Multidimensional Problem Solving in Lucid[*]

Technical Report SRI-CSL-93-03

A.A. Faustini
Department of Computer Science
Arizona State University
Tempe, Arizona 85287, U.S.A.
phone: (602) 965-3983, email: faustini@lu.eas.asu.edu

and

R. Jagannathan
Computer Science Laboratory
SRI International
Menlo Park, California 94025, U.S.A.
phone: (415) 859-2717, email: jaggan@csl.sri.com

[*]Expanded version of paper titled "Indexical Lucid" by the authors in the Fourth International Symposium on Lucid and Intensional Programming (ISLIP 91) held at SRI International in April, 1991.

Abstract

One of the limitations of Original Lucid is its inability to deal easily with multidimensional data structures such as arrays and trees. Over the past ten years, Lucid has evolved sufficiently to express and manipulate multidimensional data structures that change — thus making multidimensional problem solving naturally possible in the latest version of Lucid, called Indexical Lucid.

In this report, we trace the evolution of Lucid from 1984 to its current form. We discuss the novel aspects of Indexical Lucid by contrasting it with Original Lucid. We also illustrate the expressiveness of Indexical Lucid by solving selected multidimensional problems.

1 Background

Lucid (Original Lucid) was invented in 1976 by Ashcroft and Wadge as a system for writing and proving properties about programs [2, 3]. The most unusual aspect of Lucid, in addition to being non-procedural, is its dynamic view of computation: one in which data is in motion that is generated and consumed by stationary operations and functions. Because of this view, commonly known as dataflow, Original Lucid can be considered to be one of the first dataflow programming languages [11].

In an Original Lucid program, a variable or an expression denotes an infinite sequence of values that is thought of as a temporal sequence. The basic Original Lucid operators `first`, `next`, and `fby` (for "followed by") appeal to this temporal view of sequences. It is also possible to specify subcomputations in the language by using nested time. In particular, variables can be defined to vary in a nested time dimension such that variables defined in outer time dimensions appear to be constant or "frozen."

One of the main drawbacks of Original Lucid is that it does not naturally support multidimensional data structures such as arrays or trees. Such structures are commonplace in many application domains, especially scientific computing. This need to solve multidimensional problems in Original Lucid resulted in two extensions: Ferd Lucid [4, 1] and Field Lucid [7, 5, 10].

Ferd Lucid Ferd Lucid advocates an "extensional" way of using multidimensional data structures, notably arrays. It does so by changing the underlying data algebra of Lucid to allow sequences to vary in space as the elements of sequences that vary in time. These space-varying sequences are called *ferds* (an obsolete word in the Oxford English Dictionary meaning "warlike arrays"). Multidimensional data structures in Ferd Lucid are defined and manipulated using the operators `initial`, `rest` and `cby` ("continued by"). This complements the `first`, `next` and `fby` of Original Lucid. Moreover, the proposed operators are information preserving. In other words, a multidimensional data structure can be defined and manipulated without losing any of its information. Ordinary temporal streams built with the operators `first`, `next` and `fby` are not information preserving because `fby` does not nest; rather it combines its arguments by taking the first value of the left argument of `fby` and combining it with all the stream of values associated with the second argument of `fby`. This means that only the first value of the left argument of `fby` can be retrieved from an object built using `fby`. The operator `cby` preserves information by augmenting the dimensionality of its result. For example, if two simple one-dimensional objects are the arguments of `cby` as in `arg1 cby arg2`, the result is a two-dimensional object. The approach is analogous to the way in which a `cons` operator of functional programming nests its first argument at the head of the list that is produced by `cons`.

The main drawback of Ferd Lucid is that the extensional treatment of arrays (i.e., a 3D object is a sequence of 2D objects and a 2D object is a sequence of 1D objects) makes the language to difficult to use it in solving multidimensional problems where the dimensions are not only orthogonal but transposable. In addition, in Ferd Lucid, one needs to be aware of the "rank" (or the dimensionality) of the multidimensional data structure one is dealing with since `cby` and `initial` operators respectively increase and decrease the rank.

