A Language with Distributed Scope

Luca Cardelli

Digital Equipment Corporation, Systems Research Center
130 Lytton Ave, Palo Alto, CA 94301, USA
luca@src.dec.com

Abstract

Obliq is a lexically-scoped, untyped, interpreted language that supports distributed object-oriented computation. Obliq objects have state and are local to a site. Obliq computations can roam over the network, while maintaining network connections. Distributed lexical scoping is the key mechanism for managing distributed computations.

1. Introduction

A simple guiding principle separates Obliq from other distributed procedural languages: adherence to lexical scoping in a distributed context. This principle has a number of interesting consequences: it supports a natural and consistent semantics of distributed computation, and enables elegant techniques for distributed programming.

In lexically scoped languages, the binding location of every identifier is determined by simple analysis of the program text surrounding the identifier. Therefore, one can be sure of the meaning of program identifiers, and can much more easily reason about the behavior of programs. In a distributed language like Obliq, lexical scoping assumes a further role. It ensures that computations have a precise meaning even when they migrate over the network: a meaning that is determined by the binding location and network site of identifiers, and not by execution sites.

Network-wide scoping becomes an issue in the presence of higher-order distributed computation, for example when a site acting as a compute server accepts procedures for execution. The question here is: what happens to the free identifiers of network-transmitted procedures? Obliq takes the view that such identifiers are bound to their original locations, as prescribed by lexical scoping, even when these locations belong to different network sites.

In the rest of this introduction we review the main notions. In section 2 we describe Obliq's object model and distributed semantics. In section 3 we illustrate the object model by means of single-threaded examples. In section 4 we present a collection of distributed programming techniques, enabled by Obliq's unique features. The most illuminating example is the compute server, in section 4.2; the most intriguing one is object migration, in section 4.6. The syntax is summarized in the appendix.

1.1 Language Overview

The principal way of structuring distributed computations in Obliq is through the notion of *objects*. Network services normally accept a variety of messages; it is then natural to see each service as a *network object* (or, more neutrally, as a network interface). Obliq supports objects in this spirit, relying for its implementation on Modula-3's network objects [7].

The Obliq object primitives are designed to be simple and powerful, with a coherent relationship between their local and distributed semantics. Obliq objects are collections of named fields, with four basic operations: selection/invocation, updating/overriding, cloning, and aliasing. There are no class hierarchies, nor complex method-lookup strategies. Every object is potentially and transparently a network object. An object may become accessible over the network either by the mediation of a name server, or simply by being used as the argument or result of a remote method.

In any framework where objects are distributed across sites, it is critical to decide what to do about mobility of *state*. To avoid problems with state duplication, objects in Obliq are local to a site and are never automatically moved over the network. In contrast, *network references* to objects can be transmitted from site to site without restrictions. Atomic object migration can be coded from our primitives, specifically from cloning and aliasing.

In addition to the distribution of data, the distribution of computations must also be designed carefully. It is clearly desirable to be able to transmit computing *agents* for remote execution. However, one should not be satisfied with transmitting just the program text of such agents. Program text cannot carry with it live connections to its originating site, nor to any data or service at any other site. Hence the process of transmitting program text over the network implies a complete network disconnect from the current distributed computation. In addition, unpredictable dynamic scoping results from transmitting and then running program text containing free identifiers.

Obliq computations, in the form of procedures or methods, can be freely transmitted over the network. Actual computations (*closures*, not source text) are transmitted; lexically scoped free identifiers retain their bindings to the originating sites. Through these free identifiers, migrating computations can maintain connections to objects and locations residing at various sites. Disconnected agents can be represented as procedures with no free identifiers; they do not rely on prolonged network connectivity.

In order to concentrate on distributed computation issues and to reduce complexity, Obliq is designed as an *untyped* language (lacking static typing). This decision leads to simpler and smaller language processors that can be easily embedded in applications. Moreover, untyped programs are somewhat easier to distribute, avoiding problems of compatibility of types at multiple sites.

The Obliq run-time, however, is *strongly typed*: erroneous computations produce clean errors that are correctly propagated across sites. The run-time data space is *heterogeneous*, meaning that there are different kinds of run-time values and no provisions to discriminate between them; heterogeneity discourages writing programs that would be difficult to typecheck in typed languages.

Because of heterogeneity and lexical scoping, Obliq is in principle suitable for static typing. More importantly, Obliq is compatible with the disciplined approach to programming that is inspired by statically typed languages.

1.2 Distributed Semantics

The Obliq distributed semantics is based on the notions of *sites*, *locations*, *values*, and *threads*.

Sites (that is, address spaces) contain locations, and locations contain values. Each location belongs to a unique site. Sites are not explicit in the syntax but are implicit in the creation of locations: when a location is created during a computation, it is allocated at the current site.

Threads are virtual sequential instruction processors. Multiple threads may be executed concurrently, both at the same site or at different sites. A given thread may stop executing at a site, and continue executing at another site. That is, threads may jump from site to site while retaining their conceptual identity.

In the Obliq syntax, *constant identifiers* denote values, while *variable identifiers* denote locations. A location containing a value may be updated by assignment to the variable denoting the location.

Obliq values include *basic values* (such as strings and integers), *objects*, *arrays*, and *closures* (the results of evaluating methods and procedures).

A value may contain *embedded locations*. An array value has embedded locations for its elements, which can be updated. An object value has embedded locations for its fields, which can be updated. A closure value may have embedded locations because of free variables in its program text that refer to locations in the surrounding lexical scope.

Values may be *transmitted* over the network. A value containing no embedded locations is copied on transmission. A value containing embedded locations is copied up to the point where those locations appear; local references to locations are replaced by network references. Because of transmission, a value may thus contain network references to locations at different sites. This semantics of value transmission has particular implications for closure values.

In general terms, a closure is a pair consisting of a piece of source text and a pointer to an evaluation stack. Transmission of a closure, in this view, implies transmission of an entire evaluation stack. Obliq, however, implements each closure as a pair of a source text and a table of values for free identifiers; this technique is well-known and applicable to lexically-scoped higher-order languages. In our context, this implementation of closures has the effect of reducing network traffic by transmitting only the values from the evaluation stack that are needed by the closure. A closure that has been transmitted may thus contain program text that, when executed, accesses remote locations (via its table of free identifiers) over the network.

Every Obliq object consists of a collection of locations spanning a single site; hence the object itself is bound to a unique site, and does not move[1]. This immobility of objects is not a strong limitation, because objects can be *cloned* to different sites, and because procedures can be transmitted that allocate objects at different sites. Hence, a collection of interacting objects can be dynamically allocated throughout the network. If migration is necessary, cloning can be used to provide the needed state duplication, and aliasing can be used to redirect operations to the clones.

