Domain-Specific and General-Purpose Aspects of Spreadsheet Languages

Alan G. Yoder and David L. Cohn

Distributed Computing Research Lab
University of Notre Dame, Notre Dame, IN 46556-5637

{agy, dlc}@cse.nd.edu

Abstract

Spreadsheets comprise a language metaphor which has been found very useful for manipulation and presentation of financial data. As such they may be considered a domain-specific language. The metaphor may have application in other areas, however. For example, spreadsheets simultaneously offer natural parallelism and ease of use. Adapting them to do general-purpose computing while preserving their natural parallelism and ease of use would be highly desirable, especially given the current trend toward massive parallelism. And the ease with which users are able to format their output would be a desirable addition to any general-purpose language.

This paper focuses on two things. The first is language features that may reasonably be added to existing spreadsheets to make them look more like general-purpose languages. These features include user-defined functions, structures, cyclical dependencies and so on. The second is on the domain-specific features of spreadsheets which are not encompassed by any general-purpose language constructs. These are primarily the addressing modes, and constitute an as yet unstandardized way of denoting cells at an arbitrary offset from a given cell.

1. Introduction

Spreadsheets are a well-known and widespread set of language systems which have been found very useful, mainly in the business world, for the manipulation and presentation of financial and other sorts of tabular data. Good interfaces to database host servers allow users to make queries to host systems and then fiddle with, transform, analyze, and format the results on a desktop PC. The fundamental language paradigm is very simple; a spreadsheet consists of rectangular *cells*, which have *values* and optionally *formulas*; these formulas calculate a cell's value using constants, some built-in functions, and the values in other cells. The immediate usefulness of the tabular layout in a financial setting, combined with the fact that spreadsheets have not really ever evolved away from this setting, marks them as domain-specific languages, at least to some degree.

Spreadsheets are interesting as language systems for a couple reasons. Firstly, users don't think of them as programming languages, yet soon become accustomed to tasks that can be reasonably described as programming. Secondly, a great many of the resulting programs are embarrassingly parallel. Adapting spreadsheets to do general-purpose computing, while preserving their natural ease of use and implicit parallelism is a highly desirable goal, especially given the trend toward massive parallelism. Processor-in-memory (PIM) technology, for instance, which is focused on reducing the processor-memory bottleneck by embedding processors directly in the memory arrays, tends toward an architecture which provides many thousands of simple processors in a grid layout [KBB 96].

The question is, then, are there any obstacles to this adaptation? Some efforts have been made to move spreadsheets toward the main stream of programming languages [YC 94]; the results have tended to look like a mainstream procedural language with a spreadsheet as a GUI, and the implicit parallelism of the metaphor has also been largely lost. In section 2, this paper sketches methods for adding iteration, structures and user-defined functions to spreadsheets without such a large sacrifice of the opportunity for parallelism. Not all general-purpose language constructs are a good fit with spreadsheet languages. This is discussed in section 3.

A further question is, do spreadsheets contain any features that cannot be accommodated by mainstream general-purpose languages as they now stand? There are a couple, it turns out. One stems from the geometric nature of spreadsheets. In a spreadsheet, cell formulas can refer to other cells at a relative offset from themselves. A small extension of standard array notation can provide this capability as well, but the resulting implicit knowledge by array elements of their own location is foreign to mainstream languages. This result is discussed in section 4.

The second major difference between spreadsheets and mainstream languages is their interactive nature. Interpreted language systems such as Lisp and ML generally display only the results of the most recent input by the user; spreadsheets display *all* the results, constrained only by display capabilities and constraints set by the user. Keeping all cell values up to date (a process known as *recalculation*) is a fundamental task for any spreadsheet system.

Since cells use other cells' values to calculate their own, a graph of dependencies is implicit in every spreadsheet. This dependency graph (DG) can be used to optimize recalculation [YC 94]. Its existence suggests that a spreadsheet could be mapped naturally to a dataflow language system. It turns out that this intuition is not entirely accurate; this interesting result is discussed in section 5.

It should be emphasized that our work is focused on the development of easy-to-use parallel languages, using spreadsheets as a starting point. Our observations on the domain-specific aspects of spreadsheets are exactly that, therefore. In attempting to build a general-purpose spreadsheet, we encountered some interesting difficulties, which we discuss here. We also attempt to answer, at least in part, whether domain-specific language constructs found in spreadsheets can be "moved into" general purpose languages. Our approach is largely orthogonal to the work of others, such as Alex Repenning [RS 95], who have tackled the problem of domain-specific problem representations, also using spreadsheets as a starting point, with fairly spectacular results. The relative ease with which the metaphor is adapted for domain-specific tasks further supports the idea that the metaphor, if not inherently domain-specific, is somehow "sympathetic" to the family of domain-specific languages.

2. A Generalized Spreadsheet Model

The purpose of a general model of spreadsheet languages is twofold. First, we need a frame of reference for talking about spreadsheets as they exist now. We term these *conventional* spreadsheets. Second, a good model of spreadsheets should accommodate constructs from general-purpose languages that we might wish to add.