Field Lucid Field Lucid advocates an "intensional" way of defining and using multidimensional data structures by allowing a sequence to vary simultaneously in multiple, orthogonal dimensions. A simple extension to the Original Lucid temporal operators deals with this type of multidimensionality but distinct operators are defined for each new dimension. They are simply definitions similar to `first`, `next` and `fby` but for particular dimensions rather than for time where both the dimension name and the associated operators are predefined. These operators are `initial0`, `succ0` and `sby0` to define objects varying in the first dimension, `initial1`, `succ1` and `sby1` the second dimension and so on.

The main drawback of Field Lucid is that it adopts an "absolute" view of the multidimensionality; the names of the multiple (orthogonal) dimensions are preordained. Thus, it is not possible to apply a function that expects its arguments to be defined over space dimension 0 to arguments defined over space dimension 1.

2 Synopsis

The objective of this report is to present a generalization of Original Lucid for multidimensional problem solving and demonstrate its expressiveness by solving selected multidimensional problems in it. The new language, which for the purposes of this report we shall refer to as "Indexical Lucid," is presented after briefly reviewing Original Lucid. The two novel aspects of the proposed language are user-defined dimensions and dimensional abstraction both of which are extensively used in the three problems that we solve: prime number generation, mergesort, and block matrix multiplication.

3 Programming in Original Lucid

In a procedural language, a variable is associated with a sequence of values. When there is an initial assignment of a value to a variable, it is usually the first time an assignment is made with respect to the control flow of the program. As control moves through the procedural program, each assignment that is encountered modifies the memory location associated with the variable. If we trace the values that were associated with the variable we can associate a sequence of values with each variable. This sequence denotes the "history" of the variable. In a conventional procedural language there is usually no trace or history of the value associated with a variable because assignment is destructive. A variable in an Original Lucid program is not associated with a destructively modifiable memory location. Like its procedural counterpart, an Original Lucid variable takes on different values at different times during the computation. The "history" or sequence of values associated with an Original Lucid variable is achieved through equational definitions rather than destructive assignment and control flow. The equation `x = 1` defines the variable `x` to be a constant, that is, `x` is equal to 1 at all times. On the other hand `y = 1 fby 2 fby 3` is an equation that says that the initial value `time = 0` of `y` is `1`, the next (`time = 1`) value of `y` is `2` and at all times beyond this the value of `y` is always `3`. There is no destructive assignment; the history of a variable is always available, and previous values of a variable can be used to define the value of other variables. The following is the syntax used in [3] to express the idea of an Original Lucid variable taking on different values at different times. The pair of equations given below define the variable `even`:

```
first even = 2
next even = even+2
```

The first equation defines the first or initial value of the `even` stream (2) and the second equation the successive values of `even` in terms of the current value of `even` plus two[1].

We can think of the above equations as equivalent to the following recurrence relation :-

$$\begin{array}{ll} t = 0 & \mathtt{even}_t = 2 \\ t > 0 & \mathtt{even}_{t+1} = \mathtt{even} + 2_t \end{array}$$

The `t` in the recurrence relation is the implicit dimension or time context of Original Lucid. There is no explicit mention of `t` or `time` other than the first equation that states that the initial value of `even` is `2` and yet the above Original Lucid equations define the stream of values $< 2, 4, 6, 8, \ldots >$ without the explicit use of indices. (In this sense we can think of Original Lucid as an intensional language, that is a language based on intensional logic [7]). In Original Lucid the implicit dimension is used simply as the time or iteration dimension. Consequently, a multidimensional problem would have to be solved using time and some form of monolithic data structure such as a list or string [6]. multidimensional problems can be expressed directly in Indexical Lucid in a manner akin to mathematical function definitions and without use of monolithic data structures. The latter has important implications for parallel and distributed implementations. multidimensional problems are found in many applications areas and specially in scientific computing and computer graphics. The rest of this report deals with the "operational" details of Indexical Lucid and how to use dimensional constructs to solve problems.

