Connection Machine® Lisp: Fine-Grained Parallel Symbolic Processing

Guy L. Steele Jr.
W. Daniel Hillis

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142

Abstract

Connection Machine Lisp is a dialect of Lisp extended to allow a fine-grained, data-oriented style of parallel execution. We introduce a new data structure, the xapping, that is like a sparse array whose elements can be processed in parallel. This kind of processing is suitable for implementation by such fine-grained parallel computers as the Connection Machine System and NON-VON.

Additional program notation is introduced to indicate various parallel operations. The symbols α and $\bullet$ are used, in a manner syntactically reminiscent of the backquote notation used in Common Lisp, to indicate what parts of an expression are to be executed in parallel. The symbol β is used to indicate permutation and reduction of sets of data.

Connection Machine Lisp in practice leans heavily on APL and FP and their descendants. Many ideas and stylistic idioms can be carried over directly. Some idioms of Connection Machine Lisp are difficult to render in APL because Connection Machine Lisp xappings may be sparse while APL vectors are not sparse. We give many small examples of programming in Connection Machine Lisp.

Two metacircular interpreters for a subset of Connection Machine Lisp are presented. One is concise but suffers from defining α in terms of itself in such a way as to obscure its essential properties. The other is longer but facilitates presentation of these properties.

1 Introduction

Connection Machine Lisp is intended for symbolic computing applications that are amenable to a primarily fine-grained, data-oriented style of parallel solution. While the language was invented with the architecture and capabilities of the Connection Machine System [11] in mind, its design is relatively hardware-independent, and may be suitable for implementation on other parallel computers, such as NON-VON [22] or the NYU Ultracomputer [21], as well as sequential machines.

Connection Machine Lisp begins with a standard dialect of Lisp, and then adds a new data type (the *xapping*) and some additional program syntax for expressing parallelism. (We use Common Lisp [23] as our base language, but Scheme [28,27,3,2] would be an attractive alternative.) The resulting language is much like APL [13,12,8,4], but with richer data structures; much like FP [1], but with variables and side effects; somewhat like KRC [29], in that one possible semantics for Connection Machine Lisp includes lazy data structures; but rather unlike QLAMBDA [5] or Multilisp [10], which introduce parallelism via control structures rather than data structures (although it may be possible to copy certain good ideas from those languages into Connection Machine Lisp without ill effect).

2 The Xapping Data Type

All parallelism in Connection Machine Lisp is organized around a data structure called a *xapping* (rhymes with "mapping"). A xapping is something like an array and something like a hash table, but all the entries of a xapping can be operated on in parallel, for example to perform associative searching. This data structure by itself is not a particularly original idea; the innovation in Connection Machine Lisp lies in the program notation used in conjunction with it.

To be precise, a xapping is an unordered set of ordered pairs. The first item of each pair is called an *index*, and the second item is called a *value*. We write a pair as *index*→*value*. An index or value may be any Lisp object. A xapping cannot contain two distinct pairs whose indices are the same; all the indices in a xapping are distinct (but the values need not be distinct). There is a question of what is meant by "same"; for now assume that the Common Lisp function **eql** determines sameness.

Here is an example of a xapping that maps symbols to other symbols:

{sky→blue apple→red grass→green}

The same xapping could have been written in this manner:

{apple→red sky→blue grass→green}

The order in which the pairs are written makes no difference.

To speak in terms of implementation on a parallel computer, one may think of an index as a label for a processor, and think of the corresponding value as being stored in the local memory of that processor. The index might or might not be stored explic-

"Connection Machine" is a registered trademark of Thinking Machines Corporation. "Symbolics 3600" is a trademark of Symbolics, Inc.

itly also. The xapping shown above might be represented, for example, by storing pointers to the symbols `apple` and `red` in processor 6, `sky` and `blue` in processor 7, and `grass` and `green` in processor 8. Additional header information indicating that the xapping is stored in three processors beginning with processor 6 must also be stored somewhere. The ingenious reader can no doubt invent many other representations for xappings suitable for particular purposes. In any case, it is well to think of indices as labelling abstract processors, and to think of two values in two xappings as being stored in the same processor if they have the same index.

Semantically a xapping really is like an array or hash table, where the indices may be any Lisp objects. A xapping may be accessed by index to obtain a value:

```
(xref '{sky→blue apple→red grass→green} 'apple)
  ⇒ red
```

(Following [23], we use the symbol ⇒ to mean "evaluates to.")

Sometimes the index and the value of a pair are the same (that is, `eql`). As a convenient abbreviation, such a pair may be written within xapping-notation as just the value, without the index or the separating arrow. For example,

```
{apple→fruit color→abstraction
 abstraction→abstraction}
```

could be abbreviated to

```
{apple→fruit color→abstraction abstraction}
```

This is most convenient in the case where *all* the pairs may be so abbreviated:

```
{red blue green yellow beige mauve}
```

means the same as

```
{red→red blue→blue green→green
 yellow→yellow beige→beige mauve→mauve}
```

but is considerably shorter. If all the elements of a xapping can be abbreviated in this manner, then the xapping is called a *xet* (rhymes with "set").

If a finite xapping has a set of indices that are consecutive nonnegative integers beginning with zero, then the xapping may be abbreviated by writing the values in order according to their indices, separated by whitespace as necessary and surrounded by brackets. For example, the notation `[red green blue]` is merely an abbreviation for `{0→red 1→green 2→blue}`. A xapping that can be abbreviated in this manner is called a *xector* (rhymes with "vector"). The use of xectors in Connection Machine Lisp is similar to the use of vectors in APL.

One can have a theory of lists (or arrays) that can speak of both finite and infinite lists and then use this theory to explain a language implementation that supports only finite lists. One might also implement a very similar language that supports apparently infinite lists by means such special representations as lazy lists or lists all of whose elements are the same. In the same manner, we have a theory that speaks of infinite xappings. For the next few sections we speak as if infinite xappings really are supported. In section 6 we address the semantic and implementation difficulties that can arise when supporting infinite xappings and various trade-offs that can be made.

It is desirable to introduce three different kinds of infinite xappings.

- A *constant* xapping has the same value for every index. A constant xapping with value v is written as `{→v}`. For example, the xapping `{→5}` has the value 5 for every index. Constant xappings are important to the implicit mapping (apply-to-all) notation discussed below in section 3.
- A *universal* xapping, written `{→}`, is the xet of all objects; that is, for every Lisp object there is a pair with that object as both index and value. There is an important operation in the language, called `domain`, that takes a xapping and returns a xet of its indices; given that constant xappings exist, universal xappings are needed so that the `domain` operation can be total.
- A *lazy* xapping uses a unary Lisp function to compute a value given an index. For example, the xapping `{ . sqrt}` maps every number to its square root. (Note the dot that is part of the notation.) Lazy xappings are a means of dealing with the mapping of arbitrary functions over infinite xappings.

Any of the three types of infinite xapping may have a finite number of explicit exceptions, where for a given index there is an explicitly represented value. The "infinite part" is conventionally written after all of the explicit pairs. For example, the lazy xapping

```
{pi→1.772453851 e→1.6487212 -1→i . sqrt}
```

is generally defined by the `sqrt` function but has explicit values for three particular indices.

3 Notation for Implicit Apply-to-All

Parallelism is introduced into Connection Machine Lisp primarily by having a way to apply a function to all the elements of a xapping at once. This notion is not new; indeed, we were inspired by the "apply-to-all" operator α of FP [1]. The apply-to-all operator takes a function f and produces ... *something* ... that, when applied to a sequence, applies f to all the elements of the sequence and yields a sequence of the results:

$$\alpha f : \langle x_1, x_2, \ldots, x_n \rangle \equiv \langle f : x_1, f : x_2, \ldots, f : x_n \rangle$$

We may do the same thing in Connection Machine Lisp:

```
(αsqrt '[1 2 3 4]) ⇒ [1 1.4142135 1.7320508 2]
```

FP is purely applicative, and so the question of order of application is irrelevant. In Connection Machine Lisp, which is not purely applicative, we must address this question, and we also specify more precisely what is that *something* that apply-to-all produces. We begin by treating it as a simple operator (or rather a read-macro character fronting for an operator, much as "'" fronts for `quote`), but are led to regard it as a complex syntactic device rather than a pure functional.

The expression αx, where x is a variable or constant, constructs a constant xapping with the value of x. For example, `α5 ⇒ {→5}`; this is "a zillion fives," loosely speaking.[1] Similarly, the expression `αsqrt` produces "a zillion square-root functions." Putting it back into pseudo-FP syntax, it is as if

$$\alpha f \equiv \langle f, f, f, \ldots \rangle$$

[1] That's "zillion," not "xillion."

We then make the rule that when a xapping is applied as a function, all of the arguments must also be xappings, and the xapping being applied must have functions as its elements (these elements may themselves be xappings). An implicit apply-to-all operation (specifically, apply-to-all of `funcall`) occurs: function elements and argument elements are matched up according to their indices. The result is a xapping that has a pair for every index appearing in the function xapping and all argument xappings. Put another way, the result is defined for all indices for which the function and argument xappings are defined. In yet other words: the domain of the result is the intersection of the domains of the function and argument xappings.

Enough! Time for an example!

```
(αcons '{a→1 b→2 c→3 d→4 f→5}
       '{b→6 d→7 e→8 f→9})
  ⇒ {b→(2 . 6) d→(4 . 7) f→(5 . 9)}
```

Note that the value of αcons, namely {→(cons function)} (again speaking rather loosely), is defined for all indices, and so does not restrict the domain of the result; the function is defined at whatever index it may be needed. On the other hand, the domains of the two arguments are both finite, and their intersection determines the domain of the result.

Operationally, this function call implicitly sets up the following calls to `cons`:

```
(cons 2 6)      (cons 4 7)      (cons 5 9)
```

These calls are executed in parallel (perhaps asynchronously—see section 6). Resynchronization occurs, at latest, when all of the parallel computations have completed and the result xapping is to be constructed.

Note that argument forms are not necessarily evaluated in parallel (as in the `pcall` construct of Multilisp [10]); that is an orthogonal notion. The parallel evaluation of arguments forms is a parallelism in `eval` (or more precisely in `evlis`). The parallelism in Connection Machine Lisp is a parallelism made manifest in `apply`. (This distinction is further discussed below in section 6.)

