The Development of the Emerald Programming Language

Andrew P. Black
Portland State University
black@cs.pdx.edu

Norman C. Hutchinson
University of British Columbia
norm@cs.ubc.ca

Eric Jul
University of Copenhagen
eric@diku.dk

Henry M. Levy
University of Washington
levy@cs.washington.edu

Abstract

Emerald is an object-based programming language and system designed and implemented in the Department of Computer Science at the University of Washington in the early and mid-1980s. The goal of Emerald was to simplify the construction of distributed applications. This goal was reflected at every level of the system: its object structure, the programming language design, the compiler implementation, and the run-time support.

This paper describes the origins of the Emerald group, the forces that formed the language, the influences that Emerald has had on subsequent distributed systems and programming languages, and some of Emerald's more interesting technical innovations.

Categories and Subject Descriptors D.3.0 [*Programming Languages*]: General; D.3.2 [*Language Classifications*]: Object-oriented languages; D.3.3 [*Language Constructs and Features*]: Abstract data types, Classes and objects, Inheritance, Polymorphism

General Terms abstract types, distributed programming, object mobility, object-oriented programming, polymorphism, remote object invocation, remote procedure call.

Keywords call-by-move, Eden, Emerald, mobility, type conformity, Washington

1. Introduction

Emerald was one of the first languages and systems to support distribution explicitly. More importantly, it was the first language to propose and implement the notion of *object mobility* in a networked environment: Emerald objects could move around the network from node to node as a result of programming language commands, while they continued to execute. Object mobility was supported by location-independent object addressing, which made the location of the target of an object invocation semantically irrelevant to other objects, although facilities were provided for placing objects on particular machines when required. At a high level, Emerald invocation could be thought of as an early implementation of remote procedure call (RPC) [14], but with a much more flexible and dynamic binding system that allowed an object to move from one node to another between (and during) invocations of methods. Furthermore, as seen from a programmer's point of view, Emerald removed the "remote" from "remote procedure call": the programmer did *not* have to write any additional code to invoke a remote object compared to a local object. Instead, all binding, marshaling of parameters, thread control, and other tedious work was the responsibility of the implementation, i.e., the compiler and the run-time system.

In addition, Emerald sought to solve a crucial problem with distributed object systems at the time: terrible performance. Smalltalk had pioneered an extremely flexible form of object-oriented programming, but at the same time had sacrificed performance. Our stated goal was local performance (within a node) competitive with standard programming languages (such as C), and distributed performance competitive with RPC systems. This goal was achieved by our implementation.

1.1 Ancient History

Emerald forms a branch in a distributed systems research tree that began with the Eden project [3] at the University of Washington in 1979. Setting the context for Emerald requires some understanding of Eden and also of technology at that time. In 1980, at the start of the Eden project, local area networks existed only in research labs. Although early Eth-

ernet [76] systems had been in use at Xerox PARC for some time, the industrial Ethernet standard had only recently been completed and the first products were still in development. To the extent that network applications existed at all, outside of PARC, they were fairly primitive point-to-point applications such as FTP and email. UNIX did not yet support a socket abstraction, and programming networked applications required explicit message passing, which was difficult and error-prone.

Eden was a research project proposed by a group of faculty at the University of Washington (Guy Almes, Mike Fischer, Helmut Golde, Ed Lazowska, and Jerre Noe). The project received one of the first grants from what seemed even at the time to be a very far-sighted program at the National Science Foundation called Coordinated Experimental Research (CER), a program that sought to build significant expertise and infrastructure at a number of computer science departments around the United States. The stated goal of Eden was to support "integrated distributed computing." While nobody had distributed systems at the time, it was clear that someday in the future such systems would be commonplace, and that programming them using existing paradigms would be extremely difficult. The key insight of the Eden team was to use objects — a computational model and technology that was still controversial in both the operating system and the programming language worlds at that time.

Eden was itself a descendant of Hydra [108], the father of all object-based operating systems, developed by Bill Wulf and his team at Carnegie-Mellon in the early 1970s. Guy Almes, a junior faculty member at UW, had written his thesis on Hydra at CMU [4], and brought those ideas to Washington. The idea behind an object-based distributed system was that every resource in the network — a file, a mail message, a printer, a compiler, a disk — would be an object. At the time we looked on an object as nothing more than some data bound together with program code that operated on that data to achieve a specific goal, such as *printing* a file object, or *sending* a mail message object, or *running* a compiler object on a file object. The code was partitioned into what are now referred to as *methods* for printing, sending, running, etc., but at the time we just thought of the code as a bunch of procedures that implemented operations on the object's data.

The great thing about objects was that client programs could use an object simply by invoking an operation in that object and supplying appropriate parameters; they didn't have to concern themselves with how the object was implemented. Thus, objects were a physical realization of Parnas' principle of information hiding [85]. The key insight behind distributed object-based computing was that the same principle applied to the *location* of the object: a client could use an object without worrying about where that object was actually located in the network, and two conceptually similar objects (for example, two files) that were located in disparate places might have completely different implementations. The Eden system would take care of finding the object, managing the remote communication, and invoking the right code, all invisibly to the programmer.

Several other research projects also explored the distributed object notion, notably Argus at MIT [71] and Clouds at Georgia Tech [2]. Argus, as a language design, was essentially complete before the Emerald project started. Thus, Argus was contemporary with Eden rather than with Emerald, and indeed shared with Eden the idea that there were two kinds of objects — "large" objects that were remotely accessible (and, in the case of Argus, transactional), and small local objects (essentially, CLU objects). Some of the Clouds ideas moved to Apollo Computer with the principals, and appeared in the Apollo Domain system; there were several visits between Apollo and Eden personnel.

This distributed object model is now the dominant paradigm for Internet programming, whether it is Java/J2EE, Microsoft .NET, CORBA, SOAP, or whatever. We take it for granted. So it is hard to convey how controversial the distributed object idea was in the early 1980s. People thought that distributed objects would never work, would be way too slow, and were just dumb. Objects had not yet been accepted even in the non-distributed world: Simula was not mainstream, C++ would not be invented for some years [100], and Java wouldn't appear for a decade and a half. Smalltalk had just been released by PARC and was gathering a following amongst computing "hippies," but unless one had a Dorado, it was not fast enough for "real" applications. Alan Borning, our colleague at Washington, had distributed copies of the influential 1981 *Smalltalk* issue of *Byte* magazine, but in our opinion, the focus of Smalltalk was on flexibility to the detriment of performance. Smalltalk, although an impressive achievement, contributed to the view that poor performance was inherent in object-oriented languages.

Emerald was a response to what we had learned from our early experience with Eden and from other distributed object systems of that time. In fact, it was a follow-on to Eden before Eden itself was finished. Before describing that experience and its implications, we discuss the team and their backgrounds and the beginnings of the project.

1.2 The People and the Beginning of Emerald

The Emerald group included four people:[1]

- Andrew Black joined UW and the Eden project as a Research Assistant Professor in November 1981. He had a background in language design from his D.Phil. at Oxford (under C.A.R. Hoare), including previous work on concurrency and exception handling. Andrew brought

[1] Another Ph.D. student, Carl Binding, participated in some of the initial discussions with a view to being responsible for a reasonable GUI for Emerald; he decided to do a different Ph.D., so the GUI of the Emerald system remained quite primitive.

the perspective of a language designer to Eden, which had been exclusively a "systems" project.

- Eric Jul came to UW as a Ph.D. student in September 1982 with a Master's degree in Computer Science and Mathematics from the University of Copenhagen. He had previous experience with Simula 67 [15, 41] and Concurrent Pascal [29, 30] at the University of Copenhagen, where he had ported Concurrent Pascal to an Intel 8080 based microcomputer [96], and also written a master's thesis [57] that described his implementation of a small OS whose device drivers were written entirely in Concurrent Pascal.
- Norm Hutchinson also came to UW in September 1982, having graduated from the University of Calgary. He had spent the previous summer on an NSERC-funded research appointment implementing a compiler for Simula supervised by Graham Birtwistle, an author of *SIMULA Begin* [15] and an object pioneer.
- Henry (Hank) Levy had spent a year at UW in 1980 on leave from Digital Equipment Corp., where he had been a member of the VAX design and implementation team. While at UW he was part of the early Eden group and also wrote an MS thesis on capability-based architectures, which eventually became a Digital Press book [70]. Hank rejoined UW as a Research Assistant Professor in September 1983, and brought with him a lot of systems-building and architecture experience from DEC.

Shortly after his return to UW in 1983, Hank attended a meeting of the Eden group in which Eric and Norm gave talks on their work on the project. Although Hank was a coauthor of the original Eden architecture paper [67], Hank wasn't up to date on the state of the Eden design and implementation, and hearing about it after being away for two years gave him more perspective. Several things about the way the system was built — and the way that it performed — did not seem right to him. After the meeting, Hank invited Eric and Norm to a coffee shop on "the Ave", the local name for University Way NE, close by the UW campus.

At the meeting, the three of them discussed some of the problems with Eden and the difficulties of programming distributed applications. Hank listened to Eric's and Norm's gripes about Eden, and then challenged them to describe what they would do differently. Apparently finding their proposals reasonable, Hank then asked, "Why don't you do it?": thus the Emerald effort was born.

1.3 The Eden System

The problems with Eden identified at the coffee shop on the Ave were common to several of the early distributed object systems. Eden applications (that is, distributed applications that spanned a local-area network) were written in the Eden Programming Language (EPL) [21] — a version of Concurrent Euclid [50] to which the Eden team had added support for remote object invocation. However, that support was incomplete: while making a remote invocation in Eden was much easier than sending a message in UNIX, it was still a lot of work because the EPL programmer had to participate in the implementation of remote invocations by writing much of the necessary "scaffolding". For example, at the invoking end the programmer had to manually check a status code after every remote invocation in case the remote operation had failed. At the receiving end the programmer had to set up a thread to wait on incoming messages, and then explicitly hand off the message to an (automatically generated) dispatcher routine that would unpack the arguments, execute the call, and return the results. The reason for the limited support for remote invocation was that, because none of the Eden team had real experience with writing distributed applications, we had not yet learned what support should be provided. For example, it was not clear to us whether or not each incoming call should be run in its own thread (possibly leading to excessive resource contention), whether or not all calls should run in the same thread (possibly leading to deadlock), whether or not there should be a thread pool of a bounded size (and if so, how to choose it), or whether or not there was some other, more elegant solution that we hadn't yet thought of. So we left it to the application programmer to build whatever invocation thread management system seemed appropriate: EPL was partly a language, and partly a kit of components. The result of this approach was that there was no clear separation between the code of the application and the scaffolding necessary to implement remote calls.

There was another problem with Eden that led directly to the Emerald effort. While Eden provided the abstraction of location-independent invocation of mobile distributed objects, the implementation of both objects and invocation was heavy-weight and costly. Essentially, an Eden object was a full UNIX process that could send and receive messages. The minimum size of an Eden object thus was about 200-300 kBytes — a substantial amount of memory in 1984. This clearly precluded using Eden objects for implementing anything "small" such as a syntax tree node. Furthermore, if two Eden objects were co-located on the same machine, the invocation of one by the other would still require inter-process communication, which would take hundreds of milliseconds — slow even by the standards of the day. In fact, things were even worse than that, because in our prototype the Eden "kernel" itself was another UNIX process, so sending an invocation message would require *two* context switches even in the local case, and receiving the reply another two. This meant that the cost of a single invocation between two Eden objects located on the same machine was close to half the cost of a remote call (137 ms vs. 300 ms). The good news was that Eden objects enjoyed the benefits of location independence: an object did not have to know whether the target of an invocation was on the same com-

puter or across the network. The bad news was that if the invoker and the target were on the same computer, the cost was excessive.

Because of the high cost of Eden objects, both in terms of memory consumption and execution time, another kind of object was used in EPL programs — a light-weight language-level object, essentially a heap data structure as implemented in Concurrent Euclid. These objects were supported by EPL's run-time system. EPL's language-level objects could not be distributed, i.e., they existed within the address space of a single Eden object (UNIX process) and could not be referenced by, nor moved to, another Eden object. However, "sending messages" between these EPL objects was extremely fast, because they shared an address space and an invocation was in essence a local procedure call.

The disparity in performance between local invocations of EPL objects and Eden objects was huge — at least three orders of magnitude.[2] This difference caused programmers to limit their use of Eden objects to only those things that they felt absolutely needed to be distributed. Effectively, programmers were using two different object semantics — one for "local" objects and one for "remote" objects. Worse, they had to decide *a priori* which was which, and in many cases, needed to write two implementations for a single abstraction, for example, a local queue and a distributed queue. In part, these two different kinds of objects were a natural outgrowth of the two (unequal) thrusts of the Eden project: the primary thrust was implementing "the system"; providing a language on top was an afterthought that became a secondary goal only after Andrew joined the project. However, in part the two object models were a reflection of the underlying implementation: there were good engineering reasons for the two *implementations* of objects.

The presence of these two different object models was one of the things that had bothered Hank in the Eden meeting that he had attended. In their discussions, Eric, Norm and Hank agreed that while the two *implementations* made sense, there was no good reason for this implementation detail to be visible to the programmer. Essentially, they wanted the language to have a *single* object abstraction for the programmer; it would be left to the *compiler* to choose the most appropriate implementation based on the way that the object was used.

1.4 From Eden to Oz

The result of the meeting on the Ave was an agreement between Eric, Hank and Norm to meet on a regular basis to discuss alternatives to Eden — a place that Hank christened the land of Oz, after the locale of L. Frank Baum's fantasy story [9]. A memo *Getting to Oz* dated 27 April 1984 (reference [69], included here as Appendix A) describes how the discussions about Oz first focused on low-level kernel issues: processes and scheduling, the use of address spaces, and so on. This led to the realization that compiler technology was necessary to get adequate performance: rather than calling system service routines to perform dynamic type-checking and to pack up data for network interchange, a smart compiler could perform the checking statically and lay down the data in memory in exactly the right format.

The memo continues:

> 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.

Eventually the name Toto was dropped; Hank thought that it was necessary to have a "more serious" name for our language if we wanted our work to be taken seriously. For a time we used the name Jewel, but eventually settled on Emerald, a name that had the right connotations of quality and solidity but preserved our connection to the Land of Oz. Moreover, the nickname for Seattle, where Emerald was developed, is *The Emerald City*.

The Emerald project was both short and long. The core of the language and the system, designed and implemented by Eric and Norm for their dissertations, was completed in a little over 3 years, starting with the initial coffee-shop meeting in the autumn of 1983, and ending in February 1987 when the last major piece of functionality, process mobility, was fully implemented. However, this view neglects the considerable influence of the Eden project on Emerald, and the long period of refinement, improvement and derived work after 1987. Indeed, Niels Larsen's PhD thesis on transactions in Emerald was not completed until 2006 [66]. Figure 1 shows some of the significant events in the larger Emerald project; we will not discuss them now, but suggest that the reader refer back to the figure when the chronology becomes confusing.

1.5 Terminology

A note on terminology may help clarify the remainder of the paper. The terms message and message send, introduced by Alan Kay to be consistent with the metaphor of an object as a little computer, were generally accepted in the object-oriented-language community. However, we decided not to adopt these terms: in the distributed-systems community, messages were things that were sent over networks between real, not metaphorical computers. We preferred the term *operation* for what Smalltalk (following Logo) called a method, and we used the term operation invocation for

[2] A local invocation in Eden took 137 ms in October 1983, while a local procedure call took less than 20 μs.

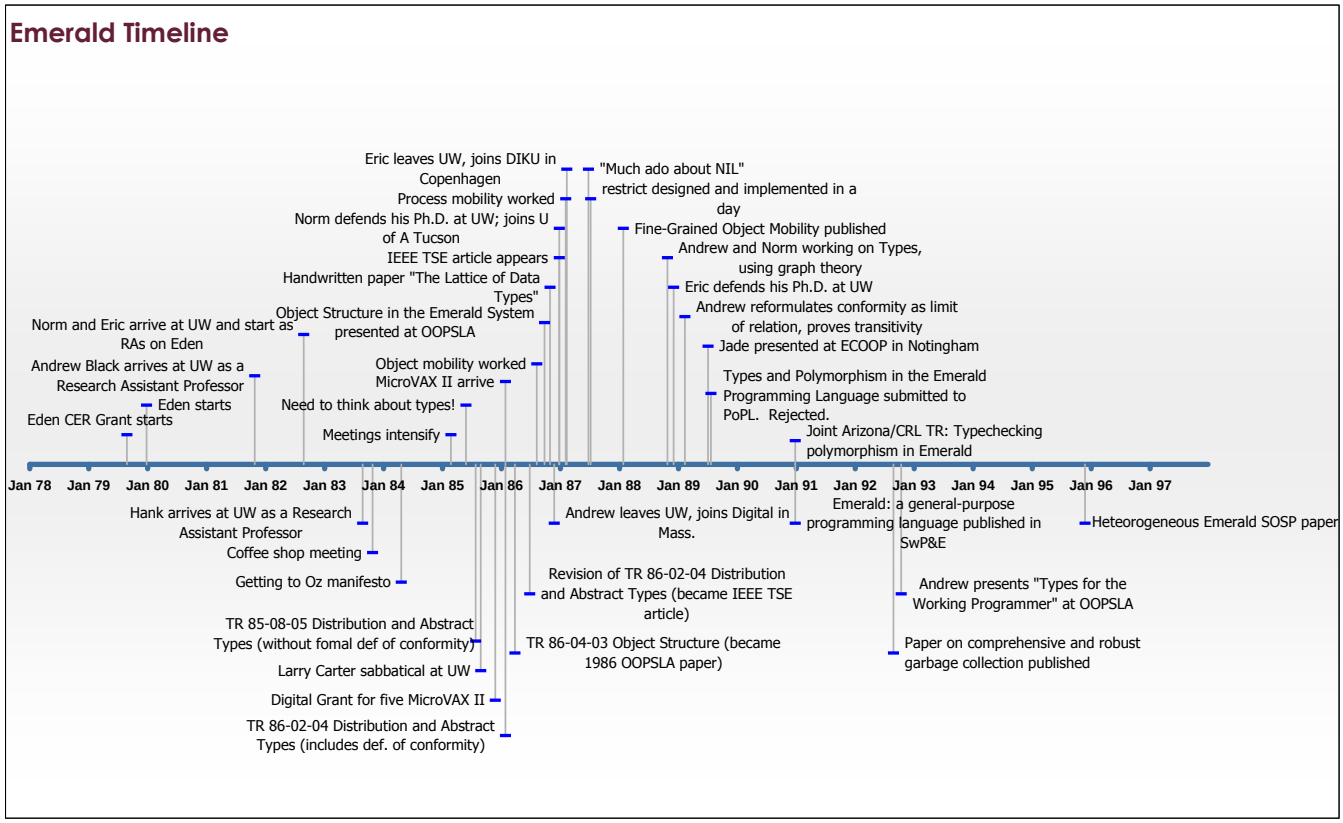

Figure 1: Some significant events in the Emerald Project

message send. In some ways this was unfortunate: language influences thought, and because we avoided the use of the terms message and message send, it was many years before some of us really understood the power of the messaging metaphor.

Emerald used the term *process* in the same sense as Concurrent Pascal and Concurrent Euclid. A process was light in weight and was protected by the language implementation rather than by an operating system address space; today the term *thread* is used to describe the same concept. Similarly, Emerald used the term *kernel* in the same sense as its predecessor languages: it meant the language-specific run-time support code that implemented object and process creation and process scheduling, and translated lower-level interrupts into operations on the language's synchronization primitives. Successor languages now often use the term *Virtual Machine* about such run-time support. Indeed, if we created Emerald today, we would have used the term Emerald Virtual Machine.

2. The Goals of Emerald

Beyond "Improving on Eden," Emerald had a number of specific goals.

- *To implement a high-performance distributed object-oriented system*. While the Eden experience had convinced us that objects were indeed a good abstraction for writing distributed systems, it did nothing to dispel doubts about the performance of distributed objects. We believed that distributed objects could be made efficient and wanted to demonstrate this *efficiency goad* by construction. Norm saw that by exploiting compiler technology, we could not only make run-time operations more efficient, but could eliminate many of them altogether (by moving the work to compile time). Eric was already concerned with wide-area distribution: even during the planning phases of the project he was thinking of sending objects from Copenhagen to Seattle, although this did not become possible for several years citeFolmer93.
- *To demonstrate high-performance objects*. Stepping back from *distributed* objects, we were also concerned with validating the ideas of object-oriented programming *per se*. Smalltalk-80 had created a lot of excitement about the power and flexibility of objects, but there was also a lot of skepticism about whether or not objects could ever be made reasonably efficient. The failure of the Intel iAPX 432 architecture [79] effort had, in the minds of some people, reinforced the view that objects would

always impose excessive overhead. Dave Ungar's dissertation research at Stanford was investigating how hardware could help overcome the inherent costs of supporting objects [102]. One of our goals was to show that operations similar to those found in a low-level language like C could be executed in Emerald with comparable efficiency. Thus, we wanted the cost of incrementing an integer variable to be no higher in Emerald than in C. Of course, the cost of invoking a *remote* object would be higher than the cost of a C procedure call. An important principle was that of *no-use, no-cost*: the additional cost of a given functionality, e.g., distribution, should be imposed only on those parts of the program that used the additional functionality.

- *To simplify distributed programming.* Although Eden was a big step forward with respect to building distributed applications, it still suffered from major usability problems. The Eden Programming Language's dual object model was one, the need to explicitly receive and dispatch incoming invocation messages was another, and the need to explicitly check for failure and deal with checkpointing and recovery was yet another. One of the goals for Emerald was to make programming simpler by automating many of the chores that went along with distributed programming. If a task was difficult enough to make automation impossible, then the goal was to automate the bookkeeping details and thus free the programmer to deal with the more substantive parts of the task. For example, we had no illusion that we could automate the placement of objects in a distributed system, and left object location policy under the explicit control of the programmer. However, the mechanisms for creating the objects on, or moving them to, the designated network node, and of invoking them once there, could all be automated.
- *To exploit information hiding.* We believed in the principle of information hiding [85], and wanted our language to support it. The idea that a single concept in the language, with a single semantics — the object — might have multiple implementations follows directly from the principle of information hiding: the details of the implementation are being hidden, and the programmer ought to be able to rely on the interface to the object without caring about those details. In the *Getting to Oz* memo [69] we wrote: "Another goal, then, and a difficult one, is to design a language that supports both large objects (typically, operating system resources such as files, mailboxes, etc.) and small objects (typically, data abstractions such as queues, etc.) using a single semantics."
- *To accommodate failure*. Emerald was intended for building distributed applications. What distinguishes distributed applications from centralized ones is that both the individual computers on which the application runs and the network links that connect them can fail; nevertheless, the application as a whole should continue to run. We realized that handling failures was a natural part of programming a distributed application, in fact, failures were *anticipated* conditions that arose from distribution: computers and programs would crash and restart, network links would break, and consequently objects would become temporarily or permanently unavailable. We saw it as the programmer's job to deal with these failures, and therefore as the language designer's job to provide the programmer with appropriate tools. It was explicitly not a goal of Emerald to provide a full-blown exception-handling facility [69], or to impose on the programmer a particular way of handling failure, as Argus was doing with its pioneering support for transactions in distributed systems [71].
- *To minimize the size of the language*. We wanted Emerald to be as small and simple as possible while meeting our other goals. We planned to achieve this by "pulling out" many of the features that in other languages were built-in, and instead relying on the abstraction provided by objects to extend the language as necessary. Our goal was a simple yet powerful object-based language that could be described by a language report not substantially larger than that defining Algol 60 [8]. We did not want to go as far as Smalltalk and rely on library code for the basic control structures: this would have conflicted with our efficiency goal. We also felt that not allowing the programmer to define new control structures would not adversely affect the writing of distributed applications.

