

# Observations on Spreadsheet Languages, Intension and Dataflow

Alan G. Yoder and David L. Cohn

*University of Notre Dame*

## Abstract

Spreadsheet languages for distributed computing are of great interest because they unite ease of use with the ability to express parallel computations. This paper discusses some issues that these languages raise, in the context of dataflow and intensional programming languages such as Lucid. First, proper design and implementation of spreadsheet languages (as we see it) places stress on both the educative and data-driven dataflow models of computation. Second, spreadsheets feature both relative and what we have called implicit addressing, which are both forms of intension; we compare these to the intensionality found in Lucid. Third, since spreadsheets are often used by unsophisticated users, the abstract reasoning skills taken for granted by scientists cannot be assumed. This elevates syntax in importance. We therefore propose an extended multi-dimensional array notation which unites simplicity, intensionality, concreteness and historical usage.

## 1. Introduction

Spreadsheet languages for distributed computing are a principal research interest for us [YC 94]. Since they are, in the main, declarative languages, they are natural candidates for parallelization; each cell can in principle be assigned to a dedicated processing unit and calculated concurrently with all other cells. This, coupled with the enormous popularity and ease of use of modern spreadsheets, gives the lie to the conventional wisdom that parallel programming is always hard.

This observation has lead us and others to attempt to design languages based on the spreadsheet metaphor which do not incorporate the numerous weaknesses which are evident in Lotus 1-2-3 and its cousins such as Microsoft Excel and Borland Quattro<sup>1</sup>. In this paper, we call these latter applications *modern spreadsheets* and call bona-fide languages which make use of these concepts *spreadsheet languages*. The generic term *spreadsheet* refers to an “instance”, a “program in operation” or “open spreadsheet file” of either type.

Properly evaluating spreadsheets in a parallel manner is non-trivial. The principles are pretty simple: first of all, one wants to avoid unnecessary recomputation. This leads immediately to the idea that *cells* (the unit of computation in spreadsheets) should be recalculated in topological order of their dependencies. Secondly, one wants to maximize use of available computational resources. These two concerns have influenced the design of the Lucid dataflow language [AFJ 91]. Du and Wadge have made use of this correspondence by designing a spreadsheet around an extension of Lucid called Plane Lucid [DW 90a], [DW 90b]. This spreadsheet uses the principle of *education* to order calculations [Ash 86]. Section 3 discusses our perspective on this approach; section 4 outlines the alternative method of eager evaluation. We believe a hybrid approach is necessary; section 5 is devoted to this.

Another issue in spreadsheet languages is the question of syntax. Modern spreadsheets usually use one of two addressing schemes: an alphabetic/numeric hybrid notation or a row/column specification notation. Du and Wadge, on the other hand, use a syntax derived from Lucid. None of these is in much accord with historical mathematical array notation, which most will agree is concise and readable. Section 6 proposes an extension to standard mathematical notation which provides for the concepts of relative addressing (introduced by Visicalc), *implicit* addressing (both implicit and relative addressing are a form of intension), *sub-range* addressing, dimensional addressing [DW 90a] (which can be interpreted as a variant of subrange addressing) and *meta-dimensional* addressing. We also show how many of the elegant semantic constructs found in Lucid can be translated into the spreadsheet idiom using this notation.

1. All trademarks mentioned herein are the property of their respective owners.



## 2. Education

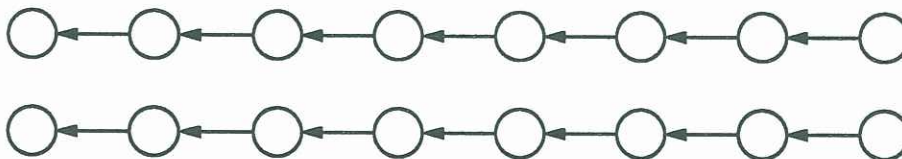
Education is a term used to describe a tagged demand-driven model of dataflow computation [Ash 86]. Data items are not submitted for calculation until a demand is made for them by the user. Calculating these items may in turn cause other data items to be needed, so they in turn are demanded. No item is actually calculated until all of the data items upon which it is dependent are available. The demands naturally propagate through the dependency graph rooted at the originally demanded item, and since items are not recalculated if they are “available”, no work is duplicated.

