ANDREW P. BLACK

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.