Consider now two forms: `(α+ α2 α3)` and `α(+ 2 3)`. The first evaluates the function and argument forms to produce {→+}, {→2}, and {→3}. The first is then applied to the other two. All three are defined for all indices, and so an infinite number of calls to + are set up, all of the form `(+ 2 3)`. (Kindly ignore for now the pragmatic and semantic difficulties of actually performing an infinite number of function calls, especially in a language with side effects. We will return to these difficulties in section 6.) All of these calls produce the result 5, and so the result is {→5}. As for `α(+ 2 3)`, we have not yet defined what "α" does when written before an arbitrary form such as (in this case) a function call, but it is tempting to define it simply to evaluate the form and then produce a constant xapping with the resulting value. If we adopt this definition, then `α(+ 2 3)` also produces {→5}, and in fact we have an important syntactic property:

"α" distributes over function calls.

Suppose that we want to add 32 to every element of a xapping c; we may write `(α+ c α32)`. Now suppose instead that we wish to multiply each element of c by 9/5 before adding 32; we write `(α+ (α* c α9/5) α32)`. Or perhaps we really want a xapping of 2-lists pairing each such computed value with the original element of c: `(αlist c (α+ (α* c α9/5) α32))`. As we construct ever more complicated expressions to be executed independently and in parallel, we find ever more apply-to-all operators creeping in. The distribution rule can be used to "factor out" these operators if *every* subform of a function call has a preceding α, but that is not the case here. We solve this problem by introducing "•" as an "inverse" to "α": by definition, $\alpha{\bullet}x \equiv x$. We can then always apply the distribution law by introducing occurrences of "α•" first. To continue our example, we begin with the expression `(αlist c (α+ (α* c α9/5) α32))` and make successive transformations:

```
 (αlist  c  (α+  (α*   c α9/5) α32)) ≡
 (αlist α•c (α+  (α* α•c α9/5) α32)) ≡
 (αlist α•c (α+ α( *  •c  9/5) α32)) ≡
 (αlist α•c α( +  ( *  •c  9/5)  32)) ≡
α( list  •c  ( +  ( *  •c  9/5)  32))
```

and derive the result `α(list •c (+ (* •c 9/5) 32))`.

We have ended up with a notation for fine-grained parallism that is similar to the familiar Common Lisp backquote notation. One may think of a backquote as meaning "make a copy of the following data structure" and of a comma as meaning "except don't copy the following expression, but instead use its value." Likewise think of "α" as meaning "perform many copies of the following code in parallel" and of "•" as meaning "except don't do the following expression in parallel, but use elements of its value (which must be a xapping)."

The template that follows a backquote indicates parts of the constructed data structure that are the same in all instances constructed by the backquoted expression, and commas indicate values that can vary from instance to instance (in time). Similarly, the template that follows an α indicates parts of the computation that are the same in all the parallel computations, and each • indicates a value that can vary from instance to instance (in space).

This notation is powerful because it allows two simultaneous points of view (as with a Necker cube). On the one hand, it can be understood as a computation with a single thread of control, operating on arrays of data. This allows one to have a global understanding of how the data is transformed, as in FP or APL. On the other hand, it can be understood as an array of processes, with each process executing the same code that follows the "α" and with "•" flagging data values that may differ among processes. This allows one to take a piece of code written for a single processor and trivially change it to operate on a processor array by annotating it with "α" and "•" in a few places. Thus the notation simultaneously supports both macroscopic and microscopic views of a parallel computation.

4 Other Useful Operations

In this section we discuss various operations on xappings, some primitive (for the purposes of this paper) and some derived. A number of programming examples are presented. Many of the examples are reminiscent of APL programming style; we point out important similarities and differences along the way.

4.1 Common Lisp Sequence Functions

Xappings are just another kind of sequence (in the Common Lisp sense). Connection Machine Lisp extends the meaning of many Common Lisp functions to operate on xappings. For example, the `length` function will return the number of pairs in a finite xapping. Many important operations on xappings, such as selection, filtering, and sorting, may be performed using ordinary Common Lisp sequence functions:

```
(subseq '[the slick brown quux jumped
          over the lazy frog]
  3 6)
  ⇒ [quux jumped over]
```

```
(remove-if-not
  #'atom
  '{ 3→(prime odd)
     6→(composite even perfect)
     e→(transcendental has-pretty-continued-fraction)
     57→boring
     pi→(transcendental has-ugly-continued-fraction)
     25→odd } )
  ⇒ {57→boring 25→odd}
```

```
(sort '{sky→blue banana→yellow apple→red
        grass→green shirt→plaid}
      #'string-lessp)
  ⇒ [blue green plaid red yellow]
```

Many of these functions, such as subseq, depend on the argument sequences to be ordered, and so are sensible only if applied to a xector. Some of the functions will still work if applied to any other kind of xapping, and will operate by first implicitly ordering the xapping (indeed, the explicit purpose of sort is to order a sequence!). Yet other operations do not require an argument to be ordered, and never implicitly order the xapping; remove-if-not is an example of this. For other functions it is considered an error for a xapping argument not to be a xector. (Unfortunately, it appears to be necessary to determine case by case which of these is the most useful behavior.)

4.2 Domain, Enumeration, and Union

The domain function takes any xapping and returns a xet of the indices.

```
(domain '{sky→blue grass→green apple→red})
  ⇒ {sky grass apple}
```

The enumerate function takes a xapping and constructs a new xapping with the same domain but with consecutive integers starting from zero as values. The net effect is to impose an (arbitrary) ordering on the domain. Enumerating the same xapping twice might produce two different results.

```
(enumerate '{sky→blue grass→green apple→red})
  ⇒ {sky→0 grass→1 apple→2}
  or {sky→1 grass→0 apple→2}
  or {sky→1 grass→2 apple→0}
  or {sky→2 grass→1 apple→0}
  or {sky→2 grass→0 apple→1}
  or {sky→0 grass→2 apple→1}
```

The xector and xet functions are like list, in that they take any number of arguments and construct a xector or xet. Note that xet must eliminate duplicate elements.

```
(xector 'red 'blue 'green 'blue 'yellow 'red)
  ⇒ [red blue green blue yellow red]
```

```
(xet 'red 'blue 'green 'blue 'yellow 'red)
  ⇒ {blue green red yellow}
```

The function xunion takes a combining function and two xappings; the result is a xapping that is the union of the sets of pairs of the argument xappings. The combining function is used to combine values for which the same index appears in both argument xappings (and furthermore xunion guarantees that the first argument to the combining function comes from the first xapping, and the second argument from the second xapping).

```
(xunion #'+ '{alburt→0 dzindzichashvili→1
              nardandrea→0 spassky→2}
            '{alburt→0 seirawan→1
              spassky→1 rachels→0})
  ⇒ {alburt→0 dzindzichashvili→1 nardandrea→0
     rachels→0 seirawan→1 spassky→3}
```

It is not necessary to have a function called xintersection, because (xintersection f x y) $\equiv$ (αf x y); all function calls implicitly perform an intersection operation.

Using xunion we can define some composition operations (whose names are taken from [18], where the operations are used to compose images):

```
(defun over (a b) (xunion #'(lambda (x y) x) a b))
```

```
(defun in (a b) (α(lambda (x y) x) a b))
```

```
(defun atop (a b) (in (over a b) b))
```

The result of (over x y) is the union of x and y, as if they were both laid on a table with x over y, so that where x and y are both defined the element from x is always taken.

```
(over '{1→a 3→b 5→c 7→d} '[% # @ + = *])
  ⇒ {0→% 1→a 2→@ 3→b 4→= 5→c 7→d}
```

The result is not printed as a xector because there is no element with index 6.

The result of (in x y) is the intersection of x and y, with values taken from x; that is, the domain of y serves as a mask on the domain of x.

```
(in '{1→a 3→b 5→c 7→d} '[% # @ + = *])
  ⇒ {1→a 3→b 5→c}
```

The result of (atop x y) has the same domain as y, but the values are taken from x for indices that appear in x, and otherwise from y:

```
(atop '{1→a 3→b 5→c 7→d} '[% # @ + = *])
  ⇒ [% a @ b = c]
```

4.3 Reduction and Combining

The α syntax can be used, in effect, to replicate or broadcast data (constants and values of variables) and to operate in parallel on data (by applying a xapping of functions). Another syntax, using the "β" character, is used to express the gathering up of parallel data to produce a single result, and to express the permuting and multiple-result combining of parallel data.

For gathering up, the expression (βf x) takes a binary function f and a xapping x and returns the result of combining all the values of x using f, a process sometimes called *reduction* of a vector.[2] (This operation is written as f/x in FP and APL.)

As an example, (β+ foo) produces the sum of all the values in foo, and (βmax foo) returns the largest value in foo. Note that in this case the indices associated with the values do not affect the result. Any binary combining function may be used, but the result is unpredictable if the function is not associative and commutative, because the manner in which the values are combined is not predictable. For example, (βf '{1 2 3}) might compute (f 1 (f 2 3)) or (f (f 1 2) 3) or (f (f 3 1) 2) or any other method of arranging the values into a binary computation tree. (If the argument xapping is a xector, then the result is predictable up to associativity: the combining function need not be commutative, but should be associative.) Note that in APL and FP the order in which elements are combined is completely predictable. We eliminate predictability in Connection Machine Lisp for two reasons: first, the domain may be unordered; second, we wish to perform reductions in logarithmic time rather than linear time by using multiple processors.

Sometimes this unpredictability doesn't matter even though the function is not associative or commutative: the expression (β(lambda (x y) y) foo) is a standard way to choose a single value from foo without knowing any of the indices of foo. This operation, though not logically primitive, is so useful that it has a standard name choice:

```
(defun choice (xapping)
  (β(lambda (x y) y) xapping))
```

Note that the combining function (lambda (x y) y) is not commutative. However, also note that executing the same expression twice might result in choosing the same element twice or two different elements. The choice is arbitrary; it might or might not be random. The expression (choice 'a b) might return a the first ten million times it is executed and then might return b thereafter. Or it might always return a.

If a matrix is represented as a xector of row-xectors, then we can perform reduction over rows or over columns by using β and α together:

```
(αβ+ '[[1 2 3] [4 5 6] [7 8 9]])        ;Sum over rows
  ⇒ [6 15 24]

(βα+ '[[1 2 3] [4 5 6] [7 8 9]])        ;Sum over columns
  ⇒ [12 15 18]
```

For a three-dimensional array (represented by nested xectors), the reduction functions indicated in APL by +/[0], +/[1], and +/[2] are written in Connection Machine Lisp as $\beta\alpha\alpha$+, $\alpha\beta\alpha$+, and $\alpha\alpha\beta$+.