In the context of spreadsheet languages the concept of education must deal with the fact that multiple items are “demanded” at once by the user. This is because multiple cells are normally displayed at once on the screen, and scrolling the spreadsheet causes the values in *all* the cells being scrolled into view to be “demanded”. This is not a problem—it is easy to see that no computation will be duplicated if the spreadsheet engine simply iterates through the cells being displayed. Suppose A and B are both cells on the same screen display; if the dependency tree of cell B happens to be a subtree of the dependency tree of cell A, then cell B will already be computed if A’s tree is done first, otherwise B will be available for A’s computation if B is done first. In fact, this is merely a generalization from the case in which the dependency graph has one root to the case in which it is a forest.

The principle of education is therefore a useful one for spreadsheet languages—up to a point. To understand its limitations, imagine the following scenario. A user opens a spreadsheet file used for analysis of a variant of the DES algorithm. Recall that the canonical algorithm subjects the data to be encrypted to a series of a dozen reversible transformations based on a symmetric key. A single row in the spreadsheet therefore might look something like this:

datum	result of iteration 1 (depends on datum)	result of iteration2 (depends on iteration 1)	...
-------	---	--	-----

Let’s say that a single screenfull of this spreadsheet consists of 20 lines and 8 columns. Each of the cells in the rightmost column depends on the cell to its left, which depends on the cell to its left and so on to the leftmost cell of the screen which contains the datum to be encrypted for that row, so the dependency forest for two rows of these cells looks like this:



You can see that bringing up a screen will result in a “demand” by the user for the values in 160 cells of the spreadsheet, 140 of which have a dependency on the cells to their left. Under the educative regime, none of these cells will be recalculated twice, which sounds like a clear win on first blush.

But suppose our user (named Sally, say) is interested in the 100’t<sup>h</sup> row of the file. Sally hits the PgDn key several times to get to the region of interest. If any time is involved in the recalculation she might prefer that the first 80 or so rows of cells not be calculated at all! Putting it another way, Sally will expect that if she loses interest in waiting on a current screen to display, scrolling away from it will cause the engine to *instantly* also lose interest in recalculating the values for the cells no longer being displayed on the screen.

This is the unfortunate reality. When a user scrolls in a spreadsheet, the normal expectation is that the cells just off the screen will be displayed faster than you can blink, thank you very much. But this only works in a demand-driven environment so long as the cells being scrolled into the display area are already up to date or easy to bring up to date. In our current example, this may not be true, because each cell depends on the value to its left and is somewhat compute-intensive.

A band-aid that springs to mind is to delay submission of the “demand” for a fraction of a second to see if the user has another scroll request to make. In our example, Sally has several more scrolling requests to make, and the cells on screen could perhaps be displayed by just drawing their outlines and ignoring their



values. Unfortunately, the spreadsheet engine has no way of knowing when Sally will submit consecutive scrolling requests, and any delay is expressly against her wish to have the cells displayed instantly. So this band-aid probably won't work.

### 3. Data-driven calculation

The above sounds like a criticism of the demand-driven dataflow model. It is not. Our example fares no better under a pure data-driven model of dataflow computation. Applying this model to a spreadsheet engine, a cell is ready to be recalculated as soon as all the other cells upon whose value it depends have been recalculated. A moment's thought will convince you that this corresponds to eager recalculation of cells in topological order.

Pure data-driven recalculation could be a disaster in a large spreadsheet. The user might be interested in an area to one side of the topological sort tree while the engine, mindlessly calculating according to the sort order, filled in the other side of the tree.

In our example, things are not that bad, but are still not good. A topological sort of the spreadsheet will yield a forest, the root of each tree being one of the cells on the leftmost row of the spreadsheet. Which tree should be recalculated first? The data-driven model gives us no guidance beyond the same idea we've already had, which is to use the cells currently being displayed as a guide to formulating a recalculation priority.

On the plus side, it's clear that eager recalculation does offer a benefit—most workstations and PC's have a lot of idle time which can be used to calculate the values of cells in an open spreadsheet in advance of the time the user "demands" them. We need a way of deciding which cells to work on.

### 4. The hybrid approach

A mixed approach seems best. The strengths of education and topological recalculation must be merged, and then augmented to allow cancellation of calculation requests. The first part of this task, adding eagerness to education, is discussed in another talk in this symposium [Jag 94]; Jagannathan's task is complicated by the fact that Lucid features infinite temporal sequences of values, only some of which should be calculated. It is possible to build spreadsheet languages which have this feature, but most will not; the concreteness of the spreadsheet view of the world is part of its appeal. We concentrate here on a protocol for spreadsheet languages in which the dependencies are bounded; it maximizes responsiveness while maintaining worst-case optimality.