[1]This syntax comes from the Original Lucid papers and is often a better way of expressing the idea of a variable changing with time than the more modern semantically equivalent equation:-
```
even = 2 fby even+2
```

4 Adding Multidimensionality

The simplest way of extending the number of dimensions available to the Lucid programmer is to allow objects in Lucid to vary over an infinite number of space dimensions. This was first suggested in [11] in which the dimensions where thought of as space dimensions whose names corresponded to the natural numbers. All the Original Lucid operators can then be extended to work over these predefined space dimensions. For example, the `fby` operator can be extended by naming a new operator for each of the predefined space dimensions `sby0`, `sby1`, `sby2`, etc. This is the absolute approach to multidimensionality.

Indexical Lucid permits the programmer to extend the dimensions explicitly giving appropriate names to the new dimensions as and when they are needed. A dimensional where clause is one that contains declarations of new dimension names (or equivalently dimension names) through the use of an dimension declaration. The scope of the dimensions introduced by the dimension declaration is the same as the scope of the variables defined in an Original Lucid where clause. Dimensional where clauses may be nested to arbitrary depths in the same way as where clauses in Original Lucid are nested. The scope rules for dimension names are the same as the scope rules for variables in where clauses [9]. Within the same scope it is not permitted to define a variable with the same name as a dimension name.

The following is an example of an dimensional where clause with two new dimension names `height` and `width`.

```
a+b where
        dimension height,width;
        a = ...
        b = ...
    end
```

In the above program the variables `a` and `b` may be defined to vary over the two new dimension names `height` and `width`. To determine how `a` or `b` vary with any of the dimensions just mentioned we need to examine the actual definitions of `a` or `b`. For example, if `a = 10;` then `a` is a constant, if `a = 1 fby a+1;` then `a` varies with dimension `time`. To make `a` or `b` vary with `height` or `width` we need to introduce operators specific to the new dimension names. All of the usual Original Lucid operators can be extended to operate on any dimension name in a simple and natural manner. For example, the infix binary operator `fby` is extended to work on say the `height` dimension through use of the operator `fby.height`. In a similar manner all of the usual Original Lucid operators can be extended to work on any dimension name that has been previously introduced by a dimension declaration and is still in scope. The following is a more complete enumeration of the extended operators.

```
first      first.time      first.h
next       next.time       next.h
prev       prev.time       prev.h
fby        fby.time        fby.h
before     before.time     before.h
upon       upon.time       upon.h
```

```
asa              asa.time              asa.h
wvr              wvr.time              wvr.h
@                @.time                @.h
```

Note that it is not necessary to write `first` as `first.time`. Any implementation of Indexical Lucid will recognize `first` as an abbreviation for `first.time`. Each of the extended Lucid operators works on its particular dimension in the same way that the ordinary Lucid operators works on the `time` dimension. This makes the new operators easy for the Lucid programmer to use.

Dimensional locators are an important part of Indexical Lucid. Each dimension name introduced by a dimension declaration also introduces a new dimensional locator having the same name as the dimension name. The dimensional locator `height` can be used in the expression part of a program and the value it takes depends on when it is used. In general, the value of `height` at position i in the `height` "dimension" is i. In other words, dimensional locators tell the programmer where they are in the dimensional space that has been built up with dimensional where clauses. Note that all dimensions in Indexical Lucid vary over the integer set (ergo, the name Indexical Lucid). It is possible to use other dimension sets but we will not discuss that here.

Here is a fragment of simple Indexical Lucid code:-

```
.... where
dimension i,j;
alt_i = 0 fby.i alt_i + 1;
alt_j = 0 fby.j alt_j + 1;
alt_i_j = (0 fby.i alt_i_j + 1) fby.j alt_i_j + 1;
end
```

In this example, the variable `alt_i` is another way of defining the dimensional locator `i`. In other words, `alt_i` at position k in the `i` "dimension" takes on the value k. In a similar manner, `alt_j` is equivalent to the dimensional locator `j`. In addition, `alt_i_j` illustrates the way in which variables can be defined over more than one dimension name. In the above example `alt_i_j` is equivalent to the dimensional locator `i` in the `i` "dimension" and equivalent to dimensional locator `j` in the `j` "dimension". In original ISWIM [9] simple variables like `alt_i` and `alt_j` can only be given a single value because of the functional nature of ISWIM. In Original Lucid, such variables can be defined to take on different values at different point in Lucid time. In other words, Original Lucid variables are temporal but Indexical Lucid variables take on values over a multidimensional space of dimension names as well as the usual temporal dimension `time`.

