

# Multidimensional Declarative Programming: The New Lucid of The New Book

Ed Ashcroft\*  
Computer Science and Engineering Department  
Arizona State University

## Abstract

A second Lucid book has been written. The authors are Ed Ashcroft, Tony Faustini, Jaggan Jagannathan, and Bill Wadge. It will be in bookstores in late December 1994 or early January 1995. The publisher is Oxford University Press. The title is "Multidimensional, Declarative Programming." Even though the title doesn't explicitly mention Lucid, the book describes the latest version of Lucid, the version that is being used by GLU. This version is going to be called "Lucid." (Surprise, surprise.)

The new version is similar to, but not a superset of, Original Lucid (by which we mean the Lucid of the first book). One of the purposes of this paper is to point out how the languages differ and why. Most of the differences have been foreshadowed earlier, particularly in my paper "Lewplewce" ("Lu+"), in ISLIP 93. In this paper, besides describing the language, I will describe ways of looking at programs in the language that simplify, I believe, full understanding of programs, proofs of their properties, and details of their implementation.

## 1 Introduction

The book is concerned with a programming paradigm (intensionality, of course) in which the existence of different dimensions, and even temporary dimensions, is natural. The latest Lucid uses dimensions in this way.

Lucid has been around for a long time. It was originally conceived twenty years ago, by Bill Wadge and myself. Originally, only one dimension was involved, and that dimension was implicit. In informal discussions we tended to refer to it as "time," but the dimension `time` never occurred in programs. Now, it *may*. It also may not, but not because it is implicit (it isn't) but because we might happen to be using only dimensions other than time. Here is such a simple (new) Lucid program that doesn't involve `time` or "time" at all.

---

\* Supported by a grant from the NSF

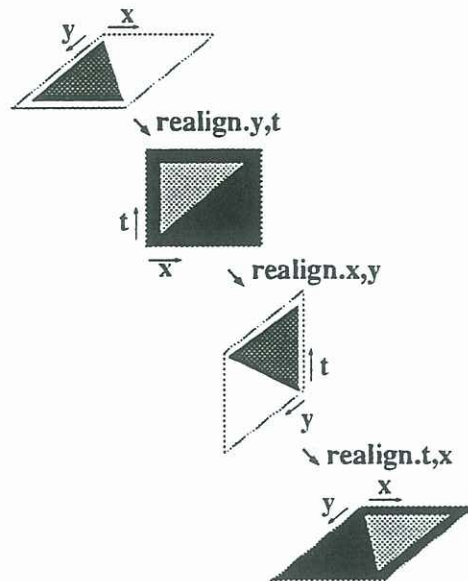


Figure 1: Matrix Transposition

## 2 An Example

```

realign.t,x ( realign.x,y ( realign.y,t ( A ) ) )
  where
    dimension t;
    realign.u,v(T) = T @.u #.v;
  end

```

This is a program for transposing the matrix  $A$ , which we assume varies in dimensions  $x$  and  $y$ . The algorithm we use can best be thought of geometrically. We start off with the matrix in the horizontal plane, say, with the  $x$  dimension going to the right and the  $y$  dimension coming towards us. (For concreteness, get hold of a micro floppy disc or something similar.) Now, the disk (matrix) is tipped up, away from us, into a new dimension,  $t$  say, which we assume goes up vertically. The disk is now facing away from us, standing on the edge that used to be furthest from us.

Continuing, we now rotate the disk towards us about its leftmost edge, resulting in the disk facing to the right and still standing on the same edge. Finally, we rotate the disk rightwards about the edge on which it is standing, so that it lies horizontally and is the transpose of the original horizontal disk, as though it had been flipped over while maintaining the back-left/front-right diagonal in its original position. (See Figure 1.)

This program probably appears a little unusual as a program introducing the new Lucid: it contains no Lucid operations like `first`, `next` or `fby`! In fact, all it

contains are three applications of the function `realign` and the actual definition of `realign`. The thing to notice is that the function is defined in terms of two dummy dimensions, `u` and `v`, and that whenever the function is used actual dimensions are specified. The definition of the function is “dimensionally abstracted,” and when the function is used it is “focussed upon” actual dimensions. The formal parameter of `realign` is `T` and the formal dimensions are `u` and `v`. The term `realign.x,t(A)` focusses on dimensions `x` and `t`, realigning the actual parameter, `A`, so that the elements that used to be aligned in the `x` dimension are aligned in the `t` dimension. (The values of `A` are not changed or realigned; `realign` produces values that are realigned versions of the elements of `A`.) In a while we will see how `realign` works. First, we will spend some time discussing *arity*.

