerald Compiler

```
const anIntegerNode ←
    object IntegerLiteral
        export getValue, setValue, generate
        monitor
            var value : Integer ← val

            operation getValue → [v : Integer]
                v ← value
            end getValue

            operation setValue[v : Integer]
                value ← v
            end setValue

            operation generate[stream : OutStream]
                stream.printf["LDI %d\n", value]
            end generate
        end monitor
    end IntegerLiteral
```

Example from Emerald Compiler

```
const anIntegerNode ←
    object IntegerLiteral
        export getValue, setValue, generate
        monitor
            var value : Integer ← val

            operation getValue → [v : Integer]
                v ← value
            end getValue

            operation setValue[v : Integer]
                value ← v
            end setValue

            operation generate[stream : OutStream]
                stream.printf["LDI %d\n", value]
            end generate
        end monitor
    end IntegerLiteral
```

Example from Emerald Compiler

```
const anIntegerNode ←
    object IntegerLiteral
        export getValue, setValue, generate
        monitor
            var value : Integer ← val

            operation getValue → [v : Integer]
                v ← value
            end getValue

            operation setValue[v : Integer]
                value ← v
            end setValue

            operation generate[stream : OutStream]
                stream.printf["LDI %d\n", value]
            end generate
        end monitor
    end IntegerLiteral
```

Parameterized Constructor

```
const IntegerNodeCreator ←
    immutable object INC
        export new

        const IntegerNodeType ←
            type INType
                function getValue → [Integer]
                operation setValue[Integer]
                operation generate[OutStream]
            end INType

        operation new[val : Integer] → [aNode : IntegerNodeType]
            aNode ←
                object IntegerLiteral
                    object constructor from previous slide
                end IntegerLiteral
    end new
end INC
```

Parameterized Constructor

```
const IntegerNodeCreator ←
    immutable object INC
        export new

        const IntegerNodeType ←
            type INType
                function getValue → [Integer]
                operation setValue[Integer]
                operation generate[OutStream]
            end INType

        operation new[val : Integer] → [aNode : IntegerNodeType]
            aNode ←
                object IntegerLiteral
                    object constructor from previous slide
                end IntegerLiteral
    end new
end INC
```

Parameterized Constructor

```
const IntegerNodeCreator ←
    immutable object INC
        export new

        const IntegerNodeType ←
            type INType
                function getValue → [Integer]
                operation setValue[Integer]
                operation generate[OutStream]
            end INType

        operation new[val : Integer] → [aNode : IntegerNodeType]
            aNode ←
                object IntegerLiteral
                    object constructor from previous slide
                end IntegerLiteral
    end new
end INC
```

Concurrency

- Processes and Monitors
  - inspired by Per Brinch Hansen's Concurrent Pascal
  - and Holt's Concurrent Euclid, used in Eden.
- All assignments to variables must be within a monitor!
- Objects could have two parts:
  - Non-monitored: concurrency ok, assignment not allowed
  - Monitored part: sequential access, updates allowed

Emerald Processes (= threads)

- Every object (in principle) has a process:

```
object doSomething
   process
   ... do something …
   end process
end doSomething
```

- Processes are not first class citizens

Distribution

- "Sea" of objects — divided into disjoint sets
- An object is on one and only one Node at a time
- A Node can crash and become unavailable — this is a normal and expected event.
- Each node is represented by a Node object
- `Locate X` returns the node where `X` *was*
- Immutable objects are omnipresent!
  - Immutability is a distributed programmer's secret weapon.

Mobility

Mobility for two reasons:

- *performance*: co-location for performance. One example is *call-by-move* where parameter moves to callee's site.
- *availability*: e.g., moving multiple copies of a replication manager to different nodes.

Reflected in two different kinds of mobility:

