*LISP AND SYMBOLIC COMPUTATION: An International Journal, ?, ??–??, 1993*


# Plurals: A SIMD Extension to EuLisp

SIMON MERRALL* (*sm@maths.bath.ac.uk*)
JULIAN PADGET (*jap@maths.bath.ac.uk*)
*University of Bath, School of Mathematical Sciences, Bath BA2 7AY, United Kingdom*

**Keywords:** Data Parallelism, Lisp, Primitive Machine Model, SIMD

**Abstract.** There are now several versions of Lisp for massively parallel SIMD architectures like the Thinking Machines Connection Machine. We describe here the extensions made to EuLisp for data-parallel programming and their implementation on a specific platform, a MasPar MP-1. Plural EuLisp, in keeping with the rest of the language, presents a collection of simple orthogonal operators which capture the essence of data parallel processing. In support of this, we demonstrate how to implement a number of higher-level abstraction from other data-parallel languages.

## 1. Introduction

Plural EuLisp is an intermediate stage in our implementation of extensions inspired by the Paralation Model described by Gary Sabot [4]. A large set of data parallel primitives allow objects to be allocated and manipulated on the processor array. In this respect it is not unlike *Lisp (section 4.1). However Plural EuLisp also includes a processor management mechanism motivated by Paralation Lisp. This allows a set of processors to be allocated leaving the remaining processors available for later allocation. When dealing with massively parallel systems this is preferable to tying up thousands of processors when only a fraction are needed.

In general, massively parallel computers have a very large number of tightly linked processing elements (PEs). Each PE comprises a simple processor and a small amount of local memory. The PEs can usually communicate with their immediate neighbours via direct connections (either a four or an eight way grid) and with any PE in the array via a router mechanism, for example a hyper cube network (CM-2) or an hierarchical cross-bar (MasPar). These computers work in an SIMD (Single Instruction Multiple Data) fashion; all the PEs execute the same instruction at the same time, but on different data. The instructions are broadcast to all PEs

*This work has been partially supported through the British Council ARC Programme, a Science and Engineering Research Council (SERC) Studentship, SERC grant GR/G31048, International Computers Limited (SERC CASE award)

by a single controller unit and each PE has an activity bit which controls whether the PE executes the current instruction. There is also often a logical OR-tree used to determine quickly if any PEs are active.

Two of the most widely used examples of massively parallel architectures, the Connection Machine and the MasPar, though matching this general configuration differ from each other significantly. These machines are connected to a conventional host computer which controls interaction with the processor array. On the Connection Machine the host computer sends a *macroinstruction* stream to the controller unit which broadcasts a *nanoinstruction* stream to the PEs. On the MasPar the control unit (ACU) is capable of independent program execution and executes a program loaded up from the host computer containing both parallel and serial instructions. This means that on the CM-2 the PEs can directly address the memory of the host computer whereas on the MasPar they can only access the ACU. In consequence the Connection Machine forms a single computer with its host, while the MasPar is a distributed system.

The dialects of Lisp for these machines which contain no high-level abstractions reflect the architectures they were developed for. Plural EuLisp encapsulates a primitive data parallel model which aims to be independent of these two architectural models. Most aspects of data parallel languages can be expressed in Plural EuLisp, but its simple nature makes it a good vehicle for describing general issues in the implementation of such languages. Similar mechanisms have been implemented for the Connection Machine suggesting Plural EuLisp is not restricted to a particular architecture.

In the next two sections we will first describe Plural EuLisp, followed by its implementation on the MasPar MP-1. We also describe an extended system, the MasPar Lisp Server, which allows several Lisp (distributed) processes to share the MasPar. This gives better utilisation of the data parallel resource and is not a feature of other languages. We then give a brief overview of other data parallel languages, and outline how they might be defined in Plural EuLisp.

