

```

LOCAL ...;
  body
}

```

When an instance of A is made, the variables declared in the PUBLIC and LOCAL statements are assigned storage, as are the methods defined there; if the body is present, then the code specified is executed to initialize the object, before a pointer to the object is returned to the caller permitting access to the new A object. Such an object is passive, since no execution will take place inside it unless calls on its method are received.

However, consider the follow scenario:

```

spawn X = makeinstance (A)
other processing
SYNCHRONIZE

```

While “other processing” is occurring, X already points to the storage area allocated to the object, and the public information is already visible, even though the body may still be executing and changing the information. It is possible for calls to be made on X, so that there is parallel execution between the body of X and its called method. X is no longer a passive object; it contains execution of its own, which may interact with calls on it from the outside.

A pointer to an active object, like a future token, is a link to an unfinished piece of computation, but whereas accessing a future token would block a thread until computation completes, an object has no definitive final state and the public part of its content can be accessed even as computation proceeds in its body or methods. However, doing this safely would require some atomic facility, and as discussed in the next section, this is done using tuples.

One could also wait for the object initialization code the end with a SYNCHRONIZE, after which the object is no longer active, and then call its methods like a normal passive object. This merely makes object initialization and execution outside the object concurrent between the class call and the SYNCHRONIZE.

5. Objects and Atomicity

An object encapsulates a relatively self contained set of data and code. In parallel processing problems, it is natural to see an object as a unit of atomicity, which arises from the imposition of sequential entry into the object, i.e., only

one call on any method is executed at any moment. Some form of call queue management is required to produce this effect. While this is not difficult to implement in itself, it is contrary to the idea of treating a method call like a simple procedure call, and has several consequences that need to be dealt with.

First, even to obtain a simple result computed in an object, we have to send a call to it and wait for the reply. In parallel processing an atomic object may not immediately respond to an incoming message because it could be engaged in some execution started by an earlier call, even though the result requested is a very simple one and is immediately available. The call and wait overheads, if incurred for a large number of simple processing steps, could make execution very inefficient.

Second, suppose an object receives a call that causes it to start some processing, e.g., pop stack, but it then discovers that the necessary information is not available, e.g., stack is empty, what does it do? Should it reply “unsuccessful” to the requester, who will have to try again, perhaps repeatedly (the equivalent of spin lock) before the request would succeed? Or, should it store away the request and wait for a new call, in the hope that some other object would do something to rectify the situation, e.g., a push request, before returning to process the unsuccessful request? which is just the monitor solution adopted in Java and other systems, and gives to the usual monitor chaining problem: if a second monitor is called from within a first monitor, and the call suspends, should both monitors be released? If so, how can the reimposition of the lock on both monitors be carried in a safe and efficient fashion?

Another idea on this point is that methods should have conditional guards, such that a call on a method succeeds only if certain conditions are satisfied, e.g., calls on stack pop function are wait queued if the stack is empty. Such decisions materially influence the execution overheads as well as programming techniques. In particular, having guarded methods leads to the so called inheritance anomaly problem discussed below.

Third, in recursive processing, an object may need to call itself. It is therefore necessary to stack the previous call in some way while the object takes the next call. This problem is unrelated to parallel execution, and the technique of causing an “unsuccessful” return for the earlier call upon a new call is not useful here.

It is necessary to point out that schemes that involving open and closed, or atomic and non-atomic, object states, with methods that can be called to

change from one state to the other, hence enabling or disabling concurrent processing, have limited usefulness in general, because this fails to allow for an object accepting some calls but not others, such as an empty stack object accepting push calls but not pop calls.

In [9], three cases of inheritance anomaly were posed.

- (a) *State Partitioning.* Suppose we have a class Stack and a sub-class DStack is defined with a new method PPop that takes the top only if there are two elements in the stack. Previously, we need to distinguish between the Empty and NonEmpty states of stack, permitting Pop calls only if a stack object is in NonEmpty state, but in a DStack object, the NonEmpty state is sub-divided into two: the One state which permits a Pop but not a PPop, and the More state in which both are permitted. A Push now changes One state to More state and Empty state to One state, while Pop changes states in reverse; hence, the introduction of the sub-class causes the redefinition of the methods of the parent class, instead of simply inheriting them, hence an inheritance anomaly. This is because states controlling the behaviour of the child class are partitions of the states of the parent class, whose methods must change to recognize these.
- (b) *History Sensitivity.* In this, the child class EStack has a QPop method that cannot be executed after a Push from the same object. Again, parent class methods must be modified to change an object between the AfterPush and AfterPop states.
- (c) *State Modification.* This concerns the introduction of orthogonal states with a sub-class that may lock an object and hence affect the processing of parent class methods.

These difficulties do not arise if parallel entries into an object are possible, because in the sub-class pre- and post-processing can be carried out with parent class methods to deal with the new situations, whereas with conditional entries, the condition and the parent method form an integrated unit. Again this reinforces the earlier discussed need that, with multiple threads in an object, some atomic structure must then be available in objects to enforce sequentialization, in shared access of wanted results or in suspending to await relevant conditions. We now discuss this issue.

6. Using BaLinda Objects

As explained earlier, the BaLinda Lisp/K parallel languages provided classes and objects within a functional framework [11, 12]. An efficient compiler for