For permuting data, the expression (βf d x) takes a binary function f and two xappings d and x and returns a new xapping z whose indices are specified by the values of d and whose values are specified by the values of x. To be more precise, the value of (βf d x) is

$$\{q \rightarrow s \mid S = \{r \mid (p \rightarrow q \in d) \wedge (p \rightarrow r \in x)\} \wedge |S| > 0 \wedge s = (\beta f\ S)\}$$

[2] We use the characters →, •, α, and β knowing full well that a portable implementation cannot use them (although a nonportable implementation on the Symbolics 3600 does use them). We have experimented with using !, !, ?, and $, respectively, to replace them (thereby burdening "!" with two purposes), but we find this aesthetically displeasing, and have found no better alternative that is portable. We will eventually have to find another syntax, not only for reasons of portability, but because of an additional, unanticipated problem with the use of "β": users have taken to referring to the process of computing the sum (or maximum, or whatever) over a xapping as "beta-reduction" of the xapping—but that term already has a very different and long-established meaning within the Lisp community!

For every distinct value q in d there will be a pair $q \rightarrow s$ in the result. If that value q occurs in more than one pair of d, then s is the result of combining all of the corresponding values from x. As an example,

```
(β+ '{grass→green sky→blue banana→yellow
      apple→red tomato→red egg→white          }
    '{grass→1     sky→2     banana→3
      apple→4   tomato→5              mango→6})
  ⇒ {green→1 blue→2 yellow→3 red→9}
```

The pair red→9 appears because the values 4 from apple and 5 from tomato were summed by the combining function +. The result has no pair with index white or value 5 because neither egg nor mango appeared as an index in *both* operand xappings.

Histogramming is a useful application of this more general form of β; many sums must be computed by counting 1 for each contributor:

```
(defun histogram (x) (β+ x '{→1}))

(histogram '[a b a c e f b c d f e b a b g d e d a])
  ⇒ {A→4 B→4 C→2 D→3 E→3 F→2 G→1}
```

This version of the β-syntax may be understood operationally as a very general kind of interprocessor communication. If we think of a fine-grained multiprocessor where a processor is assigned to every element of a xapping, then the index of a xapping pair is a processor label. If for some p there is a pair $p \rightarrow q$ in d and a pair $p \rightarrow r$ in x, it means that the processor labeled p has a datum r and a pointer to another processor q. The βf operation sends the datum r to processor q. Of course, many such messages may be sent in parallel; the combining function f is used to resolve any message collisions at each destination. (This idea of a combining function that resolves collisions recurs in many places throughout the language, almost everywhere parallel data movement is involved. Recall, for example, the function xunion presented above in section 4.2.)

Note the following similarity between the two forms of β syntax: (βf x) $\Rightarrow$ v if and only if (βf '{→q} x) $\Rightarrow$ {$q \rightarrow v$}. Operationally speaking, (βf '{→q} x) causes all the values of x to be sent to processor q and combined there, whereas (βf x) causes all the values of x to be sent to the host (or to a neutral corner, if you please) and combined there. It is this relationship that prompted us to use β for both purposes.

4.4 Other Functional Operations

Early in the development of Connection Machine Lisp there was a raft of other functional operators as well, which consumed a fair portion of the Greek alphabet. We have found that "α" and "β" along with the domain, enumerate, xunion and a very few other functions seem to constitute a comfortable set of primitives from which many other parallel operations are easily constructed. This is of course similar to the experience of the APL community, except that we have settled on a different set of primitives.

Consider for example the function order that takes any xapping and imposes an ordering on its values (thereby producing an equivalent xector):

```
(defun order (x) (β@ (enumerate x) x))
```

(The binary function @ merely signals an error when it is called:

```
(defun @ (&rest x)
  (error "Combining function erroneously called ~
         ~@[ on argument list ~S~]"
         x))
```

It is conventionally used to indicate that collisions should not occur.) Compare order, which produces an arbitrary ordering (that is, an ordering at the discretion of the implementation) to sort, which imposes an ordering according to a user-specified binary ordering predicate.

A useful operation defined in terms of β is transport, which uses a unary Lisp function to compute new indices from old ones:

```
(defun transport (f g x)
  (βf α(g •(domain x)) x))
```

(Actually, this definition is not correct. To render transport into legitimate Common Lisp syntax, we must first explain that (βf *d x*) ≡ (combine #'f *d x*), and then write

```
(defun transport (f g x)
  (combine f α(funcall g •(domain x)) x))
```

Similarly, (βf *x*) ≡ (reduce #'f *x*). The fact that Common Lisp, like most Lisp dialects, has separate function and variable namespaces makes the use of functional arguments awkward. All remaining code in this paper adheres to Common Lisp syntax.)

Here are some examples of the use of transport.

```
(defun double (x) (+ x x))

(transport #'@ #'double '[a b c d])
   ⇒ {0→a 2→b 4→c 6→d}
```

Compare this to a similar use of the APL expansion function:

```
      (7ρ1  0)\3  4  5  6
3  0  4  0  5  0  6
```

Whereas APL requires vector elements to have contiguous indices, and must therefore pad the expansion with zeroes, Connection Machine Lisp allows a xapping simply to have holes.

The inverse of the index-doubling operation shown just above is a contraction operation with combining:

```
(defun halve (x) (floor x 2))

(transport #'+ #'halve '[1 2 3 4 5 6]) ⇒ [3 7 11]
```

In this example, every element in [1 2 3 4 5 6] is transported to an index equal to half (rounded down) of its original index. The net effect is to combine adjacent pairs. This effect is rather more difficult to achieve in standard APL.

```
(defun rotate (x j)
  (transport #'@
             #'(lambda (k) (mod (- k j) (length x)))
             x))

(defun shift (x j)
  (transport #'@ #'(lambda (k) (- k j)) x))

(rotate '[a b c d e f] 2) ⇒ [c d e f a b]

(shift '[a b c d e f] 2)
   ⇒ {-2→a -1→b 0→c 1→d 2→e 3→f}
```

APL provides an function ⌽ equivalent in effect to rotate, but has nothing quite like shift, which can renumber the indices of a vector so as to begin at any origin, even a negative origin. Such shifted vectors are handy on occasion, as in the definition of scan presented further below.

The function inverse computes the inverse of a xapping considered as a mapping; for every pair $p \rightarrow q$ in the argument, a pair $q \rightarrow p$ appears in the result. As usual, a combining function must be provided against the possibility of duplicate values q.

```
(defun inverse (f x)
  (combine f x (domain x)))    ;i. e., (βf x (domain x))

(inverse #'@ '[a b c d]) ⇒ {a→0 b→1 c→2 d→3}

(inverse #'+ '[a b c b]) ⇒ {a→0 b→4 c→2}
(inverse #'max '[a b c b]) ⇒ {a→0 b→3 c→2}
(inverse #'min '[a b c b]) ⇒ {a→0 b→1 c→2}
```

The function compose computes composition of two xappings considered as mappings; for every pair $p \rightarrow q$ in the first argument, there must be a pair $q \rightarrow r$ in the second argument, and the pair $p \rightarrow r$ appears in the result.

```
(defun compose (x y) α(xref y •x))

(compose '{p→x q→y r→x s→w} '{x→3 y→4 z→5 w→6})
   ⇒ {p→3 q→4 r→3 s→6}
```

As in APL, it is helpful to have a primitive operator iota to generate xectors of a given length.

```
(iota 10) ⇒ [0 1 2 3 4 5 6 7 8 9]
```

Note that [0 1 2 3 4 5 6 7 8 9] is the same data structure as {0 1 2 3 4 5 6 7 8 9}.

4.5 Examples Using Matrices

We take as primitive the transpose operation. If x is a xapping whose values are all xappings, then (transpose x) is also a xapping whose values are xappings, and (transpose x) contains a pair $p \rightarrow y$ where xapping y contains a pair $q \rightarrow r$ just in case x contains a pair $q \rightarrow z$ where xapping z contains a pair $p \rightarrow r$. To put it another way,

```
(xref (xref (transpose z) j) k) ≡ (xref (xref z k) j)
```

for all j and k, and if one side of the equivalence is undefined then so is the other side. Note also that

```
(transpose (transpose x)) ≡ x
```

as one might expect.

Examples of using transpose:

```
(transpose '[[1 2 3]
             [4 5 6]])
   ⇒ [[1 4]
       [2 5]
       [3 6]]

(transpose '[[0 1 2 3]
             [4 5 6]
             [7 8]
             [9]])
   ⇒ [[0 4 7 9]
       [1 5 8]
       [2 6]
       [3]]

(transpose '{a→[1 2] b→[3 4 5] c→{0→6 2→7}})
   ⇒ [{A→1 B→3 C→6} {A→2 B→4} {B→5 C→7}]
```

```
(transpose '{0→{4→14} 1→{1→67 3→23} 3→{1→89}
             4→{0→93 3→35 4→56}})
   ⇒ {0→{4→93} 1→{1→67 3→89} 3→{1→23 4→35}
      4→{0→14 4→56}}
```

Note that the subxappings of the argument need not all have the same domain; the argument and result may be "ragged" (or even sparse) matrices. The last result above might be expressed as

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 14 \\ 0 & 67 & 0 & 23 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 89 & 0 & 0 & 0 \\ 93 & 0 & 0 & 35 & 56 \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 93 \\ 0 & 67 & 0 & 89 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 23 & 0 & 0 & 35 \\ 14 & 0 & 0 & 0 & 56 \end{bmatrix}$$

in mathematical notation, but note that the xappings shown in that example do not represent the zero entries explicitly. The correspondence between the dense and sparse representations may be seen by properly formatting the sparse one:

```
{ 0→{                                  4→93 }
  1→{          1→67          3→89           }

  3→{          1→23                    4→35 }
  4→{ 0→14                             4→56 } }
```

The support of sparse arrays is one of the strengths and conveniences of the xapping data structure.

The function `inner-product` takes two functions f and g and two xappings p and q and computes an inner product. This would be written $p\ f.g\ q$ in APL, $+ . \times$ being, for example, the standard vector inner product.

```
(defun inner-product (f g &optional (p nil pp) q)
  (if (not pp)
      #'(lambda (p q) (inner-product f g p q))
      (reduce f α(funcall g •p •q))))

(funcall (inner-product #'+ #'*) '[1 2 3] '[4 5 6])
   ⇒ 32

(inner-product #'max #'min '[5 2 7] '[2 8 5]) ⇒ 5
```

Note that `inner-product` is written so as to allow optional currying. One may supply all four arguments at once, or supply only two and get back a closure that will accept the other two arguments and then perform the operation. This technique avoids some of the awkwardness of notation that would otherwise be required when using functional arguments and values in a Common Lisp framework.[5]

The function `outer-product` takes one n-ary function f and n xappings, where $n > 0$, and computes an outer product; for binary f this would be written $p \circ . f\ q$ in APL. (The definition of `outer-product` is also written so as to allow optional currying.)