As you can see, `realign` needs an argument (`A`, for example) and has to be focussed on two dimensions (`x` and `t`, for example). The argument has to be a term, while the dimensions to focussed on have to be represented by alphanumeric strings. (If we want to, we can refer to dimensions 1, 2, and 3 as well as to dimensions `x`, `y`, and `t`.) This will be true throughout programs. Every function and operation in the language takes arguments and dimensions, as do variables and constants. These four basic types of entities, functions, variables, operations, and constants, we will call *atoms*.

### 3 Arity

All atoms have two *arities*, which we will call their Roman and Greek arities. (The names “Roman” and “Greek” are used to allude to the Roman and Greek languages, not to any particular qualities of the Roman and Greek peoples.) Greek arities, when they have a name, end in “adic”, while Roman arities, when they have a name, end in “ary.” For example, some small Roman arities are “unary,” “binary,” and “ternary.” (These clearly are Roman-arity 1, Roman-arity 2, and Roman-arity 3, respectively.) Corresponding small Greek arities are “monadic,” “dyadic,” and “triadic.”

The Roman arity indicates the number of arguments the atom takes, in the usual way. For example, we would say that a function is unary if it takes one argument, and that an operation is ternary if it takes three arguments. (The arguments of functions are called actual parameters, while the arguments of operations are called operands.)

The Greek arity indicates the number of dimensions that the atom needs to be focussed on. For example, we would say that a function is monadic if one dimension needs to be focussed on, and that an operation is dyadic if two dimensions need to be focussed on.

We thus see that the atom `realign` is a unary and dyadic function, because it is defined to take one argument (indicated in the definition by a single formal parameter) and it requires the specification of two dimensions to focus on (indicated in the definition by a sequence of two formal dimensions). Applying this terminology to Original Lucid, we see that the atom `fby` is a binary, monadic operation and the



atom `next` is a unary, monadic operation. The atom `+` is a binary, nulladic operation. (Note that a *nullary* atom takes no arguments and we use the invented terminology *nulladic* to describe an atom that needs no dimensions specified.) The individual variables in Original Lucid programs are nullary, nulladic functions.

Nullary functions are exactly the variables, and nullary operations are exactly the constants. (We therefore could have avoided introducing variables and constants as syntactic entities at all, but it doesn't seem worth the minor syntactic simplification of doing so. We will see that it is actually often very useful to be able to use the terms "variable" and "constant." In fact, because we are going to use the terms "variable" and "constant", whenever we say "function" and "operation" it can be assumed that they are *not* nullary, unless we have explicitly stated otherwise.

It should always be possible to determine the arities of an atom by inspection, as explained below.

Constants are always nullary, and are usually nulladic. (Examples are `3`, `true`, and `eod`.) The main example of a constant that is not nulladic is `#`; it is monadic, and its behavior and purpose will be explained later. Basic operations like `+` and `if then else fi` are always nulladic. What used to be called the Lucid operations, such as `first`, `next`, `fby`, and `wvr`, are all monadic. At the moment (September 1994), there are no dyadic, triadic, etc., operations, but reclassifying `realign` as a (dyadic) operation may be in the works.

The arities of a variable or function can be determined by looking at the left-hand-side of its definition, if it has one. (Every *function* that is used must have a definition, but some *variables* may not.) The number of formal parameters is the Roman arity and the number of formal dimensions is the Greek arity.

Every variable that is used but has no definition must be either a formal parameter or an input variable. In both cases, it will be nulladic.

The Greek arity of a variable, say, has nothing to do with the dimensionality, or rank, of the value the variable denotes. We will discuss "rank" later.

We are now ready to return to the example program.

## 4 The Function `realign`

The definition of `realign` is very simple:

```
realign.u,v(T) = T @.u #.v;
```

The binary operation `@` ("at") is monadic. It has to be focussed on the dimension from which we are going to get the values "at" particular points. Here, it is focussed on the dummy dimension `u` (one of the formal dimensions of the definition of `realign`).