In Original Lucid, the "is current" declaration within a where-clause is used to define a nested computation (or subcomputation) [6]. The where clause is said to define a "frozen environment" since variables in it can only vary in the new dimension. The following program is an example of a nested computation in Original Lucid to compute the running root mean square of its input. Note the use of the `is current` in the definition of `sqroot`. In Original Lucid, where clauses with `is current` declarations implicitly introduces a new dimension into the program. That is, for each point in time outside an "is currented" where-clause, there corresponds a possibly different set of time sequences for each of the variables in the clause.

```
sqroot(avg(square(a)))
```

```
     where
       square(x) = x*x;
       avg(y) = mean
                  where
                    n = 2 fby n+1;
                    mean = y fby mean + d;
                    d = (next y - mean) / n;
                  end;
       sqroot(z) = approx asa err < 0.0001
                    where
                     Z is current z;
                     approx = Z/2 fby (approx + Z/approx)/2;
                     err = abs(square(approx)-Z);
                    end;
      end
```

The dimensional version of the same program makes this implicit dimension visible (as in dimension `i`). In a similar manner "is currented" where clauses nested within other "is currented" clauses introduce a new implicit dimension for each level of nesting. This can again be implemented using dimensional where clauses nested in the same manner but explicitly introducing a new dimension of each level of nesting of the "is currented" where clauses.
An equivalent program written using a dimensional where clause:-

```
    sqroot(avg(square(a)))
     where
       square(x) = x*x;
       avg(y) = mean
                  where
                    n = 2 fby n+1;
                    mean = y fby mean + d;
                    d = (next y - mean) / n;
                  end;
       sqroot(z) = approx asa.i err < 0.0001
                    where
                     dimension i;
                     approx = z/2 fby.i (approx + z/approx)/2;
                     err = abs(square(approx)-z);
                    end;
      end
```

The introduction of extra dimensions whether implicit, as in Original Lucid or Field Lucid, or explicit, as in Indexical Lucid, means that the language can directly define objects in terms of objects of higher dimensionality. In other words the subject part of an "is currented" or dimensional where clause is defined in terms of functions and variables inside the body of the where clause and these functions and variables will probably be defined in terms of the extra dimensions introduced by the dimension declaration (otherwise there would be no point in having the subcomputation). The question then arises as to which of the values associated with the extra dimensionality is used to give meaning to the outside objects ? In Field Lucid, a diagonalization method is used. That is, for each point in time (say k) in the outer world, we associate the i^{th} point in time of the inner world. That is, if a variable `v` is defined in terms of an “is currented” where clause, the value the variable will take on for each point in time, say k, in the outer environment is the value of the subject part of the “is currented” where clause at inner time `k`. This works well in Original Lucid because inner subcomputations are always one dimension higher than their immediately enclosing lexical environment.

In Indexical Lucid diagonalization is not appropriate. The reason for this is that if a variable `v` is defined directly in terms of an dimensional where clause, `v` itself might already be in an environment in which it varies with respect to many dimensions (not just time as in Original Lucid). Which of these dimensions will be used to diagonalize the inner subcomputation? Fortunately, we can use a solution that is far simpler than diagonalization. If a variable `v` (which may vary in many dimensions) is defined directly in terms of an dimensional where clause (which can vary in even more dimensions) we can make the inner dimensionality compatible with the outer dimensionality by setting all the new dimensions created by the dimension declaration of the where clause to 0. This means that whichever result that is being computed by the dimensional where clause is carried to the outside by being placed at the origin of the dimensional where clause. We call this method of collapsing the “origin” method. Note that in practice most Original Lucid programs use the origin method since the subject part of an “is currented” where clause is usually defined in terms of an `asa` operator. In Indexical Lucid values can be pinned to the origin by the use of `asa` or `@` operators.