## 2. Plural EuLisp

Plural EuLisp is a data parallel extension of EuLisp. The extensions supply a new sequence data structure called a `plural`, which is similar to a vector, each element of which is allocated on a separate processing site. A plural is created by the function `make-plural`. It takes the length of the desired plural as its argument. For example:

```
(setq a (make-plural 5))
=> #P(() () () () ())
```

The initial value of each element of the plural is `()` (the empty list). We can set and reference elements of the plural using the function `plural-ref` and its updater:

```
((setter plural-ref) a 1 '(1 a))
=> #P(() (1 a) () () ())
```

Plural EuLisp has a set of primitive functions which can be applied to plurals. These are data parallel versions of typical lisp primitives. They are usually distinguished by a `-s` suffix (e.g. `car-s` `null-s`), but where there is an appropriate generic function the data parallel version has been added as a method (e.g. `+`). When the function is applied to a plural it is as though the serial version of the function were applied to each value in the plural and the result is a new plural containing these values (in the same order).

```
(null-s a)
=> #P(t () t t t)
```

The resulting plural will be allocated on the same set of processing sites as the argument plural since its values were created on those sites. In this case, we say that the two plurals are *conformal* or belong to the same *conformal set*. There is an additional function, `bang`, which has no serial counterpart. This projects a singular value into a plural, for example:

```
(setq b (bang 55 a))
=> #P(55 55 55 55 55)
```

This creates a new plural conformal to `a` with each element set to `55`. If a data parallel function takes more than one argument (e.g. `cons-s`) then they must be conformal. So

```
(cons-s b a)
=> #P((55) (55 1 a) (55) (55) (55))
```

is correct, but `(cons-s b (make-plural 5))` signals an error as the new plural would not be conformal to `b`. Therefore, to make it easier to allocate a conformal plural, the argument to `make-plural` can also be a plural, in which case the result is a plural conformal to the one supplied as the argument. Similarly the conversion functions `list-to-plural` and `vector-to-plural` accept a plural as the optional second argument and

the result will be conformal to this plural—padding or truncating the list or vector data as necessary.

In order to write any non-trivial parallel functions we need one more function, `if-s`, a parallel version of `if`. The arguments are three expressions which deliver (conformal) plural values. The values of the first plural are interpreted as booleans which are used to modify the activity of the elements of the virtual processor set before each of the remaining expressions are evaluated. The plurals resulting from these two expressions are merged to form the result of the conditional expression. Before executing the two forms the fast or mechanism is used to check there are some active processors, if not the expression is simply ignored; this is important when defining recursive functions:

```
(defun list-length-s (list-s)
  (if-s list-s (+ (bang 1 list-s) (list-length-s (cdr-s list-s)))
    (bang 0 list-s)))
```

If both the `if-s` forms were evaluated regardless of the current activity this intuitive definition of `list-length` would recurse indefinitely. This shows how it is, in principle, straightforward to define parallel functions. This is the processing side of the model. The other side is communication and that mechanism is modelled closely on that in Paralation lisp.

Given two plural arguments the function `match` creates a relation between the conformal sets of the plurals called a *mapping*. This defines which elements from the source set are mapped to each site in the destination set. It can be thought of as a collection of arrows between the two sets connecting the sites that were equal in the original plurals (those given to `match`). Given a mapping and a plural in the source conformal set the function `move` creates a plural in the destination conformal set. This can be thought of as the values in the plural moving down the arrows in the map to sites in the destination conformal set. It is possible that there will be no arrows pointing to a site in the destination and a default value is supplied for this case. If more than one arrow points to a site then a given binary function is used to combine the values.

```
(setq from (list-to-plural '(nowhere 1st 1st 2nd nowhere)))
=> #P(nowhere 1st 1st 2nd nowhere)

(setq map (match (list-to-plural '(1st 2nd 3rd)) from))
=> #<mapping>

(move (list-to-plural '(a b c d e) from) map cons-s 'empty)
=> #P((b . c) d empty)
```

In the example above a collision between two objects occurs in the first element and they are made into a cons pair (note that the order in which the arguments are presented to the combining function is undefined). The last element has no counterpart in the source and takes the default value `empty`. A detailed summary of the operators defined in the plural module appears in the appendix.

## 3. Implementation

A plural is a collection of objects allocated on a set of processing sites, one per site. Two plurals are conformal if they are allocated on the same set of sites. A plural is specified by two components, its conformal set and its values. As well as specifying the physical sites the values of a plural are allocated on, the conformal set also identifies the internal activity (as modified by `if-s`) of the plural. These two components make a natural division in the parallel system:

1. The Parallel Lisp Kernel (PLK): Functions Allocating and manipulating lisp objects in parallel.
2. The Context Management System: Allocating conformal sets

Both sections are written in mpl (Maspar's data-parallel variety of C) and, along with the special functions handling communication, constitute all the code running on the MasPar.

