BALINDA K, A PARALLEL LISP DIALECT FOR IMPERATIVE PROGRAMMERS

C K Yuen, M D Feng and K K Fong

Abstract. The parallel language BaLinda Lisp is presented in an alternative form taking on an appearance closer to C. The language combines features of functional and sequential execution models. Examples of different applications are shown. BaLinda K forms the basis for developing a parallel language with data/class type checking and generic functions, while retaining the current list processing and recursion capabilities.

1. Introduction

Lisp is a language that arouses strong feelings. Its adherents consider it vastly superior to other languages (see for example [1]), while others consider it strange and incomprehensible. We ourselves see important advantages in its method of program composition through recursive functions and storage management through garbage collection, but do not expect the majority of computer users to readily take to it. While it is a simple matter to add syntactic sugar to the language and make its appearance closer to that of imperative languages [2][3][4][5], the results have generally met with limited approval.

In this paper we make a fresh attempt that takes into account modern trends, this time "imperitizing" a parallel Lisp dialect BaLinda Lisp to take on a resemblance to the popular language C, but without changing its Lisp essence. The result is BaLinda K (the letter K being a "hard C".)

We start with a brief description of BaLinda Lisp. The BaLinda suite of parallel languages originated from a research project on dataflow architectures. (The term BaLinda stands for BIDDLE And Linda, while BIDDLE stands for BIdirectional Data Driven Lisp Engine.) To identify independently executable parts of a program, BaLinda languages use the EXEC keyword. Thus, in BaLinda Lisp

```
(...
 (EXEC F ...)
 (next expression)
 ...)
```

generates a parallel task to execute the function F, while the main task continues to execute the next part of the program. The parallel task for F ends with the) that ends the F expression, while the main task will synchronize with the parallel task at the next unmatched). For example, in the code (F1 (EXEC F2 ...) (F3 ...)), F2 is executed as a separate task, which ends with F2's own). F1 is entered when the main task, having executed F3, synchronizes with F2 at the final).

To illustrate, the following is a program that multiplies N factors evaluated in separate processes. ...

```
(DEFUN Multiply (Start End)
    (COND ((= Start End) (Factor Start))
        (T (LET ((Middle (/ (+ Start End) 2)))
             (* (EXEC Multiply Start Middle)
               (Multiply (+ Middle 1) End))))))

(Multiply 1 N)
```

In each call on Multiply, the group of factors from Start to End is divided into two parts and sent to two child functions, unless the group size is already down to 1, in which case the lone factor is evaluated and returned. The two Multiply functions execute in parallel because of the EXEC prefix before the first Multiply, and their returned results are combined and returned for further multiplication.

For inter task communication, we use the Linda tuplespace mechanism. The system assumes a shared data space containing multi-field records or tuples. A task puts a tuple into the tuplespace using an OUT command:

```
(OUT exp1 exp2 ... expN)
```

where each expression defines a value of any type, whether numerical, logical, text or even array/list. Tuples are not accessed by name or address, but by content, using the IN or RD commands:

```
(IN exp1 exp2 ... expM ? name1 name2 ... nameN-M)
(RD exp1 exp2 ... expM ? name1 name2 ... nameN-M)
```

These will retrieve from the tuplespace an N-field tuple whose first M fields match the result of the M expressions in the IN/RD, and then store the values of the last N-M fields into the N-M variables specified in the IN/RD. IN causes the tuple to be removed from the space, while RD leaves it for others to access. If no matching tuple is found, the task executing the IN/RD is suspended until another task OUTs the required tuple.

Below is an another parallel multiplication program that spawns factor tasks one by one until the index reaches N, at which point it generates a product tuple. Each task competes for the tuple and multiplies its factor into the product. When a return is made to the main program, all the parallel tasks have ended and the product is ready.

```
...
(DEFUN Multiply (Index)
    (COND ((< Index N)
          (EXEC Multiply (+ Index 1))
          (LET ((X (Factor Index))
               (Y NIL))
              (IN 'Product ? Y)
              (OUT 'Product (* X Y))))
         ((= Index N)
          (OUT 'Product (Factor N)))))

(Multiply 1)
(IN 'Product ? Result)
```

An efficient compiler for BaLinda Lisp, including good load balancing and tuplespace distribution, is now available for a PC/Transputer system and a multiprocessor SUN workstation, and is being ported to other machines. [6][7][8]

2. BaLinda K

We now introduce the new language through a series of examples. Below are the two parallel multiplication programs recast in BaLinda K:

```
{ FUNCTION Multiply <- Start, Finish; { PROCEDURE Multiply <- Index;
  LOCAL X, Y, Middle;                   LOCAL X, Y;
  { IF Start==Finish                  { IF Index==N
      => Factor (Start);                  => OUT ('Product, Factor (N));
   T => { Middle=(Start+Finish)/2;        T => { EXEC Multiply (Index+1);
          EXEC SHARING (X)                      X=Factor (Index);
            X=Multiply (Start, Middle);        IN ('Product ? Y);
          Y=Multiply (Middle+1, Finish);         OUT ('Product, Y*X);
          SYNCHRONIZE;                         SYNCHRONIZE
          X*Y                   } }       };
 };                          Multiply (1);
Result=Multiply (1, N);              IN ('Product ? Result);
```

LOCAL starts a Lisp LET enclosing the whole body of a procedure or function. NIL is assigned to any name declared but not given a value. LOCAL is a single statement ending with ;, but may have multiple declarations separated by ,. IF, like LISP's COND, has a series of clauses each containing a Boolean guard and a set of actions, separated by =>. The guards are tested one by one until T is found, which causes the attached actions to be performed and the rest of IF to be skipped. No ELSE is used, as the effect is achieved by putting T in the last clause. Some pairs of parentheses have been replaced by { }, which are used as BEGIN..END pairs, while others have been made unnecessary through the use of ; to separate statements. An explicit SYNCHRONIZE statement has been introduced, whereas in BaLinda Lisp, tasks implicitly synchronize at the next unmatched). As each SYNCHRONIZE waits for the ending of the most recent task, waiting for multiple parallel tasks is last in first out. SHARING (X) prevents the subtask from creating its own copy of X, ensuring that the task returns its result to the main task.