5 Dimensional Abstraction

In Original Lucid, function definitions are used to encapsulate or abstract temporal or stream functions as well as simple first order data functions as in the following examples.

```
g(x) = first x + next x;
f(x,y) = x*y+4;
```

The function definitions of Original Lucid need to be generalized if we want to use them to define multidimensional functions. In Original Lucid the `time` dimension is absolute and so ordinary function definition notation is adequate. This means that function like `g(x)` above can be defined in Original Lucid without difficulty. This does not extend to multidimensional user-defined dimension names. In other words, we have no means of abstracting dimension names in function definitions. For example the function, `VectorSum` sums the first 100 elements along the `x` dimension:-

```
......
 where
        dimension x,y;
        .....
```

```
          VectorSum(V1)  = sum asa.x x>100
                                          where
                                             sum = 0 fby.x V1+sum;
                                          end;
         A = 0.2 fby.x A + 1;
         B = 3.3 fby.y B + 2;
         sum1 = VectorSum(A);
         sum2 = VectorSum(B);
     ....
 end
```

The problem with the above definition of `VectorSum` is that it will only compute the sum of the first `100` elements along the `x` dimension. Thus the value of `sum1` will be as expected and the value of `sum2` will be the $100 * \texttt{B}$ because `B` does not vary with respect to the `x` dimension. In order to abstract dimensional names so that we can write one vector sum function for any dimension, we need to augment the way in which Original Lucid (and ISWIM) defines and uses functions. We choose to augment Original Lucid function definition by permitting the declaration of formal dimensional names on the left hand side of equations. To be more precise a function name may be followed by a sequence of formal dimension names each prefixed with a period (.). Here are two examples:-

```
f.k(x) = ...
g.a,b(x,y) = .....;
```

The following is a rewritten version of the vector sum example with both dimensional parameters and value parameters :-

```
 ....
  where
         dimension x,y;
         .....
         VectorSum.k(V1) = sum asa.k k>100
                                          where
                                             sum = 0 fby.k V1+sum;
                                          end;
         A = 0.2 fby.x A + 1;
         B = 3.3 fby.y B + 2;
         sum1 = VectorSum.x(A);
         sum2 = VectorSum.y(B);
     ....
  end
```

The `k` dimension is a formal dimension name, the actual dimension name is determined when the function `VectorSum` is used, as in `VectorSum.x(A)` or `VectorSum.y(B)`. In the first of these, the actual dimension name is `x`; this name is used in the actual computation of the vector sum by the function `VectorSum`. We note that simple variable definitions can also have dimensional parameters. They can be thought of as nullary function definitions, that is, function definitions with dimensional parameters but without value parameters. Here are some examples:-

```
a.x = .....;
b.x,y = .....;
```

This may seem less useful than the non-nullary form but there are instances in which it is useful. For example if `m` is a multidimensional variable with formal dimension parameters `x` and `y` as in the following example :-

```
dimension h,i,j;
m.x,y = if(x==y) then 1 else 0 fi;
use1 = m.h,i;
use2 = m.i,j;
......
```

The variable `use1` is a unit matrix with respect to the `h` and `i` dimension names and `use2` a unit matrix with respect to the `i` and `j` dimension names.

We can now show how the built-in Indexical Lucid functions can be defined. It is interesting to note that all dimensional functions can be defined using the primitive operator `@.<dimension name>`, dimensional locators and the Lucid `if-then-else-fi` operator. To illustrate this we give definitions to all of the Lucid built-in operators in terms of these operators. The reader should also note that none of the definitions are in terms of recursive non-nullary functions.

```
first.i(x) = x @.i 0
next.i(x) = x @.i (i+1)
prev.i(x) = x @.i (i-1)
fby.i(x,y) = if i<=0 then x @.i i else prev.i y fi
before.i(x,y) = if i>=0 then y @.i i else next.i x fi
asa.i(x,y) =  x @.i ( c1 @.i  0)
where
  c1 = if y then i else next.i c1 fi;
end;
upon.i(x,y) = x @.i t1
where
  t1 = 0 fby.i if y then t1+1 else t1 fi;
end;
swap.i.j(x) = x @.i j @.j i
rightChild.i(x) = x @.i (2*i+1);
leftChild.i(x) = x @.i (2*i);
```

6 Selected Multidimensional Applications

With the above description of Indexical Lucid, we will now illustrate its expressiveness in solving multidimensional problems. The three "standard" problems we consider are prime number generation using the sieve of Eratosthenes algorithm, sorting using the mergesort algorithm, and matrix multiplication using a recursive block decomposition algorithm.