The monadic constant `#` simply returns the current position in the dimension on which it is focussed (here, `v`). It is the generalization to multidimensions of the Original Lucid constant `index`. (We could have decided to use `index.u`. It is just a question of taste. In fact, we can avoid using `#` in many cases by defining a

nulladic variable that has the same effect as the monadic constant applied to a particular dimension. It has been proposed that we use the variable with the same name as the dimension for this purpose; that is, instead of  $\# . u$  we just use  $u$ . To do this we must have a place for the following definition

$$u = \# . u;$$

This is difficult if the occurrence of  $\# . u$  we want to replace is not within a where clause.)

Individually the pieces are simple. How can they possibly do the job? Let us assume that  $T$  varies in dimensions  $u$  and  $w$ . The rank analysis that we are going to look at shortly tells us that the term  $T @ . u \# . v$  varies in dimensions  $v$  and  $w$ , and not in dimension  $u$ . If we want the value of the term at point  $i$  in dimension  $v$  and point  $j$  in dimension  $w$ , the value of  $\# . v$  is  $i$  and the desired value is therefore the value of  $T$  at point  $i$  in dimension  $u$  and point  $j$  in dimension  $w$ . The term really does denote the realignment of  $T$ !

## 5 Temporary Dimensions

The program for transposing (flipping over) the matrix  $A$  needs to use a dimension different from both dimension  $x$  and dimension  $y$ . The declaration `dimension t;` in the where clause introduces a new dimension  $t$  that is orthogonal to all other dimensions. Rank analysis of the subject of the where clause shows us that the subject does not vary in the new dimension, so dimension  $t$  was just temporary. (If the subject did vary in dimension  $t$ , the value returned would be the value of the subject at point 0 in dimension  $t$ , so, again, the value returned does not vary in the new dimension and the dimension itself was just used temporarily.)

## 6 Syntax

For completeness, we will give the abstract syntax of the new `Lucid`, together with a little of the concrete syntax.

### 6.1 Abstract Syntax

A *program* is a term.

A *term* is a where clause of size 1 or it is an atom, together with actual dimensions and arguments. In the latter case, if the atom has Greek-arity  $i$  and Roman-arity  $j$ , the term will have a length  $i$  sequence of dimension names (the actual dimensions) and a length  $j$  sequence of terms (the arguments). The arguments are called *actual parameters* if the atom is a function (and the term itself is a *function call*), and they are called *operands* if the atom is an operation.



A *where clause of size  $n$*  is an  $n$ -tuple of terms, called the *subject* of the clause, a sequence of dimension names, and a set of definitions (of various sizes) of distinct variables and functions. The sequence of dimension names forms a *declaration*, a declaration that those dimensions are new and temporary. Together, the sequence of dimension names and the set of definitions form the *body* of the where clause.

A *definition of size  $n$*  consists of a left-hand-side and a right-hand-side. The *right-hand-side* is simpler, so we will explain it first; it is simply a where clause of size  $n$  or, provided  $n$  is 1, a term. The *left-hand-side* consists of an  $n$ -tuple of distinct functions or variables (which we will call the *siblings* of the definition if  $n$  is greater than 1), all with the same arities, both Greek and Roman (which we will call the Greek and Roman arities of the definition), and two sequences: a sequence of distinct dimension names (the *formal dimensions*) and a sequence of distinct nulladic variables (called the *formal parameters*). The lengths of the two sequences must match the Greek and Roman arities of the definition, respectively.

## 6.2 Concrete Syntax

A program is just a particular term.

When describing the concrete syntax for terms, we first will look at the simpler of the two cases, which is when the term in question is just an atom applied to other terms, focussed, perhaps, onto particular dimensions. The “other terms” are the arguments of the atom. If the atom is a function, it is written prefix (with parentheses), and if it is an operation it is written infix. The “particular dimensions” are the actual dimensions of the atom. This sequence of actual dimensions follows the atom, with a period between the two. (The period may be missing, of course, if there are no actual dimensions.)

The more complicated of the two cases for terms is when the term in question is a where clause of size 1.

In a where clause, the subject comes before the body. The body of a where clause begins with the keyword *where* and ends with the keyword *end*. The declarations in the body come before the definitions, and, since they are declaring new dimensions, they begin with the keyword *dimension*. They are terminated with semicolons.

