01 Jan 1984

Parallelism in the Language, Natural

Thomas J. Sager

Parallelism in the Language, Natural

Thomas J. Sager

CSc-84-2

Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401 (314)341-4491

Parallelism in the Language, Natural

by Thomas J. Sager

Abstract

Natural is a language designed to provide a vehicle for the expression of abstract programming concepts clearly and precisely in a natural and mathematical form. The concept of parallelism can be expressed both explicitly and implicitly in the language, Natural. Due to relative freedom from side-effects and the use of a special value, undef, subexpressions can often be evaluated in parallel. The for and do statements both allow for a parallel mode of execution. A builtin functional, prlleval, creates functions which can evaluate their arguements in a parallel mode. In addition, the concept of module allows for the definition of and communication among processes.

Parallelism in the Language, Natural

The language, Natural, [7] was designed in 1983 with the explicit goal of providing a vehicle for expressing abstract programming concepts clearly and precisely in a natural and mathematical form. A project to implement the language, Natural is currentrly underway. Because of its specific goal, Natural has a somewhat different emphasis and flavor then most programming languages. In this short paper we will look at the manner in which the concept of parallelism is expressed in Natural.

The following criteria were considered of primary importance in the design of Natural:

1. To merge what the designer felt were the strong points of the general purpose languages such as Pascal [3] and Pl/1 [5] with what the designer felt were the strong points of the functional languages such as Lisp [4] and FP [1]. Strong typing and nested scope rules were borrowed from the general purpose languages. Relative freedom of side-effects, dynamic creation of functions and lack of a clear distinction between variable and function were borrowed from the functional languages.

2. To provide a large variety of programming and data structures. However, it was felt that structures with similar abstract qualities should have similar syntactic definitions. For example functions, procedures and arrays are all represented syntactically as functions.

3. To provide for explicit and implicit expressions of parallelism. Implicit parallelism within Natural source code should be locatable with relative ease. In this regard, Natural has been influenced by concurrent extentions of pascal such as Ada [6], Concurrent Pascal [2] and Modula-2 [8].

4. To make no attempt to "enforce" upon the programmer what the designer feels constitutes "good programming practices". Rather, the language allows for consistent use of what the designer feels constitutes good programming practices.

These criteria have led to a language with marked differences from existing languages. In adhering to the above criteria, the following methods of expressing parallelism were adopted.

1. Expressions:

With the exception of expressions that directly or indirectly contain common objects, all expressions are free of side-effects. A common object is one declared at the highest level within a module. Thus the above case withstanding, subexpressions may be evaluated in parallel.

a. Functions may not have side effects except as stated above. Globals within a function are read only and refer to the value at time of definition not invocation. For example:

```
let
    a: int <- 3;
    f: func( x: int -> y: int ) <- do y <- a * x
do(
    a <- 5;
    a <- f(2) );  >> assigns a the value 6, not 10.
```

b. Expressions may take on the value undef and

"true or undef = true" and "false and undef = false",

thus: "a[i] = x or i > n"

evaluates to true when i > n even if a is defined only on the range [1..n]. Thus conditional and complete evaluation of boolean expressions will yield the same results provided no common objects are involved.

2. Statements:

```
<for statement>  -> forall id in <expr> <fmode> do <statement>
<fmode>          -> ascending | descending | parallel | ϵ
<do statement>   -> do <do mode> <statement list>
<do mode>        -> parallel | ϵ
```

Both the do and the for statement as described by the above BNF grammar can execute in parallel mode. In parallel mode the for statement will cause parallel execution of the statement on the right, once for each member of the (set-valued) expression. In parallel mode, the do statement will cause each statement in the statement list to be executed in parallel.

3. Functionals:

The builtin functional, prlleval, takes two arguements, a binary function, f, assumed to be associative and commutative and a value assumed to be the identity element for f. Prlleval produces a function, g, whose arguement is a sequence. g applies f to each member of its arguement without regard to sequentiality or order producing a value.