We have stressed so far how Obliq computations can evolve into webs of network references. However, this is not necessarily the case. For example, a procedure with no free identifiers forms a completely self-contained computing agent. The execution of such an agent may be carried out autonomously by a remote compute server; the agent may dynamically reconnect to the originating site to report results. Intermediate situations are also possible, as with semi-autonomous agents that maintain low-traffic tethers to their originating site for status queries.

[1] In the implementation, network references are generated to objects and arrays, not to each of their embedded locations. However, it is consistent and significantly simpler to carry out our discussions in terms of network references to locations.

1.3 Discussion

The distributed semantics of Obliq is defined so that data and computations are network-transparent: their meaning does not depend on allocation sites or execution sites (of course, computations may receive different arguments at different sites). At the same time, Obliq programs are network-aware: distribution is achieved by explicit acts that give full control on communication patterns. Central to network transparency is the notion of distributed lexical scoping.

The combination of lexical scoping with strong run-time typing and interpreted execution can provide network security guarantees. Consider the situation of a server executing incoming foreign agents. Because of lexical scoping, these agents have access only to the data and resources that they can reference via free identifiers or that they explicitly receive in the form of procedure parameters. Hence, foreign agents cannot access data or resources at the server site that are not explicitly given to them. As a concrete example, operations on files in Obliq require file system handles that are provided only as global lexically-bound identifiers at each site. A foreign agent can use the file system handle of its originating site, simply by referring to it as a free identifier. But the file system handle at the server site is outside its lexical scope, and hence unobtainable except with the cooperation of the server. Degrees of file protection can be represented by file system handles with special access rights.

In summary, distributed lexical scoping makes it easy to spread computations over multiple network sites, since computations are likely to behave correctly even when they are carried out at the wrong place (by some measure). This flexibility in distribution can, however, result in undesirable network traffic. Obliq relieves some of the burden of distributing data and computations, but care and planning are still required to achieve satisfactory distributed performance.

2. Objects

Obliq is an object-oriented language based on objects, rather than classes. An object is a self-contained exemplar of behavior that can be either constructed directly or cloned from other objects. The Obliq language is therefore *prototype*-based [10], but is not *delegation*-based [22]. Obliq belongs to a category of prototype-based languages that we may call *embedding*-based[2] [31]. This name indicates that all the methods of an object, as well as its value fields, are embedded in the object itself (at least in principle) rather than being located in other objects or classes. In spirit, this model is close to Borning's original prototype-based proposal [10], and to recent languages that are not delegation-based [9, 30].

The embedding-based model is straightforward, and is well suited to network applications because of the self-contained nature of the objects. The delegation-based model, in contrast, maximizes sharing across objects; this is not always desirable in a distributed context. For example, when an Obliq object is cloned over the network it carries with it its embedded methods, thus it can work locally and autonomously when it reaches its destination. In a delegation-based model it would be more difficult to obtain the complete relocation of an object and its methods. Typically, this would require the coordinated migration of the object's *parents* [33], and would affect other objects that share the same parents.

2.1 Fields

An Obliq object is a collection of fields containing methods, aliases, or other values. A field containing a method is called a *method field*. A field containing an alias is called an *alias field*. A field containing any other value, including a procedure value, is called a *(proper) value field*. Each field is identified by a *field name*. Syntactically, an object with n fields has the form:

$$\{\ x_1 \Rightarrow a_1,\ \ldots\ , x_n \Rightarrow a_n\ \}$$

where n≥0, and x_i are distinct field names. The terms a_i are *siblings* of each other, and the object is their *host object*.

A value field is, for example:

```
x => 3
```

A method field has the form:

$$x \Rightarrow \textbf{meth}(y, y_1,\ \ldots\ , y_m)\ b\ \textbf{end}$$

The first parameter, `y`, denotes *self*: the method's host object. The other parameters, for m≥0, are supplied during method invocation. The body of the method is `b`, which computes the result of an invocation of `x`.

Methods and procedures are supported as distinct concepts. Procedures start with the keyword **`proc`** instead of **`meth`** and have otherwise the same syntax. The main differences between the two are as follows. Methods can be manipulated as values but can be activated only when contained in objects, since self needs to be bound to the host object. In contrast, procedures can be activated by normal procedure call. Further, a procedure can be inserted in a value field and later recovered, while any attempt to extract a method from an object results in its activation.

An alias field has the form:

```
x => alias y of b end
```

Operations on the `x` field of this object are redirected to the `y` field of the object `b`. If that field is another alias, the

---

[2] The terms *concatenation*-based and *copy*-based have also been used.

redirection continues recursively. (However, aliasing operations are not themselves redirected; see section 2.3.)

As we said, Obliq fields (including methods) are stored directly in objects, not indirectly in classes or shared prototypes. Therefore, field lookup is a one-step process that searches a field by name within a single object: there is no class or delegation hierarchy to be searched iteratively. Field lookup is based on a nearly constant-time caching technique that does not penalize large objects. A separate cache is used for each operation instance; the cache records the position where a field was last found in an object [15].

2.2 Simple Examples

Let us examine some simple examples, just to became familiar with the Obliq syntax and semantics. A full explanation of object operations is given in the next section.

The following object has a single method that invokes itself through self (the `s` parameter). A **`let`** definition binds the object to the identifier `o`:

```
let o = { x => meth(s) s.x() end };
```

An invocation of `o.x()` results in a divergent computation. Divergence is obtained here without any explicit use of recursion: the self-application implicit in method invocation is sufficient.

The object below has three components: a value field `x`, a method `inc` that increments `x` through self and returns self, and a method `next` that invokes `inc` through self and returns the `x` component of the result.

```
let o =
 { x => 3,
   inc => meth(s,y) s.x := s.x+y; s end,
   next => meth(s) s.inc(1).x end };
```

Here are some operations that can be performed on `o`:

`o.x`	Selecting the `x` component.
`o.x := 0`	Setting the `x` component to zero.
`o.inc(1)`	Invoking a method, with parameters.
`o.next()`	Invoking a method with no parameters.
`o.next := meth(s) clone(s).inc(1).x end`	Overriding the `next` method so that it no longer modifies its host object.

2.3 Operations

We now examine the object operations in some detail. Apart from object creation, there are four basic operations on objects.

Selection (and Invocation)

This operation has two variants for value selection and method invocation:

```
a.x
a.x(b1, ... ,bn)
```

The first form selects a value from a value field `x` of `a` and returns it. The second form invokes a method from a method field `x` of `a`, supplies $n \geq 0$ parameters, and returns the result produced by the method; the object `a` is bound to the self parameter of the method. For convenience, the first form can be used for invocation of methods with zero parameters.

When a value field of a remote object is selected, its value is transmitted over the network to the site of the selection (see the transmission semantics in section 1.2). When a method of a remote object is invoked, the arguments are transmitted over the network to the remote site, the result is computed remotely, and the final value (or error, or exception) is returned to the site of the invocation.