To make a spreadsheet engine work according to the user's expectations, we must have a way of cancelling current "demands" if the user so desires. A simple way of doing this involves using an educative recalculation engine capable of assigning a session-unique identifier to each "demand", and tagging every recalculation request generated by the demand with the same identifier. Assuming the engine keeps a queue of pending work prioritized by "demand ID", it is a simple matter to delete all requests with a given tag (in a lazy manner) if the user scrolls out of the screen area which generated it. Of course, if a language such as Lucid is used to implement this engine, this capability must be present in it.

This educative method with cancellation is both optimal and maximally responsive in the case that we are suffering from a paucity of computing resources. But most modern workstation and PC clusters represent a huge unused compute resource, so a distributed spreadsheet implementation should try to use eager evaluation, when possible, to calculate values we may wish to see in advance of when we express this desire by pulling them up on the screen.

What spreadsheet users want is a spreadsheet which displays enough information about each cell to enable us to navigate the spreadsheet, while calculating completely all those cells which truly interest us. This perfection is unattainable, so we propose a reasonable approximation:

- The recalculation engine determines the dependency forest by performing a topological sort on the current spreadsheet.
- The engine submits a low-priority "demand" consisting of the roots of all the trees in the topological sort forest. This is eager evaluation; the engine can proceed with this so long as there is no more pressing business at hand. Sophisticated engines might interact with the operating system to lower



their own process priorities during eager evaluation, or use a separate low-priority process for this work.

- Any time the user submits a command such as a scrolling request or changing the value of a cell, a new high-priority “demand” is made to the engine, with every cell changed or exposed to view by the scrolling request being tagged with the same new unique demand ID. The engine satisfies all high-priority demands before returning to work on the low-priority demands generated by eager evaluation.
- Subsequent scrolling requests cause those recalculation requests with the current demand ID still remaining in the engine’s recalculation queue to be deleted, and new requests to be submitted for those cells in the new viewing area.

To summarize, neither demand-driven nor data-driven dataflow really satisfies the needs of a spreadsheet user interface. A mixed approach is the result.

## 5. Array syntax for intensional and multi-dimensional uses

When we discuss spreadsheet languages, we use the term *cell* to mean a unit of computation and *block* to mean a hyper-rectangular array of cells. In the following discussion, the terms *block* and *array* are equivalent, as are the terms *cell* and *array element*; the useful additional connotation is that cells are *active* entities, i.e. array elements which (may) refer to other array elements.

Spreadsheets provide a natural way of viewing vectors and matrices; they can also be used to provide views of two-dimensional “slices” of arrays of higher dimensionality. Because of this, a good form of array notation is a must; because of the concrete nature of the view, highly abstract syntaxes tend to be unsuitable. A further factor—the unsophisticated nature of many spreadsheet users—mitigates in favor of a concrete and simple syntax, even at the expense of some expressive power. While computer scientists and language developers are justly proud of their powerful and hard-won abstract thinking skills, it is unreasonable to expect novice and casual users to possess or even *want* to learn them. Our position is therefore that concrete forms must be used when possible.

On the other hand, a prominent feature of both modern spreadsheets and natural languages is the use of *intensional logic* [AFJ 91]<sup>1</sup>, exemplified (in part) in modern spreadsheets in the form of relative addresses. Novice users seem to grasp this concept with great ease and joy.

In an effort to unite the above concerns with historical usage, we propose an extended array notation for spreadsheet and other concrete languages. Wherever possible we have borrowed from existing notations, languages and conventions. The following five addressing modes describe the behavior of array subscripts:

$A[m, n, \dots]$	<i>Absolute addressing.</i> The $m, n, \dots$ <sup>th</sup> element of the array $A$ , corresponding to standard mathematical usage.
$A[\pm m, \pm n, \dots]$	<i>Relative addressing.</i> The element in $A$ at offset $m, n, \dots$ from the position in its home array of the element which contains the reference.
$A[m_1..m_2, n_1..n_2, \dots]$ $A[m_1, n_1, \dots]..A[m_2, n_2, \dots]$	<i>Subrange addressing.</i> A sub-array whose first dimension begins at element $m_1$ and whose size is $m_2 - m_1$ , whose second dimension begins at element $n_1$ and whose size is $n_2 - n_1$ , and so on. Note the two forms, which are equivalent.
$A[* , \dots]$ $A[i..* , \dots]$	<i>Dimensional addressing.</i> A unary star (*) denotes all the elements in the respective dimension of array $A$ . A trailing star in a subrange address allows selection of part of an infinite or unknown-length sequence.