[5] This trick is so useful that we briefly considered introducing a lambda-list keyword `&curry` so that we could write simply

```
(defun inner-product (f g &curry p q)
  (reduce f α(funcall g •p •q)))
```

but then we thought better of it.

```
(defun outer-product (f &rest args)
  (labels ((op (list-of-xappings list-of-args)
             (if (null list-of-xappings)
                 (apply f list-of-args)
                 α(op (cdr list-of-xappings)
                      (cons •(car list-of-xappings)
                            list-of-args)))))
    (if (null args)
        #'(lambda (&rest args) (op args '()))
        (op args '()))))

(outer-product #'+ (iota 5) (iota 6)) ⇒
   [[0 1 2 3 4 5]
    [1 2 3 4 5 6]
    [2 3 4 5 6 7]
    [3 4 5 6 7 8]
    [4 5 6 7 8 9]]

(outer-product #'substitute
               '[p q]
               '[a b]
               '[(c a b) (b a b)])
   ⇒ [[[(C P B) (B P B)]
       [(C A P) (P A P)]]
      [[(C Q B) (B Q B)]
       [(C A Q) (Q A Q)]]]
```

Here we have a standard matrix multiplication example; compare this to the definition in FP given by Backus [1]. A matrix is represented as a xector of xectors; the second argument is transposed, and then the two matrices are combined by an outer product that uses an inner product as the operation.

```
(defun matrix-multiply (f g &optional (x nil xp) y)
  (if (not xp)
      #'(lambda (x y) (matrix-multiply f g x y))
      (outer-product (inner-product f g)
                     x
                     (transpose y))))

(matrix-multiply #'+ #'*
                 '[[1 2] [3 4] [5 6]]
                 '[[0 1 0 1] [1 1 1 0]])
   ⇒ [[2 3 2 1] [4 7 4 3] [6 11 6 5]]
```

(This result would be expressed as

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 2 & 1 \\ 4 & 7 & 4 & 3 \\ 6 & 11 & 6 & 5 \end{bmatrix}$$

in mathematical notation.)

4.6 A Large Example: Scan

The scan operation, written $f \backslash x$ in APL, computes a vector of the f-reductions over all prefixes of a vector x. (This operation is also referred to as a "parallel f-prefix" computation.) In Connection Machine Lisp, using xectors, we have the examples

```
(scan #'+ '[1 2 3 4 7 2]) ⇒ [1 3 6 10 17 19]

(scan #'* '[1 2 3 4 5 6]) ⇒ [1 2 6 24 120 720]

(scan #'max '[1 6 2 7 3 4]) ⇒ [1 6 6 7 7 7]
```

This operation can be implemented so as to execute in time logarithmic in the number of β-permutation operations (with no combining needed), `iota` operations, and αf operations, as follows:

```
(defun scan (f x)
  (if (xectorp x)
      (scan1 f x)
      (let ((q (enumerate x)))
        (β@ (inverse #'@ q) (scan1 f (β@ q x))))))

(defun scan1 (f x)
  (do ((j 1 (+ j j))
       (z x (over α(funcall f
                              •(shift (in z (iota
                                              (- n j)))
                                      (- j))
                              •z)
                  z))
       (n (length x)))
      ((>= j n) z)))
```

Here `scan1` is the central part of the algorithm; it takes a xector and computes an APL-style scan in a number of iterations logarithmic in the length of the xector. (It should be noted that arguably any implementation of the `shift` or `iota` operation might require time logarithmic in the length of the shifted or generated xector, for an overall time complexity of $O(\log^2 n)$.) The result of the call to `in` is the first $n - j$ elements of `z`; these are shifted so as to align with the last $n - j$ elements of `z` for combining. The results replace the last $n - j$ elements of `z` (via the function `over`).

The main function `scan` takes care of handling arbitrary xappings; a non-xector is enumerated to impose an ordering q, the resulting xector is scanned, and the scan results are then projected back onto the original domain.

As an example of the power of the scan operation in a nonnumerical application, consider the problem of lexing, that is, dividing a character string into lexical tokens. The essential work of this can be done in time polylogarithmic in the length of the string. Lexing is normally performed by a finite-state automaton that makes transitions as it reads the characters sequentially from left to right, and the states of the automaton indicate token boundaries. One can view a character (or a single-character string) as a function that maps an automaton state into another state; taking string concatenation to be isomorphic to composition of these functions, a string may therefore also be viewed as such a function. We can represent state-to-state functions as xappings, and their composition by the `compose` function. Therefore a single scan operation using the `compose` function can compute the mapping corresponding to each prefix of a source text; applying each such mapping to the start state yields the state of the automaton after each character of the input.

Let us consider a simple example where a token may be a sequence of alphabetic characters, a string surrounded by double quotes (where an embedded double quote is represented by two consecutive double quotes), or any of `+`, `-`, `*`, `=`, `<`, `>`, `<=`, and `>=`. Spaces, tabs, and newlines delimit tokens but are not part of any token.

Our automaton will have nine states:

`n` means the last character processed is not part of a token. This is the initial state.

`a` means the character is the first in an alphabetic token.

`z` means the character is in an alphabetic token but not first.

`<` means the character is `<` or `>`.

`=` means the character is the `=` in a `<=` or `>=` token.

`*` means the character is `+`, `-`, `*`, or an `=` that is not part of a `<=` or `>=` token.

`q` means the character is the `"` that begins a string.

`s` means the character is part of a string.

`e` means the character may be the `"` that ends a string, unless the next character also is `"` in which case the string continues.

(There is no reason why the states could not have multicharacter names other than conciseness of presentation.)

Characters are transformed into xappings by the following function:

```
(defun character-to-state-map (ch)
  (if (alpha-char-p ch)
      '{n→a a→z z→z <→a =→a *→a q→s s→s e→a}
      (ecase ch
        ((#\+ #\- #\*)
         '{n→* a→* z→* <→* =→* *→* q→s s→s e→*})
        ((#\< #\>)
         '{n→< a→< z→< <→< =→< *→< q→s s→s e→<})
        ((#\=)
         '{n→* a→* z→* <→= =→* *→* q→s s→s e→*})
        ((#\")
         '{n→q a→q z→q <→q =→q *→q q→e s→e e→s})
        ((#\Space #\Newline #\Tab)
         '{n→n a→n z→n <→n =→n *→n q→s s→s e→n})
        )))
```

This function computes the state of the automaton after every character of an input string (represented as a xector of character objects):

```
(defun compute-all-states (x)
  α(xref •(scan #'compose α(character-to-state-map •x))
         'n))
```

The function `lex` takes an input string and returns a xector of xectors; each subxector contains the characters for one token. The first subxector is always empty.

```
(defun lex (x)
  (let* ((s (compute-all-states x))
         (first α(not (not (member •s '(a < * q)))))
         (nums (scan #'+ α(if •first 1 0)))
         (masked-nums α(if (eq •s 'n) 0 •nums))
         (lengths (over '[0] (histogram masked-nums)))
         (origins (inverse
                    #'(lambda (x y) 0)
                    α(if •first •masked-nums 0))))
    α(subseq x •origins (+ •origins •lengths))))
```

The bulk of the work is done by `compute-all-states` and is nonnumerical in nature. A little bit of numerical trickery involving a sum-scan and a histogram is used to perform the actual chopping of the string.

As an example of the operation of `lex`, let us consider how the input string `"foo + "a+b"<=bar"` would be processed. This string is rendered as a xector as follows:

```
[#\f #\o #\o #\Space #\+ #\Space #\" #\a #\+
 #\b #\" #\< #\= #\b #\a #\r]
```

The intermediate variables computed by `lex` have these values in the example:

```
s ⇒ [A Z Z N * N Q S S S E < = A Z Z]

first ⇒ [T NIL NIL NIL T NIL T NIL NIL
          NIL NIL T NIL T NIL NIL]

nums ⇒ [1 1 1 1 2 2 3 3 3 3 3 4 4 5 5 5]

masked-nums ⇒ [1 1 1 0 2 0 3 3 3 3 3 4 4 5 5 5]

lengths ⇒ [0 3 1 5 2 3]

origins ⇒ [0 0 4 6 11 13]
```

In `masked-nums`, entries with the same value correspond to characters belonging to the same token, except that zero values indicate whitespace belonging to no token. Note the use of `over` to replace the first element of `lengths` with zero, and the use of a constant function with `inverse` to force the first element of `origins` *to* be zero.

The final computed value is:

```
[{} [#\f #\o #\o] [#\+] [#\" #\a #\+ #\b #\"]
 [#\< #\=] [#\b #\a #\r]]
```

If you believe that the implementation actually executes `lex` in $O(\log^2 n)$ time, then lexing a megabyte of text should take about four times as long as lexing a kilobyte of text (assuming that enough processing resources are available to take advantage of the parallelism specified in the algorithm.)

5 A Metacircular Interpreter

Table I presents a metacircular interpreter for a subset of Connection Machine Lisp. It is similar in style to published interpreters for Scheme [28,26] but differs in three respects. First, it uses such Common Lisp constructs as `case` and `etypecase` to discriminate forms. Second, like Common Lisp, it maintains the distinction between ordinary forms (including ordinary variables) and functional forms (including names of functions). Third, it allows for the body of a `lambda` expression to be an implicit progn, which Common Lisp also allows and some interpreters of Scheme do not.