Updating (and Overriding)

This operation deals with both value field update and method field override:

```
a.x := b
```

Here the field `x` of `a` is updated with a new value `b`. If `x` contains a method and `b` is a method, we have method override. If `x` and `b` denote ordinary values, we have value update. The other two possibilities are also allowed: a value field can be turned into a method field, and vice versa.

When a field of a remote object is updated, a value is transmitted over the network and installed into the remote object. Remote method override involves the transmission of a method closure.

Cloning

Our third operation is object cloning, generalized to multiple objects:

```
clone(a1, ... ,an)
```

In the case of a single argument, a new object is created with the same field names as the argument object; its fields are initialized to the similarly named values, methods, and aliases of the argument object.

In the case of $n \geq 2$ arguments, a single object is produced that contains the values, methods, and aliases of all the argument objects (an error is given in case of field name conflicts). Useful idioms are **`clone`**`(a, {...})`, to *inherit* the fields of `a` and add new fields, and **`clone`**`(a1, a2)`, to *multiply inherit* from a_1 and a_2.

When a collection of remote or local objects is cloned, the clone is created at the local site. Its contents (including method closures) may have to be fetched over the network.

Aliasing

Our final operation is aliasing, which is the replacement of field contents with aliases (section 2.1). The syntax is similar to updating, but this is really a separate operation:

```
a.x := alias y of b end
```

Further operations on x of a are redirected to y of b; either object may be local or remote. An aliasing operation replaces field contents with aliases regardless of whether those fields are already aliased.

For a method invocation `a.x(c)`, the field `x => alias y of b end` behaves just like the field `x => meth(s,z) b.y(z) end`; that is, an aliased invocation behaves like an indirect method invocation. However, aliasing redirects also method override, as well as value selection and value update.[3]

A special construct can be used to alias all the components of an object at once:

```
redirect a1 to a2 end
```

The effect is to replace every field x_i of a_1 (including alias fields) with **alias** x_i **of** a_2 **end**; this is particularly useful for network redirection.

Aliasing is implicit in the distributed-systems notion of *local surrogate* of a remote object: we have simply lifted this mechanism to the language level. By doing this, we are able to put network redirection under flexible program control, as shown later in the case of object migration.

For method invocation, aliasing redirections behave differently from the redirections typical of delegation-based languages [22]: in aliasing, self is bound to the redirection target; in delegation, self is bound to the redirection source. Aliasing is more satisfactory than delegation when the redirection target is a remote object: after an initial aliasing redirection over the network, further accesses to self are local.

2.4 Self-inflicted Operations

Our four basic object operations can be performed either as external operations on an object, or as internal operations through self. This distinction is useful in the contexts of object protection and serialization, discussed in the next two sections, which are essential features of distributed objects. In preparation, we discuss the general notion of self-inflicted operations.

When a method operates on an object other than the method's host object, we say that the operation is *external* to the object. By contrast, when a method operates directly on its own self, we say that the operation is *self-inflicted*:

- If *op*(o) has the form `o.x`, `o.x:=b`, `clone(...,o,...)`, or `o.x:=alias...end`, then *op*(o) is *self-inflicted* (on o) iff o is the same object as the self of the *current method*.
- *op*(o) is *external* (on o) iff it is not self-inflicted.

Here, by the *current method* (if it exists) we mean the last method that was invoked in the current thread of control and that has not yet returned. Therefore, the notion of self for self-inflicted operations is preserved through procedure calls, but not through external method invocations or thread creation.

Whether an operation is self-inflicted can be determined by a simple run-time test. Consider, for example the object:

```
{ p => meth(s) s.q.x end,  q => ... }
```

Here the operation `s.q` is self-inflicted, since `s` is self. But the `.x` operation in `s.q.x` is self-inflicted depending on whether `s.q` returns self; in general this can be determined only at run-time.

2.5 Protected Objects

It is useful to protect objects against certain external operations, to safeguard their internal invariants. Protection is particularly important, for example, to prevent clients from overriding methods of network services, or from cloning servers. Even protected objects, though, should be allowed to modify their own state and to clone themselves.

A *protected* object is an object that rejects external update, cloning, and aliasing operations, but that admits such operations when they are self-inflicted. The syntax is:

```
{ protected, x1 => a1, ... , xn => an }
```

Therefore, for example, methods of a protected object can update sibling fields through self, but external operations cannot modify such fields.

Note that a protection mechanism based on individual fields would not address protection against cloning.

2.6 Serialized Objects

An Obliq server object can be accessed concurrently by multiple remote client threads. Moreover, local concurrent threads may be created explicitly. To prevent race conditions, it must be possible to serialize access to objects and their state.

We say that an object is *serialized* when (1) at most one thread at a time can operate on the object or run one of its methods. Moreover, we want to ensure that (2) a method can call a sibling through self without deadlock. Note that requirement (2) does not contradict invariant (1).

The obvious approach to implementing serialized objects, adopted by many concurrent languages, is to asso-

[3] Note that, for simplicity, we delayed the discussion of redirection in our previous explanation of selection and update.

ciate a *mutex* with each object (for example, see [3]). Such mutexes are acquired when a method of an object is invoked, and released when the method returns, guaranteeing condition (1). This way, however, we have a deadlock whenever a method calls a sibling, violating condition (2). We find this behavior unacceptable because it causes innocent programs to deadlock without good reason. In particular, an object that works well sequentially may suddenly deadlock when a mutex is added. Brewer and Waldspurger [11] give an overview of previous solutions to this serialization problem.

A way to satisfy conditions (1) and (2) together is to use reentrant mutexes, that is, mutexes that do not deadlock when re-locked by the "same" thread (for example, see [17]).

On the one hand, reentrant mutexes may be too liberal, because they allow a method to call a method of a different object, which then can call back a method of the present object without deadlocking. This goes well beyond our simple desire that a method should be able to call its siblings; object invariants may be compromised, since objects become vulnerable to unexpected activations of their methods.

On the other hand, reentrant mutexes may be too restrictive, because the notion of "same" thread is normally restricted to an address space. If we want to consider control threads as extending across sites, then an implementation of reentrant mutexes might not behave appropriately.

We solve the serialization problem by adopting an intermediate locking strategy, which we call *self serialization*, based on the notion of self-inflicted operations described in section 2.4.

Serialized objects have an implicit associated mutex, called the object mutex. An object mutex serializes the execution of selection, update, cloning, and aliasing operations on its host object, according to the following rules of acquisition:

- External operations always acquire the mutex of an object, and release it on completion.
- Self-inflicted operations never acquire the mutex of their object.

Note that a self-inflicted operation can happen only after the activation of an external operation on the object that is executed by the same thread. The external operation has therefore already acquired the mutex.