6.1 Prime Number Generation

The sieve of Eratosthenes is an ancient method for computing prime numbers. The algorithm begins with a finite sequence of all natural numbers from 2 in increasing order. Successive passes are made over this sequence filtering out multiples of numbers that are found to be primes. The process begins by removing multiples of 2 from the sequence. The next number after 2, in this case 3, is the next prime and it then is used to filter out all multiples of 3. This process continues until only prime numbers are left in the finite sequence. The following sequence of sequences illustrates this process:-

```
2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17.
2,3,5,7,9,11,13,15,17.
2,3,5,7,11,13,17.
```

In Lucid we can write this application using two dimensions or we can use one dimension and function recursion. Either way we begin with a sequence of all numbers greater than or equal to 2. We will call this `natsFrom2`. This sequence or one dimensional variable is infinite and so our program will be a little more general than the process described above. Once we have `natsFrom2` we need to sieve the sequence to remove multiples of numbers that we know to be primes. In the first program below we will do the sieving using a recursive function `Sieve`. Each call of the recursive function will work on a new sequence of numbers to filter. This corresponds to each sequence in the above sequence of sequences except that out sequence of sequences is infinite. The second example program does not use recursive functions. It uses a new dimension in which to express the sequence of sequences. In either program the same filtering mechanism is used. The `wvr` operator, appropriately indexed, is used to do the actual work of filtering. In our case we want the control stream of the `wvr` to be `true` when ever the current data element is not a multiple of the first element of the stream we are filtering. This is why the first program uses the expression `n wvr ( n mod ( (first n) ne 0) )` where `n` is the stream being filtered. and the second program uses the expression `sieve wvr.i ( sieve mod ( ( first.i sieve ) ne 0 ) )`. (Note the use of dimensional locator `i` in the definition of `natsFrom2` in Program 2.)

```
// Program 1

Primes(natsFrom2) where
  Primes(n) = n fby Primes(n wvr ( n mod ( ( first n ) ne 0 ) ) );
  natsFrom2 = 2 fby natsFrom2+1;
end

// Program 2

primes where
```

```
    dimension i;
    primes = first.i sieve;
    natsFrom2 = i+2;
    sieve = natsFrom2 fby.time
                  ( sieve wvr.i ( sieve mod ( ( first.i sieve ) ne 0 ) ) );
  end
```

6.2 Mergesort

The merge sort problem is to sort a list of data items by repeatedly merging sorted lists starting with each data item as a sorted singleton list. The solution to the mergesort problem in Indexical Lucid uses three dimensions. Dimension `t` is used as the iteration dimension. Two other dimensions are required. In the program, they are `v` and `h`. The `h` dimension is used as the dimension in which the unsorted data is initial put. The `v` dimension is used to accumulate sorted sub-sequences. At time 0 the `mergeSort` is simple the original input spread out along the `h` dimension. At successive points in time the function `merge` combines the ordered sub-sequences that are being accumulated in the `v` dimension by previous applications of the `merge` function. As the algorithm iterates through time, the sorted data are accumulated in the `v` dimension.

```
  rOt8.v,time( output )
  where
    dimension v,h,t;
    output = mergeSort asa.t  inputSize == sizeOfMerge;
    merge(x,y) = if iseod xx then yy
                    else if iseod yy then xx
                         else if xx <= yy then xx else yy fi fi fi
      where
                      xx = x upon.v iseod yy || xx <= yy;
                      yy = x upon.v iseod xx || xx > yy;
                    end;
    mergeSort = Inputdata(h) fby.t
                    merge(leftChild.h(mergeSort), rightChild.h(mergeSort));
    sizeOfMerge = 1 fby.t 2*sizeOfMerge;
    rOt8.a,b( x ) = x @.a b;
  end
```

The program computes values of `output` over time by rotating (using `rOt8`) the values of `output` which are computed over dimension `v`. In particular, the value of the program computed at time t is the value of `output` and hence `mergeSort asa inputSize == sizeOfMerge` at `t`-context 0 and `v`-dimension-context t. The expression `mergeSort asa inputSize == sizeOfMerge` is evaluated as follows: the expression `inputSize == sizeOfMerge` is evaluated from `t`-context 0 until equality holds at which time `mergeSort` is evaluated.