The  $n$ -tuples of terms that are subjects of where clauses of size  $n$ , and the  $n$ -tuples of distinct variables in the left-hand-sides of definitions of size  $n$ , will be enclosed in braces (unless  $n$  happens to be 1).

In definitions, the left-hand-side and right-hand-side are separated by  $=$ . They thus, deliberately, look like equations.

In function calls or in the left-hand-sides of function definitions, the sequence of actual or formal dimensions comes before the sequence of actual or formal parameters. A useful mnemonic for this is “the Greeks came before the Romans.” In a left-hand-side that contains a sequence of formal dimensions, there is a period separating the  $n$ -tuple of variables or functions from the sequence of formal dimensions, unless that sequence is empty.

“*Scope*” Note: The functions and variables that are defined in the body of a where clause are “local” to that where clause; the definitions apply to both the subject and the body (but not outside). The same is also true of the new dimension names that are declared in the body. The formal dimensions and formal parameters used in the left-hand-side of the definition of a variable or function can only be referred to in the right-hand-side of the definition.

There is no automatic run-time renaming of new dimensions to avoid clashes with existing dimensions, but users can almost always consistently rename new dimensions statically to avoid such run-time clashes if they wish. Here is an example that cannot be so renamed:

```
f.c(#.c)
  where
    f.e(x) = realign.d,e(y)
              where
                dimension d;
                y = x fby.d f.d(next.e y);
              end;
    realign.u,v(T) = T @.u #.v;
  end
```

The problem is that more and more dimensions called *d* will be set up. (Notice that the recursive call of *f* is focussed on the latest dimension *d*.)

This program is not defined to do anything useful but it is interesting that it does do *something*, nontrivially. Finding a program that makes recursive calls like this but actually *does* something useful is an interesting challenge.

## 7 Incompatibilities with Original Lucid

Several features of Original Lucid are no longer supported.

### 7.1 Implicitness of Dimensions

The most obvious is the implicitness in Original Lucid of the dimension that Lucid operators are working in. We can no longer say

```
x = 1 fby x+1;
```

We have to say

```
x = 1 fby.time x+1;
```

or

```
x = 1 fby.d x+1;
```

(say).

We can even say

```
x.a = 1 fby.a x+1;
```

so that we can get the effects of the previous two definitions of `x` by simply using `x.time` and `x.d`.

Whatever we do, we always have to make it clear what dimension we are working in. (Several ways have been proposed for having dimensions implicit in some circumstances, but we have found no generally agreed-upon way of doing that. Consequently, no implicitness of dimensions is allowed at the moment.)

## 7.2 Subcomputations

In the new Lucid, temporary dimensions are used to set up subcomputations. In Original Lucid, subcomputations were set up using `is current` declarations. Such declarations are no longer in the language.

Here is a typical Original Lucid program using `is current` which raises `x` to the `n`-th power by repeated multiplication:

```
p asa index eq N
  where
    N is current n;
    X is current x;
    p = 1 fby X * p;
  end
```

This is how the same program would now appear:

```
p asa.d #.d eq n
  where
    dimension d;
    p = 1 fby.d x * p;
  end
```

In Original Lucid, we have to say which things are frozen (here, `n` and `x`). Everything else (there *is* nothing else here) is not frozen. In the new Lucid, we introduce a new dimension (`d`), and when we define something in that dimension (such as `p`) whenever we refer to things we had before (such as `x` and `n`) they are automatically frozen because they do not vary in the new dimension. The program is therefore simpler because we do not have to explicitly freeze anything.

If, however, we want to refer to something that is not frozen, in the new Lucid we have to explicitly unfreeze it. In fact, what we do is restart it in the new dimension, using `realign`! For example, here is a program that returns the first value of `y` that is greater than twice the average absolute divergence from the mean. We first will give it in Original Lucid and then in the new Lucid.



```

y asa y > 2*(avDiv
  where
    Y is current y;
    avDiv = avg(abs(Y - avg(y)));
  end)
where
  avg(A) = s/(index + 1)
  where
    s = A fby s + A;
  end;
end

```

In new Lucid, we assume that  $y$  varies in dimension time.

```

y asa.time
y > 2*realign.d,time(avDiv)
  where
    dimension d;
    avDiv = avg.d(abs(y -
      avg.d(realign.time,d(y))));
  end)
where
  avg.e(A) = s/(#.e + 1)
  where
    s = A fby.e s + A;
  end;
end

```

In the new Lucid case, the program appears more complicated, because of the use of `realign`, but it can be argued that `realign` is being used exactly in those places where something tricky is going on; the Original Lucid program glosses over the trickiness.