The serialization attribute of an object is specified as follows:

{ serialized, x_1 **=>** a_1, ... ,x_n **=>** a_n **}**

With self-serialization, a method can modify the state of its host object and can invoke siblings without deadlocking. A deadlock still occurs if, for example, a method invokes a method of a different object that then attempts an operation on the original serialized object. A deadlock occurs also if a method forks an invocation of a sibling and waits on the result.

In addition to mutual exclusion, Obliq provides *conditional synchronization* over implicit object mutexes. Conditional synchronization (where threads wait on a mutex and a condition) allows multiple threads to be simultaneously present "inside" an object, although at most one thread is active at any time. Producer-consumer behavior can be handled this way [5].

A **watch** statement is provided to wait on a condition in conjunction with the implicit mutex of an object. This statement must be used inside a method of a serialized object; hence, it is always evaluated with the object mutex locked:

watch *c* **until** *guard* **end**

The **watch** statement evaluates *c* to a condition and, if *guard* evaluates to **true**, terminates leaving the mutex locked. If the guard is **false**, the object mutex is unlocked (so that other methods of the object can execute) and the thread waits for the condition to be signaled. When the condition is signaled, the object mutex is locked and the boolean guard is evaluated again, repeating the process.

The interaction of conditional synchronization with certain object operations requires some attention. Objects with implicit mutexes can be cloned: a fresh implicit mutex is created for the clone. Consider then the case of a thread blocked on a condition within an object that is being cloned: the thread remains blocked within the original object, not the clone. Consider now the case of a thread blocked on a condition within a method that is being overridden or aliased. When the thread resumes, the blocked method runs to completion with a non-trivially modified self. Object protection, when used in conjunction with serialization, alleviates these worries since it prevents external cloning and updates.

In summary, mutual exclusion, amended for self-inflicted operations, handles common situations conveniently, for example for network servers maintaining some internal state. In addition, conditional synchronization can be used for standard concurrency-control problems. More complex situations may require sophisticated uses of explicit mutexes; for this, Obliq supports the full spectrum of Modula-3 thread primitives [5, 20]. Explicit mutexes, conditions, and threads cannot be transmitted, since these values are strongly site-dependent.

There is no automatic serialization for variables or arrays. If necessary, their access can be controlled through serialized objects or explicit mutexes. Even for objects, serialization is neither compulsory nor a default, since its use is not always desirable. In some cases it may be sufficient to serialize server objects (the concurrent entry points to a site) and leave all other objects unserialized.

2.7 Name Servers and Execution Engines

Obliq values can flow freely from site to site along communication channels. Such channels are initially established by interaction with a name server. A name server for Obliq programs is an external process uniquely identified by an IP address; it simply maintains a table associating text strings with network references [8].

The connection protocol between two Obliq sites is as follows. The first site registers a local, or remote, object under a certain name with a known name server. The second site asks the name server for (the network reference to) the object registered under that name. At this point the second site acquires a direct network reference to the object living in the first site. The name server is no longer involved in any way, except that it still holds the network reference. Obliq values and network references can now flow along the direct connection between the two sites, without having to be registered with a name server. This protocol is coded as follows, using a built-in `net` module:

Server Site:

```
net_export("obj", Namer, site1Obj);
```

Client Site:

```
let site1Obj =
 net_import("obj", Namer);
site1Obj.opA(args);                 (remote invocation)
site3Obj.opB(site1Obj);    (re-export to a third site)
```

where `"obj"` is the registration name for the object, `site1Obj` is the object, and *`Namer`* is a string containing the IP address or name of the machine running the desired name server. The object is now available through the name server, as long as the site that exports it is alive. Objects are garbage collected at a site when they are no longer referenced, either locally or via the network [6].

We shall see soon that compute servers are definable via simple network objects. However, compute servers are so common and useful that we provide them as primitives, calling them *execution engines*. An execution engine accepts Obliq procedures (that is, procedure closures) from the network and executes them at the engine site. An engine can be exported from a site via the primitive:

Server Site:

```
net_exportEngine("Engine1@Site1", Namer,
 arg);
```

The `arg` parameter is supplied to all the client procedures received by the engine. It may contain local data as well as site-specific procedures (*services* [29]). Multiple engines can be exported from the same site under different names.

An engine, once imported, behaves like a procedure of one argument. Implementing engines as remote procedures, instead of as remote objects, allows self-inflicted operations to extend across sites; this turns out to be important for object migration, as discussed in section 4.6.

A client may import an engine and then provide a procedure to be executed remotely.

Client Site:

```
let atSite1 =
 net_importEngine("Engine1@Site1", Namer);
atSite1(proc(arg) 3+2 end);
```

Communication failures produce exceptions that can be trapped. These failures may mean that one of the machines involved has crashed, or that an Obliq address space was terminated. There is no automatic recovery from network failures.

3. Local Techniques

In this section we discuss a collection of single-threaded examples to illustrate Obliq's sequential features. A collection of concurrent and distributed examples is given in section 4; the impatient reader may want to skip forward. In both these sections the emphasis is on advanced, rather than tutorial, examples.

3.1 Recursion and Iteration

We start with a simple example, to illustrate the use of definitions, local variables, and control constructs. The factorial function is defined in recursive and iterative style.

```
let rec recFact =
 proc(n)
  if n is 0 then 1
  else n * recFact(n-1)
  end;
 end;

let itFact =
 proc(n)
  var cnt = n; var acc = 1;
  loop
   if cnt is 0 then exit end;
   acc := cnt * acc; cnt := cnt - 1;
  end;
  acc;
 end;
```

Identifiers are declared by **`let`**, and updatable variables by **`var`**. Recursive definitions are obtained by **`let rec`**. The identity predicate is called **`is`**. A sequence of statements separated by semicolons returns the value of the last statement; hence the iterative factorial program returns `acc`.

3.2 The Object-Oriented Numerals

The next example illustrates the expressive power of the object primitives by encoding the natural numbers purely in terms of objects.

```
let zero =
 { case =>
    proc(pz,ps) pz() end,
   succ =>
    meth(self)
     let o = clone(self);
     o.case := proc(pz,ps) ps(self) end;
     o
    end };
```

The numeral `zero` has two fields. The `succ` field produces successive numerals by appropriately modifying the current numeral. The `case` field is used to discriminate on zero: the idiom `(n.case)(`**`proc`**`() b `**`end`**`, `**`proc`**`(p) c `**`end`**`)` is read, informally, as "if `n` is zero then return `b`, else bind the predecessor of `n` to `p` and return `c`".

The code of the `succ` method depends heavily on Obliq peculiarities: it clones self, and embeds the current self into a procedure closure, so that it can be used later. For example, the numeral `one`, computed as, `zero.succ()`, is:

```
{ case => proc(pz,ps) ps(zero) end,
  succ => (as for zero) }
```

Hence, `one.case(pz,ps)` correctly applies `ps` to the predecessor of `one`.