It should however be kept in mind that BaLinda K is Lisp, not C. Every statement in the two programs corresponds to a BaLinda Lisp equivalent. Only the presentation has changed, and instead of writing a new compiler, a simple pre-processor could be used to change BaLinda K programs to BaLinda Lisp. (In our own implementation the conversion is directly to the internal representation.) Nevertheless, there is some change in programming style, e.g., instead of (* (Multiply ..) (Multiply..)), we now have the equivalent of (SETQ X (Multiply ..)) (SETQ Y (Multiply ..)) (* X Y) because that is more natural in BaLinda K. Also, local variables are more naturally declared at the start of a procedure or function, rather than as needed by additional LETs, even though they may be needed only under certain conditions and not others.

To take another example, below is the reader/writer problem programmed in BaLinda K. There is a control tuple with a number showing the number of active reads and a Boolean flag. Readers are able to start new reads only when the Boolean flag is T, and the Writer starts a write only when the number of reads is 0; if not, the writer changes the flag to NIL to prevent new reads from being started, while it waits for the number to return to 0.

```
{ PROCEDURE Writer;                    { PROCEDURE Reader;
 LOCAL Readers;                      LOCAL Readers, Flag;
 IN ('Control, - ? Readers);          IN ('Control, T ? Readers);
 { IF Readers>0                    OUT ('Control, T, 1+Readers);
    => { OUT ('Control, NIL, Readers); ...reading...
         IN ('Control, NIL, 0) }       IN ('Control, ? Flag, Readers);
  }                              OUT ('Control, Flag, Readers-1);
 ...writing...                    ...
 OUT ('Control, T, 0)                  },
 ...                             ...
},
...
```

To show how BaLinda K processes lists, below is a function that searches for Element in Alist, returning its position, or NIL if absent. $ is the Lisp function CAR, and & is CDR, applied left to right, e.g., &&$ (Alist) takes the third element of Alist, i.e., $(&(&(Alist))), and &&$&$, the second element of the third element, i.e., $(&($(&(&(Alist))))). The reverse operation CONS, e.g., CONS (New, AList), may also be written as {$ New & AList}. Similarly LIST (X, Y, Z) may also be written as {$ X & {$ Y & {$ Z & NIL}}} or CONS (X, CONS (Y, CONS (Z, NIL))).

```
...
{ FUNCTION Search <- Element, Alist, Number;
 { IF NULL (Alist)
      => NIL;
    $(Alist) == Element
      => Number;
    T => Search (Element, &(Alist), Number+1)
} }
...
Where = Search (What, ListXXX, 0);
...
```

Another example is a function that inserts New in front of Element if it is present, or at the end of the list if not:

```
...
{ FUNCTION Insert <- Element, Alist, New;
 { IF NULL (Alist)
      => LIST (New);
    $(Alist) == Element
      => {$ Element & Alist};
```

```
      T => {$ $(Alist) & Insert (Element, &(Alist), New)}
 } }
...
Where = Search (What, ListXXX, 0);
...
```

Now we show a Quicksort program, with two loops, one searching from the top for elements larger than Pivot and one from the bottom for smaller elements. Note that array AnArray has to be declared globally, and Index and Index2 outside the two Loop functions so that they can be shared.

```
...
{ PROCEDURE Quicksort <- Start, Finish;
  LOCAL Swap,
        Middle = (Start+Finish)/2,
        Pivot  = AnArray[Middle],
        Index  = Start,
        Index2 = Finish,
        { PROCEDURE Loop;
          { IF AnArray[Index] < Pivot
                 => { Index = Index+1;
                      Loop
                      }
               T => Loop2
          } },
        { PROCEDURE Loop2 ;
          { IF AnArray[Index2] > Pivot
                 => { Index2 = Index2-1;
                      Loop2
                      }
               Index<Index2
                 => { Swap = AnArray[Index2];
                      AnArray[Index2] = AnArray[Index];
                      AnArray[Index] = Swap;
                      Loop
          } }         };
  Loop;
  { IF Start < Index-1 => Quicksort (Start, Index-1) }
  { IF Index+1 < Finish => Quicksort (Index+1, Finish) }
 }
```

In both of the above programs, recursion is used to implement iteration in accordance with functional practice. However, the concept of “in place” processing of arrays exists, in contrast to the pure functional and list processing concept of constructing new data from old, disposing the latter by garbage collection.

3. Loops

Should there be a looping construct? Consider a simple loop to sum an array, which may be programmed by defining a local block with a looping function:

```
Y = { LOCAL
      { FUNCTION Loop <- Sum, I;
        { IF I>N => Sum;
             T   => Loop (Sum+X[I]), I+1)
       } };
      Loop (0., 1)
     }
```

Several matters make the program unnatural to imperative programmers. First, the exit condition is tested at the start of the loop rather than the end, because otherwise X[I] might go beyond the array bound. Second, the arithmetic is not performed in the loop body but in the recursive call, by passing a new argument Sum to the next iteration. Third, the loop index increment (+ I 1) is also embedded in the recursive call. Finally, the loop initialization, I=1 and Sum=0, are embedded in the initial loop call, which comes after the loop definition. While these might seem to be very minor problems to Lisp programmers, it is by no means simple to convert others to this way of thinking. Writing complex nested loops this way is much harder still.