move X to Y	just performance hint, can't fail
fix X at Y	must commit or fail

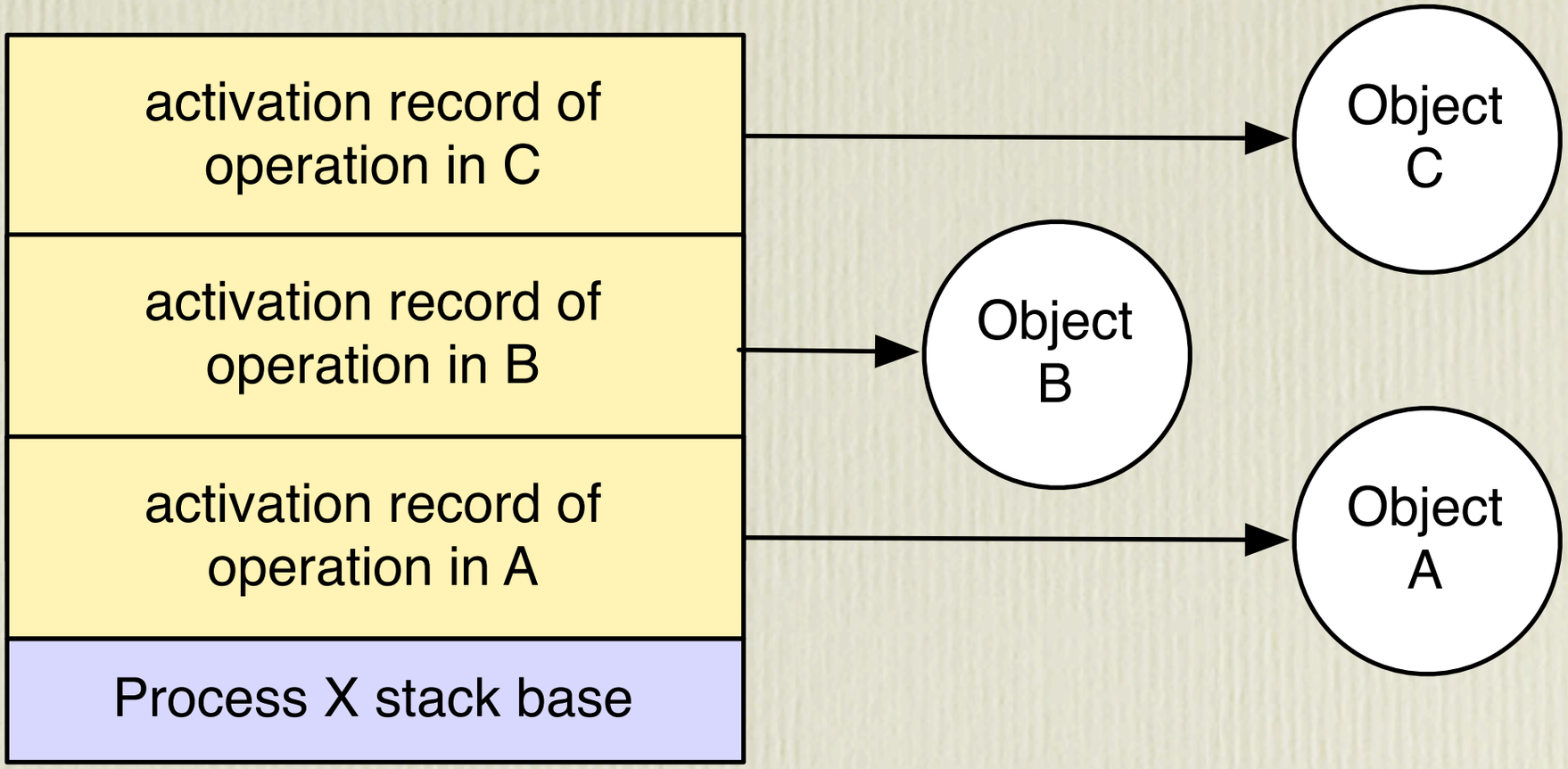
On-the-fly process mobility

```
object Kilroy
  process
    const origin ← locate self
    const up ← origin.getNodes
    for n in up
      move self to n.getTheNode
    end for
  end process
end Kilroy
```

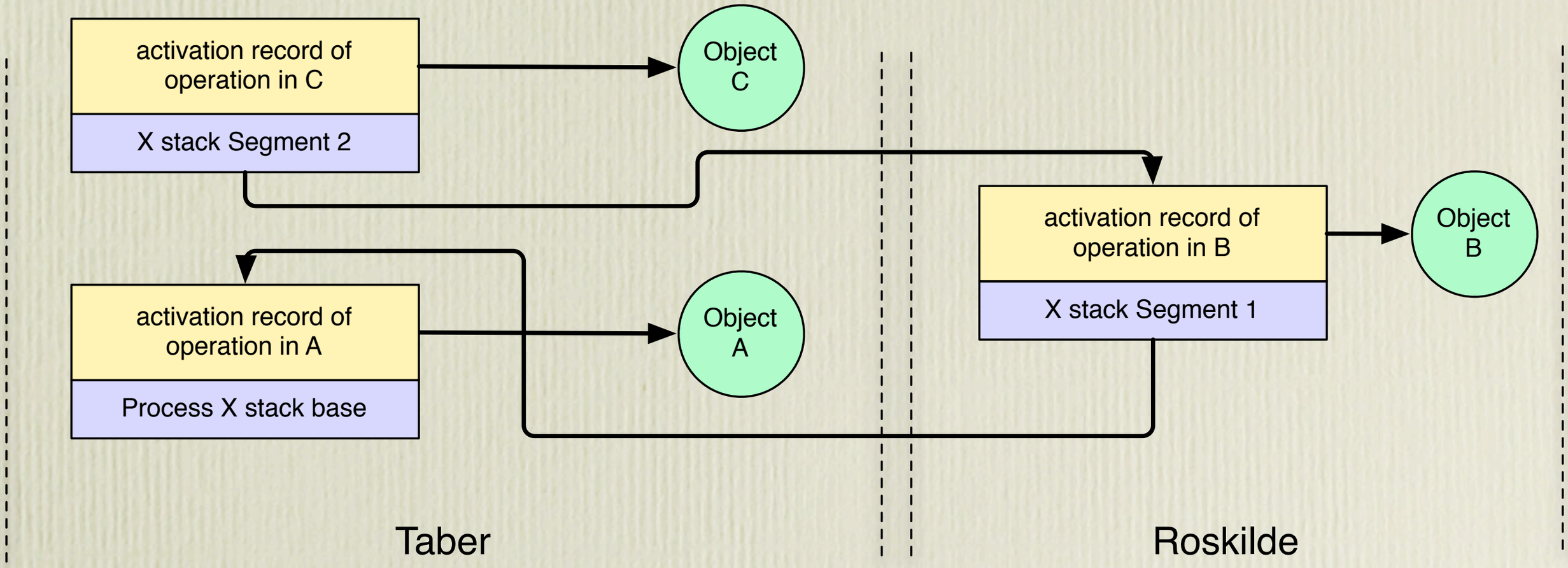
- Object moves itself to all available nodes
- On the original MicroVAX implementation: 25 moves/second, 40 ms/move while a remote invoke was 28ms
- The process moves along with the object