---

1. First exposure to this term is often humorous. One student asked us, “*Intensional logic*, uh, is that, like, stuff that makes sense on purpose?”



A[... , i*]	<i>Meta-dimensional addressing.</i> A postfix star means that all remaining dimensions of the array being indexed are included. A prefix star selects all the leading dimensions of the array up to and including the one denoted by the integer it prefixes. The pound (#) sign works the same way as the star, but denotes exclusion instead of inclusion.
A[*i, ...]	
A[... , i#]	
A[#i, ...]	

By contrast with these subscripting modes, what we have termed *implicit addressing* concerns the identity of the array being subscripted; this concept is fully orthogonal to the subscripting modes (we use the absolute mode in the example):

@[m, n, ...]	<i>Implicit (absolute) addressing.</i> The m,n,... <sup>th</sup> element of the array occupied by the element containing the reference. The ampersand (@) symbol resolves to the name of the containing array, in other words.
--------------	--

The subscripting modes are all orthogonal “per dimension”, in other words the somewhat boggling construct A[m, +n, p..q, \*] is legal (It denotes a 2-dimensional subarray of A, anchored at A’s *m*th row and *n*th column relative to the current position, and containing A’s *p*th through *q*th “piles” in the *z* dimension, each of which is itself a vector). Of course, the implicit reference @[m, +n, p..q, \*] is also legal.

All the array indexing modes depend to varying degrees on knowledge of two things—the *shape* of the containing array, and the *location* of the element making the reference within this containing array—which they then apply to produce a new array reference. There are times when we need direct knowledge of these things. We define for the purpose (but do not propose as a standard), the following primitives:

<i>Sh</i>	The shape of the enclosing container in vector form. Both the implicit and explicit versions are legal.
<i>Sh</i> (A)	
<i>Loc</i>	The offset of the current location within the enclosing container, in vector form.

*Sh* returns the shape of an array, which is defined in More’s array theory [Mor 73] to be a vector of the same length as the dimensionality of the array, containing in each element the size of the corresponding dimension of the array. *Loc* returns the offset of an array element within its container, also in the form of a vector. Suppose A is a 3 x 4 x 5 array and shapeA = *Sh*(A). Then shapeA = [3, 4, 5], shapeA[1] = 3, shapeA[2] = 4 and so on. *Loc*, invoked in any of A’s elements, will evaluate to [n, m, p] where 1 ≤ m, n, p ≤ 3.

In the next sections, we describe each of the addressing modes in more detail.

## 5.1 Absolute Addressing

This is the mathematical standard. We purposely do not specify the basis of the array; many language developers like zero-based arrays, while others prefer one-based arrays. There are valid arguments in either direction, depending on the purpose of the language. We will use one-based arrays for the remainder of this discussion.

## 5.2 Relative Addressing

Our proposed notation distinguishes between two forms of intension; relative addressing is one of them. In the literature, an *intension* is the context associated with a variable or expression, so it amounts to a table of all the possible values the variable or expression can have. For example, the intension for the expression *my age* is (in many cases) a table of date/age pairs beginning at my birth and ending at the present, with my age on each date being the second part of the corresponding pair. This concept of intension is closely related to the idea of a function, which is a special kind of relation, i.e. a set of all possible domain/range pairs.

The relative addressing found in modern spreadsheets is a form of intension, because the value of a cell which uses relative addressing is dependent on the location of the cell, in other words, its context. It is easy to be misled by the syntax of Lotus 1-2-3 and cousins into thinking that this is not so. For example, in a



Lotus worksheet a reference in cell A1 to cell B2 (“@+1, +1” in our notation, “B2” in Lotus) looks like an ordinary reference to an absolute offset from the beginning of the spreadsheet. But this appearance only holds until the formula is copied into another cell. At that point, the spreadsheet engine changes the relative reference so that it continues to match the pattern @+1, +1 with respect to its new location.