Now consider the alternative of using a Pascal REPEAT loop:

```
Sum:= 0;
I:= 1;
REPEAT Sum:= Sum+X[I];
       I:= I+1
UNTIL I>N;
```

Loop initialization has been moved to the front, and the summing action is separately shown, but the programmer still has to explicitly put in the exit test and the index increment. Programming this way does not seem to be much simpler than recursion. In contrast the FOR loop solution

```
Sum = 0.
FOR I = 1 TO N DO
   Sum = Sum+X[I];
```

provides a more significant improvement, by simplifying the looping control. Hence, if one wishes to provide loops to help the programmer, then a FOR loop seems more helpful then others.

Having a loop construct in a recursive language is not just a trivial matter of adding some preprocessing, because there are certain special requirements for loops in functional programming. Consider the two Pascal loops: Neither the loop as a whole nor the individual iterations return any result, and information carrying from one iteration to the next is achieved through side effect, by changing Sum. Though these practices are natural in imperative languages, they are contrary to functional principles.

To conform to these, the BaLinda K FOR loop uses the CONTINUE statement to call the next iteration, with the possibility of passing over arguments so that the results of one iteration may pass to the next. The FOR part starts the first iteration, establishing initial values of the arguments as well as specifying iterative changes and termination conditions.

To take an example, summing an array is done as:

```
{ FOR Sum = 0.,
     I = 1 TO N RETURN Sum
  DO CONTINUE (Sum+X[I], -)
 }
```

Sum and I are both initialized in FOR..DO, but a new value of Sum is passed in CONTINUE, whereas the change for I (increment by 1) is already specified in the FOR part and so is not required in CONTINUE. Its place is taken by the “don’t care” symbol - (also used in tuple IN/RD statements to avoid retrieving unwanted fields). In both the Lisp and K programs, Sum and I are internal to the loop and their values are not accessible from the outside. Hence, in BaLinda K an explicit RETURN construct is needed to obtain the results of loops.

The next example is a slightly more complex loop which multiplies elements of list X into those of array Y, summing the products to get the inner product of X and Y. It also provides for a FOR loop terminating prematurely, i.e., if the array or list has a zero then an error is found requiring immediate loop exit. This is handled by using RETURN inside the loop. Here Z “zips” through the elements of X (a method also used in some functional languages), while I goes from 1 to N. Reaching either the end of X or Y causes the return of Sum. We see that the ZIP feature allows the FOR loop to be used like a WHILE:

```
{ FOR Sum = 0.,
     Z = ZIP (X) RETURN Sum,
     I = 1 TO N RETURN Sum
  DO { IF Z==0.    => RETURN NIL;
```

```
      Y[I]==0. => RETURN NIL;
      T        => CONTINUE (Sum+Z*Y[I], -, -)
}     }
```

which is equivalent to

```
(DEFUN Loop (Z I Sum)
    (COND ((NULL Z) Sum)
        ((> I N) Sum)
        ((= (CAR Z) 0.) NIL)
        ((= Y[I] 0.) NIL)
        (T (Loop (CDR Z) (+ I 1)
              (+ Sum (* Y[I] (CAR Z)))))))

(Loop X 1 0.)
```

This might also be compared with the Common Lisp version:

```
(CATCH Zero
     (LOOP (INITIALLY (I 1)
                 (Z X)
                 (Sum 0.))
         (WHILE (AND (<= I N) (NOT (NULL Z))))
         (DO (IF (= (CAR Z) 0.)
               (THROW Zero NIL)
               (IF (= Y[I] 0.)
                  (THROW Zero NIL)))
            (SETQ Sum (+ Sum (* (CAR Z) Y[I])))
            (SETQ I (+ I 1))
            (SETQ Z (CDR Z)))
         (FINALLY Sum)))
```

which is rather like Pascal except for the provision of premature exit by CATCH-THROW. The Scheme DO loop is a little simpler in structure than the Common Lisp loop but the difference is minor.

To give a larger example with both looping and recursion, below is a second Quicksort program:

```
{ PROCEDURE Partition <- Start, Finish, Continue;
 LOCAL Pivot = { IF NULL(Continue) => X[(Start+Finish)/2];
                 T            => Continue
              },
      Swap,
      Large = { FOR I = Start TO Finish
              DO { IF X[I]>Pivot
                   => RETURN I
              }    },
      Small = { FOR I = Finish DOWNTO Large
              DO { IF X[I]<Pivot
                   => RETURN I
              }    };
 { IF Large<Small
      => { Swap = X[Large];
           X[Large] = X[Small];
           X[Small] = Swap;
           Partition (Large, Small, Pivot)
          }
    T => { { IF Start<Large-1
            => Partition (Start, Large-1, NIL)
           }
          { IF Small+1<Finish
```

```
          => Partition (Small+1, Finish, NIL)
 } }       } }
```

```
Partition (1, N, NIL)
```

Another example is a breadth-first 8-queens program:

```
{ FOR Init = '((1)(2)(3)(4)(5)(6)(7)(8)),
    Length = 2 TO 8 RETURN Init
  DO CONTINUE
    ({ FOR Soln = NIL,
         Row = ZIP (Init) RETURN Soln
       DO CONTINUE
         ({ FOR Result = Soln,
              New = 1 TO 8 RETURN Result
            DO { IF { FOR Old = ZIP (Row) RETURN T,
                        Column = Length-1 DOWNTO 1
                      DO { IF Old==New OR Length-Column==ABS(New-Old)
                             => RETURN NIL
                    }    }
                     => CONTINUE (CONS (CONS (New Row), Result), -);
                   T => CONTINUE (Result, -)
           }    },
          -)
      },
     -)
 };
```

A full explanation of the algorithm may be found in [9]. Note that the innermost FOR loop does not require a CONTINUE statement as no arguments need to be passed from iteration to iteration.