To show that the encoding is fully general, we define the successor, the predecessor, and the test for zero procedures:

```
let succ =
 proc(n) n.succ end;

let pred =
 proc(n)
  (n.case)(proc() zero end, proc(p) p end)
 end;

let iszero =
 proc(n)
  (n.case)
   (proc() true end, proc(p) false end)
 end;
```

3.3 The Prime Numbers Sieve

This example shows an interesting case of methods overriding themselves, and of objects replicating themselves by cloning. The program below prints the prime numbers when the method `m` of the `sieve` object is invoked with successive integers starting from 2. Each time a new prime p is found, the sieve object clones itself into two objects. One of the clones then transforms itself into a filter for multiples of p; non-multiples are passed to the other clone. **`ok`** is a trivial constant.

```
let sieve =
 { m =>
    meth(s, n)
     print(n);                    (defined elsewhere)
     let s0 = clone(s);
     s.m :=
      meth(s1,n1)
       if (n1 % n) is 0 then ok
       else s0.m(n1)
       end
      end;
    end };
```

(print the primes < 100)

```
for i = 2 to 100 do sieve.m(i) end;
```

At any point in time, if n primes have been printed, then there exists n filter objects plus a clone of the original sieve object.

3.4 A Calculator

This example illustrates method overriding, used here to store the "pending operations" of a pocket calculator.

```
let calc =
 { arg => 0.0,          (the "visible" argument display)
   acc => 0.0,          (the "hidden" accumulator)

   enter =>             (entering a new argument)
    meth(s, n)
     s.arg := n;
     s
    end,

   add =>               (the addition button)
    meth(s)
     s.acc := s.equals;
     s.equals := meth(s) s.acc+s.arg end;
     s
    end,

   sub =>               (the subtraction button)
    meth(s)
     s.acc := s.equals;
     s.equals := meth(s) s.acc-s.arg end;
     s
    end,

   equals =>            (the result button, and operator stack)
    meth(s) s.arg end,
```

```
  reset =>                         (the reset button)
   meth(s)
    s.arg := 0.0;
    s.acc := 0.0;
    s.equals := meth(s) s.arg end;
    s
   end };
```

For example:

```
calc.reset.enter(3.5).equals;                    (3.5)
calc.reset.enter(3.5).sub.enter(2.0)
 .equals;                                        (1.5)
calc.reset.enter(3.5).add.equals;                (7.0)
calc.reset.enter(3.5).add.add.equals;           (10.5)
```

3.5 Surrogates

Here we create a non-trivial surrogate for the calculator object of section 3.4. Unlike the original calculator, this object is protected against outside interference. Some of the calculator fields are shared by aliasing, some are hidden, some are renamed, and one is added.

```
let publicCalc =
 { protected,
   enter => alias enter of calc end,
   pi => meth(s) s.enter(3.141592) end,
   plus => alias add of calc end,
   minus => alias sub of calc end,
   equals => alias equals of calc end,
   reset => alias reset of calc end };
```

4. Distributed Techniques

In this section we code some distributed programming techniques in Obliq. Each example is typical of a separate class of distributed programs, and illustrates the unique features of Obliq.

4.1 A Serialized Queue

We begin with an example of ordinary concurrent programming to illustrate the threads primitives that are used in the sequel. We implement a queue that can be accessed consistently by concurrent reader and writer threads.

The queue is implemented as a serialized object with `read` and `write` methods. These methods refer to free identifiers that are hidden from users of the queue. The object mutex is used, implicitly, to protect a private variable that contains an array of queue elements. Another private variable contains a *condition* `nonEmpty` used for signaling the state of the queue.

The write method adds an element to the queue, and *signals* the non-empty condition, so that at least one reader thread waiting on that condition wakes up (a similar *broadcast* operation wakes up all waiting threads). The object mutex is locked throughout the execution of the write method, therefore excluding other writer or reader threads.

When a read method starts executing, the object mutex is locked. Its first instruction is to watch for the non-empty condition, and for the existence of elements in the queue. If the queue is non-empty, the reader simply goes ahead and removes one element from the queue. If the queue is empty, the reader thread is suspended and the object mutex is released (allowing other reader and writer threads to execute). The reader is suspended until it receives a signal for the non-empty condition; then the object mutex is locked, and the reader thread proceeds as above (possibly being suspended again if some other reader thread has already emptied the queue).

What is important here is that a reader thread may be blocked inside a method, and yet a writer thread can get access and eventually allow the first thread to proceed. Hence, even though only one thread at a time can run, multiple threads may be simultaneously present “in” the object.

Here, `[...]` is an array, `#` is array-size, and `@` is array-concatenation.

```
let queue =
 (let nonEmpty = condition();
  var q = [];                      (the hidden queue data)

  { protected, serialized,
    write =>
     meth(s, elem)
      q := q @ [elem];             (append elem to tail)
      signal(nonEmpty);            (wake up readers)
     end,

    read =>
     meth(s)
      watch nonEmpty               (wait for writers)
      until #(q)>0                 (check size of queue)
      end;
      let q0 = q[0];               (get first element)
      q := q[1 for #(q)-1];        (remove from queue)
      q0;                          (return first element)
     end; });
```

Let us see how this queue can be used. Suppose a reader is activated first when the queue is still empty. To avoid an immediate deadlock, we fork a thread running a procedure that reads from the queue; this thread blocks on the **`watch`** statement. The reader thread is returned by the **`fork`** primitive, and bound to the identifier `t`:

```
let t =                    (fork a reader t, which blocks)
 fork(proc() queue.read() end, 0);
```

Next we add an element to the queue, using the current thread as the writer thread. A non-empty condition is immediately signaled and, shortly thereafter, the reader thread returns the queue element.

```
queue.write(3);                    (cause t to read 3)
```

The reader thread has now finished running, but is not completely dead because it has not delivered its result. To obtain the result, the current thread is joined with the reader thread:

```
let result = join(t);              (get 3 from t)
```

In general, `join` waits until the completion of a thread and returns its result.

4.2 Compute Servers

The compute server defined below receives a client procedure `p` with zero arguments via the `rexec` method, and executes the procedure at the server site. This particular server cheats on clients by storing the latest client procedure into a global variable `replay`. Another field, `lexec`, is defined similarly to `rexec`, but `rexec` is a method field, while `lexec` is a value field containing a procedure: the operational difference is discussed below.