We have kept the name and tightened up the definition. *Relative addressing* applies the offset of the cell within its enclosing block to some other block. For example, the reference B[+1, +1] appearing in cell A[2, 3] refers to cell B[3, 4], while the same reference appearing in cell C[0, 0] refers to cell B[1, 1]. Putting it another way, the expression A[+m, -n] signifies the element in A  $m$  rows to the right and  $n$  columns above the current position.<sup>1</sup>

Formally,

$$A[m, n, \dots, p] = A[\text{Loc}[1] \ m, \text{Loc}[2] \ n, \dots, \text{Loc}[\text{length}(\text{Loc})] \ p],$$

where  $\text{length}(x)$  returns the number of elements in vector  $x$ .

There is a syntax problem to be resolved in a grammar which incorporates this notation; it conflicts with the unary plus and minus operators found in many languages. The resolution is not difficult, however. Array indices are rarely negative, and users almost never use unary plus; the simple expedient of requiring the unary plus and minus operators to be parenthesized in an array index expression resolves the conflict. For those who do not mind an additional level of operator precedence, defining the indexing unary plus and minus operators to be of higher precedence than ordinary unary plus and minus also does the trick.

### 5.3 Implicit Addressing

We have termed the other form of intension *implicit addressing*. Consider for example the implicit absolute address @[m, n, ...]. In this case the intension is the set of *spreadsheet variables* for which the reference is legal (we use the term *spreadsheet variable* in the same sense as Du and Wadge in [DW 90a] and [DW 90b]). Formally,

Let

$curr$  be the name of the current spreadsheet variable

Then

@[<subscripting info>] =  $curr$ [<subscripting info>]

The two forms of intension we have defined correspond to intension of position and intension of scope (relative and implicit addressing, respectively). The positional form would seem to be closer to that found in most variants of Lucid. For example, in the Lucid doublet

first evens = 2

next evens = evens + 2

the value of evens at any point in time is defined by the point just before it in time, i.e. its position in the time dimension..

Intension of scope is a much-used language feature, but it is almost never used in its pure form. In class-based object-oriented languages such as C++ [Str 93], for example, scope intensionality appears in the form of *virtual functions*. If class A defines the virtual function  $foo()$ , for example, and classes B and C each inherit from A and override the definition of  $foo()$ , then a call to  $foo()$  will generate results according to the original class of the object which calls it (*irrespective* of any type casting that has been done). Effectively, the original class of the object forms an implicit context which is used to decide the result of the function. But this effect is not available for *data members* of objects in C++.

By contrast, a spreadsheet or array-based language which implements implicit addressing as we have proposed it provides intensionality of scope in its pure form; both data and functions may be addressed intensionally. As an illustration of its usefulness, this allows straightforward implementations of the powerful concept of *roles*, in which an object's behavior and/or instance data changes depending on the role it is currently playing. For example, Ted Smith has a certain personality and phone number known to his friends, but in his role as Dr. Smith his demeanor and phone number will probably be different.

1. This is the mathematical standard. In some respects it is a very bad one, because the novice user expects (in our experience) that array indices will proceed in  $x, y, z$  order, whereas convention uses  $z, y, x$ .



## 5.4 Subrange Addressing

This mode is another familiar one to modern spreadsheet users. In Lotus, the range B2..E4 is the block denoted by @ [2..4, 2..5] or @ [2, 2]..@ [4, 5] in a one-based version of our notation. Formally,

$A[m_1..m_2, n_1..n_2, \dots, p_1..p_2]$ , where  $m_1 < m_2, n_1 < n_2, \dots, p_1 < p_2$ , =  
 an array  $R$  of shape  $[m_2 - m_1, n_2 - n_1, \dots, p_2 - p_1]$ ,  
 where  $\forall m_i : m_1 \leq m_i \leq m_2, n_i : n_1 \leq n_i \leq n_2, \dots, p_i : p_1 \leq p_i \leq p_2$ ,  
 $R[m_i, n_i, \dots, p_i] = A[m_i - m_1 + 1, n_i - n_1 + 1, \dots, p_i - p_1 + 1]$

It is interesting to digress on the relative compactness of the Lotus notation. As it stands, the notation is somewhat awkward because the x-component of a cell address is radix-26, so addresses like BX104..CY105 are not uncommon. The basic idea is sound, though: let the odd-numbered dimensions use a different symbol set from the even-numbered ones. To fix the radix problem, let the odd-numbered dimensions be numbered by radix-10 numbers using the alphabetic digits {O, A, B, C, D, E, F, G, H, I} corresponding to the numerical digits {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Our example range BX104..CY105 then becomes FE104..AOC105 (corresponding to [+104, +76]..[+105, +103]). Multidimensional constructs are possible; the four-dimensional reference AB98FO104..AC98FO104 is rather Romanesque, but denotes the same thing as @ [+98, +11..+12, +70, +104] in our notation. Still, we prefer the latter, partly because it integrates smoothly into the historical standard, and partly because it does not impose the burden of learning a new decimal symbol set.