  We did decide to build in only the simplest kind of fixed-length Vector, and to rely on library code to provide all the common data structures, such as arrays of variable size, and lists. This meant that the type system had to be adequate to support user-defined container objects; striving to do this led us to make several innovations in object-oriented type systems, even though inventing a type system was not one of our primary goals. The same idea of pulling out functionality into a library was applied to integer arithmetic. We had to decide whether to include infinite-precision or fixed-length integers. Infinite-precision arithmetic was clearly more elegant and would lead to simpler programs and a more concise language definition, but it would not help us in our quest to make $x \leftarrow x + 1$ as fast as in the C language. So we included fixed-precision integers in the base language, and allowed the programmer to define other kinds of integers in library code.
- *To support object location explicitly*. We knew that the placement of objects would be critical to system performance, and so our information-hiding goal could not be allowed to hide this performance-critical information from the programmer. So "we decided that the concept of object location should be supported by the language"

and that "the movement of objects should be easily expressible in the language" [69]. Early on we envisaged language support for determining the location of an object, moving the object to a particular host, and fixing an object at a particular host. The idea of *call-by-move* was floated in the 1984 *Getting to Oz* memo; call-by-move was a new parameter passing mechanism in which the invoker of an operation indicated explicitly that a parameter should be moved to the target object's location. By March 1985 call-by-move was established, and Andrew was arguing for a separation of concerns between the language's *semantics*, which he thought should be similar to those of any other object-based language (for example, all parameters would be passed by object reference), and its *locatics* — a term that we invented to mean the part of the language concerned with the placement of objects. We adopted this separation: the primitives that controlled the placement of objects were designed to be orthogonal to the ordinary operational semantics of the language. For the most part we were successful in maintaining this separation, but locatics does inevitably influence failure semantics. For example, if two objects are co-located, then an invocation between them can never give rise to a *object unavailable* event.

3. How We Worked

The Emerald group was informal and largely self-organized. Hank kept Eric and Norm in line by requiring regular meetings, minutes, and written progress reports. He was also the prime mover in getting the first external funding that the project received: a grant of five MicroVAX computers from Digital Equipment Corp. Andrew attended some of the meetings, and had many impromptu as well as scheduled discussions with Norm about the language design, and in particular about the type system.

Minutes from March 1985 reveal an intent to meet three times per week. They state: "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." This captured the major distribution of work: Norm implemented the compiler, while Eric worked on the run-time system, which we called the Emerald kernel.

They pursued an incremental implementation strategy. The first compiler was for a simplified version of the language and produced byte-codes that were then interpreted. The interpreter ran on top of a kernel that provided threads. Initially there was no I/O other than a `println` instruction provided by the interpreter. In this way, they very quickly had a working prototype, even though it could execute only the simplest of programs. Over time, the compiler was modified to generate VAX assembly language, which ran without needing interpretation, but which still called kernel procedures for run-time support. This incremental strategy brought new functionality and better performance every day, and was a real catalyst for the development of the prototype. Looking back, we realize that we pioneered many of the techniques that have now become popular as part of agile development practices such as XP [10], although there is no evidence to suggest that those who invented XP were aware of what we had done.

A hallmark of Emerald was the close integration between the compiler and the run-time kernel. In part, this was achieved by putting the compiler writer (Norm) and the kernel implementor (Eric) in the same office, together with two workstations. (It was unusual at the time to allocate expensive workstations to graduate students.) Norm and Eric could clarify even minor details immediately. But there was also a technical side to this integration. The compiler and the kernel had to agree on several basic data structures. For example, the structure of an object descriptor was laid down in memory by the compiler, and manipulated by the kernel. The alternative approach in which this structure is encapsulated in a single kernel module would have simplified the software engineering, but it would also have meant that whenever the compiler wanted the code that it was generating to create an object, it would be forced to lay down code that made a call to the run-time kernel. This would have conflicted with our efficiency goal.

To ensure compatibility between the compiler and the kernel, all of these basic data structures were defined in a single file, both in C (for the compiler) and in assembly language (for the kernel). Careful use of the C preprocessor meant that this file could be treated either as a C program with assembly language comments, or as assembly language with C comments. The development of the shared description of the data structures was aided by the presence of a whiteboard where the important data structures were described; by convention, the truth was on the whiteboard, and the rest of the system was made to conform to it (see Figure 2). We avoided divergence by rebuilding the compiler and run-time system completely every night and during lunch breaks. Even language changes were handled efficiently. Because all existing Emerald programs were stored on Norm's and Eric's MicroVAX workstations, we could decide to change the syntax or semantics of the language and within hours change the compiler, the run-time system and *all existing Emerald programs* to reflect the modification. Typically, such changes were made before lunch so that a rebuild could happen over lunch with a full afternoon to verify that things still worked.

Initial development took place on the VAX 11/780 and two VAX 11/750s of the Eden project, which were called June, Beaver, and Wally. In December 1985 we obtained our grant of MicroVAX workstations from DEC; the workstations themselves arrived a few months later. Three of these (Roskilde, Taber, and Whistler) were used as personal development machines by Eric, Norm and Hank. The others were

Figure 2: The whiteboard that defined the basic data structures in the Emerald system, February 1986.

used remotely in distribution experiments. We used micro-benchmarks to find the fastest instruction sequences for common operations, and found that sometimes sequences of RISC operations were faster than the equivalent single CISC instruction. One example of this was copying a small number of bytes, where repeated reads and writes were faster than using the VAX block move (MOVC3) instruction.

The interpreter itself was written in C, but in a way that treated C as a macro-assembler. We would first identify the most efficient assembly-language sequence for the task at hand; for example, we found that non-null loops were more efficient if the termination test came at the end rather than at the beginning, because this avoided an unconditional branch. We then wrote the C code that we knew would generate the appropriate assembly language sequence, for example, using **if** (*B*) **do** . . . **while** (*B*) rather than **while** (*B*) . . .

Over time, more and more data structures were defined to enable the compiler to transmit information to the interpreter. These included a source-code line number map, which enabled us to map back from the program counter to the source code line for debugging, and *templates* for the stack and for objects, which enabled us to distinguish between object-references and non-references. The latter information was essential for implementing object mobility, because object references were implemented as local machine addresses, and had to be translated to addresses on the new host when an object was moved. (The same information is of course useful for garbage collection, but the garbage collector was not implemented until later.)

By June 1985 we had realized that we needed to think seriously about types. We had decided that utility objects like variable-sized arrays and sets should not be built into the language, but would be provided in a library, as they were in Smalltalk. This would keep the language simple. However, we wanted these objects to have the same power, and offer the same convenience to programmers, as the built-in arrays and sets of Pascal. We had also decided early on that Emerald was to be statically typed, both for efficiency and to make possible the early detection and reporting of programming errors. We therefore needed a way of declaring that an array or a set would contain objects of a certain type — in other words, we needed parameterized types. We were influenced by the Russell type system [43], one of the few contemporary systems that supported parameterized types. We were also aware of CLU's decision to provide a parameterization mechanism that allowed procedures and clusters to be dependent on manifest values, including types; this mechanism (which used square brackets and operated at compile time) was completely separate from the mechanism used to pass arguments to procedures (which used parentheses and operated at run time). However, we realized that an open distributed system could not make a clear distinction between

compile time and run time: it is possible to compile new pieces of program and add them to the running system at any time. Thus, while we planned to check types statically whenever this was possible, we knew from the outset that sometimes we would have to defer type checking until run time, for the very good reason that the object that we needed to check arrived over the network at run time, having been compiled on a different computer. So we followed Russell, and made types first-class values that existed at run time and could be manipulated like any other values. In an object-oriented language, the obvious way to do that was to make types objects, so that is what we did in Emerald.

4. Technical Innovations

As we have discussed, the goals of Emerald centered on simplifying the programming of distributed applications. The way in which Emerald supports distribution has been described in a journal article [59] that was widely cited in the 1980s and 1990s but may not be familiar to this audience. In addition, Emerald made a number of other, lesser-known technical contributions that are nevertheless historically interesting. In this section we first summarize Emerald's object model and then discuss a few of these other contributions.

4.1 Emerald's Object Model

Key to any object-based language or system is its object model: the story of how its objects can be created and used, and what properties they enjoy. We have previously described the motivation for Emerald's single object model: the problem with having both heavyweight and lightweight objects visible to the programmer, and the need for the programmer to choose between them. This was a reflection of the state of the art in the early 1980s, which recognized two very distinct object worlds: object-oriented programing languages, such as Smalltalk, and object-based operating systems, such as Eden and Hydra. In object-oriented languages, the conceptual model of an object was a small, lightweight data structure whose implementation was hidden from its clients. In object-oriented operating systems, the conceptual model of an object was a heavyweight, possibly remote operating system resource, such as a remote disk, printer, or file. Consequently, these systems frequently used an OS process to implement an object.

4.1.1 A *Single* Object Model

An important goal of Emerald was to bridge the gulf between these two worlds and to provide the best of each in a unified object model. On the one hand, we wanted an Emerald object to be as light in weight as possible — which meant as light as an object provided by a procedural programming language of the time. On the other hand, we wanted an Emerald object to be *capable* of being used in the OS framework, for example as a distributed resource that was transparently accessible across the local-area network.

Emerald achieved this goal. It provided a single, simple object model. An object, once created, lived forever and was unique, that is, there was only one instance of it at any time. Each object had (1) a globally unique *name*, (2) a *representation*, i.e., some internal state variables that contained primitive values and references to other objects, and (3) a set of *operations* (methods) that might be invoked on the object. Some of these operations could be declared to be *functional*, which meant that they computed a single value but had no effect and did not access any mutable state outside of the object itself. Any Emerald object could also have an associated *process*, which meant that an object could be either active (having its own process) or passive (executing only as a result of invocation by another process); the process model is described fully in Section 4.3.1.

4.1.2 Immutability

One of our more powerful insights was to recognize the importance of immutability in a distributed system. With ordinary, mutable objects, the distinction between the original of an object and a copy is very important. You can't just go off and make a copy of an object and pass it off as the original, because changes to the original won't be reflected in the copy. So programmers must be kept aware of the difference, and the object and the copy must have separate identities. However, with immutable objects, the copy and the original will always have the same value. If the language implementation lies and tells the programmer that the two objects have the same identity — that they are in fact one and the same — the programmer will never find out.

The idea of an immutable object was familiar to us from CLU. While this idea did at first seem strange — after all, one of the main distinctions between objects and values was that objects can change state — a little thought showed that it was inevitable. Even the most ardent supporter of mutable objects would find it difficult to argue that 3, π and 'h' should be mutable. So, the language designer is forced either to admit non-objects into the computational universe or to allow immutable objects. Once immutable primitive objects have been admitted, there are strong arguments in favor of allowing user-defined immutable objects, and little reason to protest. For example, Andrew knew that CLU's designers had eventually found it expedient to provide two versions of all the built-in container objects, one mutable and the other immutable, even though they had initially decided to provide only mutable containers [28, 92].

Once we accepted that some objects were immutable, we found that there were many benefits to formally distinguishing immutable from mutable objects by introducing the **immutable** keyword. Emerald made wide use of the fact that some objects were known to be immutable. Although the language semantics said that there was a unique integer object 3, it would have been ridiculous to require that there really be only a single 3 in a distributed system. Because inte-

gers were immutable, the implementation was free to create as many copies of 3 as was expedient, and to insist (if anyone asked, by using the == operation to test object identity) that they were all the very same object. The implementation was lying to the user, of course, but it was lying so well and so consistently that it would never betray itself.

Another place where Emerald used immutability was for *code objects*. Logically, each Emerald object owned its own code, but if a thousand similar objects were created containing the same code, there was clearly no need to create a thousand copies of the code. But neither was it reasonable to insist that there could be only a single copy: in practice, the code needed to be on the same computer as the data that it manipulated. The obvious implementation was to put one copy of the code on each computer that hosted one of the objects; if an object were moved to a new computer, a copy of the code would be placed there too, unless a copy were there already. Because the code was immutable, we could pretend that all of these copies were logically the same, and this pragmatic use of copying by the implementation did not "show through" in the language semantics. Indeed, from the programmer's point of view, code objects did not exist at all.

Immutability applied to programmer-defined objects as well as built-in objects. Indeed, any object could be declared as immutable by the programmer: immutability was an assertion that the *abstract* state of the object did not change. It was quite legal, for example, to maintain a cache in an immutable object for efficiency purposes, provided that the cache did not change the object's abstract state.

Functional operations on immutable objects had the property that they always returned the same result. This meant that they could be evaluated early without changing the program's semantics. In particular, functional operations on immutable objects could be evaluated at compile time: they were manifest. This turned out to be vitally important to Emerald's type system, because we required types of variables and operations to be manifest expressions: the declarator *Array.of*[*Integer*] is an invocation of the function *of* on the immutable object *Array*, with the immutable object *Integer* as argument.

Our decision to trust the programmer to say when an object was immutable or an operation was functional, rather than attempting to enforce these properties in the compiler, arose from our experience with Concurrent Euclid (CE). Concurrent Euclid distinguished between functions and procedures, and in CE the compiler would enforce the distinction by emitting an error message if a function attempted to do something that had an effect. We had found it to be enormously frustrating to get an error message when we were trying to debug a program by putting a *print* statement in a function. (The compiler was smarter than we thought. We eventually found out that the error messages were actually warnings: the compiler generated perfectly good code despite the existence of the *print* statement!) What we learned from this was that programmers resent tools that get in their way: a language should help programmers to express what they need to express, rather than always trying to second-guess them. Interestingly, in "A History of CLU" Liskov explains that she rejected the idea of immutability declarations exactly because the compiler would not have been able to check them [72, p. 484–5].

4.1.3 Objects Were Encapsulated

Because of our background in distributed systems, we took encapsulation much more seriously than did those who thought that encapsulation was just something that one did to get software engineering brownie points. The only way to access an object in Eden was to send a message to the machine that hosted it: there were no "back doors" that could be left ajar. In Emerald, because all of the objects that were co-located shared a single address space, it would have theoretically been possible to provide a back door through which a co-located object could sneak a look at some private data. But this would clearly be a really bad idea: such a back door would need to be slammed shut if the target object moved to another machines, and objects could move at any time. Moreover, as system implementors, we had to know *all* of the references leading into and out of each object, and we had to know that those references were used only through legitimate invocations of the object. This knowledge is what made mobility possible; if an object could somehow "cons up" a direct reference to the internal state of another object, then mobility would be next to impossible. Moreover, Emerald objects were concurrent (see Section 4.3), so it was necessary to ensure that any access to the internal state of an object was subject to some kind of synchronization constraint.

In the Emerald language, we indicated encapsulation by distinguishing between operations that could be invoked on an object from the outside (which were flagged with the **export** keyword), and operations that could be invoked only on the object itself. The compiler used the list of exported operations to determine the signature of the object: attempts to invoke other operations were not only forbidden by the type system, but would in any case not be supported at run time.

It is worthwhile comparing this approach to encapsulation to that taken by Java. In Java (and in C++), encapsulation is class-based rather than object based: a field or a method that is designated as private can in fact be accessed by any other object that happens to be implemented by the same class. However, access to fields of remote objects is not supported by Java RMI, and consequently Java can't make local and remote objects look the same.

4.1.4 One Object Model, Three Implementations

Although many of the newer languages with which we were familiar — Alphard, CLU, Euclid, and so on — claimed to support "abstract data types", we realized that most of them didn't. To us, *abstract* meant that two different implementations of the *same* type could coexist, and that client code would be oblivious to the fact that they were different. Other languages actually supported *concrete* data types: only a single implementation of each data type was permitted.

If we took encapsulation seriously — and, as we have discussed in Section 4.1.3, distribution meant that we had to — then Emerald objects would be characterized by truly abstract types. This meant that so long as two objects had the same set of operations, client code would treat them identically, and the fact that they actually had different implementations would be completely hidden from the programmer. When the object implementation and the client code were compiled separately — this might mean separate in space as well as separate in time — the difference would also be hidden from the compiler.

However, when the implementation and the client code were *not* compiled separately, then the compiler could take advantage of the ability for multiple implementations of the same type to coexist. The same source code could be compiled into different representations, if the compiler decided that there was a reason to do so. Thus, the compiler could choose a customized representation that took advantage of the fact than an object was used in a restricted way. Norm designed the compiler to examine the object constructors in the code that it was compiling and to choose between two alternative implementations for the object under construction.

Global was the most general implementation. A global object could move to other machines and could be invoked by any other object regardless of location. References to a global object were implemented as pointers to an *object descriptor*.

Local was an implementation optimized for objects that could never be referenced from a remote machine. The compiler chose this implementation when it could ascertain that a reference to the object could never cross a machine boundary. For example, if object *A* defined an internal object *B* but never passed a reference to *B* outside of its own boundary, then the compiler knew[3] that *B* could only be invoked by code in *A*. This allowed the compiler to strip away all code related to remote referencing, including the object descriptor.

There was actually a third implementation, *direct*, which we used to implement objects of "primitive" types (*Boolean*, *Character*, *Integer*, etc.). However, direct objects were a bit of a cheat, because for these types we gave up on type abstraction. Primitive types are discussed in Section 4.2.5.

Letting the compiler choose the implementation meant that programs that did not use distribution could be compiled into code similar to that produced for a comparable C program. In particular, integer arithmetic and vector indexing were implemented by single machine instructions on the VAX. Moreover, we were able to provide multiple representations without boxing, and could thus represent integers by a direct bit pattern, just as in C. The result was that Emerald achieved performance very close to that of C — for the kind of simple non-distributed programs that could have been written in C. Nevertheless, the very same source code, used in a program that exploited distribution, could create remotely accessible distributed objects.

4.1.5 Object Constructors Replace Classes

During the initial development of Emerald we were unhappy about adopting Smalltalk's idea of a class that could change dynamically. Smalltalk allowed a class to be modified, for example by adding an instance variable or a method, while instances of that class were alive; all of the instances immediately reflected the changes to their class. This worked well in a single-user centralized system, but we did not see how to adapt it to a distributed system. For performance reasons, the class would have to be replicated on every node that hosted an instance of the class; if the class were subsequently modified, we would be faced with the classic distributed update problem. The problem was compounded in Emerald by the fact that some machines might be inaccessible when the update occurred. For us, the semantics of class update seemed impossible to define in a satisfying manner: the only implementable semantics that we could think of was that the update would take effect *eventually*, but perhaps at widely differing times on different machines, and possibly far in the future.

The difficulty of defining update for classes was one of the reasons that we considered classes harmful. [4] Another reason was that we wanted Emerald to be a simple language in which everything was an object. This implied that if we had classes they should also be objects, which would in turn need their own classes. Smalltalk's metaclass hierarchy resolved this apparent infinite regression, but at the cost of significant complexity, which we also wanted to avoid.

We noted that in classic OO languages, such as Simula 67 and Smalltalk, the concept of class was used for several different purposes: as a classification scheme for object instances, as a template describing the internal structure of those instances, as a repository for their code, and as a factory for generating new instances [17, p. 85].

[3] More precisely: the compiler could figure this out, using techniques from what is now called escape analysis.

[4] During our car trip from Seattle to Portland for the first OOPSLA in 1986, we bounced around the idea of writing a paper about *Classes Considered Harmful*.

Because we had a type system, we did not need to use classes for classification. Code storage was managed by the kernel, because it was heavily influenced by distribution. To describe the internal structure of an instance, and to generate new instances, we invented the idea of an object constructor, based on the record constructor common in contemporary languages. An object constructor was an expression that, each time it was executed, generated a new object; the details of the internals of the object were specified by the constructor expression. Object constructors could create fully-initialized instances; in combination with block structure, this meant that the initial state of an object could depend on the parameters of the surrounding context. Moreover, because object constructors could be nested, it was easy to write an object that acted like a factory, that is, an object that would make other objects of a certain kind when requested to do so. It was also easy to write an object that acted like a prototype, in that it created a clone of itself when requested to do so.

4.1.6 Objects Are *Not* Fragmented

An important early decision was that the whole of an object would be located on a single machine. The alternative would have been to allow an object to be fragmented across multiple machines, but this would then require some other inter-machine communication mechanism (apart from operation invocation), thus making the language much larger. It would also seriously complicate reasoning about failure. We were also not convinced that fragmented objects were necessary: a distributed resource could after all be represented by a distributed collection of communicating (non-distributed) objects that held references to each other. One of the consequences of this design decision was that if any part of an object were locally accessible, then all other parts were also locally accessible. This allowed the compiler to strip away all distributed code for intra-object invocations and references.

4.1.7 The Unit of Mobility and the Concept of Attachment

Because we had decided that Emerald objects would be mobile, it might seem that we had also determined the unit of mobility: the object. Unfortunately, things were not quite that simple. An object contained nothing more than variables that were references to other objects, so moving an object to a new machine would achieve little: every time the code in the object invoked an operation on one of its variables, that operation would be remote.

If the variable referred to a small immutable object, it would clearly make sense to copy that object when the enclosing object was moved. Mutable objects could not be copied, but they could be *moved* along with the object that referenced them. A working memo written by Eric in August 1986 discusses various ways of defining groups of objects that would move together; the memo describes both imperative and declarative language features. We eventually decided to introduce the concept of *attachment* by allowing the **attached** keyword to be applied to a variable declaration, with the meaning that the object referenced by the attached variable should be moved along with object containing the variable itself. Although this appeared to make attachment a static property, we realized that this was not in fact the case. An object could maintain two variables *a* and *u* that referenced the same object, *a* being attached and *u* not attached; by assigning *u* to *a*, or assigning **nil** to *a*, attachment could be controlled dynamically. Thus, the concept of attachment was powerful enough to implement dynamically composable groups efficiently. Joining or leaving a group could be as simple as a single assignment. Furthermore, in many cases when a new object joined a group, it would be assigned to a variable anyway, and so the additional cost would be zero. Attachment was also very simple to implement, because it required nothing more than a bit in the template that described the object's layout in memory (described in Section 4.6) and thus there was no per-object overhead, nor was there any cost for an object not using the concept.

4.2 The Emerald Type System

At the time we started the Emerald project, none of us knew very much about type theory, and innovation in type theory was not one of Emerald's explicit goals. However, we found that the goals that we had set for Emerald seemed to require features that were not mainstream; indeed, in 2007 some of them are still not mainstream. So we set about figuring out how to support the features we needed. The aim of this section is to describe that process and its end point; it draws from material originally written by Andrew and Norm in an unpublished paper dated 1989.

4.2.1 Emerald's Goals as They Relate to Types

We had decided from the first that Emerald needed the following features.

- Type declarations and compile-time type checking for improved performance. We had all done most of our programming in statically typed languages, and felt that Smalltalk's dynamic type checking was one of the reasons for its poor performance.
- A type system that was used for classification by behavior rather than implementation, or naming. This was demanded by our view that programming in Emerald consisted of adding new objects to an existing system of objects (as in Smalltalk) rather than writing standalone programs, (as in Euclid or Object Pascal). We might know nothing at all about the implementation of an object compiled elsewhere; we could demand that the object supported certain operations, but not that it had a particular implementation, or inherited from some other implementation.

- A small language in which utilities such as collection objects were not built-in, but could instead be implemented (and typed) in the language itself.

The second goal was most influential. The Smalltalk idea of operation-centric *protocols* formed the starting point for our work on what came to be known as abstract types and conformity-based typing.

4.2.2 The Purpose of Types

As we implied in Section 4.2.1, Emerald's compile-time typing was more an article of faith than a carefully reasoned decision: we believed that compile-time typing would help the compiler to generate more efficient code. However, the decision to declare types in Emerald did not tell us what those declarations should mean. In a language like Pascal or Concurrent Euclid, a type declaration determined the data layout of the declared identifier. In Simula, a declaration determined the class of the object referenced by the identifier. But, as discussed in Sections 4.1.4 and 4.1.5, neither of these functions of types would be applicable in Emerald. So, what purpose should type declarations serve?