The variable `mergeSort` is defined over context `t`. It is initially the values of `Inputdata(h)` defined over dimension `h`. (Note that in dimension `v`, values of `Inputdata(h)` at all but the initial `v`-context is assumed to be `eod` (or end of data).) At some later `t`-dimension context (say T), the value of `mergeSort` for a particular `h`-dimension-context (say i) is obtained by applying function `merge` to the appropriate values of `mergeSort` at the previous time ($T-1$). (The appropriate values would be at `h`-dimension contexts $2i$ and $2i+1$ as computed by `leftChild` and `rightChild`.) Function `merge` essentially takes two sorted sub-sequences in the `v` dimension and produces a single sorted sequence in the `v` dimension for a particular point in the `h` dimension.

Assuming the value of `inputSize` is n, `mergesort` initially consists of n singleton sequences across dimension `h`, and half-as-many sequences of twice the size at each subsequent `t`-dimension context, and eventually a single sequence of length n in dimension `v` when `inputSize` is the same as the value of `sizeOfMerge` ($\log_2(n)$).

6.3 Block Matrix Multiplication

The recursive block decomposition algorithm for matrix multiplication can be expressed as follows:- Let A and B be two n by n matrices where n is a power of two (without loss of generality). We can divide each of A and B into four $n/2$ by $n/2$ matrices and express the product of A and B in terms of these $n/2$ by $n/2$ matrices. Product matrix C (also n by n) can be obtained according to the following formula.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$C_{11} = A_{11}B_{11} + A_{12}B_{21},$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22},$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21},$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

The Indexical Lucid program that embodies this algorithm is given below. The unusual aspect of this program is that its intensional realization of the extensional divide-and-conquer basis of the algorithm, this being possible because of multidimensionality.

The matrices to be multiplied, `A` and `B` are of order given by `n` and the elements of the matrices are stored in dimensions `aux_i` and `aux_j` corresponding to row and column respectively. The desired result is a particular element of the product matrix at row and column position given by `r` and `c`. Function `matmult` performs the multiplication and it uses two dimensionally abstract functions in doing so: `rOt8.a.b` for rotating dimensions (i.e., dimension `a` becomes dimension `b`) and `append.dim` that "appends" its second argument to its first argument of size given by the third argument, with the first two arguments assumed to vary in dimension `dim`.

```
( matmult( A, B, n ) @.aux_i r ) @.aux_j c where
  dimension aux_i, aux_j;
  matmult( aux_a, aux_b, n ) = c asa.t  s == (n*n*n) where
    dimension t, i, j, k;
    s = 1 fby.t 8*s;
    d = 1 fby.t 2*d;
```

```
      a = r0t8.aux_i.i( r0t8.aux_j.j( aux_a ) );
      b = r0t8.aux_i.i( r0t8.aux_j.j( aux_b ) );
      c = (a @.j k) * (b @.i k)
          fby.t
          ( ( reduce @.k (2*k) ) @.j (2*j) ) @.i (2*i) where
            reduce = block( c + next.k c,
                            next.j c + next.k next.j c,
                            next.i c + next.k next.i c,
                            next.j next.i c + next.k next.j next.i c,
                            d
                          );
            block( tl, tr, bl, br, d ) = p where
              p = append.aux_i( append.aux_j( tl, tr, d ),
                                append.aux_j( bl, br, d ),
                                d
                              );
              append.dim( a, b, d ) = if( dim < d ) then a
                                      else b @.dim (dim-d) fi;
            end;
          end;
          r0t8.a.b( x ) = x @.a b;
      end;
    end
```

Function `matmult` introduces four dimensions:- `i`, `j`, and `k` as the row, column and inner product dimensions over which matrix blocks are defined, each of whose elements are defined over the orthogonal and outer dimensions `aux_i` and `aux_j`; and `t` as the iteration dimension over which product matrix is iteratively built starting with singleton matrices.