Much of the machinery will be familiar to those who know Scheme. The function `eval` takes an expression and a lexical environment, as usual, but also takes a list of indices whose purpose is explained below. Numbers and strings are treated as self-evaluating. Ordinary variables are looked up in a lexical environment structure, here represented as an association list in the time-honored fashion. The following non-atomic forms are distinguished:

- `(QUOTE` x`)` evaluates to x.
- `(FUNCTION` f`)` evaluates f as a functional form by calling `fneval`.
- `(IF` p x y`)`, as usual, first evaluates p; then one of x and y is evaluated depending on whether the value of p was non-`nil` or `nil`.
- `(ALPHA` x`)` represents the construction αx. Briefly, this causes many evaluations of of x, one for every possible index. (Difficulties with this idea are discussed below.) Each of these many calls to `eval` gets a different `indices` argument, obtained by consing the index for that call to `eval` onto the previous list of indices. Note the use of a universal xapping $\{\rightarrow\}$ to allow each of the parallel calls to obtain its own index.
- `(BULLET` x`)` represents the construction $\bullet x$. The `indices` list must contain one entry for each α that is dynamically controlling the current call to `eval`. The first entry in the list corresponds to the innermost α, which is the one that the bullet of this expression should cancel. Therefore the expression x is evaluated (it must produce a xapping) using the rest of the indices, and then the first index is used to select an element of the resulting xapping.
- Any other list is a function call. The first element is evaluated as a functional form, `evlis` is called in the usual way to evaluate the argument forms, and then the function is applied to the arguments.

The function `fneval` constructs closures from `lambda` expressions. Symbols are treated as primitive operators (this suffices to allow the interpreter to execute properly in Common Lisp).

The function `apply` processes closures in the usual manner, adding parameter/argument pairs onto the environment of closure before evaluating the body of the `lambda` expression. The function `evprogn` handles the evaluation of the body. When the function is a xapping, then many applications must be performed. The function `list-xapping-transpose` merely takes a list of argument xappings and produces its transpose, a xapping of lists.

This interpreter is not satisfactory for a number of reasons. One is that the form x in `(BULLET` x`)` is evaluated many times, once for every index being processed by the corresponding α. This shouldn't matter in a pure theory, but we are interested in explicating side effects within the language, and prefer a semantics where x is evaluated only once (at least as far as the corresponding α is concerned; additional occurrences of α surrounding it might cause repeated evaluations).

A second objection is that when a form `(ALPHA` x`)` is processed an apparently infinite number of recursive calls to `eval` are performed, one for every possible index (meaning every possible Lisp object!), for there is nothing in the syntax of the call to limit it. Theoretical objections aside, this is difficult to implement.

The most telling objection, however, is that this interpreter explains α in terms of itself in such a way that one cannot tell, just to look at the code of the interpreter, whether or not α actually processes anything in parallel. This is similar to the difficulties noted by Reynolds [19] for any interpreter that defines a construct in terms of itself.

In the next section we discuss a number of difficult semantic problems and then present a second metacircular interpreter that avoids defining α in terms of itself and also explicates some issues of parallelism and synchrony.

6 Conditionals, Closures, Infinities, Side Effects, and Other Hard Things

In section 3 we arrived at a clear operational idea of what α*form* ought to mean: lots of processors should each execute the same *form*, and "$\bullet$" indicates where each must use its own data (or drop out if it has no data in that xapping). What then are we to make of "α`(if` p x y`)`"? Each processor should execute the predicate p; those that compute a true result should execute x, and those that compute a false result should execute y.

```
(defun eval (exp env indices)
  (etypecase exp
    ((OR NUMBER STRING) exp)
    (SYMBOL (lookup exp env))
    (CONS (case (car exp)
            (QUOTE (cadr exp))
            (FUNCTION (fneval (cadr exp) env indices))
            (IF (if (eval (cadr exp) env indices)
                    (eval (caddr exp) env indices)
                    (eval (cadddr exp) env indices)))
            (ALPHA α(eval (cadr exp) env (cons •'{→} indices)))
            (BULLET
              (if indices
                  (xref (eval (cadr exp) env (cdr indices)) (car indices))
                  (error "Misplaced '•' before ~S" (cadr exp))))
            (t (apply (fneval (car exp) env indices)
                      (evlis (cdr exp) env indices)))))))

(defun lookup (exp env)
  (let ((pair (assoc exp env)))
    (if pair (cddr pair) (symbol-value exp))))

(defun fneval (fnexp env indices)
  (cond ((symbolp fnexp) fnexp)
        ((and (consp fnexp) (eq (car fnexp)  lambda))
         (make-closure fnexp env indices))
        (t (error "Bad functional form ~S" fnexp))))

(defun evlis (argforms env indices)
  (and argforms
       (cons (eval (car argforms) env indices)
             (evlis (cdr argforms) env indices))))

(defun apply (fn args)
  (etypecase fn
    (SYMBOL (apply fn args))
    (CLOSURE (evprogn (cddr (closure-function fn))
                      (pairlis (cadr (closure-function fn)) args (closure-env fn))
                      (closure-indices fn)))
    (XAPPING α(apply •fn •(list-xapping-transpose args '{→()})))))

(defun evprogn (body env indices)
  (cond ((null (cdr body)) (eval (car body) env indices))
        (t (eval (car body) env indices)
           (evprogn (cdr body) env indices))))

(defun list-xapping-transpose (list-of-xappings xapping-of-lists)
  (if (null list-of-xappings) xapping-of-lists
      (list-xapping-transpose (cdr list-of-xappings)
                              α(cons •(car list-of-xappings) •xapping-of-lists))))
```

Table 1. A Simple Metacircular Interpreter for Connection Machine Lisp

That is the microscopic view. Let us now map this back to the macroscopic view, in terms of xappings. The predicate p is first evaluated (many times, in effect, once for each index) to produce a xapping. Now if side effects were not an issue, we could simply specify that both x and y are also evaluated and then combined according to the truth values in p; `if` would then be a purely functional conditional. But side effects *are* an issue, and we want `if` to be the usual control construct, not a function. We find that `if` is best macroscopically explained by postulating that x is evaluated only for indices in which p has true values, and that y is evaluated only for indices in which p has false values. In other words, evaluation of an expression that is under the control of an α must be dependent on a *context* that is a set (or xet, if you wish) of "active indices."

Suppose that there are nested occurrences of α. Then in general the context must be a nested structure. We can let `t`, say, represent the global context where no α is controlling, and in the general case a context is a tree of uniform height composed of nested xappings. Suppose that the value of `rebus` is a familiar quatrain:

```
rebus ⇒ [[Y Y U R] [Y Y U B] [I C U R] [Y Y 4 ME]]
```
[4]

Then in the expression

```
αα(if (eq ••rebus 'y) 'q ••rebus)
   ⇒ [[Q Q U R] [Q Q U B] [I C U R] [Q Q 4 ME]]
```
[5]

the consequent expression `'q` is evaluated in the context

```
[[t t nil nil] [t t nil nil]
 [nil nil nil nil] [t t nil nil]]
```

and the alternative expression `••rebus` (the second occurrence) is evaluated in the context

```
[[nil nil t t] [nil nil t t]
 [t t t t] [nil nil t t]]
```

A context, therefore, may be understood to be exactly the places where the controlling predicates have succeeded or failed as necessary to enable execution of the current expression.

It is technically convenient to eliminate the occurrences of `nil` from contexts (recursively eliminating any resulting empty xappings as well). The contexts just exhibited would therefore actually appear as

```
{0→[t t] 1→[t t] 3→[t t]}
```

and

```
[{2→t 3→t} {2→t 3→t} [t t t t] {2→t 3→t}]
```

respectively. The more complex metacircular interpreter exhibited below will use contexts of this form to determine the indices for which to evaluate an expression. That will take care of the interaction between α and conditionals.

It gets worse. Consider the interaction of α with closures. What does

```
α(mapcar #'(lambda (x z) (list x •y z)) a •b)
```

mean? Taking the microscopic view, for every index we execute the computation

(`mapcar #'(lambda (x z) (list x` *y* `z)) a` *b*)

where y is the value for that index within `y`, and b within `b`. If

```
a ⇒ (three five)
y ⇒ [little blind]
b ⇒ [(kittens monkeys) (mice men)]
```

then the result should be

```
[[(three little kittens)
  (five little monkeys))
 ((three blind mice)
  (five blind men))]
```

But now let us take the macroscopic view:

```
α(mapcar #'(lambda (x z) (list x •y z)) a •b)
```

means the same as

```
(αmapcar α#'(lambda (x z) (list x •y z)) αa b)
```

Thus a xapping containing a zillion `mapcar` operations must be applied to three other xappings. The second is a constant xapping with the value of `a`, and the third is the xapping named by `b`. But what is the first xapping? Is it a constant xapping of closures? No, because each closure behaves slightly differently: each uses a different value from `y`, according to index! We conclude that "α" does not distribute over `(lambda ...)` in a simple manner; rather, a closure must close not only over free lexical variables but also implicitly over the current set of indices.

An intuitive way to understand this is the following technique for distributing "α" over lambda-expressions:

(`α(lambda (x y ...)` *body*`) ≡ (lambda (αx αy ...) α`*body*`)`

That is, a xapping of closures of the same lambda-expression can be understood to be a simple closure that accepts xappings as arguments and "destructures" them before executing the body in parallel. This can be made more formal:

```
α(mapcar #'(lambda (x y) (list x y •z)) a •b) ≡
(αmapcar α#'(lambda (x y) (list x y •z)) αa b) ≡
(αmapcar #'(lambda (αx αy) α(list x y •z)) αa b) ≡
(αmapcar #'(lambda (αx αy) (αlist αx αy z)) αa b) ≡
(αmapcar #'(lambda (qx qy) (αlist qx qy z)) αa b) ≡
(αmapcar #'(lambda (qx qy) α(list •qx •qy •z)) αa b)
```

where the penultimate step is merely a renaming of the "variables" αx and αy to be the otherwise unused names qx and qy, and the last step is a factoring back out of α. This mode of understanding is still only vaguely intuitive, because the result of `α#'(lambda ...)` must really after all be a xapping and not a closure. One might go a step further and specify that in a function call where the function is a constant xapping of `funcall` operations, any argument that is not a xapping may be a closure that takes xappings as arguments; in other words, we arrange to "transpose" the levels of closureness and xappingness.

The complex metacircular interpreter that we yet promise to show you (please have patience, Gentle Reader) allows exactly that sort of transposition. It turns out that an appropriate representation for closures has *four* components: the `lambda` expression, the lexical environment, the context, and the indices, all as of the point of closure. The context and indices information trade off against each other. A closure containing a context c that is a xapping (that is, anything except `t`) may regarded as representing a xapping whose values are closures whose context parts are the values in c. For concreteness assume that a closure is represented as a 5-list beginning with the symbol `closure`:

(`closure` *exp env context indices*)

Then we have the following identity:

[4] Too wise you are, / Too wise you be; / I see you are / Too wise for me.