## 5.5 Dimensional Addressing

The unary star in dimensional addressing is a convenient extension to subrange addressing. For example, the subrange B[1..2, \*] is the first two rows of B (in a one-based language). If B is a 10 x 10 array then B[1..2, \*] = B[1..2, 1..10].

The trailing star in the subrange version of dimensional addressing allows operations on infinite sequences. For example, if B is an infinite vector, B[2..\*] denotes the same thing as the Lucid construct *next B*. Formally,

Let  $A =$   
 an infinite sequence  $[a_1, a_2, \dots, a_k, a_{k+1}, \dots]$   
 Then  
 $A[k..*] =$   
 an infinite sequence  $[a_k, a_{k+1}, \dots]$

Dimensional addressing may also be useful for expressing the sort of intensional operations on infinite sequences found in Lucid, depending on the semantics of the language. Consider the Lucid phrase

evens = 2 fby evens + 2

In a C-style language using our array notation this might be something like

```
evens[0] = 2
tmp = &evens[1];
tmp[*] = evens[*] + 2;
```

This is less elegant than the Lucid version, but does the job. A spreadsheet language can do better, but may not use the dimensional addressing mode for the purpose. The mapping from Lucid to spreadsheet languages using our notation is the subject of section 6.

## 5.6 Meta-dimensional addressing

Some of the more sophisticated operations we might wish to perform on arrays require the ability to manipulate their shape. In the case of user-defined functions, we must often do this without knowing which elements of the shape will ultimately be affected (Indexical Lucid [FJ 93] was designed to deal with this same problem). Unfortunately this addressing mode is significantly more abstract, and therefore more difficult for average users than the others we have presented. Languages—those aimed at a wide audience, at least—which employ it should do so as little and as unobtrusively as possible.

The meta-dimensional addressing modes operate on subscript groups instead of individual subscripts; Suppose C is a 3-dimensional array of shape [3, 3, 3]. Then C[1\*] is a 2-dimensional array of shape [3, 3]

which contains the first 2-dimensional row of C, and  $C[*2, 1]$  is a 2-dimensional array containing C's first "face" in the z dimension. The pound sign (#) allows exclusion instead of inclusion of dimensions, so  $C[\#2, *]$  yields a 1-dimensional array of shape [3] containing  $C[1, 1, 1..3]$ . The postfix pound sign is included for completeness; normal usage is such that  $C[1\#] = C[1]$ . Formally,

Let  $A =$   
an  $n$ -dimensional array of shape  $[s_1, s_2, \dots, s_k, s_{k+1}, \dots, s_n]$

Then  $A[\langle \text{subscript}_1 \rangle, \langle \text{subscript}_2 \rangle, \dots, \langle \text{subscript}_k \rangle *] =$   
an  $n$ -dimensional array  $A' =$   
 $A[\langle \text{subscript}_1 \rangle, \langle \text{subscript}_2 \rangle, \dots, \langle \text{subscript}_k \rangle, 1..s_{k+1}, \dots, 1..s_n]$

And  $A[k*, \langle \text{subscript}_{k+1} \rangle] =$   
a  $(k + 1)$ -dimensional array  $A'' =$   
 $A[1..s_1, 1..s_2, \dots, 1..s_k, \langle \text{subscript}_{k+1} \rangle]$

And  $A[\#k, \langle \text{subscript}_{k+1} \rangle] =$   
a vector  $A''' =$   
 $A[1_1, 1_2, \dots, 1_k, \langle \text{subscript}_{k+1} \rangle]$

And  $A[\langle \text{subscript}_1 \rangle, \langle \text{subscript}_2 \rangle, \dots, \langle \text{subscript}_k \rangle \#] =$   
a vector  $A'''' =$   
 $A[\langle \text{subscript}_1 \rangle, \langle \text{subscript}_2 \rangle, \dots, \langle \text{subscript}_k \rangle, 1_{k+1}, \dots, 1_n]$