Expression `c asa.t s == (n*n*n)` captures at which iteration the product matrix is built, i.e., in $\log_2(n^3)$ iterations. Variable `c` can be thought of as a 3-dimensional array in dimensions `i`, `j`, and `k`, each dimension of size given by `n` although it is not such a monolithic object. Initially, (when `t` is 0) the each element of `c` identified by (i, j, k) is the product of element (i, k) of `a` and element (k, j) of `b`. Variables `a` and `b` are equivalent to `A` and `B` except that they vary in dimensions `i` and `j` instead of dimensions `aux_i` and `aux_j`, this being achieved using the `r0t8` operator. At each successive iteration (iteration index given by `t`), variable `c` can be thought of as a 3-dimensional array with 1/8th the number of elements in the dimensions `i`, `j`, and `k` except that each element can be thought of as a 2-dimensional array in dimensions `aux_i` and `aux_j` of four times as many elements as in the previous iteration as per the formula given above. This building of matrix blocks in dimensions `aux_i` and `aux_j` is achieved by using function `block` which is defined in terms of `append`. When $log_2(n^3)$ iterations have been completed, the size of the 3-dimensional array that `c` can be thought of as denoting is precisely 1 and the size of the 2-dimensional array, which represents the matrix product, it denotes in the `aux_i` and `aux_j` dimensions is the same as that of matrices `A` and `B`.

It is worth reiterating that while we have referred to variables as denoting monolithic objects of various dimensionality, this is simply meant to be a mental device. In actuality, all objects are defined and manipulated elementwise.

7 Conclusions

Indexical Lucid, Lucid circa 1993, enables the programmer to implicitly and intensionally define and manipulate multidimensional data structures that change. The features of the language that make this possible and natural are user-defined dimensions and dimensional abstraction.

Being able to define dimensionality in Indexical Lucid enables the programmer to naturally express solutions to diverse multidimensional problems. We have illustrated this by expressing three dissimilar applications in Indexical Lucid: prime number generation, mergesort, and matrix multiplication. Being able to define dimensionally-abstract functions in Indexical Lucid enables the programmer not only to write succinct programs but also to reuse these functions in diverse applications.

One of the important practical consequences of Indexical Lucid is that it expresses parallelism succinctly — an aspect of the language we have not discussed in this report. In fact, Indexical Lucid is the parallel composition language of GLU, a coarse-grained system for programming conventional parallel computers [8]. Through GLU, it has been extensively used in implicitly expressing parallelism in applications ranging from scientific computations to graphics to real-time processing.

References

[1] E.A. Ashcroft. Ferds – massive parallelism in Lucid. In *Proc. 1985 Phoenix Computer and Communications Conference*, pages 16–21. IEEE, March 1985.

[2] E.A. Ashcroft and W.W. Wadge. Lucid - a formal system for writing and proving programs. *SIAM Journal on Computing*, 5(3):336–354, September 1976.

[3] E.A. Ashcroft and W.W. Wadge. Lucid, a nonprocedural language with iteration. *CACM*, 20(7):519–526, 1977.

[4] E.A. Ashcroft and W.W. Wadge. The syntax and semantics of Lucid. Technical Report CSL-146, SRI International, Menlo Park, CA 94025, 1984. Computer Science Laboratory.

[5] W-C. Du. *Indexical Programming*. PhD thesis, University of Victoria, Victoria, B.C., Canada, 1991. Department of Computer Science.

[6] A.A. Faustini, S.G. Matthews, and A.A.G. Yaghi. The pLucid Programmer's Manual. Technical Report TR84-004, Arizona State University, Computer Science Department, Tempe, Arizona 85287, U.S.A., 1984.

[7] A.A. Faustini and W.W. Wadge. Intensional programming. In J.C. Boudreaux, B.W. Hamill, and R. Jernigan, editors, *The Role of Languages in Problem Solving 2*. Elsevier Science Publishers B.V. (North-Holland), 1987.

[8] R. Jagannathan and A.A. Faustini. The GLU Programming Language. Technical Report SRI-CSL-90-11, SRI International, Computer Science Laboratory, Menlo Park, California 94025, USA, 1990.

[9] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.

[10] B.K. Szymanski, editor. *An Intensional Parallel Processing Language for Applications Programming*, chapter 2. ACM Press, 1991.

[11] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press U.K., 1985.