We turned for inspiration to two recently published papers on types. According to Donahue and Demers [44], the purpose of a type system was to prevent the misinterpretation of values — to ensure that the meaning of a program was independent of the particular representation of its data types. This independence would be necessary if we wished to abstract away from representation details — for example, so that some of them could be left to the compiler. Cardelli and Wegner [40] made the same point more colorfully:

> A major purpose of type systems is to avoid embarrassing questions about representations, and to forbid situations in which these questions might come up. A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use.

In fact, Donahue and Demers stated that a programming language was strongly typed exactly when it prevented this misinterpretation of values [44].

However, for the most part these papers discussed typing for values rather than objects. One of the properties of an object was encapsulation: the object's own operations were the only ones that could be applied to its data. The only mechanism that was needed to enforce this encapsulation was static scoping as found in Simula 67 and Smalltalk. There was thus no need for Emerald's type system to forbid "embarrassing questions about representations": object encapsulation already did exactly this. So, should we abandon the idea of making Emerald statically typed? We thought not: we felt that a type system could usefully serve other goals. Amongst these goals were the classification of objects, earlier and more meaningful error detection and reporting, and improved performance.

In Smalltalk, objects were classified by class, and the inheritance relation between classes was important to understanding a Smalltalk program. However, classes conflated two issues: how the object behaved, and how it was implemented. Because Emerald took encapsulation seriously (see Section 4.1.3), we had to separate these issues. We decided that the programmer should use types to classify an object's behavior, while implementation details should be left to the compiler. This led to the view of a type being the set of operations understood by the object, as discussed in Section 4.2.3.

A second problem we perceived with Smalltalk was that (almost) the only error that we saw was "message not understood", and we didn't see this error until run time. As programmers who had grown up with Pascal and Concurrent Euclid, when we accidentally added a boolean and an integer we expected an explicit *compile time* error message. We wanted to see similar error messages from the Emerald compiler whenever possible — which was quite often (see Section 4.2.4).

We also believed that at least some of Smalltalk's performance problems were caused by the absence of static typing We felt that if only the compiler had more information available to it about the set of operations that could be invoked on an object, it could surely optimize the process of finding the right code, called method lookup. We may have been right, although subsequent advances such as inline caches have largely eliminated the "lookup penalty". The way that we used this extra information to eliminate method lookup is described in Section 4.2.5.

In summary, we came to the conclusion that there were three motivations for types in Emerald: to classify objects according to the operations that they could understand, to provide earlier and more precise error messages, and to improve the performance of method lookup. We now discuss in more detail the consequences of each of these motivations for types on the development of Emerald's type system.

4.2.3 Types Were Sets of Operations

We were aware from our experience with Eden that a distributed system was never complete: it was always open to extension by new applications and new objects. Today, in the era of the Internet, the fact that the world is "under construction" has become a cliché, but in the early 1980s the idea that all systems should be extensible — we called it the "open world assumption" — was new.

A consequence of this assumption was that an Emerald program needed to be able to operate on objects that did not exist at the time that the program was written, and, more significantly, on objects whose *type* was not known when the application was written. How could this be? Clearly, an application must have *some* expectations about the operations that could be invoked on a new object, otherwise the appli-

cation could not hope to use the object at all. If an existing program *P* had minimal expectations of a newly injected object, such as requiring only that the new object accept the *run* invocation, many objects would satisfy those expectations. In contrast, if another program *Q* required that the new object understand a larger set of operations, such as *redisplay*, *resize*, *move* and *iconify*, fewer objects would be suitable.

We derived most of Emerald's type system from the open world assumption. We coined the term *concrete type* to describe the set of operations understood by an actual, concrete object, and the term *abstract type* to describe the declared type of a piece of programming language syntax, such as an expression or an identifier. The basic question that the type system attempted to answer was whether or not a given object (characterized by a concrete type) supported enough operations to be used in a particular context (characterized by an abstract type). Whenever an object was bound to an identifier, which could happen when any of the various forms of assignment or parameter binding were used, we required that the concrete type of the object *conform* to the abstract type declared for the identifier. In essence, conformity ensured that the concrete type was "bigger" than the abstract type, that is, the object understood a superset of the required operations, and that the types of the parameters and results of its operations also conformed appropriately.

Basing Emerald's type system on conformity distinguished it from contemporary systems such as CLU, Russell, Modula-2, and Euclid, all of which required equality of types. It also distinguished Emerald's type system from systems in languages like Simula that were based on subclassing, that is, on the ancestry of the object's implementation. In a distributed system, the important questions are not about the implementation of an object (which is what the subclassing relation captures) but about the operations that it implements.

For our purposes, type equality was not only unnecessary: it was counterproductive. Returning to the example above, there was no need to require that new objects presented to *P* supported *only* the operation *run*; no harm could come from the presence of additional operations, because *P* would never invoke them. The idea of conformity was that a type *T* conformed to a type *U*, written $T \circ\!\!> U$, exactly when *T* had all of *U*'s operations, when the result types of those operations conformed, and when the argument types conformed *inversely*. We chose the symbol $\circ\!\!>$ to convey the idea that the type to the left had more *o*perations than the type to the right. Nowadays, this relation is usually written $<:$ and called *subtyping*, which seems to convey exactly the opposite intuition.

Having settled on a conformity-based type system, we still had to address the question of whether conformity should be deduced or declared. In other words, would it be sufficient for an object to *have* all of the operations demanded by a type, or would it also be necessary for the programmer to *say* that it had them? Because URLs would not be invented for another 10 years, and even local-area distributed file systems were rare and primitive, there was no simple way for a programmer to state that one type (declared right here) conformed to another type (declared in some other program on some other computer). Also, our experience with Eden had taught us that we would often not appreciate the need for a "supertype" (a type with fewer operations) until after someone else had written a program using a subtype. It seemed quite impractical to have to ask some other programmer, possibly in some other organization, to change his or her code to say that one of the types that it used conformed to a supertype definition that we had written later. It also seemed pointless: it was easy enough to check type conformity directly.

This question of whether type compatibility should be deduced or declared is currently still open; the current jargon is to call deduced conformity *structural* and declared conformity *nominal*. Four or five years after we had chosen structural equivalence for Emerald, Cardelli and his colleagues wrote:

> there is a strong argument for switching to structural equivalence, which is that structural equivalence makes sense between types that occur in different programs, while name equivalence makes sense only between types that occur in the same program. This advantage becomes significant when type-safety is extended to distributed systems… or to permanent data storage systems [39, p. 207].

Modula-3, a version of Modula designed for distributed systems, moved from the nominal typing of Modula-2 to structural typing [38].

We were not, of course, the first to use structural equivalence — Algol–68 and Euclid were there before us. Neither were we the first to realize that in an object-based language, "structure" meant not the layout of data fields, but the availability of operations — Smalltalk had done that, and its lead has since been followed by more recent languages such as Python and Ruby, which call it "duck typing". (The name comes from the idea that if it looks like a duck, walks like duck, and quacks like a duck, it must *be* a duck [107].) But we were perhaps the first to apply static duck typing to objects.

4.2.4 Type Checking and Error Messages

Another consequence of the open world assumption was that sometimes type checking had to be performed at run time, for the very simple reason that neither the object to be invoked nor the code that created it existed until after the invoker was compiled. This requirement was familiar to us from our experience with the Eden Programming Language [21]. However, Eden used completely different type systems (and data models) for those objects that could be

created dynamically and those that were known at compile time.

For Emerald, we wanted to use a single consistent object model and type system. Herein lies an apparent contradiction. By definition, compile-time type checking is done at compile time, and an implementation of a typed language should be able to guarantee at compile time that no type errors will occur. However, there are situations where an application must insist on deferring type checking, typically because an object with which it wishes to communicate will not be available until run time.

Our solution to this dilemma provided for the consistent application of conformity checking at either compile time or run time. If enough was known about an object at compile time to guarantee that its type conformed to that required by its context, the compiler certified the usage to be type-correct. If not enough was known, the type-check was deferred to run time. In order to obtain useful diagnostics, we made the design decision that such a deferral would occur only if the programmer requested it explicitly, which was done using the **view**...**as** primitive, which was partially inspired by qualification in Simula 67 [15, 41].

Consider the example

var *unknownFile: File*
...
r ← (**view** *unknownFile* **as** *Directory*).*Lookup* ["*README* "]

Without the **view**...**as** *Directory* clause, the compiler would have indicated a type error, because *unknownFile*, as a *File*, would not understand the *Lookup* operation. With the clause, the compiler treated *unknownFile* as a *Directory* object, which would understand *Lookup*. In consequence, **view** ... **as** required a dynamic check that the type of the object bound to *unknownFile* did indeed conform to *Directory*. Thus, successfully type-checking an Emerald program at compile time did not imply that no type errors would occur at run time; instead it guaranteed that any type errors that *did* occur at run time would do so at a place where the programmer had explicitly requested a dynamic type check.

The **view**...**as** primitive later appeared in C++.

Partially inspired by the **inspect** statement of Simula 67 [15, 41], we also introduced a Boolean operator that returned the result of a type check. This allowed a programmer to check for conformity before attempting a **view**...**as**.

4.2.5 Types and efficiency

A primary goal of Emerald was to demonstrate the viability of using a single object model for both small (Integer) and large (Directory) objects. One of our performance goals was to achieve the performance of C for simple operations like adding integers and invoking operations on local objects. We believed that static typing would lead to improved efficiency and we used information from the type system in two places: primitive types and operation invocation.

Primitive types

We realized that a few primitive types must behave correctly for the language to be usable. In particular, consider the Boolean type. The correctness of the **if** statement and **while** loop depend on the proper behaviour of the two Boolean objects **true** and **false**[5]. We therefore insisted that a few types would not follow the normal rules for conformity: no non-primitive type conforms to Boolean. Eventually, we extended this notion for performance as well as correctness and defined a collection of primitive object types that included *Boolean*, *Character*, *Integer*, *Real*, *String*, and *Vector*. When the compiler knew that a variable had a primitive type, it also knew the implementation of the object bound to the variable, and used the direct object implementation shown in Figure 9. This meant that operations on such objects could be inlined and be made as efficient as in a conventional language.

Operation Invocation

A performance problem plaguing object systems that were contemporary with Emerald was the cost of finding the code to execute when an operation was invoked on an object. This process was then generally known by the name "method lookup"; indeed it still is, but we in the Emerald team called it operation invocation. In Smalltalk, method lookup involved searching method dictionaries starting at the class of the target object and continuing up the inheritance class hierarchy until the code was located. We thought that if Emerald didn't do static type checking, each operation invocation would require searching for an implementation of an operation with the correct name, which would be expensive — although, because we did not provide inheritance, not as expensive as in Smalltalk. In a language like Simula in which each expression had a static type that uniquely identified its *implementation*, each legal message could be assigned a small integer and these integers could be used as indices into a table of pointers to the code of the various methods. In this way, Simula was able to use table lookup rather than search to find a method (and C++ still does so). We though that static typing would give Emerald the same advantage, and this was one of the motivations for Emerald's static type system.

However, even with static typing, there is still a problem in Emerald: except for the above-mentioned primitive types, knowing the type of an identifier at compile time tells us *nothing* about the implementation of the object to which it will be bound at run time. This is true even if the program submitted to the compiler contains only a single implementation that conforms to the declared type, because it is al-

[5] Even in Smalltalk, in which conditional statements are represented by message sends, messages such as ifTrue:ifFalse: are known to the compiler and treated specially; it is not in practice feasible to re-implement Boolean.

ways possible for another implementation to arrive over the network from some other compiler. Thus, the Emerald implementation would still have to search for the appropriate method: the only advantage that static typing would give us would be a guarantee that such a method existed.

It is often the case that *dataflow analysis* can be used to ascertain that an object has a specific concrete type, and the Emerald compiler used dataflow analysis quite extensively to avoid method lookup altogether, by compiling a direct subroutine call to the appropriate method. However, the point that we did not fully appreciate when we started the Emerald project was that static typing, in itself, would *not* help us to avoid method lookup.

In those cases where dataflow analysis could not assign a unique concrete type to the target expression, we avoided the cost of searching for the correct method by inventing a data structure that took advantage of Emerald's abstract typing. This data structure was called an *AbCon*, because it mapped *Ab*stract operations to *Con*crete implementations. The run-time system constructed an AbCon for each $\langle type, implementation \rangle$ pair that it encountered. An object reference consisted not of a single pointer, but of a pair of pointers: a pointer to the object itself, and a pointer to the appropriate AbCon, as shown in Figure 3.

The AbCon was basically a vector containing pointers to some of the operations in the concrete representation of the object. The number and order of the operations in the vector were determined by the abstract type of the variable; operations on the object that were not in the variable's abstract type could never be invoked, and so they did not need to be represented. In Figure 3, the abstract type *InputFile* supports just the two operations *Read* and *Seek*, so the vector is of size two, even though the concrete objects assigned to f might support many more operations. AbCon vectors were created where necessary when objects were assigned to identifiers of a different type, and were cached whenever possible to avoid recomputing them. AbCons increased the cost of each assignment slightly, but made operation invocation as efficient as using a virtual function table. In practice it was almost never necessary to generate them during an assignment, because the number of different concrete types that an expression would take on was limited, often to one. We compare AbCons with more recent technologies in Section 6.4.

4.2.6 Type:Type

As we mentioned in Section 3, the occasional need to defer type checking until run time implied that types would have to be representable at run time. Because Emerald was object-based, it seemed like an obviously good idea that types should themselves be represented as objects. The alternative would be to increase the size of the language dramatically by providing one set of declaration and parameterization constructs for objects and another parallel set for types. Our minimality goal discouraged full exploration of this alternative. A consequence of this decision was that Emerald's type system would have the Type:Type property, that is, the property of an object being a type would be a type property, just like the property of being an integer or the property of being a set. At this time, the papers investigating Type:Type [36, 77] had not yet been published, and we didn't see Type:Type as a bad thing. Later, we realized that one of the consequences of Type:Type was that Emerald's type system was undecidable: there were certain pathological type checks involving infinite type objects that would not terminate. But this didn't seem to be a real problem either: these infinite type checks would occur only at run time, and the possibility of computations that did not terminate at run time had always been with us.

Once types were objects, the distinction between types and non-types was no longer one of syntax, but one of value. Thus, arbitrary expressions might appear in positions that required types. Such expressions were evaluated by the compiler, resulting in type objects, the *values* of which were used to do type checking. For example, in the declaration

var *x* : *Integer*

the expression *Integer* was evaluated, resulting in an object v. The type system then inspected v (i.e., it looked at v's *value*) in order to assign a type to the identifier *x*. Clearly, the context implied that v's value should be a type, in other words, that $v \diamond\!\!> \mathit{Type}$[6]; if it did not, the compiler signaled an error. Thus, we see that the values of certain objects, called type objects, were manipulated at compile time to do type checking. These same type objects were also available at run time to perform dynamic type checking.

For pragmatic reasons, the compiler restricted the expressions that could appear in a type position to those that were *manifest*. Intuitively, a manifest expression was one that the compiler could evaluate without fear of either nonterminating computations or (side) effects. We could guarantee a computation to be free from effects by insisting that only functions on immutable objects with immutable arguments that returned immutable results were evaluated at compile time. In addition, while it was obviously not decidable whether or not an arbitrary computation would diverge, the compiler placed restrictions on what it was willing to evaluate to guarantee that compilation terminated. It turned out that we never found a need for alternation (**if**) or itera-

[6] The identifier that we initially chose to denote the type of all types was *AbstractType*; we used the keyword **type** to signify the start of a type constructor, a special form of an object constructor that created a type object. Later, we reversed this decision: we used *Type* for the type of all types, and **typeobject** for the special object constructor. In this article we use the more recent syntax consistently; we felt that changing notation part way though the text, although historically accurate, would be unnecessarily confusing.

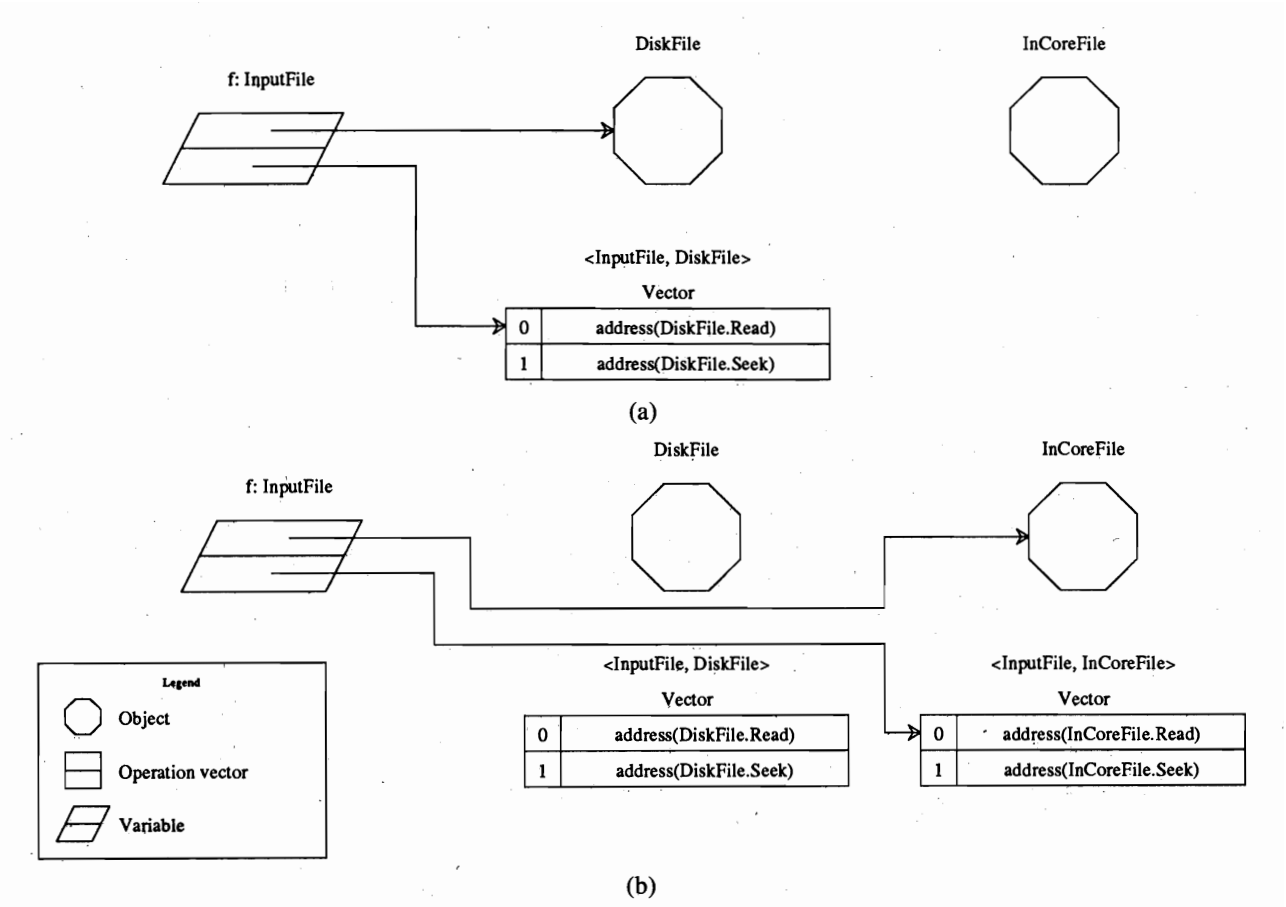

Figure 3: This figure, taken from reference [18], shows a variable *f* of abstract type *InputFle*. At the top of the figure (part a), *f* references an object of concrete type *DiskFile*, and *f*'s AbCon (called an Operation vector in the legend) is a two-element vector containing references to two *DiskFile* methods. At the bottom of the figure (part b), *f* references an object of concrete type *InCoreFile*, and *f*'s AbCon has been changed to a two-element vector that references the correspondingly named methods of *InCoreFile*. (Figure ©1987 IEEE; reproduced by permission.)

tion (**for**, **while**) in our manifest expressions. Thus, when we wrote

var *v* : *Vector.of*[*Integer*]

although *Vector* was just a constant identifier that happened to name a built-in object, and *of* was just an operation on that object, because the objects named by *Vector* and *Integer* were both immutable, and because *of* was a function, the evaluation of the expression *Vector.of*[*Integer*] could proceed at compile time; we knew that the value of this expression would be the same at compile time as it would be at run time.

Once types were first-class objects, and comparison for conformity (rather than equality) was the norm, we realized that it would be a small step to allow an object that was a type also to have additional operations. For example, the object denoted by the predefined identifier *Time*, in addition to being a type, also had a *create* operation that made a new time from a pair of integers representing the number of seconds and μs since the epoch. The Emerald programmer was thus able to define objects that acted as types and *also* had arbitrary additional operations; this was particularly useful for factory objects, which could be both the creators of new objects and their types.

4.2.7 Conformity, nil, and the Lattice of Data Types

The Nature of Conformity. The role of conformity in Emerald was so central that in our early discussions we

treated it as a relation between an object and a type. Other operations on objects could be defined in terms of conformity. For example, to ascertain whether or not an object o possessed an operation f with one argument and one result, we could evaluate:

$o \diamond\!\!>$ **typeobject** T
  operation[7] $f[None^{8}] \rightarrow [Any]$
end T

However, in working with Larry Carter in 1985-6 to formalize the definition of conformity, we realized that conformity needed to be a relation between *types*: the definition of conformity is recursive and depends on the conformity of parameters and results. Consequently, the paper that presents that definition [18] is somewhat inconsistent, referring in places to objects conforming to types, and elsewhere to types conforming to each other.

Unlike languages with which we were familiar, the notion of conformity meant that every object had not one but many types. Referring to Figure 4, any object that had type *ScanableDirectory* also had type *Directory*. *DeleteOnlyDirectory*, *AppendOnlyDirectory* and *Any*, among others. Thus, we found ourselves talking not about "the" type of an object but about its "best-fitting type", meaning the type that captured all of the operations understood by the object. We realized that $\diamond\!\!>$ induced a partial order on types, as depicted in Figure 4. Because in Emerald the type *Any* had no operations, it was the least element in this partial order. Some types were incomparable: the type with *Add* as its only operation seemed to be incomparable to the type with only *Delete*. Nevertheless, these two types had *Any* as their greatest lower bound.

The Partial Order of Types. Although this partial order was for the most part simple and intuitive, we were aware of two problems with it. The first problem, which confused us for a long time, arose when two types had operations with the same name but with different arities, i.e., different numbers of arguments or results. For example, consider two types, one with the operation *Add* [*String*] $\rightarrow$ [ ] (*Add* with a single *String* argument and no result), and one with the operation *Add* [*Integer*] $\rightarrow$ [*Integer*] (*Add* with a single argument and a single result, both *Integer*). Not only are these types incomparable, but they seemed to have no upper bound: there was no type to which they both conformed, because no type could have an single *Add* operation with both of these arities.

A second problem was how to type **nil**, the "undefined" object. We typically wish to use **nil** in assignments:

var d : *Directory*
$d \leftarrow$ **nil**

[7] The syntax **operation** *name*[T] $\rightarrow$ [U] means that the method *name* takes one argument, which is of type T, and returns one result, of type U.

[8] *None* is the type that includes every possible operation, and *Any* is the type that includes no operations, as described later in this Section.