The extended EuLisp has an extra intrinsic module written in C called `plural` and the exported functions invoke the appropriate functions on the MasPar. The `plural` functions are still very basic and so they are wrapped in a Lisp module which defines the class `plural` along with its operators. Figure 1 shows the system organisation.

### 3.1. Parallel Lisp Kernel

This set of parallel lisp primitives manipulate a Lisp object on each currently active PE in a conformal set. We refer to these collections as Parallel Lisp Objects (PLOs).

#### *3.1.1. Parallel Lisp Objects*

Each processing element (PE) contains a small garbage-collected heap. To conserve space we have adopted a 16-bit addressing system in which an address is an index into an array. We use a compacting garbage collector and so allocation is simply a matter of moving the *heap top* pointer. This

Figure 1: System Organisation

makes it easy to allocate different sized objects on different PEs in parallel and so we can support heterogeneous plurals without difficulty. This takes advantage of the local indirect addressing available on the MasPar where although all the processors execute the same instruction stream it need not be applied to the same address on each PE.

In general the objects in a PLO will be at different locations on each PE and it would be impractical for the host computer (i.e. the lisp process) to keep track of all these addresses. A section of the heap called the plural space is used to give the host a handle on parallel lisp objects. It also forms a part of the processor management mechanism (A more complete description is given in [3]). To give the host a handle on the PLO we allocate a slice (i.e. the same location on each PE) of the plural space and store the address of the objects within this slice. This means the host only requires one value to identify a PLO.

### *3.1.2. Parallel Lisp Functions*

A number of the fundamental lisp operations, such as `cons`, `car` etc., have direct equivalents in mpl. But some functions, such as `binary-` and `binary/` have been combined into a single function (e.g. `bin_op`) which takes an additional argument to identify the operation. This reduces the amount of code but also, as the opcode can denote a parallel operation,

different operations can be executed on different PEs which will share the allocation and argument checking phases of the operation. These functions all have the same function prototype.

`int` *function_name*`(` `MP_PluralHeap` *MPPH_arg*, [ `MP_PluralHeap` *MPPH_args* ], `MP_PluralHeap` *MPPH_result* `);`

Obviously the number of arguments will vary and functions like `bin_op` have an extra argument `op_id`.

The mpl handle on a PLO is the parallel address of a 16-bit heap address and is defined as follows:

```
typedef plural natural *plural MP_PluralHeap;
```

And `natural`, which has the role of an address, is defined as follows:

```
typedef unsigned short int natural;
```

We pass addresses because many of the operations are not functional and, instead, we treat the return value as a completion code. Currently if any of the processors should fail then the operation fails globally (note the function has a singular result). Below is the simplified mpl code (error checking has been omitted) for the function `cons`. The explanation follows the code.

```
int cons( MPPH_car, MPPH_cdr, MPPH_pair )

MP_PluralHeap MPPH_car, MPPH_cdr, MPPH_pair;

{
  plural cons_cell *plural new_cell;
  plural natural    tmp;
  MP_PluralHeap     MPPH_tmp = &tmp;

  mp_alloc(MP_CONS, (plural int) 1, MPPH_tmp);

  new_cell = (plural cons_cell *plural) OA_data(MPPH_tmp);
  new_cell->car = OA_offsets(MPPH_car);
  new_cell->cdr = OA_offsets(MPPH_cdr);

  OA_offsets(MPPH_pair) = OA_offsets(MPPH_tmp);

  return SUCCESS;
}
```

First we request a new cons cell on each active PE, we create a temporary handle for these as the arguments may not be independent. The macro `OA_data` extracts the physical address of the PLO and we cast this to a plural pointer to the type of object we are dealing with, in this case the structure `cons_cell`. The contents of this pair are set to the addresses, that is the offsets into the 16-bit heap, of the two arguments. The macro `OA_offsets` extracts this value from the PLO handle. To finish we copy the contents of the temporary handle (`MPPH_tmp`) into the result handle (`MPPH_pair`) and return `SUCCESS`.

In this section we have described how collections of objects are allocated and manipulated by the functions in the Parallel Lisp Kernel (PLK). Next we look at the other component of plurals, the conformal sets.