[5] Two cues, your / Two queues, Eubie; / Icy ewer / Took youse for me.

```
(xref '(closure ,exp ,env ,context ,indices) k) ≡
'(closure ,exp ,env ,(xref context k) ,(cons k indices))
```

We will take this identity as the definition of a closure containing a context other than t. It can be used to convert such a closure into a xapping, and if the original context is nested the process can be iterated to produce a nested xapping whose structure will be the same as that of the original context and whose leaves will be closures whose contained context is t, that is, ordinary closures. Indeed, we officially modify the definition of xref as follows:

```
(defun xref (x k)
  (etypecase x
    (XAPPING (primitive-xref x k))
    (CLOSURE
      (make-closure
         (closure-exp x)
         (closure-env x)
         (primitive-xref (closure-context x) k)
         (cons k (closure-indices x))))))
```

The advantage of this dual representation, as we shall see in due course, is that it allows us to specify certain kinds of synchrony in the complex metacircular interpreter.

First, however, let us turn to the matter of infinite xappings. Suppose we were to write

α(+ (random 8) •x)

This seems clear enough; we wish to add a random number (from 0 to 7) to each element of the xapping x. But shall a single random number be chosen, and that result added to every element of x? Or shall there be distinct computations of random numbers for each element of x? Our distribution law states that the previous expression means the same as

(α+ (αrandom α8) x)

and this clearly calls for many instances of the random function to be applied to many instances of 8, thereby producing many random numbers to be added. But *how* many? There is no problem with the addition operation if x is finite; the rule about the intersection of domains in a function call causes only a finite number of calls to + to occur. But the code calls for an infinite number of calls to random to occur, one for every possible Lisp object. This is difficult to implement effectively. The reason is that random has a side effect. (If it did not, we could simply make one call to it and then effectively replicate the result. That is what we did earlier when we claimed that (α+ α2 α3) ⇒ {→5}.)

We have investigated two ways out of this problem. One is to use lazy xappings: in that case

(αrandom α8) ⇒ { . (lambda () (random 8))}

more or less. This approach has the disadvantage that side effects can occur out of order, sometimes unexpectedly late in the progress of the program. (This same problem can occur with futures in Multilisp [10]. Indeed, a lazy xapping is in effect merely a collection of futures, all of a particular form.) For instance, in the code fragment α(foo (bar)) we cannot guarantee that all side effects caused by calls to foo occur after all side effects caused by calls to bar. Nevertheless, we have implemented (on a single-processor system, the Symbolics 3600) an experimental version of Connection Machine Lisp with lazy xappings and have found it tremendously complicated to implement but useful in practice.

The other way out is to forbid infinite xappings. Unfortunately, a total ban on infinite xappings greatly restricts the utility of the α-notation. We could allow just constant xappings, but then side effects cannot be treated consistently (the example of α(random 8) could not be made to work, for example), and warts appear such as the domain function not being total.

We have found the following intermediate position tractable. We introduce two rules:

- One must not execute an infinite number of function calls or an infinite number of IF forms.
- An expression beginning with an explicit "α" must not produce an infinite xapping.

Violations of these restrictions be detected syntactically (by a compiler, for example). They allow infinite xappings to arise "virtually" in the notation, but an implementation can always arrange never to have to represent them explicitly. One consequence of these rules is that any function call or IF form within the control of a "α" must have as an immediate subform either a form preceded by "•" [basis step] or another function call or IF form [induction step]. Another consequence is that the distributivity of "α" over function calls is partly destroyed in practice.

The metacircular interpreter shown in Table 2 makes use of a special representation for constant xappings, but only for the sake of representing contexts. Infinite xappings can become visible to "user code" only if provided as part of the input to the interpreter.

The code in Table 2 takes advantage of the Common Lisp type specifier hierarchy in two ways. First, the type specifier (MEMBER T) specifies a type to which belongs the object t and no other object. Second, the type specifiers CONSTANT-XAPPING and FINITE-XAPPING represent subtypes of the type XAPPING. The function choice is assumed to operate properly on a constant xapping by returning the object that is the value of that xapping at every index.

The code in Table 2 is quite similar to that in Table 1. We remark here primarily on the differences.

The function eval of course takes an additional argument, context, which is the third argument and not the fourth for no good reason other than historical accident. An important invariant to understand is that the value returned by a call to eval will match the context argument in its overall structure; that is, it will be a copy of the context with suitable values substitituted for the occurrences of t.

The handling of numbers, strings, and quoted objects is a bit different in that the context must be taken into consideration. The function contextualize in effect makes a copy of the context, substituting the value for each occurrence of t; it thus replicates a value so as to match the context structure. For symbols, the function lookup performs a similar contextualization. (The strange maneuver involving the function lookup-contextualize is discussed below in conjunction with closures.)

The description of the processing of ALPHA forms no longer uses α in any essential way. A form (ALPHA x) is processed by evaluating the subform x in an extended context, one in which every occurrence of t in the current context has been replaced by {→t}, thereby increasing the height of the context tree by one. From this one can see that the topmost xapping in a nested context structure corresponds to the outermost controlling α, and a xapping that contains a t corresponds to the innermost α. The function extend-context performs the straightforward mechanics of context extension.

The processing of (BULLET x) becomes more complicated. If any indices are provided, then one is used as in Table 1. Other-

```
(defun eval (exp env context indices)
  (etypecase exp
    ((OR NUMBER STRING) (contextualize exp context))
    (SYMBOL (lookup exp env context))
    (CONS (case (car exp)
            (QUOTE (contextualize (cadr exp) context))
            (FUNCTION (fneval (cadr exp) env context indices))
            (ALPHA
              (eval (cadr exp) env (extend-context context) indices))
            (BULLET
              (if indices
                  (xref (eval (cadr exp) env context (cdr indices)) (car indices))
                  (context-filter (eval (cadr exp) env (trim-context context) nil)
                                  context)))
            (IF
              (let ((test (eval (cadr exp) env context indices)))
                (let ((truecontext (ifpart t test))
                      (falsecontext (ifpart nil test)))
                  (if truecontext
                      (if falsecontext
                          (merge-results (eval (caddr exp) env truecontext indices)
                                         (eval (cadddr exp) env falsecontext indices))
                          (eval (caddr exp) env truecontext indices))
                      (if falsecontext
                          (eval (cadddr exp) env falsecontext indices)
                          (error "Internal error: failed context split for ~S" exp))))))
            (t (apply (fneval (car exp) env context indices)
                      (evlis (cdr exp) env context indices)))))))

(defun fneval (fnexp env context indices)
  (cond ((symbolp fnexp) (contextualize fnexp context))
        ((and (consp fnexp) (eq (car fnexp) 'lambda))
         (make-closure fnexp env context indices))
        (t (error "Bad functional form ~S" fnexp))))

(defun evlis (argforms env context indices)
  (and argforms
       (cons (eval (car argforms) env context indices)
             (evlis (cdr argforms) env context indices))))

(defun apply (fn args)
  (etypecase fn
    (SYMBOL (apply fn args))
    (CLOSURE
      (evprogn (cddr (closure-exp fn))
               (let ((h (context-height (closure-context fn))))
                 (pairlis (cadr (closure-exp fn))
                          (mapcar #'(lambda (a) (cons h a)) args)
                          (closure-env fn)))
               (closure-context fn)
               (closure-indices fn)))
    (XAPPING
      (let ((index-set (get-finite-context fn args)))
        (construct-xapping index-set
                           (aplis #'(lambda (x)
                                      (apply (xref fn x)
                                             (mapcar #'(lambda (a) (xref a x)) args)))
                                  index-set))))))
```

Table 2. A Complex and Not Quite So Metacircular Interpreter for Connection Machine Lisp

```
(defun evprogn (body env context indices)
  (cond ((null (cdr body)) (eval (car body) env context indices))
        (t (eval (car body) env context indices)
           (evprogn (cdr body) env context indices))))

(defun aplis (fn index-set)
  (and index-set
       (cons (funcall fn (car index-set))        ;No parallelism is shown here.
             (aplis fn (cdr index-set)))))       ;See the text for a discussion.

(defun contextualize (value context)
  (etypecase context
    ((MEMBER T) value)
    (CONSTANT-XAPPING (make-constant-xapping (contextualize value (choice context))))
    (FINITE-XAPPING α(contextualize value •context))))

(defun lookup (exp env context)
  (let ((pair (assoc exp env)))
    (if pair
        (lookup-contextualize (- (context-height context) (cadr pair))
                              (cddr pair)
                              context)
        (contextualize (symbol-value exp) context))))

(defun lookup-contextualize (j value context)
  (cond ((< j 0) (error "Misplaced '•'"))
        ((= j 0) (context-filter value context))
        (t α(lookup-contextualize (- j 1) value •context))))

(defun trim-context (context)
  (etypecase context
    (CONSTANT-XAPPING
      (if (eq (choice context) t) t
          (make-constant-xapping (trim-context (choice context)))))
    (FINITE-XAPPING
      (if (eq (choice context) t) t
          α(trim-context •context)))))

(defun extend-context (context)
  (let ((alpha-t (make-constant-xapping t)))
    (labels ((ec (context alpha-t)
               (etypecase context
                 ((MEMBER T) alpha-t)
                 (CONSTANT-XAPPING
                   (make-constant-xapping (ec (choice context) alpha-t)))
                 (FINITE-XAPPING α(ec •context alpha-t)))))
      (ec context))))

(defun get-finite-context (fn args)
  (coerce (reduce #'(lambda (p q) (α(lambda (x y) y) p q))
                  (mapcar #'(lambda (a)
                              (domain (etypecase a
                                        (XAPPING a)
                                        (CLOSURE (closure-context a)))))
                          (cons fn args)))
          'LIST))
```

Table 2 (continued).

```
(defun ifpart (kind context)
  (etypecase context
    ((NOT XAPPING) (if context kind (not kind)))
    (CONSTANT-XAPPING
      (let ((z (ifpart kind (choice context))))
        (and z (make-constant-xapping z))))
    (FINITE-XAPPING
      (let ((z (remove-nils α(ifpart kind •context))))
        (and (not (empty z)) z)))))

(defun merge-results (x y) (xapping-union #'merge-results x y))

(defun context-filter (value context)
  (if (eq context t) value
      (etypecase value
        (CLOSURE
          (make-closure (closure-exp value)
                        (closure-env value)
                        α(context-filter •(closure-context value) •context)
                        (closure-indices value)))
        (XAPPING α(context-filter •value •context)))))

(defun context-height (context)
  (etypecase context
    ((MEMBER T) 0)
    (XAPPING (+ 1 (context-height (choice context))))))
```

Table 2 (concluded).

wise the context is "trimmed" to reduce its height by one. Because a bullet must cancel the innermost α, this reduction must take place near the leaves. The function `trim-context` performs the straightforward mechanics of context trimming. The function `context-filter` eliminates any values that are not relevant to the current context.