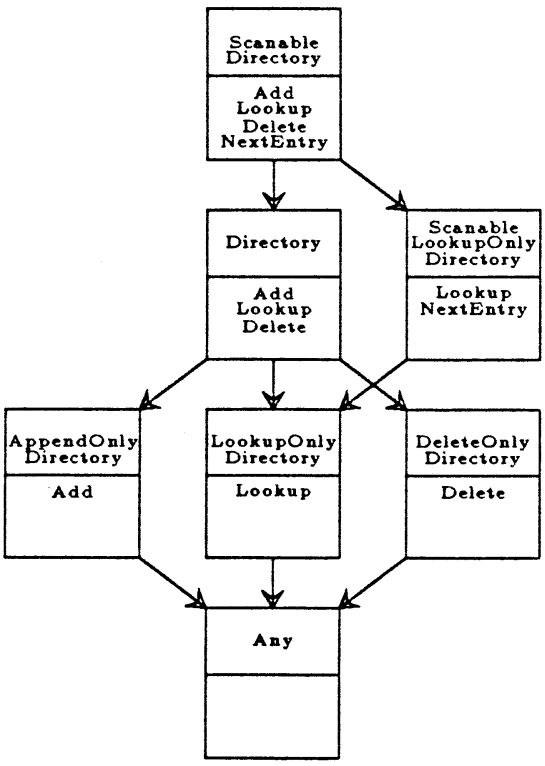

Figure 4: An example of a directed acyclic graph (July 1988) that illustrates the partial order on types induced by conformity. Each box represents a type; above the line is the name of the type, and below is a list of the type's operations. The arrows represent the conformity relation, for example, *ScanableDirectory* $\diamond\!\!>$ *Directory* because it has a superset of *Directory*'s operations.

and in tests:

if $d \neq$ **nil then** $r \leftarrow d.Lookup$[*"key"*] **end if**

It should be clear that for **nil** to be assignable to d, it must support all the *Directory* operations. Similarly, for **nil** to be assignable to **var** i : *Integer* it must support all of the *Integer* operations. By extension, **nil** must possess all of the operations of all possible types that might ever be constructed! This seemed like a contradiction, because operationally we knew that **nil**, far from being an all-powerful object, actually did nothing at all.

The conventional solution to this dilemma, and the one that we initially adopted for Emerald, was to make **nil** a special case. Either **nil** would refer to a single special object that did not otherwise fit into the type system, or, as in Algol 68, there would be a separate **nil** for each reference type, and a syntactic mechanism for disambiguating the overloaded symbol **nil** that denoted them all [103, §2.1 and §3.2].

From Partial Order to Lattice. We eventually stumbled on an elegant solution to both the arity problem and the nil problem. In the partial order so far described, every pair of

types T and U had a "meet" (also known as greatest lower bound) $T \sqcap U$ that contained just those operations that were common to T and U. If T and U had no operations in common their meet was *Any*, so *Any* was the least element in the partial order. If T and U had in common just operation α $[f] \rightarrow [g]$ then the type containing just α with that signature was their meet $T \sqcap U$. However, because of the arity problem, we could not see how to define the "join" (least upper bound) of arbitrary types. When an operation α had different arities in types T and U, the meet of T and U was well-defined (it would omit α completely), but it seemed that the join did not exist.

Meets and joins arose rather naturally when performing type checking. For example, if a variable could take on either a value of type T or a diferent value of type U, all that we could say about the type of that variable is that it is the meet of T and U. Because of contravariance, computing this meet might involve computing a join, for example, if T and U both supported operations α with the same arity but with different signatures, say, α $[f_T] \rightarrow [g_T]$ and α $[f_U] \rightarrow [g_U]$, then $T \sqcap U$ was the type containing α $[f_T \sqcup f_U] \rightarrow [g_T \sqcap g_U]$ — *provided that $f_T \sqcup f_U$ (the join of the f_τ) was defined*. This definition generalized in the obvious way to types with more than one operation, and to operations with multiple arguments and results.

The problem with this definition was the inelegant caveat that the required meets and joins must all exist. Because of contravariance, once some joins did not exist, it was also possible for some meets not to exist: the problem cascaded. Black had studied at Oxford under Stoy and Scott, and knew that the conventional mathematical solution to this problem was to ensure that *all* upper and lower bounds exist, i.e., to embed the partial order in a complete lattice. He also knew that we would not lose any generality by this embeding, because such a lattice always exists [98, pp. 88-91, 414]. However, for a long time we could not see how to construct it, because we could not see what to do about an operation that had different arities in T and U.

The solution, like many good ideas, is quite simple and in hindsight quite obvious: we needed to treat operations with different arities as if they had different names. For practical purposes this meant that we changed the language to allow overloading by arity; formally, we equipped every operation name with two subscripts, representing the number of arguments and results. This meant, for example, that the two add operations mentioned above became $Add_{1,0}$ and $Add_{1,1}$. Now that the operations had distinct names, the upper bound could contain both of them, and as a consequence we were able to turn the partial order into a lattice.

None. Of course, every lattice has a unique top element; what surprised us initially was the discovery that the top element of Emerald's type lattice was semantically useful. We realized that it provided a type for **nil**, so we called it *None*; this also seemed like an appropriate name for the dual of *Any*.

We defined $T \sqcap U$ as the largest type (that is, the type with the largest number of operations) such that $T \circ\!\!> (T \sqcap U)$ and $U \circ\!\!> (T \sqcap U)$, and defined $T \sqcup U$ as the smallest type such that $(T \sqcup U) \circ\!\!> U$ and $(T \sqcup U) \circ\!\!> T$. The type of all objects, which we called *Any*, was then nothing more than the bottom element of the lattice induced by the conformity relation: *Any* was the maximal type such that for all types T, $T \circ\!\!>$ *Any*. Dually, we defined *None* to be the top of this lattice: *None* was the minimal type such that for all types T, *None* $\circ\!\!> T$. The keyword **nil** then simply denoted the (unique) object of type *None*. By this definition, **nil** supported all possible operations, with all possible signatures. In other words, **nil** supported the operation *Add* with 1, 3, and 17 arguments of the most permissive types, as well as the operation *Halt* [*Turing-machine, Tape*] $\rightarrow$ [*Boolean*]. This didn't initially sound like the **nil** with which we were familiar! However, we finally realized that the conformity lattice spoke only about type checking. If any of these all-powerful operations were actually *invoked*, the implementation would immediately break — which is exactly the behavior we expected of **nil**.

Restriction and Capabilities. In Eden, object were addressed using a capabilities that included not only a reference to the object but also a set of access rights that described which operations were invocable using the capability. Applications could create capabilities with restricted access rights and send them to clients. For example, the creator of a mail message object would have the right to perform all operations on it, but might send the recipients restricted capabilities that gave then the right to read the message but not to modify it. The access rights were implemented as a fixed-length bit vector, which provided a small name space and did not mesh well with subtyping [22].

Emerald's type system provided a similar facility, in that clients of an object could be given a reference with a restricted type. However, the **view**...**as** facility meant that the type restriction could always be circumvented. Apart from this fatal flaw, the type-based mechanism was better than Eden's access rights: it resolved the small-name-space problem, and it was compatible with subtyping.

During the summer of 1987, Norm visited Seattle and met Eric at a Lake Washington café. Over breakfast they designed a new language primitive **restrict**...**to**, which removed the flaw by limiting the power of the **view** facility. To see how **restrict** works, consider the code in Figure 5, which references a *Directory* object using the types of Figure 4. In the figure, a single object conforming to *Directory* is bound to *d*, *l*, and *lr*. Both *l* and *lr* have type *LookupOnlyDirectory*, but whereas *l*'s reference is unrestricted, and can be "lifted" back up to *Directory* by a **view** expression, *lr*'s reference is restricted to *LookupOnlyDirectory*. Thus, the view expres-

```
var d, d1, d2 : Directory
var l, lr : LookupOnlyDirectory
d ← ...
l ← d
lr ← restrict d to LookupOnlyDirectory
d1 ← view l as Directory
d2 ← view lr as Directory
```

Figure 5: The **restrict**...**to** primitive can be used to limit the operations invokable on an object. The types *Directory* and *LookupOnlyDirectory* are shown in Figure 4.

sion in the assignment to *d1* will succeed, whereas the view expression in the assignment to *d2* will fail. This is because the restricted reference stored in *lr* cannot be viewed above the level of the *LookupOnlyDirectory* in the type lattice.

The restrict mechanism was implemented by late afternoon of the day on which it was designed. The implementation was simple. AbCons already contained a reference to the concrete type of the object; all that was necessary was to introduce an extra field that specified the largest type allowed in a **view**...**as** expression. Upon creation of an AbCon this field was initialized to the concrete type, but a **restrict**...**to** *R* expression would create an AbCon containing the restricted type *R*.

Overloading. In spite of this rather theoretical genesis, our decision to include overloading by arity but not by type was made because it worked well for the practicing programmer. We now believe that overloading by arity represents a "sweet spot" in the space of possibilities for overloading. It supports the most common uses, such as allowing unary and binary versions of the same operation, and allowing the programmer to provide default values for parameters. Nevertheless, it is always easy for the programmer (and the compiler) to know which version of the operation is being used. In contrast, overloading by type, as provided in Ada at the time of Emerald's development and as now adopted in Java, is simply a bad idea in an object-oriented language. Many programmers confuse overloading with dynamic dispatch; it is difficult for both the programmer and the compiler to disambiguate; and it does little more than encourage the programmer to be lazy in inventing good names, for example, by permitting both *moveTo(aPoint)* and *moveBy(aVector)* to be called *move*. In a procedural or functional language, it is hard to argue that procedures like *print* and functions like $=$ and $+$ should not be overloaded. But in an object-based language, these things are just operations on objects, and several operations with the same name can be implemented by different objects without any need for type-based overloading.

***Implementing* nil.** There was no difficulty in implementing **nil** for objects that were represented by pointers: we just chose a particular invalid address for **nil** whereby we had a free hardware check for invocations of **nil**. However, the attentive reader will recall from Section 4.1.4 that reals and integers were primitive and were represented directly, without using pointers. We wanted to avoid a software-based (and thus highly inefficient) check for **nil** on integer and real operations. We couldn't find a perfectly clean representation for **nil** that was also efficient, and in the end in our implementation of Emerald we cheated just a little. We reserved a special **nil** value that could be assigned to real and integer variables. The bit pattern 0x80000000 was illegal as a VAX floating-point number, so by using this value to represent **nil** we could be sure that a legitimate floating-point operation would never result in **nil**. Moreover, the hardware would trap if this bit pattern were used in a floating-point instruction, so it was easy to detect an attempt to invoke an operation on **nil**. For integers, the same bit pattern represented -2^{31}, so by using this value for **nil** we were "stealing" the most negative value from the range of valid integers. If the program asked whether or not an integer variable was **nil**, the answer would be right. However, we did not implement checks to ensure that invocations on integer **nil** always failed or that normal arithmetic operations didn't generate **nil** by accident. We were not happy with the idea that multiplying -2^{30} by 2 would evaluate to **nil**; neither were we happy that multiplying -2^{30} by 3 would give the wrong answer. The decision not to implement checks for integer overflow and underflow was entirely a matter of efficiency: we wanted built-in integers to be as efficient as C's integers, and C didn't do any checks, so we didn't do any either. Of course, if a programmer wanted a integral numeric type that did all of the checking, or had a greater range, or whatever, such a integer could be implemented as an Emerald object — with the consequent performance penalty. But the built-in *Integer* type was quick, and in this case just a little bit dirty.

4.2.8 Polymorphism

In 1985, we felt that any "modern" programming language should have a type system that provided support for polymorphic operations, that is, operations that might be invoked in several different places with an argument of different types. The contemporary authority on types, a survey paper by Cardelli and Wegner [40], distinguished two broad classes of polymorphism, ad hoc and universal, We were not particularly concerned with ad hoc polymorphism; of universal polymorphism they wrote:

> Universally polymorphic functions work with an infinite number of types, so long as these types share a common structure. As suggested by the name, only universal polymorphism is considered *true* polymorphism. Within this broad class are two sub-divisions:
>
> **inclusion polymorphism**
> An object can be viewed as belonging to many different types that need not be disjoint.
>
> **parametric polymorphism**
> A function has an implicit or explicit type param-

eter which determines the type of the argument for each application of the function.

We found these definitions confusing because the distinction between parametric polymorphism and inclusion polymorphism appeared to break down in Emerald. Consider the passing of an argument *o* of type *S* to an operation that expects an argument of type *T* such that $S \circ\!\!> T$. This could be said to be an example of inclusion polymorphism because the object *o* has a number of different types including *S* and *T* (as well as *Any* and a host of others). An alternative explanation is that, because *o* knows its own type, the type of *o* is an implicit argument to every operation to which *o* is passed, making this an example of parametric polymorphism.

We finally realized that the distinction was still a valid one for Emerald: it lay in the way that the operation *used* its argument. If the operation treated *o* as having a fixed type *T*, then the inclusion polymorphism view was appropriate, but if it treated *o* parametrically, then the parametric view was appropriate (See the typechecking technical report [26] for some examples.)

As a consequence, we included two features in the Emerald type system, one for each of these two kinds of polymorphism. First, basing type checking on conformity rather than equality supported inclusion polymorphism: every operation that expected an argument of type *T* would also operate correctly on an argument whose type conformed to *T*.

Second, Emerald types were values (more precisely, objects), and we allowed arbitrary expressions to appear in syntactic positions that required types. This implied that types could be passed as parameters to operations. An operation could also return type as its result, and that type could then be used in a declaration. Figure 6 is an example of the sort of parametric polymorphism that we provided in Emerald: a polymorphic pair factory that insists that both elements of the pair are of the same type. A trivial application of this object to create a pair of strings might appear as follows.

```
const p ← pair.of[String].create["Hello", "World"]
```

The *of* function of *pair* accepts any type as its argument, but returns a pair that has no operations other than *first* and *second*. Suppose that we want the pair to also have an equality operation = that tests the component-wise equality of one pair against another. Obviously, such a pair cannot be constructed for arbitrary element types: the element type must itself support an = operation. We therefore need to be able to express a constraint on the argument to the *of* operation on *pair*. This can be done by adding a constraint on the *value* of the argument *T* passed to *of*. The constraint that we need is that:

```
T ◇> typeobject eq
    function = [eq] → [Boolean]
end eq
```

```
const pair ← immutable object pPair
    export function of[T : Type] → [r : PairCreator]
        where
            PairCreator ← typeobject PC
                operation create[T, T] → [Pair]
            end PC
        where
            Pair ← typeobject P
                function first → [T]
                function second → [T]
            end P
        forall T[9]
        r ← immutable object aPairCreator
            export operation create[x : T, y : T] → [r : Pair]
                r ← object thisPair
                    export function first → [r : T]
                        r ← x
                    end first
                    export function second → [r : T]
                        r ← y
                    end second
                end thisPair
            end create
        end aPairCreator
    end of
end pPair
```

Figure 6: A polymorphic pair factory (after an example in a November 1988 working paper).

Syntactically, we gave Emerald a **suchthat** clause to capture this constraint. Types constrained in this way provide what Canning and colleagues later called *F-bounded polymorphism* [34] because the bound on the type *T* is expressed as a function of *T* itself. The **where** clause of CLU [73] provided similar power, although we were unaware of this at the time. We eventually realized that the $\circ\!\!>$ relation in the **suchthat** did not denote the conformity relation that we had defined between types, but was actually a higher-order operation on *type generators*. In the types technical report [26] we called this operation matches, and denoted it by the symbol $\triangleright$. Subsequently, Kim Bruce adopted an equivalent definition for an operation that he denoted by $\leq_{meth}$ [32], and eventually also called matches [33].

Two other changes are necessary in Figure 6 to define a pair with equality. The first is, obviously, the addition of the = operation to the type *Pair*; the second is the addition of a corresponding implementation of = to the object *thisPair*. The complete factory for pairs with equality is shown in Figure 7.

[9] The **forall** keyword was added quite late in Emerald's development. In a sense the **forall** is unnecessary, because it expresses the empty constraint on *T*. However, without it there is no declaration for the identifier *T*, and we felt that *every* identifier should be declared explicitly; the alternative of making the programmer write **suchthat** *T* $\circ\!\!>$ *Type* seemed overly pedantic.

```
const pair ← immutable object pPair
    export function of[T : Type] → [r : PairCreator]
        where
            PairCreator ← typeobject PC
                operation create[T, T] → [Pair]
            end PC
        where
            Pair ← typeobject P
                function first → [T]
                function second → [T]
                function = [P] → [Boolean]
            end P
        suchthat T ◇> typeobject eq
            function = [eq] → [Boolean]
        end eq
        r ← immutable object aPairCreator
            export operation create[x : T, y : T] → [r : Pair]
                r ← object thisPair
                    export function first → [r : T]
                        r ← x
                    end first
                    export function second → [r : T]
                        r ← y
                    end second
                    export function = [other : Pair] → [r : Boolean]
                        r ← other.first = x and other.second = y
                    end =
                end thisPair
            end create
        end aPairCreator
    end of
end pPair
```

Figure 7: A Pair Factory that is polymorphic over types that support equality

4.2.9 Publications on Types

Our initial ideas on types were published in an August 1985 technical report [19]. However, that report did not contain a formal definition of the conformity relation, not because we thought it unimportant, but because we were not sure how to do it. The obvious definition of conformity was as a recursive function, and we did not know how to ensure that the recursion was well-founded. Fortunately, Larry Carter from IBM was on sabbatical at UW in the fall of 1985, and we were able to convince him to help us formulate the definition. Larry's definition, which constructed the conformity relation as the limit of a chain of approximations, appeared in a revised version of the technical report [16], and was eventually published in *Transactions on Software Engineering* in January 1987 [18].

As we deepened our understanding of the issues around types, and in particular looked harder at polymorphism, we revised a number of the decisions made in these early papers and developed more formal underpinnings for the type system. We also redefined conformity using inference rules, a technique that was then becoming popular. Andrew and Norm were responsible for this evolution and for authoring several documents describing it, none of which was formally published. Our earliest document is a handwritten paper "The Lattice of Data Types", dated 1986.11.04. Sometime in 1987 this paper acquired a subtitle "Much Ado About NIL"; it eventually evolved into "Types and Polymorphism in the Emerald Programming Language", which went through a series of versions between July 1988 and July 1989, before finally morphing into a technical report "Typechecking Polymorphism in Emerald", which Andrew and Norm finished over the Christmas holidays in 1990 [26].

4.3 Concurrency

We knew that concurrency was inherent in any distributed system that made use of multiple autonomous computers. Moreover, we realized that even in a non-distributed program, Emerald's object model implied that each object was independent of all others and was capable of acting on its own. This meant that every object should be given the possibility of independent execution; the consequence was that we needed a concurrency model that allowed concurrency on a much finer grain than that provided by operating system processes in separate address spaces.

4.3.1 The Emerald Process Model

In designing Emerald's process and concurrency control facilities, we were inspired by Concurrent Pascal [29], with which Eric had worked extensively while writing his Master's thesis [57] at the University of Copenhagen, and Concurrent Euclid [50, 51], which we had used in Eden. Note that, as we explained on page 4, we use the unqualified term *process* to mean what it meant at the time: one of a number of lightweight processes sharing an operating system address space. Today this would more likely be called a *thread*; when we mean an operating system process (in a protected address space), we say so explicitly.

In Eden, each object was a full-blown UNIX process and thus had its own address space, within which the Eden Programming Language provided lightweight processes. The two levels of scheduling and the costs of operating system interprocess communication inherent in this scheme were part of the cause of Eden's poor performance. So, for Emerald, we knew that we needed to find a way to develop a single process model for all concurrency. We also knew that we would have to implement it ourselves, rather than delegating that task to an underlying operating system.

Our first idea was to have a separate language construct to define a process, modeled on the **process** of Concurrent Pascal. However, we we soon realized that this would not be adequate. The Concurrent Pascal construct is static: all the processes that can ever exist must be defined at compile

time. This restriction could be lifted by making the creation of processes dynamic, for example, by allowing a **new process** construct similar to **new** in Simula. However, we were not happy about the idea of introducing another first class-citizen besides objects.

Object constructors had let us avoid introducing classes as first-class citizens: instead we nested one object constructor inside another. We realized that we could also nest processes *inside* objects. We added an optional **process** section to object constructors so that, when an object was created, a process could be created as one of the object's components. The process would start execution as soon as the object had been initialized. If the process section were omitted, the object would be passive and would execute only in response to incoming invocations. This design allowed us to have active objects that could execute on their own, as well as conventional passive objects. It also had the benefit of making it clear where a program should start executing. Some languages express this with various kinds of ad-hocery, such as a "special" process called "main". Emerald needed no "special" process. Instead, we saw the Emerald world as a vast sea in which objects floated. When a program created a new object (by executing an object constructor), the object was merely added to the sea. If the new object had no process, nothing more happened (until the new object was invoked). If it did have a process, the process started execution concurrently with all the other Emerald processes.

Thus, instead of needing a "main procedure", an Emerald program was simply a collection of object declarations. When the program was loaded, these declarations were elaborated, and any objects thus created were added to the sea. A sequential Emerald *Hello, world!* program looked like this.

```
const simpleprogram ← object myMainProgram
    const i ← 1
    process
        stdout.PutString[i.asString || ": hello, world!\n"]
    end process
end myMainProgram
```

This program uses a simple object constructor to create an essentially empty object whose only purpose is to house the process that prints "1: hello, world", and then terminates.

We also considered allowing more than one **process** section in an object. This would have been of limited use: as a static construct, it would not have helped the programmer to create a variable number of processes in the object. The more general case of dynamically created processes was already provided: any number of processes could be created simply by defining internal objects that contained not much more than a process, and then creating such objects. This is illustrated in the example below.

```
const p ← object multiProgram
    export operation workerBody[i: Integer]
        stdout.PutString[i.asString || ": hello, world!\n"]
    end workerBody
    process
        var i: Integer ← 17
        var x: Any
        loop
            if i <= 0 then exit end if
            x ← object worker
                    process
                        multiProgram.workerBody[i]
                    end process
            end worker
            i ← i − 1
        end loop
    end process
end multiProgram
```

This program contains a single object that creates 17 other *worker* objects each housing a process. These 17 processes thereafter execute in parallel and will print a message in some (indeterminate) order.

The following example shows a generalized worker process that is parameterized by a function containing the actual work to be done.

```
const WorkToDoType ← typeobject wtd
    operation doWork[ ]
end wtd

const workerCreator ← object wc
    export operation createWorker[work: WorkToDoType]
        var x: Any
        x ← object worker
                process
                    work.doWork[ ]
                end process
        end worker
    end createWorker
end wc

const exampleWork ← object hello
    var i: Integer ← 0
    export operation doWork[ ]
        i ← i + 1
        stdout.print[i.asString || ": hello world! \n"]
    end doWork
end hello

const exampleProgram ← object exampleProgram
    process
        workerCreator.createWorker[exampleWork]
        workerCreator.createWorker[exampleWork]
    end process
end exampleProgram
```

The *workerCreator* operation is used to generate a process that executes the *doWork* operation of any object given to it as argument. In this example, the *hello* objects merely print messages. The object *exampleProgram* creates two worker

processes; the order in which the messages are printed is undefined because the two processes execute concurrently.

4.3.2 Synchronization

In general, many processes could be executing inside an object simultaneously. However, if they updated the same variables, race conditions could readily occur. Traditionally, such problems were resolved by protecting the shared variables with some form of synchronization construct. The final example in Section 4.3.1 contains a serious race condition in the object *hello* over the update of the variable *i* in the *doWork* operation. Because it was easy to write programs with unwanted race conditions, we provided a way for the programmer to state that an object's variables would be protected inside a monitor; the externally visible operations on that object would then become the monitor entry operations.[10] Thus the *hello* object in the example becomes:

```
const exampleWork ← monitor object hello
    var i: Integer ← 0
    export operation doWork[ ]
        i ← i + 1
        stdout.PutString[i.asString || ": hello world! \n"]
    end doWork
end hello
```

We did not spend a lot of time considering alternatives to the monitor. Innovation in concurrency control was not a goal; we were familiar with monitors, they were known to be adequate, and they were widely taught and understood (although some of Black's prior work [1] indicated that they were not as well understood as we had thought!) We adopted Hoare semantics [49] for monitors, including the facilities for signaling and waiting on so-called condition variables. However, condition variables themselves posed a bit of a problem. In Hoare's original design, condition variables could be declared only inside a monitor, and consequently had a scope that was limited to the enclosing monitor. It was impossible to export a reference to a condition variable, and it made no sense to wait on a condition variable in one monitor and signal it in another. So conditions seemed to be much less general than objects, which were always known by reference and which could be passed around freely.