The program may be compared with the Lisp version:

```
(DEFUN NoCheck (I J Row)
    (COND ((= I 0) T)
        ((= J L) NIL)
        ((= (- K I) (ABS (- L J))) NIL)
        (T (NoCheck (- I 1) (CAR Row) (CDR Row)))))
(DEFUN Extend (Soln Row L)
    (COND ((> L 8)
         (COND ((NULL Soln) NIL)
             (T (Extend (CDR Soln) (CAR Soln) 1))))
        ((NoCheck (- K 1) (CAR Row) (CDR Row))
         (CONS (CONS L Row) (Extend Soln Row (+ 1 L)))))
        (T (Extend Soln Row (+ 1 L)))))

(DEFUN Init (Soln K)
    (COND ((> K 8) Soln)
        (T (Init (Extend (CDR Soln) (CAR Soln) K 1) (+
1 K)))))

(Init '((1)(2)(3)(4)(5)(6)(7)(8)) 2)
```

and the Miranda version[10]:

```
NoCheck Row L = ALL [(J<>L)AND((K-I)<>ABS(L-J)) | (I,J)<-
ZIP([1..#Row], Row)]
           WHERE K = #Row+1
```

```
Queens 1      = [[1][2][3][4][5][6][7][8]]

Queens (K+1)  = [Row++[L] | Row <- Queens K; L <- [1..8];
NoCheck Row L]
```

Note that the Lisp Extend function implements two loops through its two recursive calls, such that, though only three functions are defined, four loops are present.

4. Objects

Objects are defined using classes. A class is called and executed like a function, except that instead of returning a result but disposing the execution environment, a class call returns a pointer to the execution environment, which may contain public functions. Calling a public function in it causes an object entry, and its execution changes the object's state that persists even after the function returns.

Below is a stack object definition. A call on Stack creates an object with entry points for the public functions Push and Pop:

```
...
{ CLASS Stack;
  PUBLIC { PROCEDURE Push <- New;
          LOCAL Pointer;
          IN ('Stack, - ? Pointer);
          OUT ('Stack, T, CONS (New, Pointer))
         },
        { FUNCTION Pop;
          LOCAL Pointer;
          IN ('Stack, T ? Pointer);
          OUT ('Stack, NOT NULL (& (Pointer)), &
(Pointer));
          $ (Pointer)
         };
  OUT ('Stack, F, NIL)
 };
X = Stack;
...
X.Push (something);
...
Y = X.Pop;
...
```

By having the stack pointer in a tuple, parallel tasks are forced to access the stack one at a time. 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 are 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.

We now consider subclasses. 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

```
C = X (A);
D = C.Y (B);
```

producing two objects C and D of classes X and Y respectively. An example will be shown in Section 6 in conjunction with type declarations.

Defining subclasses separately from parent classes is also done by using a dotted name, e.g.,

```
{ CLASS X.Y <- B;
  ...}
```

defines subclass Y of a previously defined class X. Its effect is as if the definition of Y had been inserted as the last element of the PUBLIC part of X. The main compiling requirement, to have access to the parent class template when the separate child class definition is encountered, does not appear to be difficult to achieve. The definition of Y must be located in a block where X is visible. Note that Y lexically inherits from X and X’s enclosing blocks, rather than from its own local block, but dynamic scoping and argument passing may be used to achieve some form of local inheritance by capturing information in the caller environment. In any case, the main advantage of such a subclass definition is to make it visible where it is required. In contrast, putting Y in the public part of X would make it visible wherever X is visible, which may not be appropriate.

For example, suppose we already have the class Vehicle, with subclasses Car, Bus, Van, Motorcycle, etc., and sub-subclasses Sedan, StationWagon, Convertible, etc. Now consider taxis, which are cars painted in a particular colour with two-way radios and trip metres and methods like sending it to some address to pick up a customer. However, though both sedans and taxis are subsets of cars, there is a clear difference between them: A car being a taxi is situational - it was bought by a taxi company, which paints it and puts in a radio/metre, turning it into a taxi (with the possibility of selling it and turning it back into a normal car). The subclass Taxi is of no general interest to callers of the Car class, and should be defined locally, but the Car subclass should be defined within Vehicle, and Sedan within Car, because of their generic applicability. The problem of a car that is both a sedan and a taxi relates to context dependence: A car, regardless of body type, should be able to acquire a set of attributes and methods applicable to taxis after it is injected into the taxi company environment. The more common method of multiple inheritance from orthogonal subclasses is not supported in BaLinda K, and types and class memberships are strictly hierarchical. In general, something akin to multiple inheritance can be achieved through context dependence, but this is not further discussed here.

5. Speculative processing

In speculative processing, a Boolean guard is executed in parallel with the THEN part, the ELSE part, or both:

```
{ IF (boolean)          { IF (boolean)          { IF (boolean)
   EXEC => then part;           => then part;      EXEC   => then part;
   ...             EXEC T => else part        EXEC T => else part
 }                 }                  }
```

Speculative tasks are dispatched at lower priority than the parallel Boolean task. When the Boolean value is returned, the execution of one part is confirmed with a raised priority, while the other speculative task (if one exists)is purged. No explicit SYNCHRONIZE statement is required as this automatically occurs at the end of the Boolean guard’s execution.