The processing of an `IF` form becomes much more complicated. The predicate expression is evaluated in the usual way to produce, of course, a value structure that matches the structure of the current context. From this two new contexts are computed; `truecontext` is that part of the original context where non-`nil` values resulted for the predicate, and `falsecontext` is that part of the original context where `nil` resulted for the predicate. The function `ifpart` computes such a context part, its first argument determining whether non-`nil` or `nil` values are sought; `ifpart` might also return `nil` if that part is entirely empty, in which case no evaluation should be performed for that arm of the `IF` expression. (We could have chosen to allow `nil` to stand generally for an empty context, and defined `eval` to return `nil` immediately if its `context` argument were `nil`. This would have simplified the code for processing `IF` forms, but would have complicated other parts of the interpreter. It struck us as needless generality.) If both arms of the `IF` form are to be evaluated, then the function `merge-results`, a one-line wonder, is used to combine the two result xappings to yield the value of the `IF` form. (To see why it works, one must realize that the results to be merged were computed in disjoint contexts; if therefore `xapping-union` recursively calls `merge-results`, the two arguments given to `merge-results` will necessarily also be xappings.)

No special changes are required in the processing of function call forms. The function `fneval` is also largely unchanged except for the call to `contextualize` and that fact that the current context is packaged up as part of a closure. Similarly `evlis` is changed only trivially.

The really interesting changes are in `apply`. When a closure is applied, the body of the `lambda` expression is executed in the usual way, but only after extending the lexical environment in an unusual manner. Instead of parameter/value pairs, the environment contains triples. The third element of each pair is information about the closure context, specifically its height. (The function `context-height` computes the height of a context. Recall that a context is a tree of uniform height.) This extra piece of information is used in the function `lookup` to deal with the implicit "destructuring" alluded to near the beginning of this section.

The application of a xapping proceeds in three stages. First, the domains of the xapping and the arguments are intersected; the result should be finite. The function `get-finite-context` computes the indices in this intersection and returns the result as a list, not as a xapping, to emphasize finiteness and eliminate any semantic confusion that might arise from using a xapping at this point. Second, for every index in this list an element of the function xapping is applied to a list of the corresponding elements from the argument xappings. Finally, the results are used to construct a new xapping that is the result of the application.

The definition of `aplis` shown in Table 2 does *not* provide for any parallelism; it performs applications one at a time. Note that `aplis` is identical in overall structure to `evlis`; hence its name. We are now in a position to distinguish between parallel argument evaluation and the parallelism in Connection Machine Lisp, as mentioned in section 3. General parallel argument evaluation is obtained by replacing the call `(cons ...)` in `evlis` with the form (pcall #'cons ...), where pcall is a Multilisp [10] special form that is like funcall in effect but evaluates all of its arguments in parallel. The data-oriented parallelism of applying a xapping-full of functions to xappings-full of arguments is made manifest by making the identical change to aplis:

```
(defun aplis (fn index-set)
  (and index-set
       (pcall #'cons         ;Allow parallelism.
              (funcall fn (car index-set))
              (aplis fn (cdr index-set)))))
```

The primary operational distinction between a xapping of closures and a closure over a context that is a xapping is one of synchrony. For a closure over a compound context, the code is executed in a synchronous fashion for all indices in that context; but applying a xapping of closures causes all the closures to execute asynchronously in parallel. There are also questions of efficiency and of distributed versus centralized control: Applying a closure involves only one call to eval, not many in parallel, and the computational overhead of interpretation is smaller but centralized. (For some computer architectures there is an advantage to expending additional resources for the interpretation of multiple, distributed copies of the same program.)

Note that the closure over a context could be treated as a xapping in apply and everything would continue to work if synchrony were not an issue. To see this, simply delete the arm of the etypecase for the type CLOSURE, and in the next arm replace the guard XAPPING with the guard (OR XAPPING CLOSURE). Thanks to the extended semantics of xref when applied to a closure, interpretation still works properly, but calls are not guaranteed to be synchronous.

7 Unsolved Problems

The interpreter in Table 2 is written so as to maintain a closure over a context in that form as long as possible, converting it into a xapping of closures only when forced to. This is done in an effort to maintain synchrony wherever possible. Control of synchrony, in turn, is desirable for two reasons. First, it gives the user more control over the behavior of the program. Second, we believe that synchronous parallelism in a program is easier to comprehend because control is always at a single place in the program text, rather than at many places simultaneously. We believe, for example, that the masterly but complex proof by Gries [9] of a relatively small program with only two processes demonstrates the difficulty of understanding parallel programs of the MIMD style.[6]

Consider this apparently straightforward code:

```
α(setf (xref x •j)
       (/ (+ (xref x (- •j 1))
             (xref x •j)
             (xref x (+ •j 1)))
          3))
```

With synchronous execution, this causes every element of a xector x specified by the indices in j to be replaced by the average of itself with its left and right neighbors. With asynchronous execution, however, all sorts of behaviors can occur, because some elements of x might be updated before the old value has been fetched for other averaging operations. That is because the thread of execution for one index might reach the setf operation before the thread for another index has performed the necessary fetch; the points of control for different indices may be at different points in the program text. This example is particularly devastating if j is an infinite xapping.

We have found the particular approach to synchrony exhibited by the code in Table 2 to be unsatisfactory in practice, because it depends on details of the operation of the interpreter and of the user program. Consider this expression:

```
α(if •p
     (funcall #'(lambda (x) (foo x •a)) •z)
     (funcall #'(lambda (x) (bar x •b)) •z))
```

All the calls to the closure that calls foo will occur synchronously, as might be expected, and similarly for the other closure. But in the superficially similar expression

```
α(funcall (if •p
              #'(lambda (x) (foo x •a))
              #'(lambda (x) (bar x •b)))
          •z)
```

the calls to the closure that calls foo will occur asynchronously, because the merging of the two closure-xappings to produce the value of the IF form requires conversion of each to a xapping of closures.

One possible patch that masks some of these symptoms is to change the code in apply that handles application of a xapping. The code can examine the elements of the xapping, and collect all the elements that are closures into equivalence classes, where members of the same equivalence class have eql expression, environment, and context components, differing therefore only in their indices lists. All the members of an equivalence class could then be processed by a single call to eval by constructing a new closure whose context is constructed from their various leading indices.

This patch seems rather hackish to us, however, and no less opaque in its operation. We would prefer that synchrony be enforced in a more manifest manner, such as a visible syntactic device.

An obvious point at which to synchronize is an explicit occurrence of α. We lean toward defining the language in such a way that an expression preceded by α is executed asynchronously for all relevant indices, but resynchronization occurs when assembling the result. For instance, in the expression α(foo (bar •x)) it might be that side effects of some calls to foo might occur before all side effects of calls to function bar had occurred. In contrast, the expression α(foo •α(bar •x)) requires that all calls to bar be completed before any calls to foo occur. The averaging example shown above could then be fixed as follows:

```
α(setf (xref x •j)
       •α(/ (+ (xref x (- •j 1))
               (xref x •j)
               (xref x (+ •j 1)))
            3))
```

In this formulation we would expect "•α" to be a widely used cliché meaning "synchronize here." In the example it forces the parallel executions to synchronize after all divisions are completed but before any values are stored.

This approach to synchronization also partly destroys the syntactic transformations involving α and •; one may always introduce •α in front of an expression, but one may not not cancel

[6] The technique of the proof, which is due to Owicki [17], is first to prove each process correct in isolation, and then to consider all possible interactions by considering all possible (actually, all "interesting") pairs of control points within the two processes. The difficulty of this technique increases exponentially (quite literally) with the number of processes.

it without endangering program correctness. Perhaps this convention is yet too delicate.

The implications of this definition for the structure of the interpreter are not entirely clear to us. The net result may be to shift the introduction of parallelism from `apply` back into `eval`, as in Table 1, but this will reintroduce all the problems of recursively calling `eval` for an infinite number of indices. We believe that this can be avoided by recasting the interpreter into the continuation-passing style [19,24,25], but that is beyond the scope of this paper.

Stylistically speaking, Connection Machine Lisp is primarily SIMD in its approach (though providing completely general communications patterns with the β operator, as opposed to the fixed communications patterns that have historically been associated with most SIMD machine architectures). Some interpretations of the semantics of α permit some slight asynchrony, but only in the evaluation of many copies of the same expression. However, there is a hook that allows expression of completely general MIMD parallelism: a function call where the function is a xapping whose elements are distinct. For example,

```
(funcall '[sin cos tan eval] '[0 1 2 (foo)])
```

causes the calls

```
(sin 0)       (cos 1)       (tan 2)       (eval '(foo))
```

to proceed in parallel; this is equivalent in effect to writing

```
(pcall #'xector (sin 0) (cos 1) (tan 2) (eval '(foo)))
```

in Multilisp syntax. We have barely begun to explore the expressive power and implementation requirements that arise from this technique.

8 Comparisons to Other Work

In this section we compare Connection Machine Lisp to seven other programming languages.

APL [13,12,8,4]. Connection Machine Lisp xappings have state (can be modified using `setf` of `xref`), whereas APL vectors are immutable. The elements of xappings may be any Lisp objects; APL array elements must be numbers or characters. Indices of APL vectors are consecutive integers beginning at 0 or 1; the indices of xappings may be any Lisp objects, and need not be contiguous. (A xapping with n pairs may be represented as a $2 \times n$ APL matrix, of course, but part of the point of xappings is notational and computational convenience.) Connection Machine Lisp has more expressive control structures, namely those of Lisp. Many of the ideas and specific operations in APL are useful in Connection Machine Lisp. The general Connection Machine Lisp β operator has no simple equivalent in APL. APL has multidimensional arrays and a useful set of operations on them; in Connection Machine Lisp we have thus far represented multidimensional arrays by nesting one-dimensional xappings. (We have considered handling multidimensional arrays by letting an index itself be a xapping: a 2×2 identity matrix would then be represented as

```
{[0 0]→1 [0 1]→0 [1 0]→0 [1 1]→1}
```

The difficulty here is that in this case we would like for two indices to be considered the same if they are `equal` rather than merely `eql`; however, the fact that xappings are mutable creates grave difficulties. What if the xector `[1 0]` used as an index in the identity matrix were mutated to be `[1 1]`? Remember that no two pairs of a xapping may have the same index. These nasty issues are the reason why indices are compared using `eql`: the equivalence of indices must remain invariant under mutability of data.)