Implementing Mobility

- Possibly-mobile objects have *guids*
  - *guids* eagerly replaced by direct local addresses
- Stack segments are treated like objects
- Hence, objects and processes (activation records) can be moved from machine to machine
  - all the cleverness was in making it fast


activation record of operation in C
activation record of operation in B
activation record of operation in A
Process X stack base
Object C
Object B
Object A

Taber


activation record of operation in C
X stack Segment 2
Object C
activation record of operation in B
X stack Segment 1
Object B
activation record of operation in A
Process X stack base
Object A
Taber
Roskilde

Emerald's Type System

- Types were an assumption, not a conclusion:
  - our experience was with Algol 60, Simula, Concurrent Euclid, Pascal, ...
  - we "knew" that type declarations and compile-time checking would improve performance
- Challenges:
  - permit the definition of typed collections *in* the language
    - Array.of [Integer] *vs.* Set.of [Mailer]
  - work in an "open world"

"Wirthian" Type Systems

- Classified objects by implementation, not by behavior
  - types were concrete data structures:
    one type ⇒ one implementation
- Assumed a closed world
  - new kinds of objects could not arrive over the network at runtime
- Programs that could not be proved type-correct at compile time could never be run

Emerald's Type System

- Types were sets of operation signatures
  - we called them abstract types: the implementations of the operations were irrelevant to the type
  - inspired by abstract Edentypes and Smalltalk protocols
- Assumed an "open world" — accept objects compiled on the other side of the world
  - If we can't type-check it now, we'll check it when it arrives
  - Regardless, the *same* type system and the *same* checking rules apply

Type *equality* was the wrong question

- Does a given object support enough operations to be used in a particular context (characterized by the operations that could be invoked on it)?

  $\Longrightarrow$ Does the object possess a *superset* of the operations that could be invoked in the current context?