### 3.2. Context Management

A context is a mechanism for identifying the set of processors corresponding to a conformal set and also its internal activity. Viewing the processor array as a sequence we can identify a contiguous subset by its start and length. We allocate a structure on the ACU containing these two values. It also contains a plural space offset, this gives the internal context and will be explained shortly. The context is defined as follows:

```
typedef struct _MP_Context {

natural start;
natural length;
natural offset;

} MP_Context;
```

To execute a PLK function within a certain context we first deactivate all those PEs not in the context. So if `MPC` is a context structure, executing a PLK function within the conditional below will ensure only those processors in the context will perform the operation.

$$(\texttt{MPC.start} < \texttt{iproc}^1 < (\texttt{MPC.start} + \texttt{MPC.length}))$$

The internal context, that is the activity of each site in the context, is given by the top of a stack allocated on each processor in the context. Each stack is a list of `nil` and non-`nil` values and these stacks taken together they form a PLO. Now when executing a PLK function within a given context we need an additional step to get the correct active set. Having

[1] `iproc` is an mpl global plural containing each PEs number.

activated only those processors in the context we then take the `car` of the context stacks and modify the active set further depending on this value.

The context stacks are manipulated by the functions `mp_if`, `mp_else` and `mp_fi` . Once the stacks have been modified they remain in that state and affect all operations in that conformal set until they are modified again. The EuLisp macro `if-s` first calls `mp_if` with the result of the boolean expression and then evaluates the consequent form. Any parallel functions in this form will be executed with respect to the modified context. `Mp_else` is called to set the context for evaluating the alternative form. Finally `mp_fi` is called to restore the context stacks to their previous state.

### 3.3. Plurals

A plural comprises an offset into the plural space and the address of the context structure on the ACU. The EuLisp module `eubang` defines the class `mp-object` with `plural` as a subclass. This is because mappings have the same representation but we wish to distinguish them from plurals. This we achieve by declaring a common superclass `mp-object` with plurals and mappings inheriting from it. The TELOS definitions for these classes are:

```
(defclass mp-object ()
  ((context
    initarg context
    reader  context)
   (offset
    initarg offset
    reader  offset))
  predicate mp-object-p)

(defclass plural (mp-object)
  ()
  constructor (allocate-plural context offset)
  predicate pluralp)
```

In general the primitive functions in the `plural` module take as their arguments a context address and a set of plural space offsets. The functions in `eubang` extract these values from the plural objects, check they are conformal by comparing the context addresses and then call the appropriate primitive function. This will then make a call to the MasPar. The result will be a new plural space offset and this, along with the context address of the arguments, is used to create a new plural object.

### 3.4. Communications

One aspect of communication is printing the contents of a plural. This is done by writing the output of each PE into a local scratch space. These strings are then printed using parallel `printf` wrapped with `#P( ...)`.

Another aspect is that of moving objects between processors. This includes transfers between the host and the array, as well as transfers between processing elements. We now examine this problem in more detail.

#### *3.4.1. Transferring Lisp Objects*

To copy an object between processors we encode it into a byte string, transfer the string and decode it to build a copy on the destination processor. For simplicity we recursively encode structures implicitly defining the references in them. For example if the decoder creates a cons cell it takes the next two objects it creates as its car and cdr. Such an algorithm means that we cannot deal with reentrant structures. We are further limited by how big a string we can construct on a processing element. We encode a lisp object as follows:

Type: 1 Byte.

Size: 1 Byte, optional, only used for vectors.

Data: This may be actual data, 4 bytes containing a value for an integer say, or for a cons cell the data is decoded to create further lisp objects which constitute the data.

There are mpl and C versions of `encode` and `decode` and the same protocol is used for both types of transfer. But during an host to array transfer it may be necessary to work in batches because the host has more scratch space than the array. The encode/decode routines are used for transferring objects by the functions `bang`, `plural-ref` and its updater. They are, of course, also used to implement mappings since we need to compare data held on different PEs.

#### *3.4.2. Mappings*

A mapping is represented by a plural conformal to the destination plural. Each element contains a list of processor ids. These are PEs that objects should be taken from to make the new plural. A mapping is another kind of `mp-object` as noted in section 3.3.

```
(defclass mapping (mp-object)
  ()
  constructor (allocate-mapping context offset)
  predicate mappingp)
```

To move a plural down a mapping we first encode it. Then we descend the lists of processor ids and each PE copies the object description from the indicated PE using the Maspar's global router. We build the object and move onto the next id consing up the new objects as we do so. The following pseudo-code expresses the algorithm for moving data down a mapping:

*moved_plural* = `nil`
**while** ($\exists x \in map$ : !null?(x))
encode(*data*)
fetch(*scratch*, car(*map*))
*new_objects* = decode(*scratch*)
*moved_plural* = cons(*new_objects*, *moved_plural*)
*map* = cdr(*map*)
**elihw**
**return** *moved_plural*

The result is a plural of lists of moved objects. From these a single value is computed using the combining function specified, or if the list is empty, the default value is used. Below is the pseudo-code for building a mapping, i.e. a plural of lists of PE ids.