The method may be used to compute parts of an infinite data structure before the values are required. In the program below, Spec (I) is executed in the Boolean part and Spec (I+1) at lower priority in the THEN part; when User requests item I, Spec (I) returns T so that computation of item I+1 is confirmed.

```
...
{ FUNCTION Compute <- I;
  ...
 },
{ FUNCTION Spec <- I;
  { IF { OUT ('Result, I, Compute (I));
         IN ('Request, I);
         T
        }
    EXEC => Spec (I+1);
 }  },
{ PROCEDURE User <- I;
  LOCAL Element;
  OUT ('Request, I);
  IN ('Result, I ? Element);
  ... use the Ith element ...
  User (I+1);
 };
...
EXEC Spec (0);
User (0);
...
```

Solving equations using Newton-Ralphson method is another example to which speculative processing can be applied: having found one approximation X, we compute f(X) and f’(X) in parallel. If f(X) is smaller than the pre-set limit, then the evaluation of f’ and the next X is abandoned:

```
LOCAL
...,
{ FUNCTION f
  ...
 },
{ FUNCTION f'
  ...
 },
...,
{ FUNCTION Newton <- X;
  LOCAL Y, Z;
  { IF { Y = f (X);
         { IF ABS (Y) > Limit
              => { OUT ('YReady);
                   T }
            T => F
         } }
       EXEC => { Z = f'(X);
                 IN ('YReady);
                 Newton (X-Y/Z)
                }
       T    => X
 } };

Limit = ...;
Init = ...;
Root = Newton (Init);
...
```

Here X is the root, and Y and Z the function and derivative values respectively. To continue iterating, the absolute value of Y must exceed some pre-set limit. The Boolean part of this IF, evaluates f, stores it in Y and then returns a Boolean result T or F depending on whether Y is larger than the limit, is executed in parallel with the THEN part, which evaluates f' and stores it in Z, and waits for Y to be ready before proceeding into the next iteration. If the Boolean part returns F, no further iteration is performed. A similar structure is applicable to adaptive integration, branch and bound algorithms and multivariate optimization using descent [11], but no details would be shown here.

6. Object and data typing:

While the main objective of recasting BaLinda Lisp into BaLinda K is an educational one, we see other benefits. The new structure makes it easier to declare all local variables at the start of a procedure or function, instead of the Lisp practice of entering new LET blocks as required. It then becomes feasible to impose the requirement that all LOCAL variables must be declared and typed at the start of each block, and to have compilers verify the correct typing of expressions and assignments. However, BaLinda K does not enforce this strictly, and would attempt to derive the types of undeclared data from the program using standard type inference methods, and may also apply run time type checking where no other method could be applied (see below).

A number of issues need to be resolved. First is how to type check lists and list elements. We can require all lists to be declared. Some may contain only one type of elements, e.g.,

```
LOCAL X : LIST OF INTEGER,
...;
```

Then $ (X), &$ (X), etc., must be of type integer, while & (X), && (X), etc., must share the type of X. To construct a tree, we first have to construct a node type, which contains pointer to its own type:

```
TYPE Node : (INTEGER, Node, Node),
    ...;
LOCAL X, Y, Z: Node,
     ...;
Z = LIST (1, NIL, NIL);
Y = LIST (2, NIL, NIL);
X = LIST (3, Y, Z);
...
```

However, under such a system, lists cannot have mixed elements except for specifically declared types like Node, and list pointers of different types must be carefully differentiated. The whole style Lisp programming would need to be changed. The result is rather like Pascal records, but with the advantage that no explicit storage allocation and release operations are required, and the disadvantage that fields cannot be accessed by name. The latter problem is usually solved by going into a LOCAL block to extract out individual fields:

```
{ LOCAL Element : INTEGER = $(X),
      Left   : Node    = &$(X),
      Right  : Node    = &&$(X),
      ...
 }
```

Alternatively, Node may be defined as an object class without internal code:

```
{ CLASS Node <- New : INTEGER;
  PUBLIC Element : INTEGER = New,
       Left   : Node,
       Right  : Node
 }
...
Z = Node (1);
```

```
Y = Node (2);
X = Node (3);
X.Left = Y;
X.Right = Z;
...
```

Each call allocates a new object of class Node, and sets its Element to New while Left and Right are set to NIL, to be reset later if necessary. However, the $, & and CONS operations can not be used on Node. An object without internal code like Node behaves like Common Lisp structures.

A more vexing question is typing tuples. An IN/RD command can cause assignments on local variables, but the types of the acquired values cannot be inferred from the tuple search keys. Dynamic type checking has to be used, e.g.,

```
IN (search keys ? X);
```

should cause an exception if the matching tuple does not assign a value to X that corresponds to its type. Applications that require lists of mixed or unknown type elements may involve the same treatment. The compiler simply inserts runtime checking of any item of unknown type to verify the correctness of its assignment to or combination with another item of known type, and performs any type conversions required, as is routinely done in Lisp. A function must have a type, which can however be unknown, ?, and must end its execution with a returned result, but a procedure does not.