We initially followed Hoare's approach and defined a special system object, *Condition*, that returned a unique condition variable when its *create* operation was invoked. However, in line with our minimality goal (Section 2), we really wanted to avoid introducing any kind of special variable into Emerald. We realized that condition variables were not really variables at all in the conventional sense: they did not refer to values. The purpose of a condition variable was merely to serve as a label for the logical condition for which a process might need to wait. Consequently, we realized that any kind of unique label would do. To avoid creating another concept, that of labels, we decided to use an arbitrary object as a label; after all, every object had a unique identity. Thus, in **signal** and **wait** statements, we allowed any object to be used to label the condition.

However, this simplification gave rise to another problem. If condition variables were not "special", how were we to enforce the restriction that a particular condition could be used within only a single monitor? Although this restriction would naturally follow from the practice of declaring the condition object locally within the monitor, this practice did not amount to a guarantee. Because condition objects were now just general-purpose objects, they could be stored inside other objects, passed as arguments, and so on: it would be impossible to limit their scope statically. We therefore decided to enforced the "single monitor" restriction dynamically. The implementation of condition variables created the structure that implemented the condition the first time that either **signal** or **wait** was applied to an object, and associated the condition structure with the enclosing monitor at that time. Subsequent applications of **signal** or **wait** checked that their condition argument had been associated with the same monitor. Because **signal** and **wait** were language statements, not operation invocations, it was trivial to ensure that they appeared only inside a monitor.

One additional advantage of using the identity of the object and not its representation was that this avoided any complications concerning distribution: nonresident objects still had resident object descriptors, so **signal** and **wait** never needed to access nonlocal data structures.

To help programmers express their intent, we retained the special system object *Condition*. A *Condition* object was an empty object without operations: *Conditions* were never invoked, but were merely used for their unique identity in **signal** and **wait** statements. In this way, we introduced the concept of a first-class label without making the language larger: labels were simply objects.

The code in Figure 8 shows two processes keeping in step with one another by alternating their execution. Each process executes an operation 10 times. They synchronize through a monitor, so that if one gets ahead of the other, it will have to wait its turn. The *Condition* object *c* represents the condition "it's my turn now"; the operations *Hi* and *Ho* **wait** on and **signal** *c*, but *c* is never invoked.

Monitors did not present any significant challenge related to mobility. The implementation structure for each monitor, including the monitor lock and any conditions, was packed up and moved along with its enclosing object. Processes that were waiting for entry into the monitor, either because they had invoked a monitored operation or because they

[10] We initially allowed any object to contain a monitored section. We later decided to instead make a whole object monitored; this simplified both the language and the compiler. The effect of an internal monitor could still be obtained with a nested object. For the sake of consistency, all of the examples in this paper use the current (whole object as monitor) syntax.

```
const initialObject ← object initialObject
    const limit ← 10
    const newobj ← monitor object innerObject
        var flip : Boolean ← true      % true => print hi next
        const c : Condition ← Condition.create
        export operation Hi
            if ! flip then
                wait c
            end if
            stdout.PutString["Hi\n"]
            flip ← false
            signal c
        end hi
        export operation Ho
            if flip then
                wait c
            end if
            stdout.PutString["Ho\n"]
            flip ← true
            signal c
        end ho
        initially
            stdout.PutString["Starting Hi Ho program\n"]
        end initially
    end innerObject
    const hoer ← object hoer
        process
            var i : Integer ← 0
            loop
                exit when i = limit
                newobj.Hi
                i ← i + 1
            end loop
        end process
    end hoer
    process
        var i : Integer ← 0
        loop
            exit when i = limit
            newobj.Ho
            i ← i + 1
        end loop
    end process
end initialObject
```

Figure 8: One of the processes in the object *hoer* invokes the *Hi* operation on *newobj* 10 times; the other invokes *Ho*. Because these operations execute inside a monitored object, they operate in mutual exclusion and the output is an alternating stream of *Hi* and *Ho* messages.

were waiting on a condition, were moved just like any other process that was executing (or waiting) within an object.

4.4 Initially

In many languages, initializing variables and data structures was a bothersome task. In Emerald, the problem was further compounded by concurrency: once created, a process ran in parallel with its creator. Consequently, race conditions could occur when creating a new object. For example, if a process P created a new object A, and during its creation A caused another process Q to be created, then Q might "outrun" P and try to invoke the new object A *before* P had finished initializing A. An object did not even need to create another process for a race condition to occur: if a new object A registered itself in a directory so that others could find it, then an aggressive process that noticed A in the directory might try to invoke A before A had finished its initialization. The same problem exists today in Java.

We solved this problem by locking an object until its **initially** section had completed. This enabled the body of the **initially** to use other objects freely, but a cycle would result in deadlock and would thus be easy to detect.

4.5 Finalization

Some object-based languages allowed the programmer to define so-called finalizers, also known as destructors. The idea was that just before an object was destroyed, its finalizer would be given a chance to "clean up", for example, to close open files or to release allocated data structures. In our minds, objects lived forever, so a finalizer did not make sense. The garbage collector could recycle objects that were no longer of any use — which meant that they were not accessible from a basic root or by an executing process. We did consider introducing a finalizer that would be invoked in this situation, but once something was executing inside the object, it would no longer be a candidate for garbage collection. So finalizers would have violated an important monotonicity property: once an object became garbage, it would stay garbage.

4.6 Compiler-Kernel Integration

The Emerald compiler and run-time kernel were very tightly integrated (see Section 3). This was essential for accomplishing our performance goal. Tight integration allowed the compiler several forms of flexibility: it could select between the three object implementations (global, local, and direct, described in Section 4.1.4) for every object reference; it could use the general purpose registers to hold whatever data it liked; and it understood the format of kernel-maintained data structures and could inspect them directly, rather than calling a kernel primitive to interpret them.

The compiler was responsible for informing the kernel about its representation choices, and because the kernel could take control at (almost) any point in the execution and might need to marshal object data, the run-time stack, and even the processor registers, the compiler had to provide descriptions of every accessible data area at all times. These descriptions, called *templates*, described the contents of an area of memory. They informed the run-time system where immediate data (direct objects), object pointers (local object

references), and object descriptor references (remote object references) were stored. A particular template described either an object's data fields or the contents of the processor registers and stack for a single activation record. Because the stack contents varied during the lifetime of an activation record we considered dynamic templates that would be a function of the program counter, but we avoided this complexity by ensuring that the variable part of the stack *always* contained full Emerald variable references, including AbCon pointers. The same templates provided layout information for the garbage collector, the debugger and, not the least, the object serializer. The object serializer was capable of marshaling *any* object, including those containing processes and active invocations, using the compiler-generated templates. This meant that, unlike other RPC-like systems (e.g., Birrell and Nelsons' RPC [14]) there was no need for the programmer to be involved in serializing objects. Completely automatic serialization was also necessary because we did not give programmers access to low-level representations.

Fast Path and Object Faulting Allowing compiled code to inspect kernel data structures was key to making local invocations of global objects as fast as procedure calls in C. The implementation of an invocation followed either a "fast path" or a "slow path". The fast-path invocation code sequence generated by the compiler checked a *frozen* bit in the object descriptor. If the object was not frozen, then the fast path code was free to construct the new activation record and jump to the appropriate method code. However, if the object was frozen, then the compiler-generated code took the slow path by calling into the kernel to perform the invocation. There were a large number of situations in which an object was frozen: it might have been still under construction, it might not have been resident on the machine, it might have failed, it might have been been in the process of being moved, or it might need to be scanned by the garbage collector. The compiler-generated invocation sequence needed to identify whether or not it could use the fast path: all of the slow paths started with a call into the kernel that then ascertained which of the various special cases applied. We called this mechanism *object faulting* because it was similar to page faulting on writes to read-only pages. Even without hardware support the object-faulting mechanism was quite efficient and was crucial for the implementation of the parallel, on-the-fly, faulting garbage collector (seen Section 5.3).

4.7 Mobility

Although we could not know it at the time, the major advance of Emerald over most of its successors was that Emerald objects were mobile. Given our experience with Eden, mobile objects were the obvious way of meeting our goal of explicit support for object location.

Although operation *invocation* was location independent in Emerald, it was never a goal that *objects* should be location independent. Indeed, we recognized that some objects, particularly those that needed to exploit hardware, would need to be placed on particular machines. We also thought that automating the placement of objects in a distributed system was in general too hard; instead we felt that it was the responsibility of the application programmer to place objects appropriately, given his or her knowledge of the application domain. We therefore gave Emerald a small number of location-dependent primitives, extending what we had done in Eden.

4.7.1 Location Primitives

Location was expressed using *Node* objects, each of which was an abstraction of a physical machine. For example, if *Y* were a node object, the statement **fix** *X* **at** *Y* locked the object *X* at location *Y*. However, *Y* was not restricted to be a *Node*; any object could be used as a location. So, if *X* was a mail message and *Y* a mail box, **fix** *X* **at** *Y* was still valid, and meant that *X* should be locked at the current location of *Y*. Andrew had first thought of the idea of using arbitrary objects to represent locations when designing Eden's location-dependent operations; the idea had worked well, so we adopted it for Emerald.

Emerald had five location-dependent operations and two special parameter passing modes that influenced location. The location dependent operations were as follows.

- **locate** an object; the answer was a *Node* object that indicated where the target resided. There was no guarantee that the object might not move immediately afterwards.
- **move** an object **to** another location.
- **fix** an object **at** a particular node; this might have involved moving it there first. An attempt to fix an object would fail (visibly) if, for example, the target object were already fixed somewhere else. An **isfixed** predicate was also provided.
- **unfix** an object: make it movable again after a **fix**.
- **refix** an object, that is, atomically **unfix** and **fix** an object at a new place.

The **move**...**to** primitive was intended to be used for enhancing performance by colocating objects, and thus reducing the number of remote invocations between them. In contrast, **fix** was intended for applications where the location of an object was part of the application semantics. For this reason, we gave **move** weak semantics: a move was treated as a performance hint. The kernel was not obliged to perform the move if a problem was encountered with it; if, for example, the destination node were unavailable, **move** would do nothing silently. Moves were also queued, so that multiple move requests following one another would usually be batched, which in many cases gave us a huge (order of magnitude) performance improvement. Even if the move

succeeded, there was no guarantee that the object concerned might not immediately move somewhere else.

The **fix**, **unfix**, and **refix** primitives were designed for use when location was part of the application semantics, as when trying to achieve high availability by positioning multiple replicas on different machines, or when implementing a program that performed load sharing (an Emerald load sharing program was written later [68]).

We therefore gave **fix** much stronger semantics and implemented **fix** transactionaly, so that after a successful **fix** the programmer could be sure that the object concerned was indeed at the specified location. An attempt to **move** a fixed object would fail, to avoid potential confusion as to whether **move** or **fix** had priority.

4.7.2 Moving Parameter Objects

As early as the *Getting to Oz* memo of April 1984 [69], we had decided that call by object reference was the only parameter passing semantics that made sense for mutable objects; this was the same semantics used by CLU and Smalltalk. However, we were also aware that call by reference could cause a storm of remote invocations, and for this reason invented two special parameter-passing modes that we called *call by move* and *call by visit*. The memo continues: "a new parameter passing mechanism we've considered is *call by move*, in which the invoked operation explicitly indicates that the parameter object should be moved to the location of the invoked object." We saw call by move as giving us the efficiency of call by value with the semantic simplicity of consistently using call by reference. However, call by move was not always a benefit; although it co-located a parameter with the target object, it would cause any invocations from the call's initiator to the parameter object to become remote, which could drastically reduce performance. The cost of the call would also be increased, albeit for smaller objects (less than 1 000 bytes) the cost was about 3.5% for call by move and 7% for call by visit [58, p. 131].

The Emerald compiler decided to move some objects on its own authority. For example, small immutable objects were always moved, because in this case the cost was negligible. In general, however, application-specific knowledge was required to decide if it was a good idea to move an object, and Emerald provided **move** and **visit** keywords that the programmer used to communicate to the compiler that a parameter should be moved along with the invocation. The use of these keywords affected locatics but not semantics: the parameter was passed by reference, as was any other parameter, but at the time of the call it was relocated to the destination node. The difference between **move** and **visit** was that, after the invocation had completed, a call-by-visit parameter was moved back to the source node when the invocation returned, whereas a call-by-move parameter was left at the destination.

Call by move was both a convenience and a performance optimization. Without call by move, it would still have been possible to request the move (using the **move**...**to** primitive), but that would have required the programmer to write more code and would not have allowed the packaging of parameter objects in the same network message as the invocation. There was also a *return by move* for result parameters.

4.7.3 Implementation

Whereas Eden objects had been implemented as whole address spaces, Emerald objects were data structures inside an address space, and so most of the implementation techniques that we needed had to be invented from scratch. A guiding principle was not to do anything that sped up mobility at the expense of slowing down local operations: the costs of mobility should rest on the applications that used it.

Moving the data structure representing an object was not conceptually difficult: we just copied it into a message and sent it to the destination machine. However, all of the references to objects in that representation were local pointers to object descriptors, and had to be translated to new pointers at the destination. To make this possible the kernel had to be able to distinguish object references from direct objects, which was achieved by having the compiler allocate all of the direct objects together, and putting a template in the code object that specifies how many direct objects and how many object references were in the object data area. A table that translated object references to *OID*s was appended to the representation of the object, and the kernel at the destination used this translation table, in combination with its own object table, to overwrite the now-invalid object references with valid pointers to local object descriptors.

Emerald objects contained processes as well as data; this included their own processes and processes originating in other objects whose thread of control passed through the object. Whereas each object was on a single machine, Emerald processes could span machines. This could occur either because of a remote invocation, or because an object moved while a process was executing one of its operations. We realized that if we treated the execution stack of a process as a linked list of activation records, and each activation record as an object referenced by a location-independent object reference, then everything would "just work": processes returning from the last operation on one machine would follow a remote object reference back to the previous machine.

Of course, in reality things were not quite so simple. Each activation record was not a separate object; instead, we allocated stack segments large enough to accommodate many activation records (the standard size was 600 bytes), and linked in a new segment only when an existing one filled up. This meant that when an object moved, we might have to split a stack segment into three pieces and then move the middle one. Also, stack segments were more complicated

that ordinary object data areas, for example, because they contained saved registers, and because the data stored in the activation record was constantly being changed by the running process.

An alternative to moving activation records with the objects to which they referred would have been to leave them in place until the executing process returned to them. We chose not to do this because it would have set up residual remote dependencies: if the machine hosting the activation record was unavailable, the executing process would be "stuck". Although this might sound like an unlikely scenario, we thought that it would actually be quite common to move all the objects off of a machine so that it could be taken down for maintenance. If, in spite of moving all of the objects, the activation records were left behind, the computation would still be dependent on the machine that was down.

Finding out which activation records to move when an object migrates is a little bit tricky. Although each activation record has a context pointer linking it to an object, the object does not normally have a pointer back in the other direction. We considered adding one, linking a list of activation records from each object, but that would have almost doubled the cost of operation invocation. However, searching through every activation record whenever an object moved would have been prohibitively expensive. The compromise we adopted was to link objects to their activation records only when there was a context switch from one process to another. After all, it was only during a context switch that an object could move; in between two context switches many thousands of activation records would have been created and destroyed without any overhead.

We used a lazy technique to ascertain what data was on the current stack when a process was moved. This technique added no overhead to normal execution, placing it all on the move. The code object contained a *static template* that described the format of the current activation record for any given code address. As the name implies, static templates could be generated at compile time. We originally thought we would need a different template for each point in the code that changed the content of the stack, leading to a large number of templates. To avoid this we devised what we called *dynamic templates*, which described the *change* in the contents of the stack. However, Norm examined the problem carefully and found out that most temporaries pushed onto the stack could just as well be stored into temporary variables *preallocated* on the stack; because these variables could be reused throughout an operation, this caused no change to the stack layout and thus no change to the template. Moreover, there was no real storage cost: the stack needed to accommodate only the maximum number of temporaries simultaneously in use at any point in the operation.

Any other temporaries that were pushed onto the stack were full object references that included a pointer to the AbCon, and so were self-describing. Thus, template information was not needed for these variables, and so dynamic templates were unnecessary. One static template, laid down by the compiler, was sufficient.

4.8 Failures

Emerald did not include a general-purpose exception handling mechanism. This was largely because Andrew was opposed to such mechanisms, and also because designing a good one would have distracted us from our goals. Because of Andrew's thesis research [20], we were aware of the distinction between exceptions — special return values explicitly constructed by the program in known situations — and failures — which occurred when programs went wrong.

An Emerald **begin** ... **end** block could be suffixed with a failure handler that specified what to do if one of the statements in the block failed, e.g., by asserting a false predicate, dividing by zero, or indexing a vector outside of its bounds. We did not regard a failure as a control structure for deliberate use, but as a bug that should eventually be fixed, and in the meantime survived. This meant that if there were some question whether or not an index was within the bounds of a vector, we expected the programmer to test the index before using it, rather than to have the indexing operation fail and then "handle" the failure. For this reason, Emerald did not include a mechanism for distinguishing between different kinds of failure.

If a failure occurred and there was no failure handler on any block that (lexically) contained the failing statement, the failure was propagated back along the call chain until a failure handler was found. Along the way, each object on the call chain was marked *failed*; this meant that any future attempt to invoke that object would fail. The language also provided a **returnandfail** statement so that an operation called with invalid parameters could cause its *caller* to fail without the invoked object itself failing.

4.9 Availability

As noted in Section 2, it was an explicit goal of Emerald to accommodate machine and network failures. At a given time, each mutable Emerald object was located on exactly one machine. Thus, if a machine crashed, the objects on it would become unavailable, and it would be temporarily impossible for another object to invoke them. We saw unavailability as a common and expected event, and felt that distributed programs had to deal with it, so we provided a special language mechanism to handle unavailability.

Our view was that unavailability was quite different from failure. The availability of an object was not like the property of an index being within bounds: programmers could not test for availability before making each invocation, because availability was a transient property of the environment rather than a stable property of the program state.

Emerald allowed programmers to attach an *unavailable handler* to any block. This handler specified what to do when, due to machine crashes or communication failures, an invocation could not be completed or an object could not be found. For example, if an object tried to invoke a name server and that name server was unavailable, it could try a second name server:

```
begin
    homeDir ← nameServer1.lookup[homeDirectoryName]
end
when unavailable
    homeDir ← nameServer2.lookup[homeDirectoryName]
end unavailable
```

An *unhandled* unavailable event was treated as a failure. So, if *nameServer2* were unavailable, the invocation of its lookup operation would fail and the object containing the above code would also fail.

4.10 Kernel Structure and Implementation

As mentioned in Section 1.5, our use of the term *kernel* followed the tradition of Concurrent Pascal and Eden. In all these systems the term meant the run-time support software that was responsible for loading programs, managing storage, performing I/O, creating and administering processes, and performing remote invocations. The Concurrent Pascal kernel actually ran on a bare machine. The Eden kernel was a UNIX process, and implemented each Eden object as another UNIX process; this led to excessive storage consumption (minimum size of an object was 300 kBytes) and execution time (invocations between Eden objects on the *same* machine took on the order of 100 ms). We wanted substantially less overhead both in storage and execution time, so we decided to implement the kernel in a single address space as a single UNIX process within which all activity remained. This saved us from expensive UNIX process boundary crossings and allowed us to make object invocations almost as fast as procedure calls in C. We handled our own storage allocation, so we were in complete control of storage layout; this let us implement mobility and prepared the system for garbage collection.

The kernel was written in C, which we considered to be an advanced assembler language: it allows detailed and efficient access to data structures including the execution stacks and let us build an invocation mechanism optimized for performance. As mentioned in Section 3, we wrote stylized C code designed to generate assembly language programs that were as efficient as hand-coded assembler, but with the advantage that portability was subtantially better. Portability was a concern to us. One of the reasons that Eden had not seen any use outside the University of Washington was that it was not portable: Eden required a modified version of the 4.1 BSD UNIX kernel and included much assembler code. We wanted Emerald to see wider use, so we strove to minimize dependence on assembly code and programmed using as generic a subset of the UNIX API as possible.

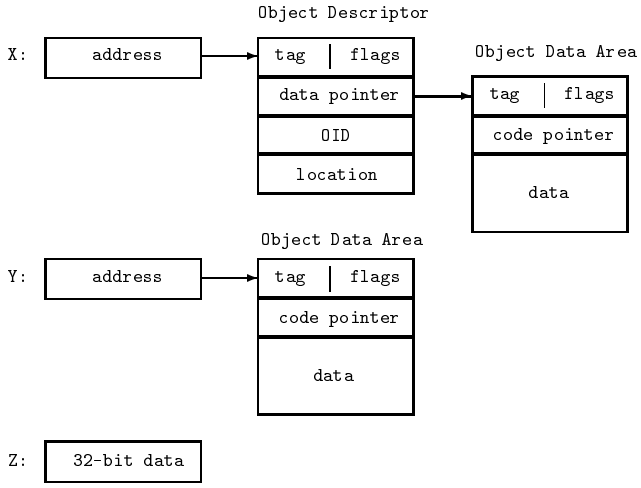

Figure 9: Emerald object layouts from Eric's thesis [58]. `X` is the two-level structure used for *global* objects, `Y` is the one-level structure used for *local* objects, and `Z` shows a *direct* object.

4.10.1 Object Layout and Object IDs

In the language, each object could potentially be accessed from a remote machine. So, in principle, every object required a globally unique object identifier, known as an *OID*. Because Emerald was developed on a local area network, we initially implemented *OID*s as 8-bit machine numbers (we used the bottom 8 bits of the machine's IP address) concatenated with 24-bit sequence numbers. In a subsequent wide-area network version of Emerald, the *OID* was expanded to be the full-32 bit IP address and a 32-bit sequence number [80]. To avoid wasting storage and *OID* space on objects that would never be referenced remotely, we did not actually allocate an *OID* until a reference to the object was exported across the network.

We expanded the use of *OID*s to other kernel data structures, which made those structures remotely addressable using the same mechanism as for objects. For example, the structure containing the code for an object was given an *OID* by the compiler (see Section 4.10.4). For this reason, in this section on kernel implementation we use the term object to mean the representation of either a real Emerald object or an object-like kernel data structure.

Figure 9 shows the layout of kernel data. There were two representations for non-primitive objects: a two-level scheme (`X`) for global objects and a one-level scheme (`Y`) for local objects. The first word of every kernel data area had a standard format: a tag that identified the data area and a number of tag bits, including the frozen bit and an indication of the representation scheme. The two-level storage scheme represented an object by the local address of an *Object De-*

scriptor. The Object Descriptor indicated whether or not the object was locally resident. If it was, the local address of the *Object Data Area* was stored in the Object Descriptor. If the object was not resident, the Object Descriptor contained the *OID* for the object and a hint as to the current location of the object. In the one-level representation, the Object Descriptor and Object Data Area were merged together; we did this to promote efficiency in access, allocation and storage space, but only for objects that could never be accessed remotely.