*map* = `nil`
**for each** ($x \in source$)
**if** ($x = y \in destination$)
*map* = cons(`iproc`($x$), *map*)
**fi**
**rof**
**return** *map*

### 3.5. The MasPar Lisp Server

Using Plural EuLisp we can partition the processor array into a collection of independent sets of processors. This is because each conformal set has its own internal context which persists between function calls. As a result, instruction streams for different conformal sets can be interleaved with no danger of them interfering with each other. To do this in practice, we have replaced the front-end's EuLisp process with a server. EuLisp processes on

any of the machines on the local area network can connect via sockets to the MasPar and allocate data parallel objects via the Plurals interface.

The MasPar has a job-swapper which allows several programs to share the processor array by dividing the PE memory between the programs. This is not an ideal method for running multiple data parallel lisp processes on the MasPar. Firstly the reduced memory makes the heap size prohibitively small, secondly it requires running several lisp processes on the host machine, a task to which it is not well suited. In contrast the lisp server allocates memory as it is required and where possible the lisp processes will be allocated disjoint processor sets. In this way several programs that need only part of the processor array can be run at the same time without affecting each other.

## 4. Related Work

In this section we will briefly describe some of the other data parallel symbolic languages and outline how they can be defined in terms of Plural EuLisp.

### 4.1. Other Languages

***Lisp** [**1**] A language devised for the CM-2, it gives fine control over the processor array via a very large number of functions which operate on *pvars*. Each pvar has as many elements as the virtual processor configuration which is being used (i.e. it is fixed). The parallel operators are distinguished from their serial counterparts by a `!!` suffix or a `*` prefix. Naturally enough, this language reflects the nature of the CM-2 where the host computer directly controls the array and the singular and parallel memory are mutually addressable.

**TUPLE** [**7**] This is a more recent language developed on the MasPar rather than the CM-2. A notable aspect of TUPLE is how its organisation mirrors that of the MasPar itself. The programmer defines data parallel variables(`defpevar`), and functions (`defpefun`) and execution is then invoked using the `ppe` form which invokes the evaluator running on the ACU. The data parallel component of TUPLE is effectively a disjoint subsystem. This reflects the MasPar system architecture where the Array Control Unit is a full control processor capable of independent program execution.

**Paralation Lisp** [**4**] A paralation is a collection of processing sites, a field belonging to a paralation has a value for each site in that paralation. A paralation is created using the function `make-paralation`. Having

allocated some new set of processing sites it then returns a field in the new paralation which numbers the sites from 0 to $n-1$, this is the *index* field.

```
(setq a (make-paralation 5))
=> #F(0 1 2 3 4)
```

Parallel code is written using the `elwise` form, this takes a list of symbols bound to fields in the same paralation and a lisp expression. Within the body of the `elwise` the lisp expression is executed in parallel on the paralation and the symbols are bound to the local values of the fields on each site rather than the entire field. In effect `elwise` is a special kind of `let` form.

```
(elwise (a) (cons a ()))
=> #F((0) (1) (2) (3) (4))
```

As mentioned earlier mappings in Plural EuLisp are almost identical to those in Paralation Lisp and are created in the same way using `match`. The only actual difference is that in paralations a singular rather than parallel combining function is specified. See section 4.2.3.

Finally, Paralation Lisp has a function `vref` which reduces a field to a single value using a given binary combining function.

```
(vref a +)
=> 10
```

**Connection Machine Lisp [5]** The data parallel objects of CM-Lisp are *xappings*: these are unordered sets of ordered pairs. The first element of each pair is the *index* the second is the *value*, e.g:

```
{tiny→rabbit small→glow major→boat}
```

Additional program notation is introduced to indicate various parallel operations. The symbol $\alpha$ indicates parallel code, and the symbol $\bullet$ cancels the affect of $\alpha$. Operations are applied to the xapping values and the index of the result will be the intersection of the argument indexes.

```
α(cons •'{0→tiny 1→small 2→major}
       •'{1→hoot 2→tree 3→boat})
=> {1→(small .  hoot) 2→(major .  tree)}
```

Communication is abstracted by $\beta$ which in general performs a reduction in the same way as `vref`, but it can also operate in a way similar to mappings. However, $\beta$ is too complicated to describe in full here; the interested reader is directed to the reference above.