The point conveyed by this section is that everything we did before can be done in the new Lucid, and a simple translation is possible from programs in Original Lucid to the new Lucid.

## 8 Rank

Now that we have many dimensions, and even temporary dimensions, the question of rank, i.e., *which* dimensions, naturally arises. The question seems to be somewhat complicated by Lucid's serendipity. An object that was defined to have a certain number of dimensions can be used in a context where it is expected to have more dimensions. The object seemingly automatically extends to the extra dimensions. This is all a consequence of the educative evaluation strategy. As far as rank is concerned, the

effect seems to be that there is no absolute rank for any object, there is just a *minimal* rank. In the semantics (see the next paper, by Bill Wadge), ranks of objects are denoted by equivalence classes.

Personally, I think the better way of looking at this is to say that objects have definite ranks and, when an object is used in a context that requires it to have a larger rank, the evaluation mechanism detects this rank disparity and throws away the contexts for the extra dimensions, so that the object is demanded only in the sorts of contexts it expects to be demanded in. Thus, this evaluation mechanism simplifies the storage of objects, because the associative memory uses shorter tags (i.e., fewer dimensions).

To make all this work, the evaluation mechanism needs to know the absolute rank of objects. To do this, we need the ranks of the terms defining objects. We will therefore concentrate now on defining the ranks of terms. (There may be some inconsistencies with what I am about to say. Some cases may not work. If so, I would like to think that this is a Naive Semantics for *Lucid*, in the sense of Naive Set Theory: it cannot handle pathological cases.)

The rank of a term or atom will be a set of dimension names. The rank of a term is defined in terms of the ranks of the variables used in the term, in the following way. (For simplicity, we will not here consider terms that contain function symbols or where clauses, the first because they are a minor complication and the second because they are a major complication.)

The terms we are considering must be atoms applied to other terms. If the atom in question is a variable, the rank of the term is the rank of the variable. Otherwise the atom must be an operation. If that operation is nulladic, the rank of the term is the union of the ranks of its operands. The only other operations are the so-called *Lucid* operations, and we will consider them separately:

If the term is `first.e A`, the rank is  $rank(A) - \{e\}$ .

If the term is `next.e A`, the rank is the union of  $rank(A)$  and  $\{e\}$ .

If the term is `#.e`, the rank is  $\{e\}$ .

If the term is `B @.e A`, the rank is the union of  $rank(A)$  and  $(rank(B) - \{e\})$ .

If the term is `B fby.e A`, the rank is the union of  $rank(A)$ ,  $rank(B)$ , and  $\{e\}$ .

If the term is `B asa.e A`, the rank is  $(\text{union of } rank(A) \text{ and } rank(B)) - \{e\}$ .

If the term is `B wvr.e A`, the rank is the union of  $rank(A)$ ,  $rank(B)$ , and  $\{e\}$ .

If the term is `B upon.e A`, the rank is the union of  $rank(A)$ ,  $rank(B)$ , and  $\{e\}$ .

Using these rules it is easy to verify the ranks in the earlier *transpose* program. For example, we can see that  $rank(T @.u \#.v)$  is  $\{v, w\}$  if  $rank(T)$  is  $\{u, w\}$ .

Being aware of rank is useful in program verification (for example, certain properties of `realign.x, y (A)` only follow if  $y$  is not a member of  $rank(A)$ , which makes sense if you think about it). Also, it is interesting that the problems with the usage count storage management scheme all seem to come from the rank disparity mentioned earlier, and I think that recognizing this fact will lead to those problems being solved.

---

## 9 Conclusion

LUCID has not changed very much syntactically since the first book, and the semantic differences are also relatively minor. Nevertheless, the language is very different in terms of what can be expressed. It has evolved in a completely different way than other languages have, and has a completely different model of computation and implementation. The message of this paper is: Read The Second Book.