Each *OID* was entered into a hashed *object table* that mapped *OID*s to Object Descriptors. This meant that an *OID* arriving over the network from another machine could be mapped to the corresponding Object Descriptor, if one already existed. This made it easy to ensure that a given object had no more than one Object Descriptor on each machine. For mutable objects, only one of the Object Descriptors in the whole Emerald system would normally reference an Object Data area. Moving an object from one machine to another therefore meant copying the Data Area from one machine to the other, and then changing a single address in the Object Descriptor on the source and destination machines. While the object was being moved, it would have a data area on both machines, so the object was flagged as *frozen* to prevent the object's operations from modifying the data. Immutable objects could be replicated; they were represented by an Object Descriptor *and* an Object Data Area on each machine where there was a reference to the object.

4.10.2 Kernel Concurrency: Task Management

The Emerald kernel needed to keep track of multiple concurrent activities, for example, an ongoing search for an object, a remote invocation, and the loading of some code over the network. At the time, good thread-management libraries were not available, and instead of writing our own thread package we decided to use event-driven programming, an idea inspired by the MiK kernel [93]. We built a single-threaded kernel that serviced a ready queue of kernel tasks, which we made sure were atomic. These tasks were generated from many sources, for example, the arrival of a invocation request message, a remote code load, or a *bootup* message from another Emerald kernel. Many events arrived in the form of a UNIX signal. For these, the signal handlers merely set a *signal-has-occurred* bit; they were not allowed to touch any other kernel data structures. To make sure that signals were not dropped, kernel tasks were required to be short and not to perform any blocking operation. If a task needed to wait, it had to do so by generating a continuation task that would run when the appropriate event arrived, and then terminating itself. Synchronization around the *signal-has-occurred* bit was complicated and took a lot of low-level hacking; the details can be found in a technical report [53].

Each waiting kernel task was potentially linked into a dependency graph: if a task B was waiting for another task A to complete, B would be linked from task A. When A completed, it would put all tasks waiting for it onto the ready queue, unless they were also waiting for another uncompleted task. For example, the arrival of a moving object would generate an *object install* task, but if the code for that object were not already resident, a *code load* task would be generated to find and load the code. The *object install* would be dependent on the *code load*. When the *code load* completed, the *object install*'s dependency on the *code load* task would be removed and, if the install task had no more dependencies, it would be put on the ready queue.

4.10.3 Choice of Networking Protocols

Eden was built on a message module that used datagrams and did not provide reliability. For Emerald, in contrast, Eric modfied the Eden Message Module to provide reliability by using a sliding window protocol taken from Tanenbaum's first *Computer Networks* book [101]. (In the process he found an error in the protocol.) The importance of a truly fault-tolerant communication protocol had become apparent to Eric during a demo of Eden, when an object on one machine had reported that another object was inaccessible. This was despite the fact that it was obviously alive because it was displaying output on another screen, and was observed as alive by an object on a third machine. After this demo Eric spent a lot of time researching fault-tolerance and recovery and worked hard to ensure that the low-level protocols would be quite robust.

We considered using TCP instead of UDP, but the version of UNIX we were using allowed only a very limited number of open connections, both because UNIX had a low limit on the number of open file descriptors and because TCP connections used a significant amount of buffer space. This meant that full machine connectivity would require us to open and close TCP connections frequently, leading to excessive overhead and high latency for remote invocation. We also wanted to have full information about when nodes failed to respond to messages, so that we could declare them dead according to our own policy. In Eden, we had noticed that nodes could be considered *up* even if they had died some time previously; even worse, nodes could be considered *dead* even though they were *up*. We realized that no failure detector could be completely reliable, but we felt that because we knew more about the Emerald system than did the generic implementation of TCP in the UNIX kernel, we could do a better job.

Eric's implementation of reliability on top of UDP strove to reduce the number of network messages to the absolute minimum. We used just two Ethernet messages for a remote invocation: the acknowledgment message, which was necessary for reliability, was piggybacked on the next interaction. Thus, a series 100 remote invocations to the same destination would require 201 Ethernet packets: 100 invoke-request packets, 100 invoke-return packets, and, after about 1 second, a single acknowledgment packet — the remaning 199 ACKs would be piggybacked onto the other packets.

4.10.4 OIDs for Compiled Entities

The compiler generated a file for each object constructor and each type. For the purpose of loading and linking, the compiler assigned a unique *OID* to each file, and any reference from one file to another used the *OID*. The compiler had its own pseudo-machine number, so in effect it had control over its own *OID* space, which was disjoint from the space of "real" objects.

Because types were objects, they needed to have *OID*s; because types were immutable, it was convenient to use the same *OID* across multiple compilations of the same type. (Recall that Emerald types contain only operation signatures, and not executable code.) Two types were the same even if the Emerald sources that defined them were different in insignificant ways; for example, if the order of operations was different in two versions of a type, or if they used different names for themselves or their component types. We also gave *OID* to Strings used as operation names; Strings were also immutable objects. The purpose of this was to speed-up the comparison of operation names when deciding if one type conformed to another.

The initial implementation of type *OID*s and and String identifiers was a file on the local network file system, which the compiler treated as a global database, and in which it stored a canonical form of the type definition. Of course, we were well-aware that this implementation would not scale to a global Emerald network, but at the time we had only five machines, and it worked quite well. Our longer-term plans involved using fingerprints of the type definitions, a technique that we had recently heard about from colleagues at DEC SRC, where it had been used in the Modula-2+ system. Fingerprinting was originally invented by Michael Rabin [88]; the techniques that made it practical were developed by Andrei Broder at SRC in the mid 1980s, although they were not published until much later [31]. Broder's Fingerprinting algorithm offered strong probabilistic guarantees that distinct strings would not have the same fingerprint. We never actually got around to using fingerprints; when Norm used Emerald in distributed systems classes, he instead implemented *Type* to *OID* and *String* to uid servers as Emerald applications.

4.10.5 Process Implementation

Processes were implemented quite conventionally using one stack for each process and one activation record for each invocation. However, our stacks were segmented: a stack could consist of multiple stack segments, each of which contained some number of activation records. When returning off the "bottom" of a stack segment, the run-time system would check to see if the process had more stack segments, in which case the process would continue execution using the top activation record of the next segment.

There were two reasons for segmenting a stack. First, segmentation allowed us to allocate a small stack and then expand it dynamically, thus saving space and letting us support a substantial number of concurrent processes. Second, segmentation allowed the process to make remote invocations. The links between stack segments were object references, and each link could thus could point to a segment on another machine. When a process performed a remote invocation, it simply continued execution on the remote machine using a new stack segment containing the activation record for the invoked operation. When the operation returned off the bottom of the stack segment, the invocation results were packaged up and sent back to the originating stack segment.

4.10.6 Dynamic Code Loading

Because Emerald objects shared a single address space, introducing a new kind of object meant loading new code into the address space. The loader had to perform "linking": references to abstract or concrete types in the loaded code had to be replaced by references to the Object Descriptor of the appropriate object.

We adopted the principle that *all* of the code files needed for an object to execute had to be loaded before execution was allowed within the object. This was an example of a general philosophy of minimizing remote dependencies: as much as possible we wanted to insulate objects from failures of remote machines. By agressively "hoarding" all the code files required by an object we ensured that the object, once it started execution, would never stall waiting for code to load, or fail because it needed code located on a machine that had become unavailable. Loading all of the code before execution started also meant that there was no need for dynamic *code-loaded* checks that would have introduced overhead into all programs, even those that didn't use distribution. This would be in violation of our *no-use, no-cost* principle mentioned in Section 2.

5. Applications and Influences of Emerald

Immediately after the implementation of Emerald was completed, indeed even before it was completed, the Emerald team dispersed. In January 1987, Norm graduated and took up a faculty position the University of Arizona; Eric went back to Denmark in February 1987 and taught at the University of Copenhagen while finishing his thesis. Andrew joined the distributed systems group of Digital Equipment Corporation in December 1986. Only Hank grew roots at Washington, where he is now department chair. Nevertheless, in addition to the initial batch of research papers, Emerald has lived on in several forms. It has been used in teaching and graduate student education and as a basis for subsequent research. Emerald also influenced the design of successor systems. In this section we summarize what we have been able to discover about the use and influences of Emerald after the initial implementation effort.

5.1 Applications of Emerald

Emerald has been used in teaching and research at various universities; in most cases Emerald was carried to other sites by those who had become familiar with the language at Washington.

Norm has used Emerald at the University of Arizona and at the University of British Columbia (UBC) in teaching graduate classes on distributed operating systems. Three M.Sc. students at UBC did their theses on aspects of Emerald including generational garbage collection [48], porting the language to the embedded processor on a network interface card [86], and improving reliability by grouping objects [45].

Eric has used Emerald at the University of Copenhagen for teaching at both the graduate and undergraduate levels. From 1994–1997, Emerald was the first object-based language taught to incoming students (their first language being ML). During these four years approximately 1000 students used Emerald for about three months, including a three-week-long graded assignment where they were to develop and write a large (1000–2000 lines of Emerald) program, such as a ray tracer. They did not use any of the distribution or concurrency facilities, but focused on the pure object-based part of the language. During the 1990s Emerald was also used to teach distribution in an introductory graduate distributed systems course at DIKU.[11] Students were required to write small (200–300 lines of Emerald) programs to implement a distributed algorithm such as election or time synchronization. About fourteen Master's students have to varying degrees based their Master's theses on Emerald [5, 54, 55, 56, 63, 65, 68, 78, 80]. In addition, two Ph.D. students conducted research based on Emerald: Niels Juul worked on distributed garbage collection [60] and Niels Larsen worked on transactions [66].

Andrew used Emerald in research while at Digital Equipment Corporation. It was used to build one of the first distributed object-oriented applications to run on Digital's internal engineering network, a distributed object-oriented mail system implemented by a summer student in 1987. This student (Rajendra Raj) later went on to develop a programming environment for Emerald [89], called Jade, that was the subject of his doctoral dissertation [91].

Other University of Washington students took Emerald with them to other parts of the globe. Ewan Tempero used it in research at the Victoria University of Wellington. Results include two theses: Neal Glew's B. Sc. Honours thesis [47] and Simon Pohlen's M. Sc. Thesis [87].

Emerald was also used in research at the University of Mining and Metallurgy (UMM) in Kracow, Poland. The research focused on multicast group communication, and was initiated by Przemek Pardyak and Eric Jul during a visit by Przemek to Copenhagen in the early 1990s. Przemek subsequently worked with Krzysztof Zielinski at UMM, resulting in two papers [82, 83] and a Master's thesis [81]. The thesis was awarded third prize in the annual Polish competition for best engineering thesis. Along the way, they also published an overview of Emerald in Polish in a book on distributed systems; this article also appeared in the main Polish CS magazine *Informatyka* [84].

[11] DIKU is the department of Computer Science at the University of Copenhagen.

5.2 Influences of Emerald

Emerald has influenced succeeding languages designs in two main areas: support for distributed objects and advanced type systems. Because the goal of Emerald was to innovate with distributed objects, the language's influence on distribution is unsurprising. (What is perhaps surprising is that in the more than twenty years since we implemented Emerald, no other language has adopted mobile objects with similar thoroughness.) In contrast, Emerald's influence on type systems was neither intended nor expected. At the time we saw our innovation here as minor: we were going to do only what was required to keep the language as small as possible. The idea of conformity has been quite widely adopted. In an interesting example of feedback, both abstract types (here called protocols) and conformity feature heavily in the ANSI Smalltalk standard. The 1997 final draft says

> The standard uses a particular technique for specifying the external behavior of objects. This technique is based on a protocol, or type system. The protocol-based system is a lattice (i.e. multiply inherited) based system of collections of fully-specified methods that follow strict conformance rules [6, p. 6].

However, Java adopted nominal rather than structural typing (see Section 4.2.3), and only partially separated types and implementation: a Java interface is a type, but a Java class is both an implementation *and* a type.

Several operating systems have followed Eden and Emerald in providing mobile objects; notable among these is SOS [94], which in addition allows an object to be fragmented across several machines. However, SOS provides only minimal language support, so its objects and proxies must be manipulated by explicit operating system primitives. (SOS does use a compiler extension to simplify the process of obtaining a proxy for a remote object.) One of the early derivatives of Emerald's object model and type system was the Advanced Networked Systems Architecture (ANSA) distributed computational model [106]. In its turn, ANSA was one of the influences behind the Open Software Foundation's Distributed Computing Environment initiative, and perhaps more significantly, a contributor to the ISO Basic Reference Model of Open Distributed Processing ISO 10746 [46]. The architecture described in part 3 of this standard includes a type system very strongly based on

Emerald, extended with support for multiple terminations, streams and signals — ideas taken from ANSA. In particular, Emerald's type conformity rules are alive and well in Section 7.2.4.3, and Annex A gives a set of subtyping judgments including top and bottom types strikingly similar to those used to formalize the Emerald type system [26].

Andrew Herbert, Chief Architect of the ANSA efforts, describes how the ANSA architecture was influenced by Emerald.

> Alongside the ANSA architecture we maintained a number of software prototypes. The first version of ANSAWare was called "Little DPL" and was preprocessor based... A separate strand of development was "Big DPL", which borrowed very heavily from Emerald. This was the foundation for the input to ISO and also to OMG CORBA. It took the view that you could do a complete language based on "distributed objects", where an object could have a number of "interfaces" (which were named typed instances and could be static or dynamic). Our invocation model was that an interface defined a bunch of methods and that each method had a bunch of "terminations", one of which generally was regarded as the "normal return" and the others as "exceptions". All parameter semantics were call by reference. Immutable objects could of course be copied, and we had an object migration facility. Migratability and other "ilities" were object properties and objects would generally have a control interface through which the "ility" could be exercised.
>
> Big DPL was never well-enough developed and not mainstream enough to compete with the C++ network object systems that came along around the same time... Over time it shifted from a separate "toy language" to DIMMA, yet another C++ distributed object system, but which carried forward explicitly the DPL type system. The final evolution was the "FlexiNet" system written in Java, which had a hybrid Emerald/Java object model and Java type system...
>
> In summary Emerald had a lot of impact on us, and through us on other bodies.[12]

Andrew Watson started working at ANSA in December 1989. At that time both the ANSA computational model and the language that realized it (DPL) were well established. There were several technical differences between Emerald and DPL, principally that DPL allowed multiple interfaces per object, that each operation in DPL could have multiple named outcomes (with different types), and that DPL had absolutely no primitive types built in. Watson writes:

> However, leaving these differences aside, the influence of Emerald on DPL was obvious — especially the conformance-based type checking and type inferencing in the DPL language. It was this type system that I chose to work on, and in particular the problem of typing constructors for collections of homogeneous and heterogeneous interfaces, and how to avoid having every object carry around its type information at run time via an Emerald-style "getInterface" operation. I did come up with a simple modification to DPL that added a separate, opaque run-time type representation which would only be created when required by the programmer — this would dramatically reduce the need for the run-time system to create and transmit run-time type tokens... [but the implementation of this modification] was never finished.[13]

Emerald also had an impact on the design of the Guide system and language at INRIA in Grenoble, France. Sacha Krakowiak writes:

> One important source of inspiration for the design of the Guide language has been Emerald, a distributed object-oriented language developed as a follow-on project to Eden. The two main features of the design of Emerald that directly influenced Guide were the separation between types and implementations, and the definition of type conformity (through covariant and contravariant relations). This definition had been proposed by Cardelli in 1984 [35], but we did not know that work.[14]

Emerald's closest descendant in the family of distributed object systems is probably the Network Objects system described and implemented by Birrell, Nelson, Owicki and Wobber [12, 13]. The authors write: "We have built closely on the ideas of Emerald [59] and SOS [94], and perhaps our main contribution has been to select and simplify the most essential features of these systems. An important simplification is that our network objects are not mobile" [12, Section 2]. Instead, the Network Objects system provided what the authors called "powerful marshaling": the ability to send a copy of an arbitrarily complex data structure across the network. They write [13, p. 10]:

> We believe it is better to provide powerful marshaling than object mobility. The two facilities are similar, because both of them allow the programmer the option of communicating objects by reference or by copying. Either facility can be used to distribute data and computation as needed by applications. Object mobility offers slightly more flexibility, because the same object can be either sent by reference or moved; while with our system, network objects are always sent by reference and other objects are always sent by copying. However, this extra flexibility doesn't seem to

[12] Andrew Herbert, personal communication.

[13] Andrew Watson, personal communication.

[14] Sacha Krakowiak, personal communication.

> us to be worth the substantial increase in complexity of mobile objects. For example, a system like Hermes [25], though designed for mobile objects, could be implemented straightforwardly with our mechanisms.

(As might be expected, we do not entirely agree with these conclusions; see the discussion in Section 6.)

Emerald was also influential in the development of the distributed systems support provided by Java and Jini, although here the influence was more indirect. Indeed, at first blush the Java RMI mechanism seems to be the antithesis of Emerald's remote object invocation, because RMI distinguishes between remote and local invocations. This is not accidental: Jim Waldo and colleagues at Sun Laboratories authored a widely referenced technical report that argues that the "unified object" view espoused by Emerald is fundamentally flawed [104]. The account here is largely based on material supplied by Doug Lea.

Java RMI was most directly influenced by CORBA and Modula-3's Network Objects. In the summer of 1994, James Gosling's *Oak* language for programming smart appliances was retargeted to the World-Wide Web and soon thereafter renamed Java. Jim Waldo's group were asked to look into adding an object-oriented distributed RPC of some form. At the same time, some people who had been in the Sun *Spring* group (who were also contributors to CORBA) were looking into the alternative approach of just providing Java with a CORBA binding. Both of these approaches had their advocates. Some people were excited by the "Emerald-ish" things one could do with extensions of Waldo's approach. Also, Cardelli's paper on *Obliq* had just appeared [37]. Obliq objects are implemented as Modula-3 network objects, and so are remotely accessible. While Obliq *objects* are fixed at the site at which they are created, distributed lexical scope allows *computations* to roam over the network. Although the people at Sun regarded Obliq as a thought experiment, it clearly demonstrated the limitations of CORBA, which was unable to express the idea of a computation moving around the network; thus Obliq provided fuel for the argument that Java needed a more powerful remote communication system. Doug Lea writes: "I wanted a system that was not only usable for classic RPC, but also for extensions of the things that I knew to be possible in Emerald. My bottom line was that I insisted it be possible to send a Runnable object to a remote host so that it could be run in a thread." After much deliberation, a committee at Sun chose Waldo's approach, and many of the researchers in Waldo's group transitioned from Sun Labs to the Java production group, and built what is now known as Java RMI. After RMI was released, they moved on to develop further what they had earlier been working on in Sun Labs, which turned into Jini [7].

As it exists today, RMI supports neither Emerald-style mobile objects nor Obliq-style mobile processes, and remote invocations are not location transparent, because different parameter-passing mechanisms can be used for local and remote invocations of the same object. But many of these restrictions are the result not of limited vision, but of compromises that had to be made to fit RMI over the Java language, whose specification was at that time largely fixed. Emerald was a force for a more adventurous design, and the ultimate decision to go with RMI rather than a CORBA binding was influenced by reading or hearing about Emerald's capabilities — these were continually brought up as the kinds of things that a forward-looking language ought to support. Jim Waldo writes:

> The RMI system (and later the Jini system) took many of the ideas pioneered in Emerald having to do with moving objects around the network. We introduced these ideas to allow us to deal with the problems found in systems like CORBA with type truncation (and which were dealt with in Network Objects by doing the closest-match); the result was that passing an object to a remote site resulted in passing [a copy of] exactly that object, including when necessary a copy of the code (made possible by Java bytecodes and the security mechanisms). This was exploited to some extent in the RMI world, and far more fully in the Jini world, making both of those systems more Emerald-like than we realized at the time.[15]

After the turn of the century, increasing use of mobile devices lead to an interest in languages that supported loosely connected devices. The ideas behind Emerald inspired Walsh's thesis work on the Taxy Mobility system [105]. Walsh introduces Emerald-like objects into Java and discusses the idea of combining weak and strong mobility. Similarily, the work of De Meuter takes a critical look at strong mobility as present in Emerald and proposes alternative, weaker, mobility mechanisms [42].

5.3 Later Developments

Various research projects have built on Emerald: to complete the implementation, to enhance the language, and to take Emerald in new directions.

Garbage collection. We realized early on (see the minutes from 18 March 1985) that Emerald would require a garbage collector, but we also realized that the implementation of the collector could be deferred until after Norm and Eric had completed their theses. In practise, Emerald did quite well without a collector. A prime reason was that the implementation was stack based and so only rarely would the implementation generate a temporary object and not be able to deallocate it.

Eric's Ph.D. dissertation [58] contains a chapter on the design of a distributed, on-the-fly, robust, and comprehensive

[15] Jim Waldo, personal communication.

garbage collector for Emerald, but there was no attempt to implement it at that time. However, Eric did prepare the Emerald prototype for the implementation of a distributed collector at some time in the future, concluding that the implementation effort would be worth a separate Ph.D. This is actually what happened: Niels Christian Juul implemented the collector for his doctorate [61, 60]. A strong point of this on-the-fly collector was that it was possible to start the collector and have it run to completion without requiring that all machines be up at the same time. In the contemporary demonstration of the collector, no more than 75% of the machines were up at any given time — we felt that this was a very robust collector.

In 2001-2002, two Master's students developed a non-comprehensive collector that could collect smaller areas of the distributed object graph and thus could reclaim garbage while parts of the system was down [55].

Emerald as a general-purpose programming language. Those of us involved in the development of Emerald naturally saw it primarily as a language for distributed programming. It took an "outsider" (Ewan Tempero) to make us realize that Emerald was an interesting language in its own right, even without its distribution features. This realization led to a paper that described the language in this light, emphasizing Emerald's novel object constructors and types [90].

The Hermes project at Digital Equipment Corporation. Andrew started the Hermes project shortly after he moved to Digital in December 1986. (This project is unrelated to Rob Strom's Nil project at IBM, which also used the name Hermes in its later years [99].) The idea was to implement as many of Emerald's distributed object ideas as possible in a conventional programming language (Modula-2+), but also to find a way to make the implementation scale to systems of around fifty thousand nodes, the size of Digital's internal DECnet-based Engineering Network at that time. The project was a technical success [24, 25] but did not have much influence on future products because, in spite of the company's early lead in networking, Digital was never able to make the transition to shipping distributed systems products.

Types. Norm and Andrew continued to work on types during the period from 1986 to 1989, after both of them had left Washington. Much of this work was unpublished, although not for want of trying. The target of our publication efforts was PoPL; at that time the PoPL community, based as it was in traditional single-machine programming language design, was developing ideas about the purpose of types and how they should be discussed that were much more conservative that those we had arrived at though our work in distributed systems. (To this day there is an enormous resistance from some members of this community to even admitting that something that might need to be checked at run time could even be called a "type".) The essence of our work was captured in a widely-cited January 1991 joint Arizona/CRL technical report [26], which had a major influence on the ANSA architecture (see Section 5.2).

Gaggles. One of the deficiencies of Emerald's pure object-based approach is that not everything is most conveniently represented as an object. In particular, in a distributed system striving for high availability, resources must be replicated: the collection of replicas is not itself an object. Of course, it is possible to put the replicas *inside* an object, such as a Set, but this recreates a single point of failure. Gaggles resolve this problem: a Gaggle is a monotonically-increasing set of replicas that can be treated as a single object with "anycast" semantics [27]. An undergraduate student from Harvard, Mark Immel, undertook the implementation.

Multicast invocations. As mentioned in Section 5.1, Przemek Pardyak [82, 83] extended Emerald with facilities for multicast invocation. In his system, one could make an invocation of a group of objects, and either take the first answer, or require all answers.

Wide-area Emerald. In 1992-3, two Master's students at DIKU implemented a Wide Area Emerald [80]. The implementation was modified to support machines on the internet in general rather than on a LAN. They increased the size of OIDs and had to spend a lot of time tuning the transport-layer protocols: their initial move of an object around the globe via seven machines took about 15–20 *minutes*, because the transport layer was optimized for LAN service. Eventually, moving an object around the world took only a few seconds.