It is legal to mix the postfix and trailing forms; the construction  $A[i..**]$  denotes the infinite sequence beginning with  $A[i]$ , each of whose members is an array of arbitrary dimensionality. There is a potential syntax conflict here for languages with an  $**$  operator; it can be resolved by the same means used in the section on relative addressing. The postfix and trailing forms do not conflict with the C language family grammars, as the pointer indirection operator  $*$  is not legal in these contexts. The prefix star, however, introduces a conflict which cannot be resolved by entirely syntactic means. Semantically, the meaning is clear, because using pointer indirection on an integer type is an error.

The dimensional and meta-dimensional forms in this notation can be confusing, but we found it necessary to develop them in order to express the constructs found in Lucid and array-based languages such as APL2 [Wil 91], Nial [GJM+89] and Falafel [Sai 90] with any sort of conciseness.

## 6. Mapping Lucid to Spreadsheet Languages

In this section we map a number of Lucid constructs to the spreadsheet language idiom using our notation. The approach is different from that of Du and Wadge [DW 90a, DW 90b], who in essence graft a spreadsheet user interface onto a variant of Lucid.

In order to actually use the notation in a spreadsheet vehicle, we must add some semantics. Let there be a special meta-cell at the "edge" of each dimension, whose formula affects all the cells in that dimension, but which is overridden by an explicit value or formula placed in any one of those cells (mimicking the usual scoping rules). In the case of conflict between two meta-cell formulas, let the lowest dimension win; i.e.  $x$  overrides  $y$  and so on.

In the following examples, values and formulas explicitly entered by the user are shown in bold, while calculated values appear in regular type. When a row or column has been given a (globally accessible) name, we show it to the left or top, respectively.

evens	<b><math>= [-0, -1] + 2;</math></b>	0	2	4	6

$evens = 0 \text{ fby } evens + 2$



The above spreadsheet uses one of the meta-cells to build the infinite array *evens*. The recurrence is anchored by manually overriding the first cell in the row with a base case value. In the next example, we take the outer product of the *evens* vector with itself. The formula in the upper left corner causes the “result” cells (those shown in white) to contain the product of the values of the respective locations in the input vectors, which in both the horizontal and vertical directions is the *evens* vector itself.

<b>h = Loc[2]; v = Loc[1]; = evens[h] * evens[v];</b>					
	0	0	0	0	0
	0	4	8	12	16
	0	8	16	24	32
	0	12	24	36	48

*evens\_outer\_prod = evens \* first evens fby evens \* next evens*

Some of the Lucid operations map easily into this spreadsheet idiom:

<b>first_evens</b>	<b>= evens[1]</b>	0	0	0	0
<b>next_evens</b>	<b>h = Loc[2]; = evens[h + 1];</b>	2	4	6	8
<b>prev_evens</b>	<b>h = Loc[2]; = evens[h - 1];</b>	NULL	0	2	4

*first\_evens = first evens  
next\_evens = next evens  
prev\_evens = prev evens*

The above examples represent the “inlined” use of functions. To emulate the actual Lucid operators, we must define them as functions. We use a hypothetical untyped spreadsheet language, which should be readable given knowledge both of C and of our notation. As in [YC 94], we use uppercase variable names to denote *blocks* (either bounded or unbounded hyper-rectangular arrays of cells). We will not bore our audience here with a complete exposition of all the built-in operators in Lucid; our interest is in the way a spreadsheet UI can use these concepts, so a couple representative operations will suffice. We define the equivalents to *asa* and *fby*:

```

asa
operator asa (A, b)
{
    static i = 1;

    while (A[i] != b)
        i++;
    return A[+(i-1)];
}

```

```

fby
operator fby(A, B)
{
    h = Loc[2];

    if (h == 1)
        return A[1];
    else
        return B[h - 1];
}

```



Using this formulation of the operators, we can apply them something like this:

evens	= fby(0, evens + 2)	0	2	4	6	8
evens10	= asa(evens, 10)	10	12	14	16	18

*evens = 0 fby evens + 2*  
*evens10 = evens asa 10*

These suffer from the same problem that inspired the development of Indexical Lucid; what we want are functions that are *dimensionally abstract* [FJ 93], but putting formulas in meta-cells requires knowledge of which dimension they live in.