Server Site:

```
var replay = proc() end;
net_export("ComputeServer", Namer,
 { rexec => meth(s, p) replay:=p; p() end,
   lexec => proc(p) replay:=p; p() end });
```

A client may import the compute server and send it a procedure to execute. The procedure may have free variables at the client site; in this example it increments a global variable `x`:

Client Site:

```
let computeServer =
 net_import("ComputeServer", Namer);
var x = 0;
computeServer.rexec(proc() x:=x+1 end);
```

(now x = 1)

When the server executes its `rexec` method, `replay` is set to (a closure for) `proc() x:=x+1 end` at the server site, and then `x` is set to `1` at the client site, since the free `x` is lexically bound to the client site. Any variable called `x` at the server site, if it exists, is a different variable and is not affected. At the server site we may now invoke `replay()`, setting `x` to `2` at the client site.

For contrast, consider the execution of the following line at the client site:

Client Site:

```
(computeServer.lexec)(proc() x:=x+1 end);
```

This results in the server returning the procedure `proc(p) replay:=p; p() end` to the client, by the semantics of remote field selection, with `replay` bound at the server site. Then the client procedure `proc() x:=x+1 end` is given as an argument. Hence, this time, the client procedure is executed at the client site. Still, the execution at the client site causes the client procedure to be transmitted to the server and bound to the `replay` variable there. The final effect is the same.

4.3 Remote Agents

Execution engines (section 2.7) can be used as general object servers; that is, as ways of allocating objects at remote sites. These objects can then act as *agents* of the initiating site, supporting multiple requests.

Suppose, for example, that we have an engine exported by a database server site. The engine provides the database as an argument to client procedures:

DataBase Server Site:

```
net_exportEngine("DBServer", Namer,
 dataBase);
```

A database client could simply send over procedures performing queries on the database (which, for complex queries, would be more efficient than repeatedly querying the server remotely [19, 29]). However, for added flexibility, the client can instead create an object at the server site that acts as its remote agent:

DataBase Client Site:

```
let atDBServer =
 net_importEngine("DBServer", Namer);

let searchAgent =
 atDBServer(
  proc(dataBase)
   { state => ...,
     start => meth ... end,
     report => meth ... end,
     stop => meth ... end }
  end);
```

The execution of the client procedure causes the allocation of an object at the server site with methods `start`, `report`, and `stop`, and with a `state` field. The server simply returns a network reference to this object, and is no longer engaged. (Client resources at the server site are released when the client garbage collects the search agent, or when the client site dies [6].)

We show below an example of what the client can now do. The client starts a remote search via `start` from background thread, and periodically request a progress report via `report`. If the search is successful within a given time period, everything is fine. If the search takes too long, the remote agent is aborted via `stop`. If an intermediate report proves promising, the client may decide to wait for however long it takes for the agent to complete, by joining the background thread.

DataBase Client Site:

```
let searchThread =
 fork(proc() searchAgent.start() end, 0);

var report = "";
for i = 1 to 10 do
 pause(6.0);
 report := searchAgent.report();
 if successful(report) then exit end;
 if promising(report) then
  report := join(searchThread); exit;
 end;
end;
searchAgent.stop();
```

This technique for remotely allocating objects can be extended to multiple agents searching multiple databases simultaneously, and to agents initiating their own sub-agents.

4.4 Application Partitioning

The technique for remotely allocating objects described in section 4.3 can be used for *application partitioning*. An application can be organized as a collection of procedures that return objects. When the application starts, it can pick a site for each object and send the respective procedure to a remote engine for that site. This way, the application components can be (initially) distributed according to dynamic criteria.

4.5 Agent Migration

In this example we consider the case of an untethered agent that moves from site to site carrying along some state [34]. We write the state as an object, and the agent as a procedure parameterized on the state and on a site-specific argument:

```
let state = { ... };
let agent = proc(state, arg) ... end;
```

To be completely self-contained, this agent should have no free identifiers, and should use the state parameter for all its long-term memory needs.

The agent can be sent to a new site as follows, assuming `atSite1` is an available remote engine:

```
atSite1(
 proc(arg) agent(copy(state),arg) end)
```

The **copy** operation is explained below, but the intent should be clear: the agent is executed at the new site, with a local copy of the state it had at the previous site. The agent's state is then accessed locally at the new site. Implicitly, we assume that the agent ceases any activity at the old site. The agent can repeat this procedure to move to yet another site.

The **copy** operation is a primitive that produces local copies of (almost) arbitrary Obliq values, including values that span several sites. Sharing and circularities are preserved, even those that span the network. Not all values can be copied, however, because not all values can be transmitted. Protected objects cause exceptions on copying, as do site-specific values such as threads.

This techniques allows autonomous agents to travel between sites, perhaps eventually returning to their original site with results. The original site may go off-line without directly affecting the agent.

The main unpleasantness is that, because of copying, the state consistency between the old site and the new site must be preserved by programming convention (by not using the old state). In the next section we see how to migrate state consistently, for individual objects.

4.6 Object Migration

In this example we use a remote execution engine to migrate an object between two sites. First we define a procedure that, given an object, the name of an engine, and a name server, migrates the object to the engine's site. Migration is achieved in two phases: *(1)* by causing the engine to remotely clone the object, and *(2)* by aliasing the original object to its remote clone (section 2.3).

```
let migrateProc =
 proc(obj, engineName)
  let engine =
   net_importEngine(engineName, Namer);
  let remoteObj =
   engine(proc(arg) clone(obj) end);    (1)
  redirect obj to remoteObj end;        (2)
  remoteObj;
 end;
```

After migration, operations on the original object are redirected to the remote site, and executed there.

It is critical that the two phases of migration be executed atomically, to preserve the integrity of the object state. This can be achieved by serializing the migrating object, and by invoking the `migrateProc` procedure from a method of that object, where it is applied to self:

```
let obj1 =
 { serialized, protected,
   ... (other fields)
   migrate =>
    meth(self, engineName)
     migrateProc(self, engineName);
    end };

let remoteObj1 =
 obj1.migrate("Engine1@Site1")
```

Because of serialization, the object state cannot change during a call to `migrate`. The call returns a network reference to the remote clone, which can be used in place of `obj1` (which, anyway, has been aliased to the clone).

We still need to explain how migration can work for protected objects, since such objects are protected against external cloning and aliasing. Note the `migrateProc(self, ...)` call above, where `self` is bound to `obj1`. It causes the execution of `engine(proc(arg) clone(obj1) end)`. Rather subtly, the cloning of `obj1` here is self-inflicted (section 2.4), even though it happens at a site different from the site of the object. According to the general definition, `clone(obj1)` is self-inflicted because `obj1` is the same as the self of the last active method of the current thread, which is `migrate` (an engine call behaves like a procedure call). The redirection operation is similarly self-inflicted. Therefore, the protected status of `obj1` does not inhibit self-initiated migration.

Migration permanently modifies the original object, redirecting operations to the remote clone. In particular, if `obj1` is asked to migrate again, the remote clone will properly migrate.

