

Here EXEC OUT (), analogous to the Linda EVAL, spawns one task to compute the single expression in the (), and returns a single field tuple, which is then removed using an IN in a subsequent loop. In more standard BaLinda, the second loop is replaced by an N-task SYNCHRONIZE:

```

...
OUT ('Product, 0, 1.);
{ FOR I = 1 TO N
  DO EXEC { LOCAL X,Y : REAL,
            X = Factor (I);
            IN ('Product? Y);
            OUT ('Product, X * Y)
          }
        }
SYNCHRONIZE (N);
IN ('Product ? Result);
...

```

### 3. Objects from Functions

The BaLinda object system provides the CLASS construct for data encapsulation through a slight change to the FUNCTION construct. Whereas a function returns a result after a function call, a call on a CLASS creates a package of data and functions by returning a pointer to the execution environment, which constitutes a newly instantiated object. It is then possible to reenter the class environment and resume execution therein. Further, the CLASS specification permits some of the internal routines to be visible from outside the package, i.e., public functions or methods. In this way, it specifies entry points into an object such that, after its creation, further calls may be made on the object to execute its enclosed functions and change its state.

Consider a Node object containing an element, two pointers, and some functions:

```

{ CLASS Node < - New : INTEGER;
  LOCAL { Left, Right } : Node;
  PUBLIC
  Element : INTEGER = New,

```

```

{ PROCEDURE AddLeft < – Leaf : Node;
  Left = Leaf }
{ PROCEDURE AddRight < – Leaf : Node;
  Right = Leaf }
{ CASE GetBranch < – Which;
  'L => Left;
  'R => Right };
... }
...
Z = Node (1);
Y = Node (2);
X = Node (3);
X. AddLeft (Y);
X. AddRight (Z);
...

```

Each call on Node allocates a new instance of the class, and sets its Element to New, while Left and Right, declared but not given values, are set to NIL. In the example, the two pointers of X are set to objects Y and Z, to constitute its left and right leaves.

A Node object looks like a three-element list, but standard list operations cannot be used on it. Instead, one has to define public functions (methods) to process object contents, and these may be directly called from the outside, though one can also invoke local functions after one has entered the object via a public function. As the Left and Right pointers are local, they cannot be accessed from the outside. However, the element is public, and can be accessed using the identifier X.Element. Note that CASE defines a function whose whole body is an IF that tests the argument(s) and selects the first true clause for execution.

Because objects are created by a call/return on a CLASS, there is no need for a make-instance function. Initial values are assigned by defining them in the LOCAL/PUBLIC statements, or by passing over appropriate arguments at object creation. CLASS definitions may be nested like functions, and objects can lexically access information from their defining environments. They can also dynamically access information from their calling environments through SPECIAL variables in order to produce context-dependent behaviour.

In BaLinda K, just as function definitions can be nested, so can class definitions, with inner classes inheriting the variables and functions of the outer definition. Further, inner definitions can be made publicly visible by placing them in the PUBLIC part:

```
{ CLASS X < - A;
  LOCAL ...;
  PUBLIC { CLASS Y < - B;
          ... }
...}
```

To call an inner class, the outer class must be entered first to establish the object attributes and methods. Hence, to create an object of class Y we could use

$$C = X.Y(A, B)$$

providing first for entry into X with argument A and establishment of object content for inheritance by Y, then for entry into Y itself. Subsequently, we can call a public function Z defined in Y using the name C.Z, which is just the usual way. Note that an object of class X was established temporarily, but not retained: X was only used as the environment for establishing Y. It is also possible to have

$$\begin{aligned} C &= X(A); \\ D &= C.Y(B); \end{aligned}$$

producing two objects C and D of classes X and Y respectively.

An object directory system has to be maintained in a way to uniquely identify all objects, including objects of the same name and class spawned from different ancestors. Because objects encapsulate a package of self contained information, they can be moved from cluster to cluster to achieve better load balancing. However, objects without their own tuples can potentially migrate more easily, because for faster searching, tuples are stored in hash tables and usually there is just one table per cluster to contain tuples from all objects at the cluster, and all associated tuples must be retrieved when an object is moved.