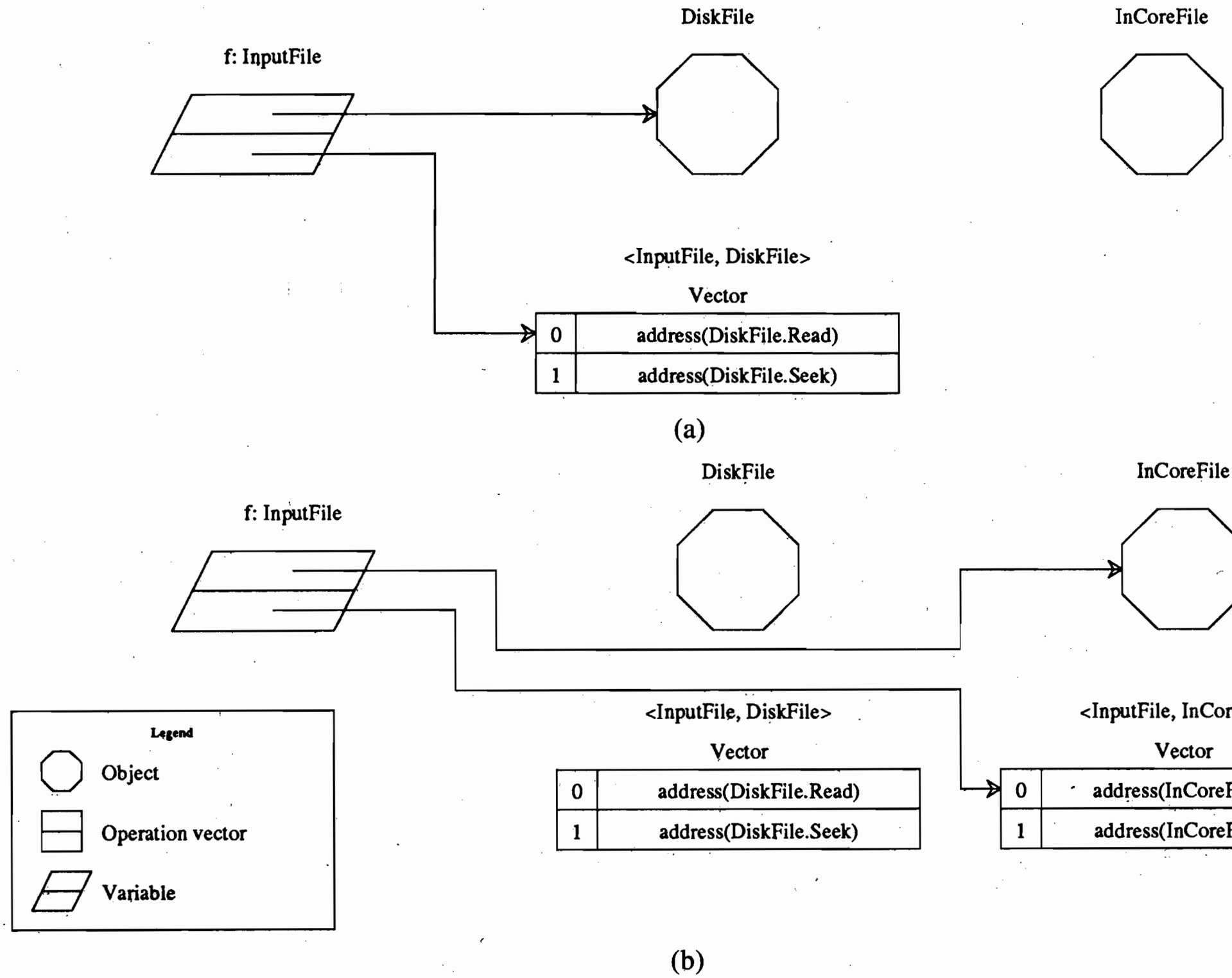
- Hence: conformity-based typing, now more usually called "subtyping"

Did types improve efficiency?

- We forbade primitive types from being re-implemented
  - *Boolean, Character, Integer, Real, String* and *Vector* were built-in and could not be re-implemented
  - So, the compiler could deduce the implementation from the type
  - This meant that operations on these type could be inlined
- This was a hack, but paid-off
  - We broke our own rule that type had nothing to do with implementation

Non-primitive types

- For non-primitives, the abstract type of an expression told us *nothing at all* about its implementation!
  - How to do efficient operation lookup?
  - Couldn't use table-lookup as in Simula
- Dataflow analysis could help eliminate operation invocation in many cases
  - context-dependent, not type-dependent
- For the rest, we invented AbCons: vectors that map operations on the type to methods



from Black et al., Distribution and Abstract Types in Emerald. IEEE TSE 1987

Parametric Polymorphism

- Open-world assumption meant that we needed a run-time representation for types
  - Obvious choice: make types objects
- Now types can parameterize operations
- Type constructors are just operations
  - Pair.of [Integer] is a perfectly normal operation invocation
  - We can evaluate it at compile time, because Pair is immutable, and of is a function

What made Emerald fast?

- Single address space
- Performance-oriented networking design
- Choice between eager and lazy evaluation
  - do as much at compile-time as possible, and defer the rest: you may never have to do it at all!
  - once you have to do it, do as much as you can as early as you can

Whatever happened to Emerald?

- Widely influential, even though not widely adopted as a language.
- Examples:
  - ANSI Smalltalk standard uses a lattice of conforming protocols
  - Mobile objects appear in several OSs: SOS, Guide, Network Objects
  - ANSA network architecture ➡OSF RPC ➡ ISO 10746 ODP

Later Influences

- Java RPC and Jini
    - “more Emerald-like than we realized at the time” — Jim Waldo
- Walsh’s Taxy mobility system
- Gaggles
- Multicast invocations

Retrospective

- Location independence
  - does not absolve the programmer from placing objects correctly.
- Mobile and Persistent objects
  - have not caught on. Why? 25 years too early? Did we need proof-carrying code?
- Static typing
  - May have been a mistake
  - Small benefit, big headaches

Not everyone was impressed...

[redacted]

#90 "Fine-Grained Mobility in the Emerald System"

Referee's Report

This is a straightforward implementation of a simple idea. It is hard to see what is unique about this operating system.