NIAL [14,20]. Many of the comments about APL apply to NIAL, except that NIAL allows nested arrays. NIAL, unlike APL, allows user-defined functions to be used with the reduction and scan operators, and indeed all operators. NIAL has a cleaner and more convenient syntax for talking about functional operators than a Connection Machine Lisp based on Common Lisp, but a Connection Machine Lisp based on Scheme would have a syntax as clean as that of NIAL. We think Connection Machine Lisp has a better notation (α and $\bullet$) for nested uses of the apply-to-all construct (which NIAL calls EACH). Such nested uses do not occur so frequently in NIAL because apply-to-all is implicit in many NIAL operations; this is possible because NIAL has a different theory of data structures than Lisp [15,16]. In NIAL data is immutable but variables are mutable. (The remarks about NIAL apply for the most part also to other "modern" APL implementations such as those of IBM, STSC, I. P. Sharp, etc.)

FP [1]. Many of the ideas and notations of FP are easily and usefully carried over into Connection Machine Lisp, and indeed we have translated some examples from Backus's paper. Like Lisp, however, Connection Machine Lisp is oriented around variables and less around functional composition; FP does not explicitly name the data to be operated on, but relies on combinator-like control of the flow of data. FP is an applicative language; data is immutable.

QLAMBDA [5]. Connection Machine Lisp organizes its parallelism around data structures rather than control structures, and thus may be more suited to a SIMD architecture than to a MIMD architecture; the opposite may be true of QLAMBDA.

Multilisp. Multilisp, like QLAMBDA has parallelism organized around control structures rather than data structures. Multilisp introduces parallelism in two ways, one structured and the other extremely unstructured. The structured way allows the elements of a very particular data structure to be computed in parallel; this data structure is the list of arguments for `pcall`. The unstructured way is the use of `future`, which allows an arbitrary computation (the argument form to `future`) to proceed in parallel with another computation of arbitrarily unrelated structure (the remainder of whatever computation surrounds the execution of `future`, that is, the continuation).

KRC (Kent Recursive Calculator) [29]. Lazy xappings are somewhat similar in their use to the infinite lists of KRC, and especially to the set abstraction expressions of KRC. Set abstraction expressions contain additional mechanisms for filtering and taking Cartesian products that lazy xappings do not have; these lend KRC a great deal of expressive power. Xappings have state and may be modified (even lazy xappings), whereas KRC is an applicative language without side effects.

Symmetric Lisp [6,7]. There is a possible confusion between our notation and that of Gelernter, because his work also involves parallelism in Lisp and uses a notation involving the word ALPHA. We regard highly his study of space-time symmetries in programming languages, but believe that our notation and our approach to parallelism are rather different from his, despite the accident of similar terminology.

9 Implementation Status

A Connection Machine Lisp interpreter that supports constant, universal, and lazy xappings has been implemented on the Symbolics 3600, a sequential processor, for experimental purposes. It has been used to test the ideas in this paper and to execute a number of smallish programs (up to fifty lines in size). All of the examples in this paper have been tried out on this interpreter.

An implementation is planned for the Connection Machine System [11], a 1000-MIPS, fine-grained, massively data-parallel computer with 65,536 (2^{16}) processors, 32 megabytes (2^{25} bytes) of memory, and a general communications network among the processors. However, we cannot now guarantee exactly when a full implementation of Connection Machine Lisp on the Connection Machine System will be ready.

10 Conclusions

We have designed a dialect of Lisp that we believe will be useful for symbolic processing problems that are susceptible to solutions with fine-grained, data-oriented parallelism. This dialect features an array-like data structure designed to be processed in parallel using operations of the kind appearing in FP, APL, and NIAL. It also features a notation, similar in form to the Common Lisp backquote construct, for expressing parallelism in a manner that facilitates both macroscopic and microscopic views of parallelism.

There remains a design space of modest size to explore, in which several important design goals are in essential conflict:

- compatible extension of an existing Lisp dialect
- convenience in using functional arguments and values
- consistency between macroscopic (arrays of data) and microscopic (code within individual processors) understanding of parallelism
- a model of data consistent with that of the base language (including that fact that data structures are mutable)
- a treatment of side effects consistent with that of the base language
- generality of the α-notation, including the rule of distribution over function calls
- control over parallelism and synchrony

We have tested a few points in this design space to determine which results in the most useful language design for practical purposes.

Other topics to explore include the integration of other notions of parallelism into Connection Machine Lisp, such as the `future` and `pcall` constructs of Multilisp, and which applications in symbolic computation are suited to this fine-grained, data-oriented style of parallel programming.

11 Acknowledgements

The term "xapping" was suggested to us by Will Clinger.

We are grateful for many comments from Michael Berry, Ted Tabloski, and others within Thinking Machines Corporation, and from the referees.

The ornamentation was taken from a volume of the Dover Pictorial Archive Series: Klimsch, Karl. *Florid Victorian Ornament.* Dover Publications (New York, 1977).

References

[1] Backus, John. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM 21*, 8 (August 1978), 613–641. 1977 ACM Turing Award Lecture.

[2] Clinger, William, editor. *The Revised Revised Report on Scheme; or, An Uncommon Lisp.* Computer Science Department Technical Report 174. Indiana University (Bloomington, June 1985).

[3] Clinger, William, editor. *The Revised Revised Report on Scheme; or, An Uncommon Lisp.* AI Memo 848. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, August 1985).

[4] Falkoff, A. D., and Orth, D. L. Development of an APL standard. In *APL 79 Conference Proceedings.* ACM SIGPLAN/STAPL (Rochester, New York, June 1979), 409–453. Published as *APL Quote Quad 9*, 4 (June 1979).

[5] Gabriel, Richard P., and McCarthy, John. Queue-based multiprocessing Lisp. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming.* ACM SIGPLAN/SIGACT/SIGART (Austin, Texas, August 1984), 25–44.

[6] Gelernter, David. *Symmetric Programming Languages.* Technical Report. Yale University (New Haven, July 1984).

[7] Gelernter, David, and London, Thomas. *The Parallel World of Symmetric Lisp.* Technical Report. Yale University (New Haven, February 1985). Extended abstract.

[8] Gilman, Leonard, and Rose, Allen J. *APL: An Interactive Approach*, second edition. Wiley (New York, 1976).

[9] Gries, David. An exercise in proving parallel programs correct. *Communications of the ACM 20*, 12 (December 1977), 921–930.

[10] Halstead, Robert H., Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems 7*, 4 (October 1985), 501–538.

[11] Hillis, W. Daniel. *The Connection Machine.* MIT Press (Cambridge, Massachusetts, 1985).

[12] *APL\360 User's Manual.* International Business Machines Corporation (August 1968).

[13] Iverson, Kenneth E. *A Programming Language.* Wiley (New York, 1962).

[14] Jenkins, Michael A., and Jenkins, William H. *Q'Nial Reference Manual.* Nial Systems Limited (Kingston, Ontario, July 1985).

[15] More, Trenchard. The nested rectangular array as a model of data. In *APL 79 Conference Proceedings.* ACM SIGPLAN/STAPL (Rochester, New York, June 1979), 55–73. Published as *APL Quote Quad 9*, 4 (June 1979). Invited address.

[16] More, Trenchard. Rectangularly arranged collections of collections. In *APL 82 Conference Proceedings.* ACM SIGPLAN/STAPL (Heidelberg, Germany, June 1982), 219–228. Published as *APL Quote Quad 13*, 1 (June 1982). Invited address.

[17] Owicki, Susan Speer. *Axiomatic Proof Techniques for Parallel Programs.* PhD thesis, Cornell University, Ithaca, New York, July 1975. Department of Computer Science TR 75-251.

[18] Porter, Thomas, and Duff, Tom. Compositing digital images. In *Proc. ACM SIGGRAPH '84 Conference.* ACM SIGGRAPH (Minneapolis, July 1984), 253–260. Published as *Computer Graphics 18,* 3 (July 1984).

[19] Reynolds, John C. Definitional interpreters for higher order programming languages. In *Proc. ACM National Conference.* Association for Computing Machinery (Boston, August 1972), 717–740.

[20] Schmidt, Fl., and Jenkins, M. A. Rectangularly arranged collections of collections. In *APL 82 Conference Proceedings.* ACM SIGPLAN/STAPL (Heidelberg, Germany, June 1982), 315–319. Published as *APL Quote Quad 13,* 1 (June 1982).

[21] Schwartz, J. T. Ultracomputers. *ACM Transactions on Programming Languages and Systems 2,* 4 (October 1980), 484–521.

[22] Shaw, David Elliot. *The NON-VON Supercomputer.* Technical Report. Department of Computer Science, Columbia University (New York, August 1982).

[23] Steele, Guy L., Jr., Fahlman, Scott E., Gabriel, Richard P., Moon, David A., and Weinreb, Daniel L. *Common Lisp: The Language.* Digital Press (Burlington, Massachusetts, 1984).

[24] Steele, Guy Lewis, Jr. *Compiler Optimization Based on Viewing LAMBDA as Rename plus Goto.* Master's thesis, Massachusetts Institute of Technology, May 1977. Published as [25].

[25] Steele, Guy Lewis, Jr. *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization).* Technical Report 474. MIT Artificial Intelligence Laboratory (May 1978). This is a revised version of the author's master's thesis [24].

[26] Steele, Guy Lewis, Jr., and Sussman, Gerald Jay. *The Art of the Interpreter; or, The Modularity Complex (Parts Zero, One, and Two).* AI Memo 453. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, May 1978).

[27] Steele, Guy Lewis, Jr., and Sussman, Gerald Jay. *The Revised Report on SCHEME: A Dialect of LISP.* AI Memo 452. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, January 1978).

[28] Sussman, Gerald Jay, and Steele, Guy Lewis, Jr. *SCHEME: An Interpreter for Extended Lambda Calculus.* AI Memo 349. MIT Artificial Intelligence Laboratory (Cambridge, Massachusetts, December 1975).

[29] Turner, David A. The semantic elegance of applicative languages. In *Proc. 1981 Conference on Functional Programming Languages and Computer Architecture.* ACM SIGPLAN/SIGARCH/SIGOPS (Portsmouth, New Hampshire, October 1981), 85–92.