As with the LET construct, each name declared is visible to the subsequent declarations, and may be assigned a value by = expressions. For multiple variables of the same type, {..names..} : type = {..expressions..} declares them to be of the type and assigns them individual values. Any missing values are automatically set to NIL, e.g., for

```
LOCAL
...
{name1, name2, name3, name4} : REAL = {expr1,,expr3},
...
```

four real variables are allocated, and names 2 and 4 are set to NIL. Note that the previous program using Node may also be written as

```
{ CLASS Node <- New : INTEGER;
  PUBLIC Element : INTEGER = New,
        {Left, Right} : Node
 }
...
{Z, Y, X} = Node {(1), (2), (3)};
X.{Left, Right} = {Y, Z};
...
```

The following example shows the evaluation of the polynomial (1+X)**N as the inner product of two arrays, one containing powers of X and the other the binomial coefficients. (Note this is not an efficient way to evaluate the polynomial, and the example is meant to illustrate the programming techniques only.) Each array is defined by a call on the CLASS Sequence, which has N+1 publicly visible values. Powers and Coeffs are generated by calling on different subclasses P and C to place different values into the N+1 cells, and the main program then computes the inner product. We have deliberately made P values real and C values integer; thus, though both are subclasses of Sequence inheriting N and Values defined in the parent class, a call on Sequence establishes N but does not cause allocation of space for Values, which is deferred until P or C is compiled: the INHERIT declaration causes the unprocessed ARRAY declaration from the parent definition to be obtained, resulting in the insertion of code that, upon a call on Sequence.P or Sequence.C, will obtain the value of N from the already-created Sequence object, so that array Values may be established in the object, and then have actual values inserted by Loop, thus creating a P or C object. Note that the two object creations execute in parallel.

```
{ PROGRAM;
  LOCAL
```

```
 { CLASS Sequence <- N;
   PUBLIC
   Values : ARRAY[1..N] OF ?,
   { CLASS P <- X : REAL;
     INHERIT Values REAL;
     LOCAL { PROCEDURE Loop <- I : INTEGER, Value : REAL;
             Values[I] = Value;
             { IF I<N => Loop (I+1, Value*X) }
            };
     Loop (1, 1.0)
    },
   { CLASS C;
     INHERIT Values INTEGER;
     LOCAL { PROCEDURE Loop <- {I, Value} : INTEGER;
             Values[I] = Value;
             { IF I<N => Loop (I+1, Value*(N-I+1)/I)) }
            };
     Loop (1, 1.0)
    };
  },
 {Result, X} : REAL,
 Max : INTEGER,
 {Powers, Coeffs} : Sequence.{P, C},
 { FUNCTION Inner : REAL <- Max : INTEGER,
                      {C, P} : Sequence.{C, P};
   LOCAL Value : REAL = 0.0;
         { PROCEDURE Loop <- I : INTEGER;
           Value = C.Values[I]*P.Values[I]+Value;
           { IF I>0 => Inner (I-1) }
          };
   Loop (Max)
  };
 Max = ...;
 EXEC Coeffs = Sequence.C (Max);
 X = ...;
 Powers = Sequence.P (Max, X);
 SYNCHRONIZE
 Result = Inner (Max, Coeffs, Powers);
 ...
}
```

It is assumed that, when the Inner function is compiled, the mixed type arithmetic multiplying an integer with a real and adding to a real to produce a real would cause the required type conversion to be inserted by the compiler. With such case-by-case determined arithmetic modes and the variable length array declarations, BaLinda K carries on the dynamic data tradition of Lisp but without suffering a performance penalty.

Still, with data and class types clearly defined, more programming errors can be detected. Consider the example of the list search program, now rewritten with partial typing:

```
...
{ FUNCTION Search : INTEGER
   <- Element : ATOM OF ?,
      Alist : LIST OF ATOM OF ?,
      Number : INTEGER;
  { IF NULL (Alist)
        => NIL;
       $ (Alist) == Element
         => Number;
       T => Search (Element, & (Alist), Number+1)
 } };
...
```

```
Where = Search (What, ListXXX, 0);
...
```

Though the type of Element and Alist is unknown, the compiler is at least able to check that What is an atom, ListXXX is a list made up of a single atomic type, NULL is applicable to Alist, $(Alist) == Element compares atom with atom, returning NIL or Number matches the type of Search, etc., as well as to insert the type testing for $(Alist) and Element before the comparison.

7. Generic functions:

With typing, it also becomes possible to introduce the CLOS system of generic functions, made up of methods selected for execution by the types of arguments:

```
{ GENERIC name <- ..argument names..;
  { METHOD name : type <- ..argument types..;
    ...
   }
  { METHOD name : type <- ..argument types..;
    ...
   }
  ...
 }
```

Note the generic function declaration only shows the argument names, and the method declaration only the types, since all the methods within a generic function must share the same number of arguments and names, but accept differing types. If a type appears in GENERIC rather than in each METHOD, then all the methods return a result of the same type. As in CLOS, T is the superset of all classes, and NIL is a member of every class. A T in the argument type list of a method means the argument may be of any type. As Winston [12] remarks, generic functions are more related to data driven execution than object oriented programming, and BaLinda K provides for the latter separately, with class membership as a form of user defined typing, and a generic function may have no object arguments at all.

Three constructs may be used to invoke a generic function: SELECT, CALL PARENT TO CHILD (or CALLv), and CALL CHILD TO PARENT (CALL^). Each is followed by a generic function name with arguments:

SELECT name (..arguments..) etc. SELECT causes the execution of one method in the generic function whose argument types match the types of the call arguments; if there are more than one matching methods, one method is chosen by taking each argument from left to right and matching to the method argument with the most specific type, e.g., given

```
{ GENERIC name : type <- X, Y;
  { METHOD name <- INTEGER, NUMBER;
    ...
   }
  { METHOD name <- NUMBER, INTEGER;
    ...
   }
  ...
 }
```

then SELECT name (integer,integer), whose argument types matches both methods, causes the execution of method 1, because its first argument type is more specific.

CALLv causes the execution of all the matching methods, the order being determined by taking each argument from left to right and matching to the method argument with the more general type first. For the above function, CALLv causes the execution of method 2 first, then method. CALL^ also executes all matching methods, but selecting by taking each argument left to right and matching to the method argument with the more specific type first, and so, would execute method 1 before method 2 in the example. Note however that in general CALL^ does not simply reverse the order of execution from CALLv, since for both calls the arguments are taken left to right, but reversing occurs if there is only one free argument. If a generic function is called without any prefix, the default is SELECT.