Ports to various architectures. Between 1987 and 1994, Emerald was ported from the original VAX architecture to SUN 3 (Motorola 68000), SUN 4 (SPARC) [75], and Digital's Alpha [66]. Tired of retargeting the compiler, Norm also developed a byte-code compiler and a virtual machine, thus allowing objects to move from one platform to another.

Eclipse plugin. In 2004, IBM funded a small project at the University of Copenhagen to implement an Eclipse plugin for Emerald. A student did the implementation during 2004–2006; the source code for the plugin — and for Emerald in general — is available at http://sourceforge.net/projects/emeraldlanguage.

Heterogeneous Emerald. Emerald's clear separation of language concepts from implementation means that the semantics of Emerald objects have a rigorous (if informal) high-level description. Any implementation of an Emerald object thus must define a translation of the language's high-level semantic concepts into a low level representation. This means that given two different implementations of an Emerald object for two different low level architectures, it should, in principle, be easy to remap the representation of an Emerald object on one architecture to the representation of that same object on another architecture.

Would this principle work out in practice? To answer that question, between 1990 and 1991, two graduate students (Povl Koch and Bjarne Steensgaard Madsen) worked with Eric to develop a version of Emerald that ran on a set of heterogeneous machines: VAX, SUN 3, and SUN SPARC. This mainly involved remapping inter-machine communication. However, object mobility also posed significant problems because an object could contain an active Emerald process. We introduced the concept of a *bus stop*, a place in the Emerald program where mobility could occur and where every implementation had to provide a consistent view. Bus stops are frequent; they are present after essentially every source-language statement. We then added so-called *bridging code* at each bus stop that could translate the state of an Emerald process or object from one architecture to another. The implementation was simplified because Emerald already had the concept of stopping points where mobility could take place.

To our knowledge, no one has since built a heterogenous object system allowing not only objects but also executing threads to move between architectures at almost any point in the program with *no* decrease in performance after arrival on the new architecture. That is, an Emerald process that moves, e.g., from a VAX to a SUN 4 SPARC and back will, when running on the SPARC, execute at the same speed as if it had originated there, and it will also execute at the original VAX speed after returning to a VAX. Cross-architecture performance measurements showed that remote invocations and object (and process) mobility took about twice as long as in the homogeneous case, mainly due to the large amount of work caused by checking for big-endian to little-endian translations, and byte-swapping if necessary. Note, however, that the programs locally always ran at full speed unimpeeded by the facilities for heterogeneity.

Eric presented the heterogeneous Emerald work at a work-in-progress session at SOSP in 1993, and generated significant attention. This caused Eric to contact one of the students, who in the meantime had joined Microsoft Research. The result was a paper presented at SOSP in 1995 [97]. At the time, Marc Weiser was advocating the then-controversial idea that researchers should make their code publicly available, so that anyone so motivated could verify the claimed results. As a consequence, the Heteogeneous Emerald implementation can be found on the 1995 SOSP CD.

Databases, Queries, and Transactions A bright graduate student, Niels Elgaard Larsen, integrated databases into Emerald as his Master's thesis project [65]. He later integrated transactions into Emerald as part of his Ph.D. project [66].

6. Retrospective

The Emerald project never came to a formal conclusion; it simply faded away as the members of the original team became interested in other research. Nevertheless, time has given us a certain perspective on the project; here we reflect on what we did and what we might have done differently.

We are all proud of Emerald, and feel that it is one of the most significant pieces of research we have ever undertaken. People who have never heard of Emerald are surprised that a language that is so old, and was implemented by so small a team, does so much that is "modern". If asked to describe Emerald briefly, we sometimes say that it's like Java, except that it has always had generics, and that its objects are mobile.

In hindsight, if we had had more experience programming in Smalltalk, Emerald might have ended up more like Smalltalk. For example, we might have had a better appreciation of the power of closures, and have given Emerald a closure syntax. Instead, we reasoned that anything one could do with a closure one could also do with an object, not really appreciating that the convenience of a closure syntax is essential if one wants to encourage programmers to build their own control structures.

6.1 The Problem with Location Independence

Emerald's hallmark is that it makes distribution transparent by providing location-independent invocation. However, there is definitely a downside to this transparency: it becomes easy to forget about the importance of placing one's objects correctly. Shortly after Eric started using Emerald in his graduate course on distributed systems, a student showed up at his office with an Emerald program that appeared to hang; the student could not find any bug in the program. Eric suggested turning on debugging and started with traces of external activities. Streams of trace output immediately showed up: the program was not hung but was executing thousands of remote invocations. What had happened was that the student had omitted an **attached** annotation on a variable declaration, which meant that he was remotely accessing an array that should have been local. Thus the program was running three to four orders of magnitude slower than anticipated. The good news was that Emerald's location independent invocation semantics means that the program was still correct; the bad news was that it ran more than a thousand times too slowly. After adding the omitted **attached** keyword the program ran perfectly!

6.2 Mobile Objects

Since we started working on Emerald, objects have become the dominant technology for programming and for building distributed systems. However, mobile objects have not enjoyed a similar success. Why is this so? The argument against mobile objects goes something like this.

The simplicity of object orientation arises because objects are good for modeling the real world. In particular, objects enable the sharing of information and resources inside the

computer, just as in the real world. Understanding object identity is an important part of these sharing relationships.

Mobile objects promise to make that same simplicity available in a distributed setting: the same semantics, the same parameter mechanisms, and so on. But this promise must be illusory. In a distributed setting the programmer *must* deal with issues of availability and reliability. So programmers have to replicate their objects, manage the replicas, and worry about "one copy semantics". Things are not so simple any more, because the notion of object identity supported by the programming language is no longer the same as the application's notion of identity. We can make things simple only by giving up on reliability, fault tolerance, and availability — but these are the reasons that we build distributed systems.

Once we have to manage replicas manually, mobile objects don't buy us very much over Birrell's "powerful marshaling" [13]: making a copy of an object at a new location. They do buy us something, but the perception is that it is not worth the implementation cost.

We feel that this argument misses the point, and for two reasons. First, the implementation cost is not high. Eric and Norm may be smart, but they are not *that* smart; if they could figure out how to make objects mobile and efficient in the mid-1980s, it should not be hard to reproduce their work in the twenty-first century. Indeed, we believe that implementing mobility right once is simpler than implementing the various *ad hoc* mechanisms for pickling, process migration, remote code loading, and so on.

Second, we now know enough about replication for availability to design a robust mechanism like Gaggles [27] to support replicated objects in the language. In the presence of such a mechanism, object identity once again becomes a simple concept: in essence, the complexity of the replication algorithm has been moved *inside* the abstraction boundary of an object. This does not make it any simpler to implement, but does permit the client of the complex replicated abstraction to treat it like any other object.

There has also been a long-running debate [62, 74] that questions whether object identity should be a language-defined operation at all, because it breaches encapsulation. Conceptually, we have a lot of sympathy for this position, but pragmatically we know that it is important to support the operations of identity comparison and hash on all objects if we want to do efficient caching — and caching is a very important technique for building efficient distributed systems. A compromise would seem to be in order. Andrew advocated just this in 1993: equality and hashing of object identity should be fast, primitive operations that do not requiring fetching an object's state, but the programmer should able to allow or disallow these operations on an object-by-object (or perhaps class-by-class) basis [23].

6.3 Static Typing

With the benefit of hindsight, static typing may have been a mistake: static types bought us very little in the way of efficiency, and cost us a great deal of time and effort in developing the theory of bounded parameterized types. The one place where static types do buy efficiency is with primitive types such as integers, but this is not because of the types themselves. The efficiency gain arises because we break our own rule and confound implementation (of an integer as a machine word) with interface. In other words, integers are efficient only because we forbid the programmer to write an alternative implementation of the *integer* type.

In the normal case of an object of user-defined type that is the target of an invocation, we know that the dynamic type of the object conforms to the static type of the identifier to which it is bound. This guarantees that the invocation message will be understood at run time, but it does *not* help us to find the right code to execute. Finding the code must still be done dynamically, because the implementation of the object is generally not known until run time. Of course, in many cases the implementation will be fixed at compile time, in which case method lookup can be eliminated altogether. But the presence of type information did not help us to implement this optimization: it relies on dataflow analysis.

Emerald's dynamic type checks rely on the compiler telling the truth: the type of an object is encoded in it as a **type** when the object is constructed, based on the information in its object constructor. If the compiler lied, type checking might succeed, but operation invocation still fails. We considered certifying compilers and having them sign their objects, but Because no one ever wrote a hostile Emerald compiler, signatures were never implemented. This is one place where Java does something simpler and more effective than Emerald: Java byte codes are typed, and code arriving from another compiler can be type-checked when it is loaded. This is an idea that we might well have used in Emerald if we had been aware of it.

6.4 Method Lookup and AbCon Vectors

AbCon vectors may be a more efficient mechanism for performing dynamic lookup than method dictionaries, although, to the best of our knowledge, the mechanisms have never been benchmarked side by side. To compare them fairly one must include the time taken to construct the AbCon on assignment. However, for many assignments the compiler is able to determine the concrete type of the object and thus does not need to generate an AbCon, or indeed perform method lookup: the appropriate method can be chosen statically. For many of the remaining assignments, e.g., those where the abstract types of the right- and left-hand sides are the same, the extra cost involved in the assignment is merely the copying of a reference to the AbCon. For some of the remaining assignments, the compiler can determine the Ab-

Con to be built; in these cases the AbCon is built at load time and a reference to it is inserted into the code, so the overhead is reduced to the store of a single constant. For the remaining assignments, the asymptotic cost is reduced by caching AbCons; after a program has run for a while, *all* of the AbCons that it needs will have been cached,[16] but some time is still expended in accessing the cache. In a similar way, the cost of method lookup is normally reduced by caching recent hits. Indeed, polymorphic inline caches [52] have proven so successful in eliminating message lookup that its cost is widely perceived as a non-problem.

6.5 What Made Emerald Fast?

As we discussed in the Introduction, Emerald grew out of our experience with Eden, which had lots of good ideas but rather disappointing performance. One of our major goals was not just to improve on Eden's performance, but to actually have good performance in an absolute sense (see Section 2). Because performance is hard to retrofit, we practiced performance-oriented design from the beginning. There was no silver bullet: we had to get the details right all along the line.

6.5.1 Single Address Space

A major design decision behind Emerald's excellent performance was placing all node-local objects and their processes in a single address space (a UNIX process, as described in Section 4.10), which was also shared with the Emerald kernel. This meant that kernel calls were simply procedure calls and any data structures in the object could be accessed by the kernel simply by dereferencing a pointer.

6.5.2 Distributed Operations and Networking

The first Eden remote invocation took approximately 1 second, during which time about 27 messages were sent (counting both IPC messages and Ethernet messages). There was considerable room for improvement: the Eden invocation module was rewritten several times, resulting in substantially fewer messages and a corresponding drop in invocation times, which ended up at approximately 300 ms for a remote invocation and 140 ms for a local invocation. A major lesson from these rewrites was that performance was more or less proportional to the number of messages sent. The best case for node-local invocations in Eden was four IPC messages: one from the source object to the Eden kernel, one from the Eden kernel to the target object, and two more to obtain the result. In Emerald, putting all node-local objects and the kernel into the same address space eliminated IPC messages entirely. The result was that Emerald node-local invocations were more than three orders of magnitude faster than Eden's, see Section 6.6.

[16] We were surprised at how few AbCons even large programs needed.

6.5.3 Choosing Between Eager and Lazy Evaluation

In theory, lazy evaluation is wonderful: nothing is ever evaluated until it is needed, but once evaluated, it is never evaluated again. However, in practice there is a cost to laziness: before a value is used, one must check to see if it is a thunk that first needs to be evaluated. In consequence, eager evaluation is a win in many situations: if it is very likely that a value will be needed, it makes sense to evaluate it early. Moreover, if "early" can be made to mean "at compile time", then the cost of run-time evaluation can be eliminated entirely.

Eager Evaluation. We applied the idea of aggressive eager evaluation in several places. Using direct references rather than *OID*s was one of these. In Eden, objects were referred to by a globally unique identifier, which meant that any operation on an object had to translate that global identifier into a reference to some local data structure. In contrast, in Emerald we translated *OID*s to pointers as soon as possible, so that, for example, a local object was represented by a direct pointer to its data area. Code, AbCons, Types — indeed, anything referenced by *OID* — was represented as a pointer to either the relevant data structure or, in the case of a remote object, to a descriptor that contained information on how to find the data structure (see Section 4.10.1). This made local operations faster, although it also made remote operations slower, because all direct pointers had to be translated when they crossed from one machine to another. However, the slowdown caused by translation was small compared to the cost of a network operation. The principle was that users should pay for a facility only when they used it: we did not want to slow down local operations just to make remote operations faster.

We also avoided lazy evaluation by changing the code to make sure that expensive computations were performed only once. Our experience with Eden had showed that many computations were done repeatedly because several different modules in the implementation needed the result of a single computation. For example, in the early days of Eden invocation, the unique identifier of the invoked object was looked up more than ten times! A quick performance fix was to equip the object table lookup function with a one-item cache, which eliminated the work of the lookup but added the overhead of the cache check. In implementing Emerald we used profiling techniques to discover such inefficiencies and fix them, usually by passing on pointers to resolved data rather than the original *OID* or pointer; this eliminated not only the work of the lookup, but also the overhead of the function call and the cache check.

The ultimate application of eagerness was to move as much computation as possible into the compiler. For example, the compiler did an excellent job of figuring out when it could make an object local rather than global, thus eliminating all the overhead associated with distribution. It also removed as

much of the type information as it could, so if you wrote and compiled a program that did not use distribution at all, then all distribution overhead would be removed from the compiled code. Consequently the program would run at a speed comparable to that of compiled C. We found that on the SPARC architecture, the Emerald version of a highly recursive computation such as the Ackermann function actually ran faster than C, because our calling sequences were more efficient.

Lazy Evaluation. Lazy evaluation was more efficient than aggressive evaluation when the work that was delayed might never need doing at all. A prime example is representing global objects. Global objects were allocated directly by compiled code, but most of the work of making them usable in a distributed environment was delayed until the first time the object had any interaction with another node, for example, by a reference to it being passed to another node. At that time the object would be "baptized" by being given an *OID* and entered into the local object table. Many objects had the *potential* to become known outside the node on which they were created, but most would never actually do so. Thus, postponing baptism was an excellent idea because it was often unnecessary.

Another place where Emerald takes a lazy approach is in moving the code for a migrating object. When an object moves from one node to another, its code is *not* moved along with its data. The reasoning behind this is that the receiving node may well already have a copy of the code, in which case the work would be unnecessary. Of course, if the receiving node does *not* have the code, it has to obtain it; this is actually more work than if the code had been sent eagerly. However, it turns out that most applications have a small working set of code objects, and so most of the time the destination node will already have the code for an inmigrating object. In the case where the code does have to be requested, multiple code objects can be requested at the same time, so we win again from a batching effect.

Another technique that used the idea of laziness is replacing computation by compiler-generated tables. One simple example is making the current source-code line number available when debugging. The Simula 67 compiler generated a load instruction that put the current line number into a register. This seemed to us to be optimizing for the uncommon case: the Emerald implementation instead generated a table in each code object that could translate the relative address of any instruction into the line number in the Emerald source code. This had a storage cost but no cost in execution time. Of course, if the debugger was actually activated, it look more time to find the line number, but that was a rare event, and not a time-critical one.

Compiler-generated tables were also used for unavailable and failure handlers. When a failure occurred, such as invoking **nil** or dividing by zero, the appropriate handler was found by using the address of the failure to search a table of handlers. The implementation of Mesa used the same idea [64]. This was in contrast with other language implementations (for example, Sherman's Ada compiler for the VAX [95]) where the appropriate failure handler was pushed onto the stack; this made finding the handler faster, but slowed down every call that did not fail.

Lazy evaluation was also the technique of choice for most of the work associated with mobility, because most objects did not move; several examples are described in Section 4.7.3.

6.6 Some Historical Performance Data

From the start, performance was important to the Emerald project; good performance was high on our list of goals, and we hoped that good performance would distinguish our work from contemporary efforts. Of course, by modern standards the performance of any 1980s computer system will be unimpressive, so we summarize here not only Emerald's performance but also that of some comparable systems available to us at the time.

Most of the performance figures given below were measured in February 1987 on a set of five MicroVAX II machines running Ultrix. The figures in Section 6.6.6 were measured on VAXStation 2000 machines, which had the same CPU as the MicroVAX II but had a different architecture that resulted in slightly longer execution times (usually about 6–8%).

6.6.1 Remote Invocations

As shown in Table 1, remote invocation of a parameterless remote operation took 27.9 ms. This included sending a request message of 160 bytes (i.e., one Ethernet payload) of which 72 bytes were transport-layer protocol headers, 64 bytes were the invocation request and 24 bytes were the return address information. The return message consisted of 82 bytes of which 72 bytes were again transport-layer headers and 12 bytes were the invocation reply.

We measured the low-level transport protocol by sending a 160-byte message and returning a 72-byte message. Such an exchange took 24.5 ms. This means that the actual handling of the remote invocation used only 3.4 ms (12%) of the 27.9 ms.

For comparison, a remote invocation in the Eden system took 54 ms on a SUN 2 (and 140 ms on a VAX 11/750). The major difference between the Eden implementation and Emerald is that Emerald used only two network messages to perform a remote invocation while Eden used four. Bennett's Distributed Smalltalk implementation used 136 ms for an "empty" remote invocation [11].

Table 1: Remote Operation Timing (MicroVAX II)

Operation	**Time**/ms
local invocation	0.019
elapsed time, remote invocation	27.9
underlying message exchange	24.5
invocation handling time	3.4

Table 2: Object Creation Timing

Creation of	**Global/ms**	**Δ**/ms	**Local/ms**	**Δ**/ms
empty object	1.06	—	0.76	—
with an **initially**	1.34	+0.28	0.92	+0.16
with a monitor	1.34	+0.28	0.96	+0.20
with one *Integer* variable	1.36	+0.30	0.96	+0.20
with one *Any* variable	1.37	+0.31	0.96	+0.20
with 100 *Integer* variables	2.41	+1.26	2.01	+1.25
with 100 *Any* variables	3.49	+2.43	3.07	+2.31
with one process	2.17	+1.11	1.85	+1.09

Table 3: Remote Parameter Timing

Operation Type	**Time/ms**	**Δ/ms**
remote invocation, no parameter	30.3	—
remote invocation, one integer parameter	31.5	+ 1.2
remote invocation, local reference parameter	32.5	+ 2.2
remote invocation, two local reference parameters	34.7	+ 4.4
remote invocation, call-by-move parameter	35.9	+ 5.6
remote invocation, call-by-visit parameter	40.3	+10.0
remote invocation, with one call-back parameter	63.5	+30.2

6.6.2 Object Mobility

Moving a simple data object took about 12 ms. This time is less than the round-trip message time because reply messages are piggybacked on other messages.

6.6.3 Process Mobility

We measured the time taken to move an object containing a small process with six variables. This took 40 ms or about 43% more than the simplest remote invocation. This included sending a message consisting of about 600 bytes of information, including object references, immediate data for replicated objects, a stack segment, and process-control information. The process control information and stack segment together consumed about 180 bytes.

6.6.4 Object Creation

Table 2 shows creation times for various global and local Emerald objects. The numbers were obtained by repeated timings and are median values — averages do not make sense because the timings varied considerably (up to +50%) when storage allocation caused paging activity. The numbers are correct to about two digits (an error of 1–2%).

In general, it took about 0.3–0.4 ms longer to create an empty global object than an empty local object because of the additional allocation of an object descriptor. An **initially** construct added another 0.2 ms as did the presence of a monitor (because the monitor caused an implicit **initially** to be added to initialize the monitor). Creating an object containing 100 integer variables cost an extra 1.2 ms.

Creating an object containing 100 *Any* variables took about 2.4 ms longer than an empty object because *Any* variables were 8 bytes long while integers were 4 bytes, so twice as much storage had to be allocated and initialized.

6.6.5 Process Creation

Creating a process took an additional 1.1 ms beyond the time required to create its containing object. We measured two processes that took turns calling a monitor. It took 710 μs for one process switch, one blocking monitor entry, one

unblocking monitor exit, and two kernel calls. By executing two CPU-bound processes and varying the size of the time slice, we estimated the process switching time to be about 300 μs ($\pm$ 5%), including the necessary Ultrixcalls to set a timer.

6.6.6 Additional Costs for Parameters

The additional costs of adding parameters to remote invocations were measured in the fall of 1988 on VAXStation 2000s, which ran slightly slower than MicroVAX IIs.

We compared the incremental cost of call by move and call by visit with the incremental cost of call by object reference. We performed an experiment in which an object on a source node S invoked a remote object R and passed as an argument a reference to an object on S. R then invoked the argument object once. Without call by move, this caused a second remote invocation back to S. When using call by move or call by visit, the remote invocation was avoided because the argument object was moved to R. The timings are shown in Table 3.

Table 4 shows the benefit of call by move for a simple argument object containing only a few variables with a total size of less than 100 bytes. The additional cost of call by move over call by reference is 3.4 ms, while call by visit adds 7.8 ms. The call-by-visit time includes sending the invocation message and the argument object, performing the remote invocation (which invokes its argument), and returning the argument object with the reply. One would expect the call-by-visit time to be approximately twice the call-by-move time. It is actually slightly higher due to the dynamic allocation of data structures to hold the call-by-visit control information. Had the argument not been moved, the incremental cost (of the consequent remote invocation) would have been 31.0 ms. These measurements are a lower bound because the cost of moving an object depends on the complexity of the object and the types of the objects it references. The lesson to take away is that call by visit is worthwhile for small parameters, even if they are called *only once*.

6.6.7 Performance of Local Operations

Table 5 shows the performance of several local operations. Integers and reals were implemented as direct objects. The timings for primitive integer and real operations were exactly the same as for comparable operations in C — which is not surprising given that the instructions generated were the same.

For comparison with procedural languages, a C procedure call[17] took 13.4 μs, a Concurrent Euclid procedure call took 16.4 μs, and an Emerald local invocation took 16.6 μs (i.e., 23% longer than a C procedure call). Concurrent Euclid and Emerald were slower because they had to make an explicit stack overflow check on each call. C avoided this overhead because UNIX used virtual memory hardware to perform stack overflow checks at no additional per-call cost.

The "resident global invocation" time in Table 5 is for a global object (i.e., one that potentially can move around the network) when invoked by another object resident on the same node. The additional 2.8 μs (above the time for a local invocation) represents cost of the potential for distribution: the time was spent checking whether or not the invoked object was resident.

6.7 A Local Procedure Call Benchmark

We used Ackermann's function as a benchmark program because most of its execution time is due to procedure calls; the only other operations performed are tests against zero and integer decrement. We wrote Ackermann's function in Emerald and in C. The Emerald version appears below:

```
function Ackermann[n: Integer, m: Integer] → [result: Integer]
    if (m = 0) then
        result ← n+1
    elseif (n = 0) then
        result ← self.Ackermann[1,m−1]
    else
        result ← self.Ackermann[
                    self.Ackermann[n−1, m], m−1]
    end if
end Ackermann
```

The C version was written twice: a straightforward version and a hand-optimized version. The straightforward version was timed when compiled both with and without the C optimizer. We compared execution times for two pairs of parameter values, namely (6,3) and (7,3). Table 6 shows the timings along with the relative difference in execution times normalized to the optimized C version. The Emerald version ran about 50% slower than the C version. When the C optimizer was used, the C timings improved by 12–13%. Careful hand optimization improved the timings for C by an additional 10%.