We can avoid chains of indirections if the migrating object `obj1` is publicly available through a name server. The `migrate` method can then register the migrated object with the name server under the old name:

```
let obj1 =
 net_export("obj1", Namer,
  { serialized, protected,
    ...
    migrate =>
     meth(self, engineName)
      net_export("obj1", Namer,
       migrateProc(self, engineName));
     end };
```

This way, old clients of `obj1` go through aliasing indirections, but new clients acquiring `obj1` from the name server operate directly on the migrated object.

4.7 Application Servers

Visual Obliq [4] is an interactive distributed-application and user-interface generator, based on Obliq. All distributed applications built in Visual Obliq follow the same model, which we may call the application server model. In this model, a centralized server supplies interested clients, dynamically, with both the client code (as a closure) and the client user interface of a distributed application. The closure transmitted to each client retains lexical bindings to the server site, allowing it to communicate with the server and with other clients. Each client may have independent local state, and may present an independent view of the application to the user. A typical example is a distributed tic-tac-toe game.

5. Conclusions

Obliq addresses a very dynamic form of distributed programming, where objects can redirect their behavior over the network, and where computations can roam between network sites. We feel that this kind of programming is still in its infancy, and that not all the fundamental issues can yet be addressed at once. Where in doubt, we have given precedence to flexible mechanism over robust methodology, hoping that methodology will develop with experience. In this spirit, for example, Obliq could be used to experiment in the design and implementation of agent/place paradigms [34], using the basic techniques of section 4.

Related Work

Obliq's features and application domains overlap with programming languages such as ML [24, 28], Modula-3 [26], and Self [33]; with scripting languages such as Sundew [19], Tcl [27], AppleScript [2], VBA [12, 23], and Telescript [34]; and with distributed languages such as Emerald [21], Orca [3], Forté [17], and Facile [32]. None of these languages, however, has the same mix of features as Obliq, particularly concerning the distribution aspects.

Stamos and Gifford [29] eloquently describe remote execution as a generalization of remote procedure call, and survey previous work on remote execution mechanisms. Their proposal, though, restricts the transmission of higher-order procedures and procedures with free identifiers, inhibiting the techniques of section 4.

Our choice of features was largely determined by the idea of a distributed lexically scoped language, by the desire for a simple object model that would scale up to distributed computation, and by the availability of a sophisticated network-objects implementation technology. The Obliq object primitives were designed in parallel with work on the semantics and type theory of objects [1]; distributed scoping and distributed semantics, however, are not treated there.

Influence of Modula-3 Network Objects

The characteristics of Modula-3 Network Objects (M3NOs) had a major influence on the Obliq language design and implementation. Thanks to the low overhead involved, all Obliq objects are M3NOs, so there is no artificial separation between local objects, and objects that may be remotely accessed. Similarly, all Obliq program variables (declared by **var**) are M3NOs: this is the basis for distributed lexical scoping. Concerns about space reclamation, specially for resources used by remote agents, are relieved by distributed garbage collection of M3NOs. Finally, the M3NOs stub generator handles automatically the transmission of all of Obliq's run-time structures.

Moral: a distributed language like Obliq is easy to implement on top of a library like Modula-3 Network Objects. Conversely, a network object library should make it easy to implement a language like Obliq, or is falling short of some important goals.

Status

Obliq has been available at Digital SRC for about a year and a half. In addition to incidental programming, it has been used extensively as a scripting language for algorithm animation [13] and 3D graphics [25], and as the basis of the Visual Obliq distributed-application builder [4]. The Obliq implementation provides access to many popular Modula-3 libraries [20], and to an extensive user interface toolkit [14] including digital video [18]. Obliq can be used as a stand-alone interactive interpeter. It can also be embedded as a library in Modula-3 applications, allowing them to interact remotely through Obliq scripts [16]. The implementation and documentation are available on the World Wide Web at http://www.research.digital.com/SRC/home.html.

Future Work

Issues of authentication, security, authority delegation, and accounting are being explored.

Acknowledgments

The Network Objects project at Digital SRC provided the infrastructure without which Obliq would not have been conceived. Alan Knaff implemented the metaparser layer of the Obliq parser.

Appendix: Syntax Overview

(See reference [16] for details.)

TOP-LEVEL PHRASES

<table>
<tr><td>a;</td><td>any term or definition ended by ;</td></tr>
</table>

DEFINITIONS (denoted by d; identifiers are denoted by x, terms are denoted by a)

<table>
<tr><td>let x_1=a_1, ..., x_n=a_n</td><td>definition of constant identifiers</td></tr>
<tr><td>let rec x_1=a_1, ..., x_n=a_n</td><td>definition of recursive procedures</td></tr>
<tr><td>var x_1=a_1, ..., x_n=a_n</td><td>definition of updatable identifiers</td></tr>
</table>

SEQUENCES (denoted by s)

<table>
<tr><td>a_1; ...; a_n</td><td>executes each a_i (term or defin.); yields a_n (or ok if n=0)</td></tr>
</table>

TERMS (denoted by a, b, c; identifiers are denoted by x, l; libraries are denoted by m)

<table>
<tr><td>x | m_x</td><td>identifiers</td></tr>
<tr><td>x:=a</td><td>assignment</td></tr>
<tr><td>ok | true | false | 'a' | "abc" | 3 | 1.5</td><td>constants</td></tr>
<tr><td>[a_1, ..., a_n]</td><td>arrays</td></tr>
<tr><td>a[b] | a[b]:=c</td><td>array selection, array update</td></tr>
<tr><td>a[b_1 for b_2] | a[b_1 for b_2]:=c</td><td>subarray selection, subarray update</td></tr>
<tr><td>option l => s end</td><td>term s tagged by l</td></tr>
<tr><td>proc(x_1, ..., x_n) s end</td><td>procedures</td></tr>
<tr><td>a(b_1, ..., b_n)</td><td>procedure invocation</td></tr>
<tr><td>m_x(a_1, ..., a_n)</td><td>invocation of x from library m</td></tr>
<tr><td>a b c</td><td>infix (right-associative) version of b(a,c)</td></tr>
<tr><td>meth(x, x_1, ..., x_n) s end</td><td>method with self x</td></tr>
<tr><td>{l_1=>a_1, ..., l_n=>a_n}</td><td>object with fields named l_1...l_n</td></tr>
<tr><td>{protected, serialized, ...}</td><td>protected and serialized object</td></tr>
<tr><td>{l_1=>alias l_2 of a_2 end, ...}</td><td>object with aliased fields</td></tr>
<tr><td>a.l | a.l(a_1, ..., a_n)</td><td>field selection / method invocation</td></tr>
<tr><td>a.l:=b</td><td>field update / method override</td></tr>
<tr><td>clone(a_1, ..., a_n)</td><td>object cloning</td></tr>
<tr><td>a_1.l_1:=alias l_2 of a_2 end</td><td>field aliasing</td></tr>
<tr><td>redirect a_1 to a_2 end</td><td>object aliasing</td></tr>
</table>