For example:

```
s:      seq(int) <- <.v1, v2, ..., vlast.>;
sum:    func( x:int, y:int -> z: int) <- do z <- x + y;
sumall: func( seq(int) -> int ) <- prlleval(sum, 0);
mean:   int <-  sumall(s) / s'ubnd
```

4. Modules:

Modules allow for definition of and communication among processes. A module consists of:

a. A set of common objects, variables and functions.

   i All common objects maintain existence throughout the life of the module. The life of a module is synonymous with its scope. Two common objects within the same module are considered local to each other.

   ii. Common objects may be designated as entry points. An entry point to a module can be accessed in a read only manner from anywhere in the module's scope.

   iii. A module or any entry point to a module may be shared. Shared means concurrent usage by more than one process is allowed. A module which does not have the shared attribute may be used by only one process not in the wait state at a time. An entry point which is not shared can be used by only one process at a time regardless of state.

b. An optional statement:

   If the statement exists, it is taken as a process and begins execution upon instantiation of the module. This statement controls the module in the sense that it can cause premature termination (interrupt) of its common functions.

An example of a Natural program for matrix multiplication appears in figure 1. The language, Natural, appears to be well adapted to clear exposition of this and other parallel algorithms.

References

[1] Backus, J.: Can programming be liberated from the von-Neumann style? A functional style and algebra of programs. Comm. ACM 21, 8, August 1978 (613-641).

[2] Brinch-Hansen, P.: The Architecture of Concurrent Programs. Prentice-Hall, Englewood Cliffs, NJ 1977.

[3] Jensen, K. and Wirth, N.: Pascal User Manual and Report. 2nd Ed. Springer-Verlag, New York 1975.

[4] McCarthy, J. et al: Lisp 1.5 Programmers Manual. MIT Press, Cambridge, Mass. 1965.

[5] OS PL/1 Checkout and Optimizing Compilers: Language and Reference Manual. IBM GC33-0009-4 5th Ed. October 1976.

[6] Reference Manual for the Ada Programming Language. US DoD 1983.

[7] Sager, T.: The Natural language report. (to appear in 1984)

[8] Wirth, N.: Programming in Modula-2. Springer-Verlag, New York 1982.

Figure 1. Matrix Multiplication using Parallel Algorithm

```
max:            int <- ? ;
floatbuf:       modtype(              >>  floating point buffer
   r:               float <- undef;
   put:             entry func( t: float -> ) <-
                        do( await(r = undef); r <- t);
   get:             entry func( -> t: float ) <-
                        do( await(r  = undef); r,t <- undef,r);
ppu:            modtype interface(    >>  parallel processing unit
   putx, puty:      func( float -> );
   initial:         func( int, int, int -> );
   final:           func( -> float ) );
matunit:        func( [1..max], [1..max] -> ppu );
ppu:            shared module(        >>  parallel processing unit
   statustype:      type <- {waiting, ready, running, done};
   status:          statustype  <- waiting;
   i, j, n:         int;
   x, y, z:         float;
   xbuf, ybuf:      floatbuf;
   putx:            entry func( t: float -> ) <- do xbuf.put(t);
   puty:            entry func( t: float -> ) <- do ybuf.put(t);
   initial:         entry func( ii: int, jj: int, nn: int -> ) <- do(
       await( status = waiting );
       i,j,n <- ii,jj,nn;
       z <- 0.0;
       status <- ready );
   final:           entry func( -> t: float) <- do(
       await( status = done ); t <- z; status <- waiting );
   repeat(          >>  process
       await( status = ready );
       status <- running;
       x,y <- xbuf.get(), ybuf.get();
       forall k in [1..n-1] do(
           z <- z + x*y;
           do parallel(
               (if( i<n -> matunit[i+1,j].puty(y) ); y <- ybuf.get() );
               (if( j<n -> matunit[i,j+1].putx(x) ); x <- xbuf.get() ) ) );
       do parallel(
           if( i<n -> matunit[i+1,j].puty(y) );
           if( j<n -> matunit[i,j+1].putx(x) ) );
       status <- done ) );
matmult:        func( n: int, a: func([1..n], [1..n] -> float), b: typ(a) ->
                    c: typ(a) ) <- do(
   forall i in [1..n] parallel, j in [1..n] parallel do
       matunit[i,j].initial(i, j, n);
   do parallel(
       forall i in [1..n] parallel, j in [1..n] ascending do
           matunit[i,1].putx( a[i,j] );
       forall j in [1..n] parallel, i in [1..n] ascending do
           matunit[1,j].puty( b[i,j] ) );
   forall i in [1..n] parallel, j in [1..n] parallel do
       c[i,j] <- matunit[i,j].final() );
```