CALL^ corresponds to the CLOS execution order for before-methods, and CALLv for the after-methods, while SELECT selects the primary method. Whereas CLOS packages the three calling procedures into a single generic function, we have kept them separate so that the same before-method or after-method can be combined with different primary methods without re-writing any generic function, and in any case, most generic function applications only have the primary method. It is also possible to include the primary method within CALL^ or CALLv. We do not provide the CLOS around-method. A CLOS generic function returns the result of the primary method, rather the after-method even though it is executed later. In BaLinda K, such a situation requires one to save the result of SELECT in a LOCAL variable and return it after the CALLv execution.

It is possible to define methods at various points of a block, rather than collecting them together within a { GENERIC ... }, which only serves as the common declaration of the argument names. However, in general it is good programming practice to bring them together.

Any type may be sub-divided into subtypes by imposing the relevant predicates. For example, the class NUMBER may be divided into subclasses { NUMBER : >=0 } and { NUMBER : <0 }. Using this idea, the generic function construct may be used to achieve conditional selection, recursion and looping, and we write the following generic function to take the absolute value of a real number, a vector or a list of real numbers: (Note that $ is the Lisp CAR, and & is CDR.)

```
{ GENERIC Absolute <- input | vector, length ;
  { METHOD Absolute <- { REAL : >=0 };
    input}
  { METHOD Absolute <- { REAL : <0 };
    -input}
  { METHOD Absolute <- { LIST : NOT NULL };
    CONS (Absolute ($ (Input)), Absolute (& (Input)))
   }
  { METHOD Absolute <- { LIST : NULL }
                  | ARRAY, { INTEGER : <=0 };
   }
  { METHOD Absolute <- ARRAY, { INTEGER : >0 };
    length = length-1;
    vector[length] = Absolute (vector[length]);
    Absolute (vector, length)
 } }
```

We see that the generic function Absolute may take one or two arguments, the latter being a vector with its dimension. The class of one dimensional arrays is divided into null arrays of length 0 and non-null ones, and the class of lists into empty and non-empty lists. For a non-null array/list, one absolute value operation is performed and the Absolute function recursively called, until the null array/list results. The above program style is rather like logic programming, though it should not be alien to Lisp programmers.

Now we show a more complex example, Quicksort on a list. The function takes an unsorted list with an initially empty list Sorted. After partitioning the unsorted list into sublists for “less than pivot” and “more than pivot”, it combines the right sublist with Sorted to produce a longer sorted list, adds the pivot to that, and then combines the left sublist into the result. The recursion continues until all sublists only have one element, the pivot. Argument names may be declared by pattern matching, reducing the definition of local variables. The generic function structure produces somewhat tidier function definitions, by putting individual clauses of an IF into separate methods.

```
{ GENERIC Sort <- { {$ Pivot & Tail} | NIL }, Sorted;
  { METHOD Sort <- NULL, LIST OF NUMBER;
    Sorted
   }
  { METHOD Sort <- { NUMBER, LIST OF NUMBER }, LIST OF
NUMBER;
    LOCAL X : LIST = Partition (Pivot, Tail);
    Sort ($(X), {$ Pivot & Sort (&(X), Sorted)})
 } }
```

```
{ GENERIC Partition <- Pivot, { {$ Element & Tail} | NIL };
```

```
 { METHOD Partition <- NUMBER, NULL;
  {$ NIL & NIL}
  }
 { METHOD Partition <- NUMBER, { NUMBER, LIST OF NUMBER };
  LOCAL X : LIST = Partition (Pivot, Tail);
  { IF Element<Pivot => {$ {$ Element & $(X)} & &(X)};
      T           => {$ $(X) & {$ Element & &(X)}}
} } }
```

This may be compared to the more standard version:

```
{ FUNCTION Sort <- { Unsorted, Sorted } : LIST OF NUMBERS;
 { IF NULL (Unsorted)
       => Sorted;
     T => { LOCAL Pivot : NUMBER = $(Unsorted),
               X : LIST = Partition (Pivot, &(Unsorted));
          Sort ($(X), {$ Pivot & Sort (&(X), Sorted)})
} }      }

{ FUNCTION Partition <- Pivot : NUMBER,
                  Remainder : LIST OF NUMBERS;
 { IF NULL (Remainder)
      => {$ NIL & NIL};
     T => { LOCAL Element : NUMBER = $(Remainder),
               X : LIST = Partition (Pivot, &(Remainder));
          { IF Element<Pivot => {$ {$ Element & $(X)} & &(X)};
              T           => {$ $(X) & {$ Element & &(X)}}
} }      } }
```

A shorter way to define generic function is by using a named CASE, which is a modified form of IF in which Boolean expressions have been replaced by predicates applied to a selector, e.g.

```
 { IF X>0 => X;
     X=0 => 0;
     X=0 => -X
  }
```

is changed to:

```
{ CASE X
   OF >0 => X;
      =0 => 0;
     <0 => -X
 }
```

A named case is a function that contains just a CASE. For example, the absolute value function may be re-written as

```
{ CASE : { Absolute }
     X | { Seq, Length }
   OF { NUMBER, >0 }
     => X;
     { NUMBER, <0 }
     => -X;
     ARRAY, >0
     => { Length = Length-1;
          Seq[Length] = Absolute (Seq[Length]);
          Absolute (Seq, Length)
         }
     ARRAY, =0 | { LIST, NULL }
```

```
      => NIL;
      { LIST, NOT NULL }
      => CONS (Absolute ($(X)), Absolute (&(X)))
 }
```