d	definition
if s_1 **then** s_2 **elsif** s_3 **then** s_4... **else** s_n **end**	conditional (**elsif**, **else** optional)
a **andif** b | a **orif** b	conditional conjunction/disjunction
a **is** b | a **isnot** b	identical/not identical predicates
case s **of**	case over the tag l_i of an option value
$l_1(x_1)$**=>**s_1,...,$l_n(x_n)$**=>**s_n **else** s_0 **end**	binding x_i in s_i (**else** optional)
loop s **end**	loop
for i=a **to** b **do** s **end**	iteration through successive integers
foreach i **in** a **do** s **end**	iteration through an array
foreach i **in** a **map** s **end**	yielding an array of the results
exit	exit the innermost loop, for, foreach
exception("exc")	new exception value named exc
raise(a)	raise an exception
try s **except** a_1**=>**s_1,...,a_n**=>**s_n **else** s_0 **end**	exception capture (**else** optional)
try s_1 **finally** s_2 **end**	finalization
condition() | **signal**(a) | **broadcast**(a)	creating and signaling a condition
watch s_1 **until** s_2 **end**	waiting for a signal and a boolean guard
fork(a_1,a_2) | **join**(a)	forking and joining a thread
pause(a)	pausing the current thread
mutex()	creating a mutex
lock s_1 **do** s_2 **end**	locking a mutex in a scope
wait(a_1,a_2)	waiting on a mutex for a condition
(s)	block structure / precedence group

References

[1] Abadi, M. and L. Cardelli, **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Springer-Verlag. 1994.

[2] Apple, **AppleScript Language Guide**. Addison Wesley. 1993.

[3] Bal, H.E., M.F. Kaashoek, and A.S. Tanenbaum, **Orca: a language for parallel programming of distributed systems**. *IEEE Transactions on Software Engineering* **18**(3), 190-205. 1992.

[4] Bharat, K. and M.H. Brown, **Building distributed applications by direct manipulation**. *Proc. UIST'94*. 1994.

[5] Birrell, A.D., **An introduction to programming with threads**. In *Systems Programming with Modula-3, Chapter 4*, G. Nelson, ed. Prentice Hall. 1991.

[6] Birrell, A.D., D. Evers, G. Nelson, S. Owicki, and E. Wobber, **Distributed garbage collection for network objects**. Report 116. Digital Equipment Corporation, Systems Research Center. 1993.

[7] Birrell, A.D., G. Nelson, S. Owicki, and E. Wobber, **Network objects**. *Proc. 14th Symposium on Operating Systems Principles*. 1993.

[8] Birrell, A.D., G. Nelson, S. Owicki, and E. Wobber, **Network objects**. Report 115. Digital Equipment Corporation, Systems Research Center. 1994.

[9] Blaschek, G., **Type-safe OOP with prototypes: the concepts of Omega.** *Structured Programming* **12**(12), 1-9. 1991.

[10] Borning, A.H., **Classes versus prototypes in object-oriented languages**. *Proc. ACM/IEEE Fall Joint Computer Conference*. 1986.

[11] Brewer, E.A. and C.A. Waldspurger, **Preventing recursion deadlock in concurrent object-oriented systems**. *Proc. 1992 International Parallel Processing Symposium, Beverly Hills, California. (Also, Report MIT/LCS/TR-526.)*. 1992.

[12] Brockschmidt, K., **Inside OLE2**. Microsoft Press. 1994.

[13] Brown, M.H., **Report on the 1993 SRC algorithm animation festival**. Report n.126. Digital Equipment Corporation, Systems Research Center. To appear. 1994.

[14] Brown, M.H. and J.R. Meehan, **The FormsVBT Reference Manual**. Unpublished. Digital Equipment Corporation, Systems Research Center. 1994.

[15] Cardelli, L., **The Amber machine**. *Proc. Combinators and Functional Programming Languages*. Lecture Notes in Computer Science 242. Springer-Verlag. 1986.

[16] Cardelli, L., **Obliq: A language with distributed scope**. Report n.122. Digital Equipment Corporation, Systems Research Center. 1994.

[17] Forté, **TOOL reference manual**. Forté, Inc. 1994.

[18] Freeman, S.M.G. and M.S. Manasse, **Adding digital video to an object-oriented user interface toolkit**. *Proc. ECOOP'94*. Springer-Verlag. 1994.

[19] Gosling, J., **Sundew: a distributed and extensible window system**. *Proc. Winter Usenix Technical Conference*. Usenix Association. 1986.

[20] Horning, J., B. Kalsow, P. McJones, and G. Nelson, **Some useful Modula-3 interfaces**. Report 113. Digital Equipment Corporation, Systems Research Center. 1993.

[21] Jul, E., H. Levy, N. Hutchinson, and A. Black, **Fine-grained mobility in the Emerald system**. *ACM Transactions on Computer Systems* **6**(1), 109-133. 1988.

[22] Lieberman, H., **Using prototypical objects to implement shared behavior in object oriented systems**. *Proc. OOPSLA'86*. ACM Press. 1986.

[23] Mansfield, R., **Visual Basic for Applications**. Ventana Press. 1994.

[24] Milner, R., M. Tofte, and R. Harper, **The definition of Standard ML**. MIT Press. 1989.

[25] Najork, M. and M.H. Brown, **A library for visualizing combinatorial structures**. *Proc. IEEE Visualization'94*. 1994.

[26] Nelson, G., ed. **Systems programming with Modula-3**. Prentice Hall. 1991.

[27] Ousterhout, J.K., **Tcl and the Tk toolkit**. Addison-Wesley. 1994.

[28] Reppy, **A higher-order concurrent language**. *Proc. SIGPLAN'91 Conference on Programming Language Design and Implementation*. ACM Press. 1991.

[29] Stamos, J.W. and D.K. Gifford, **Remote Evaluation**. *ACM Transactions on Programming Languages and Systems* **12**(4), 537-565. 1990.

[30] Taivalsaari, A., **Kevo, a prototype-based object-oriented language based on concatenation and module operations**. Report LACIR 92-02. University of Victoria. 1992.

[31] Taivalsaari, A., **A critical view of inheritance and reusability in object-oriented programming**. Jyväskylä Studies in computer science, economics and statistics No.23, A. Salminen ed. University of Jyväskylä. 1993.

[32] Thomsen, B., L. Leth, S. Prasad, T.-M. Kuo, A. Kramer, F. Knabe, and A. Giacalone, **Facile Antigua Release Programming Guide**. ECRC-93-20. European Computer-Industry Research Centre. 1993.

[33] Ungar, D. and R.B. Smith, **Self: the power of simplicity**. *Lisp and Symbolic Computation* **4**(3). 1991.

[34] White, J.E., **Telescript technology: the foundation for the electronic marketplace**. White Paper. General Magic, Inc. 1994.