One object can only engage in tuple exchange with another object by calling a method in it, or alternatively, if the latter object calls a function defined outside it. Hence, tuple exchanges are restricted to sibling modules in the same

object. This “flattens” the task relations, as interacting tasks that exchange tuples must share a scope. It is reasonable to expect that such enforcement of task relations should make program design and debugging more structured, in addition to the more direct effect of simplifying the management of the tuplespace by integrating it with the management of objects.

While the attachment of tuplespaces to objects restrict tuple exchanges to objects that share the same scope, this sharing can either be direct, in the sense that X is within the scope of Y so that Y can call X.method and exchange tuples with the body or another method of X, or indirect, with Y and Z entering X at different times and leave tuples in X’s space for each other, provided X’s space is defined as public. The remote execution of objects in a distributed system automatically leads to the distribution of tuplespaces, and the maintenance of an object directory on each remote node is associated with the tuplespace maintenance: tuples are entered into the node tuplespace with visibility flags identifying their owner objects, and they only respond to tuple access operations with the same flags. This logically partitions the tuplespace into subspaces for the different objects executing at that node. Tuples themselves are never sent from one node to another: a task on one node must call a method in an object on another node to access that object’s tuples, and information in a tuple pass between nodes as call arguments and method results.

Certain requirements need to be met by the object directory system, which lists on each node the objects that are executing on the node and also shows the locations of the objects that have been dispatched elsewhere for execution. The scoping information of each object is also stored to facilitate the determination of the legitimacy of any particular call, i.e., the call X.method from Y is permitted only if X is in the scope of Y, and in case the same name is reused, which particular X is in the scope of Y. Having found the object and called the relevant method, any tuple operations executed by the method are picked up by the recipient node’s tuplespace manager, and the correct visibility flags are attached to identify the object which the operations come from. The process of locating a particular tuple is decoupled into two parts: first calling the object and then searching for the tuple, each using fairly standard techniques. The old problems of information security and search efficiency are under better control, the former because of well defined object scoping rules, and the latter because tuples are distributed and for both tuple producers and

consumers, the correct location for any tuple is clearly defined by the object location.

With the distribution of tuples within objects, security and efficiency can both be expected to improve over the global object space. An object makes its tuplespace accessible to other objects by declaring its tuple functions IN/RD/OUT public; in contrast, if these functions are declared local, then its tuplespace is private. In either case, a tuple search to execute an object's tuple functions, is confined to the object's local space. Since the space is usually small, searching for a tuple should be faster compared with a shared global tuple space. By not inheriting the tuple functions, an object indicates that it does not require tuples, thus simplifying its compilation and instantiation. Each object can access the global tuple space using .IN etc., i.e., by using an empty object name with the tuple functions.

Cluster-based systems aim to support scalable parallel computing with a two level hardware structure of tightly coupled processing units making up a set of loosely coupled clusters [1, 2]. The tightly coupled clusters, such as multiple processors on a single PC bus, run tasks with high communication requirements with each other, such as threads sharing a single environment, but engage in only occasional communication with remote clusters, such as to spawn off functions that return results, but otherwise do not access the main thread environment.

Objects provide an obvious software structure to match this hardware structure, because they encapsulate highly self contained data and code. Methods spawned off for parallel execution in the same object share direct access to the object's encapsulated information, but are less likely to share information with methods spawned from other objects. As a general rule, methods of the same object should execute on the same cluster, and whole objects are dispatched to other clusters for execution. Objects change their state in the course of execution, and receive occasional calls from other objects so that the computed results of the objects' methods may be retrieved. From time to time, object migration to another cluster may also be required, either to better balance the workload of the whole system, or to better meet intensive communication needs with objects which are already there. We expect close links between cluster based systems and concurrent object languages [3-8].

The intersection of parallel processing and object oriented programming generates a number of language design problems that must be addressed. The following sections deal with two basic issues that arise.