Methods of a generic function and functions of a class may be defined externally and exist on separate nodes of a multiprocessor system, including a heterogeneous system with nodes that specialize in certain functions. A generic function or an object may then be used as the vehicle to bring together the results of external executions. To show a simple example of containing parallelism, objects, external methods and generic functions, consider a sequence of lines being read and written on separate I/O processors. Two buffers are used. The program starts with one read, and after transferring the line read in to the out buffer, continues with a read and a write in parallel, until a NIL line is read in signifying end of file:

```
{ LOCAL
  { CLASS Transfer <- InDev, OutDev;
    EXTERNAL PROCEDURE Read (InDev,), Write (OutDev,);
    LOCAL
    Buffer1 : STRING;
    { GENERIC Get <- Finish;
      { METHOD Get <- { BOOLEAN : ==TRUE };
       }
      { METHOD Get <- { BOOLEAN : ==FALSE };
        Read (, Buffer1);
        OUT ("New", Buffer1);
        IN ("Next");
        Get (NULL (Buffer1))
     } }
    { GENERIC Put <- Buffer2;
      { METHOD Put <- { STRING : NULL };
       }
      { METHOD Put <- { STRING : NOT NULL };
        Write (, Buffer2);
        OUT ("Next");
        IN ("New" ? Buffer2);
        Put (Buffer2)
     } };
    Read (, Buffer1);
    EXEC Get (NULL (Buffer1));
    EXEC Put (Buffer1);
   }
  ...;
  X = Transfer (DevA, DevB);
  Y = Transfer (DevC, DevD);
  ...
  SYNCHRONIZE (2*N)
 }
```

Each call on Transfer creates an object containing a reading task and a writing task, with the former passing each line to the latter in a tuple until a NIL line is read in. Note that the EXTERNAL declaration pre-sets the read/write device selection parameter since drivers for different devices may reside on different processors and the device number is needed to link with the correct node at the start, rather than for each individual read/write. SYNCHRONIZE (2*N) waits for 2N tasks to end. The above objects have no public parts, but it might be useful to add some I/O status information visible from the outside. As each object has a private tuplespace, the two tasks need not establish another private space.

8. Discussion:

By eliminating the large number of parentheses and using the infix notation for arithmetic expressions, BaLinda K presents a more approachable appearance to the average programmer. The introduction of a type system and generic functions should make programming easier and more reliable. We thus see BaLinda K as the starting point of turning BaLinda Lisp, currently a succinct and somewhat idiosyncratic "hacker language", into a more supportive user language. Further, one of the reasons for slow Lisp execution is the dynamic data typing, such that arithmetic operations cannot be compiled to a fixed type, but must test for datatypes at run time. With definite typing, BaLinda K should suffer less from such performance problems, making it more realistic to aim for C-like speed in Lisp execution. Thus, while the slower BaLinda Lisp could be used for prototyping, BaLinda K has the prospect of success as a production language.

However, let us reiterate that K is not C. To use BaLinda K effectively, one need to adapt to the use of recursive functions and the idea of storage management by cell allocation and garbage collection. While the absence of the effort of requesting and releasing space should be simplifying, the programmer need to develop an awareness of list traversal and cell structures in order to understand what a program does. We believe that for most programmers this is both a feasible and worthwhile investment, in view of the value of Lisp in programming algorithms that use highly dynamic data structures. We hope that BaLinda K, by also providing parallel tasking and tuple operations and being available on low cost systems, would prove to be a positive development in equipping programmers with an additional computing tool, beyond other ways of bringing Lisp to imperative programmers. [13]

References

[1] G L Steele and R P Gabriel, The evolution of Lisp, ACM SIGPLAN Notices, 28(3), pp. 231- 270, 1993.

[2] W A Martin and R J Fateman, The MACSYMA system, 2nd Symposium on Symbolic and Algebraic Manipulation, pp. 59-75, 1971.

[3] V R Pratt, CGOL: An alternative external representation for Lisp users, AI Working Paper No. 121, MIT AI Laboratory, 1976.

[4] A C Hearn, Reduce 2: A system and language for algebraic manipulation, 2nd Symposium on Symbolic and Algebraic Manipulation, pp. 128-133, 1971.

[5] Apple Corp, Dylan Language Reference, 1994.

[6] M D Feng and C K Yuen, Dynamic load balancing on a distributed system, Sixth IEEE Symposium on Parallel and Distributed Processing, October 1994, Dallas, USA. (Proceedings published by IEEE Computer Society Press, Los Amitos, California, 1994, pp 318-325.)

[7] M D Feng, Y Q Gao and C K Yuen, Distributed Linda tuplespace algorithms and implementations, CONPAR94-VAPP VI, Springer LNCS No. 854, pp. 581-593, 1994.

[8] M D Feng, W F Wong and C K Yuen, Design and implementation of abstract machine for parallel Lisp compilation, International Conference on Parallel Processing, August 1995, Oconomowoc, Wisconsin.

[9] C K Yuen and M D Feng, Breadth-first search in the eight queens problem, ACM SIGPLAN Notices, 29, No. 9 (1994): 51-55.

[10] R Bird and P Wadler, Introduction to Functional Programming, Prentice-Hall, 1988, p. 161.

[11] C K Yuen and M D Feng, Iterative computation and speculative processing, Software- Concepts and Tools, 16, pp. 41-48, 1995.

[12] P H Winston and B K P Horn, Lisp, Addison-Wesley, 3rd edition, p. 187.

[13] B W Benson, libscheme: Scheme as C library, USENIX Symposium on Very High Level Language, 1994