The operations may also be defined to be dimensionally abstract, but at an added cost in code complexity and semantics. We will define the equivalents to the dimensionally abstract Indexical Lucid [FJ 93] operations *asa.i* and *fby.i*. (but without discussing the additional semantics required):

<p><b>asa</b></p> <pre> operator asa (i, A, B) {   len = length(A);   for (j=1; j&lt;=len &amp;&amp;       A[* (i - 1), j*] != B; j++)     ;   if (A[* (i - 1), j*] == B)     return A[* (i - 1), j..**];   else     return EMPTY; }</pre>	<p><b>fby</b></p> <pre> operator fby(i, A, B) {   len = length(A);   R = A[*i];   for (j=1; j &lt;= Sh[i]; j++)     if (len &lt; 2)       R[j] = B;     else       R[* (i - 1), j] = B;   return R; }</pre>
--	---

These operations work best in a spreadsheet language with a nested block facility, i.e. a cell may contain a block of cells of arbitrary dimensionality. An example applications of the *asa.i* function then looks something like this:

<b>= asa(1, evens, 6)</b>						
6	8	10	12	14	16	18

*evens asa.1 6*

## 7. Conclusion

Spreadsheets as a concept have much in common with dataflow languages. Implementing them properly requires the use of concepts from both the eductive and data-driven varieties of dataflow. Spreadsheets which use our proposed array notation (or duplicate its capabilities) are also able to offer much of the same sort of intensional expressive power found in the Lucid language family.

## 8. Acknowledgments

Jaggan Jagannathan encouraged us to investigate the concepts presented in this paper and answered many questions about the Lucid language. Any mistakes we have made with reference to Lucid, however, are wholly ours.



## 9. References

- [Ash 86] E.A. Ashcroft. Dataflow and education: data-driven and demand-driven distributed computation. In *Current Trends in Concurrency*, G. Goos and J. Hartmanis, eds., pp 1-50. LCNS 224. Springer-Verlag, 1986.
- [AFJ 91] E.A. Ashcroft, A.A.Faustini and R. Jagannathan. An Intensional Language for Parallel Applications Programming. In *Parallel Functional Languages and Compilers*, Boleslaw K. Szymanski, ed. pp. 11-49. Addison-Wesley, 1991.
- [AU 91] Arvind, L. Bic and T. Ungerer. Evolution of Data-Flow Computers. In *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.
- [Den 91] The Evolution of "Static" Data-Flow Architecture, Jack Dennis; in *Advanced Topics in Data-Flow Computing*, J.L. Gaudiot, L. Bic, eds. (35-91). Prentice-Hall, 1991.
- [DW 90a] Weichang Du and William W. Wadge. The Educative Implementation of a Three-dimensional Spreadsheet. In *Software—Practice and Experience* 20(11): 1097-1114. November 1990.
- [DW 90b] Weichang Du and William W. Wadge. A 3D Spreadsheet Based on Intensional Logic. In *IEEE Software*, May 1990, pp. 78-89.
- [FJ 93] A.A. Faustini and R. Jagannathan. Multidimensional Problem Solving in Lucid. Expanded version of paper entitled "Indexical Lucid" in ISLIP 91. From the authors.
- [GJM+89] Janice Glasgow, Michael Jenkins, Carl McCrosky, Henk Meijer. Expressing Parallel Algorithms in Nial. In *Parallel Computing 11*, pp. 331-47, North-Holland, 1989
- [Jag 94] R. Jagannathan. Adding Eagerness to Education (Extended Abstract). To appear in *Proceedings of ISLIP 94*. From the author.
- [Mor 73] Trenchard More. Axioms and Theorems for a Theory of Arrays. In *IBM Journal of Research and Development* 17(2):135-175, March 1973.
- [Sai 90] Ken Sailor. *A First Implementation of Arrays in Falafel*. Masters thesis, Univ. of Saskatchewan. 1990. From Carl McCrosky (thesis advisor).
- [Str 93] Bjarne Stroustrup. *The C++ Programming Language*, 2nd ed. Addison-Wesley, 1993.
- [Wil 91] R. G. Willhoft. Parallel Expression in the APL2 Language. In *IBM Systems Journal* 30(4): 498-512, 1991.
- [YC 94] Alan G. Yoder and David L. Cohn. Real Spreadsheets for Real Programmers. In *Proceedings of the 1994 IEEE Conference on Computer Languages (ICCL '94)*. pp. 20-30.