**NESL** [**2**] NESL is a strongly typed, applicative data parallel language with a lisp-like syntax. Parallelism is supplied through a set of data parallel constructs based on vectors, including the `over` form which applies any function over the elements of a vector in parallel, and a broad set of parallel functions that manipulate vectors.

### 4.2. Implementation using Plurals

We now briefly indicate how these languages could be implemented using Plural EuLisp.

#### *4.2.1. Parallel Data Structures*

Most of the parallel data structures in these languages are easy to represent using plurals. The parallel variables in TUPLE and *Lisp are equivalent to plurals which have as many sites as there are physical processors. Adding an extra slot for the length and ignoring the excess elements gives us the vectors of NESL.

Fields and paralations correspond closely to plurals and conformal sets. Adding an additional slot to the plural class which contains the paralation index field gives us a field class.

Xappings are best implemented by using the *rendezvous* mechanisms described by Hillis and Wholey [6]. Every object which is used in the range of a xapping has a unique rendezvous location. This would be a site in a sufficiently large plural, the range object is stored in this slot and is referred to by its rendezvous location. Xappings are represented by a pair of conformal plurals, the range and the value. The value contains objects but the range contains the position of the rendezvous location. The reasons for this will become clear shortly.

#### *4.2.2. Execution*

With the exception of *Lisp all the languages here have a special form used to indicate parallel code. The code within the form is an ordinary lisp expression which is executed in parallel. So to implement any of these languages in Plural EuLisp requires a kind of compiler which rewrites the special forms into Plural EuLisp expressions. This is, in general, a fairly simple process: functions must be replaced with their parallel counterparts and singular values must be wrapped with code which will replicate them at run time.

In TUPLE, *Lisp and NESL the functions are executed in the single global context. In Paralation Lisp we must check that the plurals are conformal i.e. belong to the same paralation. In CM-Lisp we must identify the intersection of the xappings and evaluate the expression for those locations only. The rendezvous mechanism makes this straightforward, albeit expensive. The values of the xappings involved in the expression are sent to the rendezvous locations given in the range. The expression is evaluated in the rendezvous plural but only on the sites which received all the values. The results are then collected into a new xapping.

### *4.2.3. Communication*

To convert plural mappings into paralation mappings we must be able to derive the parallel version of the combining function from the singular argument to `move`. This is basically the same conversion process used by `elwise`. Most of the various communication functions in TUPLE, *Lisp and Nesl can be implemented using mappings. Those functions using fixed and regular communication patterns, e.g. the reduction operators, could use a library of pre-computed mappings eliminating the cost of `match`. Other functions would have to create mappings which provided the desired communication pattern as they were needed.

The $\beta$ operator of CM-Lisp also makes use of the rendezvous mechanism. The communication patterns defined by $\beta$ are implicit in the act of sending objects to their rendezvous location and so much of the mechanics of $\beta$ are present in the rendezvous mechanism. It only remains to combine any collisions and collect the results into a new xapping to complete the operation.

### *4.2.4. Nested Parallelism*

In CM-Lisp and Paralation Lisp the parallel forms, i.e. $\alpha$ and `elwise`, can be nested. Early versions of these languages did not execute nested expressions fully in parallel, instead they sequentialised on the outer forms and only the inner-most parallel form was executed in parallel. NESL however executes nested forms fully in parallel at all levels by flattening out the nested forms at compile time, the same techniques have been used in later versions of paralation lisp. The high-level expressions are compiled into a more primitive language (e.g. SV-Lisp or VCODE) in much the same way we rewrite the expressions into Plural EuLisp. These primitive languages motivate extensions to Plural EuLisp which would allow us to fully support nested parallelism, in particular the segmented scan operations which allow a collection of vectors to be represented by and manipulated as a single vector.