An analysis of the code generated by the C and Emerald compilers revealed that the Emerald version was slower than the C version for three reasons. First, as mentioned earlier, Emerald invocations were 23% slower than C procedure calls. Second, Emerald's parameter-passing mechanism was more expensive than C's because Emerald also transfered type information. Third, in Emerald all variables were initialized.

In 1992, Emerald was ported to the SUN SPARC architecture [75]. The SUN C compiler used the SPARC register window, while the Emerald implementation did not. As a consequence, Emerald invocations on the SPARC were almost 15% *faster* than C procedure calls.

[17] On a MicroVAX II using the Berkeley portable C compiler.

Table 4: Incremental Cost of Remote Invocation Parameters: elapsed time in addition to a call-by-reference invocation

Parameter Passing Mode	**Time**/ms
empty remote invocation	30.3
call-by-move	+ 3.4
call-by-visit	+ 7.8
call-by-reference, one call-back	+31.0

Table 5: Local Emerald Invocation Timing — MicroVAX II

Emerald Operation	**Example**	**Time**/μs
primitive integer invocation	$i \leftarrow i + 23$	0.4
primitive real invocation	$x \leftarrow x + 23.0$	3.4
local invocation	localobject.no-op	16.6
resident global invocation	globalobject.no-op	19.4

Table 6: Ackermann's Function Benchmark (Time in Seconds)

Version	**(6,3)**	**Δ%**	**(7,3)**	**Δ%**
C hand optimized	3.7	−10%	14.9	−10%
C with optimizer	4.1	0%	16.6	0%
C version	4.6	+12%	18.7	+13%
Emerald version	6.6	+61%	27.7	+67%

7. Summary

Emerald is a research programming language that was developed at the University of Washington from 1983 to 1987, and has subsequently been used extensively in teaching and research at the Universities of Copenhagen, British Columbia, and Arizona, at Digital Equipment Corporation, and at Victoria University of Wellington. It was originally implemented on DEC VAX hardware, later on the Motorola 68000-series, then on a portable virtual machine, then on Sun SPARC and Digital Alpha, and most recently in a heterogeneous setting in which objects run native code but can move between architectures. Emerald is object-based in the sense that it supports objects but not classes or inheritance; its distinguishing feature is support for distribution, which is arguably more complete than that of any other language available even now, more than 20 years later. Along the way, Emerald also made significant strides in type theory, including implementations of what have since become known as F-bounded polymorphism and generic data structures.

The primary problems that Emerald sought to address were the costs of object invocation and the coupling between the way that an object was represented in the programming language and the way that the object was implemented. Contemporary object-based distributed languages, notably Argus and the Eden Programming Language, had two notions of object: "small objects", which were efficiently supported within a single address space but could not be accessed remotely, and "large objects", which were accessible from remote address spaces but were thousands of times more costly. Emerald's contribution was the realization, obvious in hindsight, that these different implementations need not show through to the source language. Instead, the Emerald language has a single notion of object and several different implementation styles: the most appropriate implementation is selected by the compiler depending on how the object is used.

The idea that the users of an object should not know (or care) about the details of its implementation is of course no more than information hiding, a principle that was well known at the time that we were designing Emerald. In addition to using the principle of information hiding in the compiler (the user didn't have to know how the compiler implemented a object), we also made it available to the programmer. We did this by thoroughly separating the notions of class (how an object is implemented) from those of type (what interface it presents to other objects). Emerald's type system is based on the notion of structural type conformity, which rigorously specifies when an implementation of an object satisfies the demands of a type, and also when one type subsumes another.

Letting the compiler choose an implementation was only possible if this did not change the semantics, so we were

forced to define the semantics abstractly. Thus, the semantics of parameter passing cannot depend on the relative locations of the invoker and the invoked. Emerald does include features to control the location of objects: to move an object to a new location, to fix and unfix an object at a specific location, to move an object to a remote location with an invocation, and to cause an object to visit a remote location for the duration of an invocation. Using these features does not change the semantics of a program, but may dramatically change its performance. Emerald also distinguishes functions (which have no effect on the state) from more general operation invocations, and immutable objects (which the implementation can replicate) from mutable objects.

Emerald's main deficiency, viewed from the vantage-point of today, is its lack of inheritance. Given the absence of classes from the language itself and the decentralized implementation strategy, it was not clear how inheritance could be incorporated. After gaining some experience with the language, and in particular with the tedium of writing two-level object constructors all the time, we did add classes, including single inheritance, as "syntactic sugar" for the obvious two-level object constructor. The resolution of inherited attributes was done entirely at compile time, implying that the text of the superclass had to be available to the compiler when the subclass was compiled. In addition, this simple scheme offered no support for "super" or otherwise invoking inherited methods in the body of subclass methods: a method or instance variable in the subclass completely redefined, rather than just overrode, any identically named method or variable inherited from the superclass. Rather than pursuing more traditional inheritance in the language, subsequent work looked at compile-time code reuse [89].

One issue that the Emerald implementation never addressed was ensuring that a remote object that claimed to be of a certain type did in fact conform to that type. Type information was present in the run-time object structures in two forms: an object structure representing the type, which was used for conformity checking, and the executable code, which actually implemented the operations required by the type. While a correct Emerald compiler always guaranteed that the object structure and the code corresponded, a malicious user on a remote node could in principle hack a version of the compiler to void this guarantee, thus breaching the type security of the whole system. We imagined overcoming this problem by certifying the compiler and having each compiler sign its own code, but this was never implemented.

Although Emerald's support for remote invocation has been widely reproduced, remote invocation has rarely been implemented with such semantic transparency. Implementations of mobile objects are still rare and languages that incorporate mobility into their semantic model rarer still, although recently proposals have been made for incorporating mobility into Java.

Acknowledgements

This paper has taken shape over a long period, and has been much improved by the contributions of many colleagues. Our HOPL referees, Brent Hailpern, Doug Lea, and Barbara Ryder, all provided useful advice. Our external referee, Andrew Watson, along with Andrew Herbert, helped to reconstruct the influence of the Emerald type system. Kim Bruce, Sacha Krakowiak, Roy Levin, and Jim Waldo helped fill in numerous details, and Michael Mahoney helped us learn how to write history.

We thank the Danish Center for Grid Computing, Microsoft Research Cambridge, the University of Copenhagen, and Portland State University for their generous support while this paper was being written.

References

[1] J. Mack Adams and Andrew P. Black. On proof rules for monitors. *Operating Systems Review*, 16(2):18–27, April 1982.

[2] J. E. Allchin and M. S. McKendry. Synchronization and recovery of actions. In *Proceedings of the 2nd Symposium on Principles of Distributed Computing*, pages 31–44, New York, NY, USA, August 1983. SIGOPS/SIGACT, ACM Press.

[3] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE11(1):43–59, January 1985.

[4] Guy T. Almes. *Garbage Collection in an Object-Oriented System*. PhD thesis, Carnegie Mellon University, June 1980.

[5] Allan Thrane Andersen and Ole Høegh Hansen. Teoretiske og praktiske forhold ved brugen af mønstre i forbindelse med udvikling af objectorienteret software-med særlig fokus på sammenhængen mellem mønstre og programmeringssprog. M.Sc. thesis, DIKU, University of Copenhagen, 1999.

[6] ANSI. *Draft American National Standard for Information Systems — Programming Languages — Smalltalk*. ANSI, December 1997. Revision 1.9.

[7] Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, and Ann Wollrath. *The Jini Specification*. Addison Wesley, 1999.

[8] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Report on the algorithmic language Algol 60. *Communications of the ACM*, 3(5):299–314, 1960.

[9] L. Frank Baum. *The Wonderful Wizard of Oz*. George M. Hill, Chicago, 1900.

[10] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[11] John K. Bennett. The design and implementation of distributed Smalltalk. In Norman K. Meyrowitz, editor, *Proceedings of the Second ACM Conf. on Object-Oriented*

Programming Systems, Languages and Applications, pages 318–330, New York, NY, USA, October 1987. ACM Press. Published as SIGPLAN Notices**22**(12), December 1987.

[12] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 217–230, Asheville, NC (USA), December 1993. ACM Press.

[13] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. Technical Report 115, Digital Systems Research Center, Palo Alto, CA, December 1994.

[14] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[15] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrtag, and Kristen Nygaard. *Simula BEGIN*. Auerbach Press, Philadelphia, 1973.

[16] Andrew Black, Larry Carter, Norman Hutchinson, Eric Jul, and Henry M. Levy. Distribution and abstract types in Emerald. Technical Report 86–02–04, Department of Computer Science, University of Washington, Seattle, February 1986.

[17] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In Norman K. Meyrowitz, editor, *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86. ACM, October 1986. Published in SIGPLAN Notices, **21**(*11*), November 1986.

[18] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract data types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[19] Andrew Black, Norman Hutchinson, Eric Jul, and Henry M. Levy. Distribution and abstract types in Emerald. Technical Report 85–08–05, Department of Computer Science, University of Washington, Seattle, August 1985.

[20] Andrew P. Black. *Exception Handling: the Case Against*. D.Phil thesis, University of Oxford, January 1982. http://web.cecs.pdx.edu/~black/publications/Black%20D.%20Phil%20Thesis.pdf.

[21] Andrew P. Black. The Eden programming language. Technical Report TR 85-09–01, Department of Computer Science, University of Washington, September 1985.

[22] Andrew P. Black. Supporting distributed applications: experience with Eden. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 181–193, New York, NY, USA, 1985. ACM Press.

[23] Andrew P. Black. Object identity. In *Proc. of the Third Int'l Workshop on Object Orientation in Operating Systems (IWOOOS'93)*, Asheville, NC, December 1993. IEEE Computer Society Press.

[24] Andrew P. Black and Yeshauahu Artsy. Implementing location independent invocation. In *Proc. 9th International Conference on Distributed Computing Systems*, pages 550–559. IEEE Computer Society Press, June 1989.

[25] Andrew P. Black and Yeshauahu Artsy. Implementing location independent invocation. *IEEE Transactions on Parallel and Distributed Syst.*, 1(1):107–119, 1990.

[26] Andrew P. Black and Norman Hutchinson. Typechecking polymorphism in Emerald. Technical Report CRL 91/1, Digital Cambridge Research Laboratory, One Kendall Square, Building 700, Cambridge, MA 02139, December 1990.

[27] Andrew P. Black and Mark P. Immel. Encapsulating plurality. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *Lecture Notes in Computer Science*, pages 57–79, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[28] Toby Bloom. Immutable groupings. CLU Design Note 61, MIT—Project MAC, August 1976.

[29] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[30] Per Brinch Hansen. *The Architecture of Concurrent Programming*. Prentice Hall Series in Automatic Computation. Prentice Hall Inc., Englewood Cliffs, New Jersey, 1977.

[31] Andrei Z. Broder. Some applications of Rabin's fingerprinting method. In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[32] Kim B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–298, Charleston, South Carolina, United States, 1993. ACM Press.

[33] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object oriented languages. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127, Jyväskylä, Finland, June 1997. Springer-Verlag.

[34] Peter S. Canning, William Cook, Walter L. Hill, John C. Mitchell, and Walter G. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 273–280, September 1989.

[35] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer, 1984.

[36] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the Fifteenth Symposium on Principles of Programming Languages*, pages 70–79, San Diego, CA, January 1988. ACM.

[37] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[38] Luca Cardelli, Jim Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language

definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[39] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 type system. In *Proceedings of the Sixteenth Symposium on Principles of Programming Languages*, pages 202–212, Austin, Texas, United States, January 1989. ACM Press.

[40] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[41] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. Common base language. Technical Report S-22, Norwegian Computing Center, October 1970.

[42] Wolfgang De Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, Vrije Universiteit Brussel, September 2004.

[43] A. Demers and J. Donahue. Revised report on Russell. Technical Report TR 79-389, Computer Science Department, Cornell University, 1979.

[44] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[45] Bradley M. Duska. Enforcing crash failure semantics in distributed systems with fine-grained object mobility. M.Sc. thesis, Computer Science Department, University of British Columbia, August 1998.

[46] International Organization for Standardization. Information technology — open distributed processing — reference model. International Standard 10746, ISO/IEC, 1998.

[47] Arthur Neal Glew. Type systems in object oriented programming languages. B.sc. (hons) thesis, Victoria University of Wellington, 1993.

[48] Xiaomei Han. Memory reclamation in Emerald — an object-oriented programming language. M.Sc. thesis, Computer Science Department, University of British Columbia, June 1994.

[49] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[50] R. C. Holt. A short introduction to Concurrent Euclid. *SIGPLAN Notices*, 17(5):60–79, May 1982.

[51] R. C. Holt. *Concurrent Euclid, the UNIX System, and TUNIS*. Addison-Wesley Publishing Company, 1983.

[52] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In P. America, editor, *Proceedings ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38, Geneva, Switzerland, July 1991. Springer-Verlag.

[53] Norm Hutchinson and Eric Jul. The handling of unix signals in the concurrent euclid kernel. Eden memo, Dept. of Computer Science, University of Washington, 1984.

[54] Christian Damsgaard Jensen. Fine-grained object based load distribution — an experiment with load distribution in guide-2. M.Sc. thesis, DIKU, University of Copenhagen, 1995.

[55] Lars Rye Jeppesen and Søren Frøkjær Thomsen. Generational, distributed garbage collection for emerald. M.Sc. thesis, DIKU, University of Copenhagen, 2002.

[56] Nick Jørding and Flemming Stig Andreasen. A distributed wide area name service for an object oriented programming system. M.Sc. thesis, DIKU, University of Copenhagen, 1994.

[57] Eric Jul. Structuring of dedicated concurrent programs using adaptable I/O interfaces. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, December 1980. Technical Report no. 82/3.

[58] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. Ph.D. thesis, Department of Computer Science, University of Washington, Seattle, December 1988.

[59] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[60] Niels Christian Juul. *Comprehensive, Concurrent and Robust Garbage Collection in the Distributed, Object-Based System Emerald*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, February 1993. DIKU Report 93/1.

[61] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *International Workshop on Memory Management (IWMM)*, volume 637 of *Lecture Notes in Computer Science*, pages 103–115, 1992.

[62] Setrag N. Khoshafian and George P. Copeland. Object identity. In Norman K. Meyrowitz, editor, *Proceedings of the First ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 406–416, Portland, Oregon, October 1986. Published as SIGPLAN Notices 21(11), November 1986.

[63] Jan Kølander. Implementation af osi-protokoller i det distribuerede system emerald vha. isode. M.Sc. thesis, DIKU, University of Copenhagen, 1994.

[64] B. W. Lampson, J. G. Mitchell, and E. H. Satterthwaite. On the transfer of control between contexts. In *Lecture Notes in Computer Science: Programming Symposium*, volume 19, pages 181–203. Springer-Verlag, 1974.

[65] Niels Elgaard Larsen. An object-oriented database in emerald. M.Sc. thesis, DIKU, University of Copenhagen, 1992.

[66] Niels Elgaard Larsen. *Emerald Database — Integrating Transaction, Queries, and Method Indexing into a system based on Mobile Objects*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, February 2006.

[67] Edward D. Lazowska, Henry M. Levy, Guy T. Almes,

Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. The architecture of the Eden system. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 148–159, Pacific Grove, California, 1981. ACM Press.

[68] Morten Jes Lehrmann. Load distribution in Emerald — an experiment. M.Sc. thesis, DIKU, University of Copenhagen, 1994.

[69] Hank Levy, Norm Hutchinson, and Eric Jul. Getting to Oz, April 1984. Included as an appendix to this article.

[70] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[71] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[72] Barbara Liskov. A history of CLU. In Thomas A. Bergin and Richard G. Gibson, editors, *History of Programming Languages*, chapter 10, pages 471–510. ACM Press, New York, NY, USA, 1996.

[73] Barbara Liskov, Alan Snyder, Robert Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[74] Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Constraints and object identity. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, volume 821 of *Lecture Notes in Computer Science*, pages 260–279, Bologna, Italy, July 1994. Springer-Verlag.

[75] Jacob Marquard. Porting emerald to a sparc. M.Sc. thesis, DIKU, University of Copenhagen, 1992.

[76] Robert M. Metcalfe and David R. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[77] Albert R. Meyer and Mark B. Reinhold. 'Type' is not a type: Preliminary report. In *Proceedings of the Thirteenth Symposium on Principles of Programming Languages*, pages 287–295, St. Petersburg, January 1986. ACM.

[78] Jeppe Damkjær Nielsen. Paradigms for the design of distributed operating systems. M.Sc. thesis, DIKU, University of Copenhagen, 1995.

[79] E. Organick. *A Programmer's View of the Intel 432 System*. McGraw-Hill, 1983.

[80] Dimokritos Michael Papadopoulos and Kenneth Folmer-Petersen. Porting the LAN-based distributed system Emerald to a WAN. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, 1993.

[81] P. Pardyak. Group communication in a distributed object-based system. Master's thesis, University of Mining and Metallurgy, Krakow, Poland, September 1992. Technical Report TR-92-1.

[82] P. Pardyak. Group communication in an object-based environment. In *1992 Int. Workshop on Object Orientation in Operating Systems*, pages 106–116, Dourdan, France, 1992. IEEE Computer Society Press.

[83] P. Pardyak and B. Bershad. A group structuring mechanism for a distributed object-oriented language. In *Proc. of the 14th International Conf. on Distributed Computing Systems*, pages 312–319. IEEE Computer Society Press, July 1994.

[84] Przemyslaw Pardyak and Krzysztof Zielinski. Emerald — jezyk i system rozproszonego programowania obiektowego (Emerald — a language and system for distributed object-oriented programming). In *Srodowiska Programowania Rozproszonego w Sieciach Komputerowych, (Distributed Programming Environments in Computer Networks)*. Ksiegarnia Akademicka, Krakow, 1994.

[85] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[86] Margaret A. S. Petrus. Service migration in a gigabit network. M.Sc. thesis, Computer Science Department, University of British Columbia, August 1998.

[87] Simon Pohlen. Maintainable concurrent software. M.Sc. thesis, Victoria University of Wellington, April 1997.

[88] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[89] Rajendra K. Raj and Henry M. Levy. A compositional model for software reuse. In S. Cook, editor, *Proceedings ECOOP '89*, pages 3–24, Nottingham, July 1989. Cambridge University Press.

[90] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: a general-purpose programming language. *Software — Practice and Experience*, 21(1):91–118, 1991.

[91] Rajendra Krishna Raj. *Composition and reuse in object-oriented languages*. PhD thesis, Department of Computer Science, University of Washington, Seattle, WA, USA, 1991.

[92] Craig Schaffert. Immutable groupings. CLU Design Note 47, MIT—Project MAC, April 1975.

[93] Bodil Schrøder. Mik — et korutineorienteret styresystem til en mikrodatamat. DIKU Blue Report 76/1, DIKU, Dept. of Computer Science, 1976.

[94] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[95] Mark Sherman, Andy Hisgen, David Alex Lamb, and Jonathan Rosenberg. An Ada code generator for VAX 11/780 with Unix. In *Proceeding of the ACM-SIGPLAN symposium on Ada programming language*, pages 91–100, Boston, Massachusetts, 1980. ACM Press.

[96] Sys Sidenius and Eric Jul. K8080 — Flytning af Concurrent Pascal til Intel 8080. Technical Report 79/9, DIKU, Dept. of Computer Science, University of Copenhagen, 1979.

[97] Bjarne Steensgaard and Eric Jul. Object and native code thread mobility among heterogeneous computers. In *SOSP*, pages 68–78, 1995.

[98] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey*

Approach to Programming Language Theory. MIT Press, 1977.

[99] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing*. Prentice Hall, 1991.

[100] Bjarne Stroustrup. The history of C++: 1979–1991. In *Proc. ACM History of Programming Languages Conference (HOPL-2)*. ACM Press, March 1993.

[101] Andrew Tanenbaum. *Computer Networks*. Prentice-Hall, first edition, 1980.

[102] David M. Ungar. *The Design And Evaluation Of A High Performance SMALLTALK System*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1986.

[103] Adriaan van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintoff, C.H. Linsey, L.G.L.T. Meertens, and R.G. Fisker. *Revised Report on the Algorithmic Language Algol 68*. Springer Verlag, 1976.

[104] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Labs, November 1994.

[105] Timothy Walsh. *The Taxy Mobility System*. PhD thesis, Trinity College Dublin, June 2006.

[106] Andrew J. Watson. Advanced networked systems architecture — an application programmer's introduction to the architecture. Technical Report TR.017.00, APM Ltd., November 1991.

[107] Wikipedia. Duck typing, September 2006. http://en.wikipedia.org/wiki/Duck_typing.

[108] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: the kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

A. Appendix: *Getting to Oz*

The *Getting to Oz* document is reproduced on the following pages. This and other historical documents are available at http://www.emeraldprogramminglanguage.org.

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

an address space and multiple address spaces. Multiple address spaces would add protection, particularly between multiple languages if they were supported. Within a single address space the compiler would provide protection. A single scheduler within Oz would schedule both light and heavy processes, making the obvious tradeoffs to schedule lightweight processes within the same address space if possible. This would remove the overhead currently caused by the two-level scheduler in Eden.

We wanted lightweight processes for two reasons. First, switching between them involves much less work and is therefore more efficient. Second, and more important, we wanted to reduce the cost of many invocations to that of a procedure call or less. This requires that objects share an address space and leads to the next issue -- compiler support.

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?

Another goal, then, and a difficult one, is to design a language that supports both large objects (typically, operating system resources such as files, mailboxes, etc.) and small objects (typically, data abstractions such as queues, etc.) using a *single* semantics. Large objects 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

typically contain data only. Within Toto, all objects are defined the same way, however the compiler implements objects differently depending on their usage. In this way, we use invocation mechanisms whose performance is commensurate with the needs of the object. For example, different objects may be invoked through direct inline code, through procedure call, or through message passing.

Oz is intended to run on a set of homogeneous nodes on a local area network. Since we're experimenting with a distributed system, we decided that the concept of object location should be supported by the language. The movement of objects should be easily expressible in the language. In fact, this is an advantage of object-based programming that we wish to demonstrate. Other systems provide message passing or even process migration but have no clean concept of *what* it is that is moved. In Oz, objects are the unit of movement and an object can consist of data, a process, or both. The language includes statements that (1) determine the location of an object, (2) move an object to a particular site, and (3) fix an object at a particular site.

We are not changing the Eden concept of location-independent invocation. Rather, we say in Oz that invocation is location-independent but objects are not. That is, location is a fundamental part of each object's state. An Oz program can locate and move other objects. A load balancing object may locate and move objects based on system usage information. However, the semantics of object invocation are identical for local and remote objects.

We have recently discussed implementation strategies for invocation parameter passing. Parameter passing is made difficult by the distributed nature of the system and by the options in object size and processing (i.e., whether an object has a process or not). In general, inovcation parameters are passed by reference. Some small immutable objects may be passed by value for performance reasons; for example, we would not pass a reference to the integer 3. A new parameter passing mechanism we've considered is *call by move*, in which the invoked operation explicitly indicates that the parameter object should be moved to the location of the invoked object. This could, in fact, be the principal facility for object movement in Oz.

Finally, because we have a distributed system, the language must allow the programmer to deal with failures. In Oz, failures are *anticipated* conditions that arise due to the distributed nature of the system. For example, applications and nodes go up and down, links break, etc. Programs should be written to expect and deal with these failures. Program bugs, on the other hand, such as array out of bounds violations or divide by zero, are not handled in Oz. We don't intend to provide full-blown exception handling for these conditions.

To summarize, we are considering building an *object-based language* that supports:

1. objects (in particular, both large and small objects are supported by a single language semantics)
2. efficient invocation (both local and remote, both within an address space and outside an address space), and

3. distribution (objects have location and are potentially movable).

Except for these three issues, the rest of the language will be extremely simple. We are currently working on a preliminary language spec which will be available shortly.