## 5. Summary

We have examined various issues in the implementation of data parallel languages here by looking at the relatively simple mechanisms in Plural EuLisp. Despite their simplicity the abstractions are central to most data parallel languages and we have outlined how they can be defined in terms of plurals. Execution of data parallel programs is currently restricted by the speed of the host computer where it is being interpreted: on the MasPar this is a VaxStation which is prohibitively slow. This partly motivated the Lisp Server which allows data parallel lisp programs to be run from faster machines. This has demonstrated an added advantage of the processor/memory management mechanisms in Plural EuLisp which make it easy for several programs or for several components of a distributed program to share the MasPar.

## References

1. **Lisp Reference Manual*. Thinking Machines Corporation (1988).

2. Blelloch, G. E. *NESL: A Nested Data-Parallel Language*. Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213 (Jan 1992). CMU-CS-92-103.

3. Merrall, S. C. and Padget, J. A. *Collections and Garbage Collection*. Proc. of International Workshop on Memory Management, IRISA/INRIA - Rennes, Campus de Beaulieu, 35042 Rennes Cedex, France (Sept 1992). LNCS 637.

4. Sabot, G. W. *The Paralation Model: Architecture Independent SIMD Programming*. MIT Press, Cambridge, MA (1988).

5. Steele, G. L., Jr., and Hillis, W. D. *Connection Machine Lisp: Fine-Grained Parallel Symbolic Processing*. ACM Conference on Lisp and Functional Programming (1986) 279–297.

6. Steele, G. L., Jr., and Wholey, S. *Connection Machine Lisp: A Dialect of Common Lisp for Data Parallel Programming*. International Conference on SuperComputing (1987). TMC Tech. Report PL87-6.

7. Yuasa, T. *TUPLE - An Extension of KCL for Massively Parallel SIMD Architecture*. Toyohashi University of Technology, Toyoyashi 441, Japan, draft of 2nd version (1992). available from author.

## A. The Plural Module

`make-plural` *plural function*

*Arguments*

*spec*: Either an integer, $n$, or a plural, $p$.

*Result*

Returns a freshly allocated object of class plural. If the argument is an integer then the new plural will have $n$ elements. If the argument is a plural, then the result will be *conformal* to $p$. Two plurals are conformal if they are allocated on the same set of processors.

`plural-length` *plural function*

*Arguments*

*plural*: An instance of `plural`.

*Result*

Returns the number of elements in *plural*.

`plural-ref` *plural function*

*Arguments*

*plural*: A plural.

*index*: An integer.

*Result*

Returns the item stored at position *index* of *plural*. It is an error if *index* is not in the index range of *plural*.

`(setter plural-ref)` *plural function*

*Arguments*

*plural*: A plural.

*index*: An integer.

*object*: An object.

*Result*

Returns the (modified) plural.

*Remarks*

Modifies *plural* so that *object* is stored at position *index* of *plural*. It is an error if *index* is not in the index range of *plural*.

---

`bang` *plural function*

---

*Arguments*

*object*: An object.

*plural*: A plural.

*Result*

Allocates a fresh plural conformal to *plural* and initializes each position with copies of the value *object*.

---

`if-s` *plural special form*

---

*Syntax*

`(if-s`*condition consequent alternative*`)`

*Remarks*

This is the parallel conditional form. It is an error unless each of the three forms operated on plurals belonging to the same conformal set. The *condition* is evaluated to deliver a plural of boolean values, which are then used to identify the active sets for the *consequent* and *alternative* forms. The results from the *consequent* and *alternative* forms are then merged to create the result plural.

---

`match` *plural function*

---

*Arguments*

$plural_1$ : A plural of integers.

$plural_2$ : A plural of integers.

*Result*

Returns a mapping from $plural_1$ to $plural_2$.

*Remarks*

Mappings are constructed by `match`. The arguments to `match` are two plurals of integers, not necessarily conformal, and constructs a mapping which represents the set of arrows between elements of $plural_1$ and $plural_2$ identified by integer equality.

---

`move` *plural function*

---

*Arguments*

*plural*: A plural—the source.

*mapping*: A mapping.

*function*: A function—to combine collisions.

*object*: An object—for positions in the target with no arrows.

*Result*

Moves the data in *plural* down *mapping* using *function* to combine collisions and *obj* as a default value resulting in a new plural. The elements of the new plural are initialized from the data in *plural* such that position $i$ receives the result of combining using *function* all the values from the positions in *plural* that point to $i$ according to *mapping*. If there are no such values for a given position it receives a copy of the value *obj*.