A	0	1
0	5	
1	A string value	23
2		2.129754
3		

Figure 1 Cells, some having values

Conventional spreadsheets suffer from a fundamental limitation which inhibits the opportunity for parallelism. This is their dependence on what are termed "macros" to carry out every form of iteration, sequencing and assignment. Spreadsheet macros cannot be parallelized, so some other method of accomplishing looping, sequencing and assignment must be found if we are to add general-purpose constructs to spreadsheets while preserving their natural parallelism.

Our generalized spreadsheet model (GSM) accomplishes these aims. This section presents the model, beginning with the constructs common to all conventional spreadsheets, and proceeding through the addition of structures, user-defined functions and iteration.

2.1. Cells

The unit of computation in a spreadsheet is the *cell*. In conventional spreadsheets, cells have *values* and *formulas*; both are optional. A cell with neither a value nor a formula is *empty*.

Besides values and formulas, one might want to store other things in a cell—documentation, for instance, or information about which other cells its formula refers to. GSM therefore defines cells as lists of *properties*. The value property is distinguished, in that a reference to a cell, whether by name or address, returns its value. The return value may be *nil*, denoting a cell with no value.

Spreadsheets are not constrained to displaying only the values of their cells; the user might wish to see the documentation information for cells instead, for example.

2.2. Cell Formulas

Cells can contain simple values, or *formulas* which compute those values. In conventional spreadsheets, these formulas have traditionally consisted of expressions based on an expression grammar. The expressions are built up out of constants, calls to a hundred or so built-in functions, references to other cells and other expressions. The cell references yield the computed values of the target cells.

In a general model, it is logical to extend the definition of a formula. GSM considers a cell formula to be a statement taken from a language, rather than an expression grammar. This language is arbitrary for the purposes of this discussion, but can be assumed to provide at least scoping, local variables and iterative and/or recursive constructs. A functional subset of Scheme with array notation added is a convenient such language, and is used in the examples in this paper.

2.3. Cell References

When cell formulas need to refer to the values of other cells, three addressing options are available in conventional spreadsheets. Cells can be referred to by *name*, by *absolute position*, or by *relative position*. These addressing modes are explored in more depth in section 4. Briefly, a name is just an alias for an address, and absolute position works similarly to the familiar array notation. A relative address refers to a cell at a relative offset from the cell making the reference. In an Excel or Lotus[1] spreadsheet, for example—in which relative addressing is the default—the address `A1`, appearing in cell `B1`, refers to the cell one to the left of `B1`. Copying `B1`'s formula elsewhere will result in automatic modification of the address `A1`, as in Figure 2, where `B1`'s formula has been copied to cells `B2` and `B3`.

	A	B
1		`=A1`
2		`=A2`
3		`=A3`
4		

Figure 2 Identical relative addresses in Excel

2.4. Groups of Cells

In conventional spreadsheets, cells can be grouped into rectangular arrays called blocks or ranges, depending on which package one is using. In GSM, these terms are given specific and separate meanings, and the concept of grouping cells is subsumed into a scheme which adds the concept of data structures to the spreadsheet metaphor, and permits the usual and familiar kinds of encapsulation.

A *block* is a hyper-rectangular container for cells. As with arrays in general-purpose languages, blocks are of fixed size. Borrowing from APL [BC 91] and array theory [Mor 73], a block's dimensionality is called its *rank*, and the vector containing the sizes of each dimension is called its *shape*.

Blocks can be nested. A cell can contain a block as its value; cells in this block can contain other blocks which contain cells that contain other blocks, and so on. We use the usual terms to denote relationships of various cells in the resulting tree structure. The containing cell of a cell's containing block is called its *parent*, and contained cells are called *child* cells or *children*. Cells in the same block are termed *siblings*.

Restricting the values in cells to be of primitive types plus blocks, and blocks to be vectors (i.e. to have rank 1), yields a fairly exact correspondence between this scheme and the concept of structure in languages such as C and Pascal.

A *range* is similar in concept and denotation to a block; it is a hyper-rectangular group of cells. But whereas blocks are containers, ranges are not; they are simply ways of denoting subsets of cells from a block.[2]

2.5. Inter-cell Dependencies

When a cell x makes a reference to another cell y, x becomes a *dependent* of y. Collectively, the depen-

1. All trademarks are the properties of their respective owners.

2. There is no obstacle, in concept, to having ranges spanning multiple blocks, if the semantics of the spreadsheet language specify that blocks are laid out in a way that accommodates such an idea.

dencies in a spreadsheet form a graph called the dependency graph (DG). While this graph is usually not constrained to be acyclic by conventional spreadsheets, cycles in it do usually give undefined results.

Conventional spreadsheets use a technique called natural-order recalculation to ensure that all cell values are up to date when recalculation is finished. This technique iterates through all the cells in the spreadsheet; when it comes to a cell *x* which depends on *y*, *y* is recalculated before *x*, and so on recursively. Cycles must be detected, and calculation stopped after some number of iterations, to prevent this method from recursing infinitely in the presence of cycles.

In GSM, cells keep track of which other cells depend on them by maintaining references to these cells on a *dependency list*. In the above example, for instance, cell *y* knows that cell *x* depends on it. When *y*'s value changes, it is responsible for letting *x* know about it. [1]

Dependency lists change when cell formulas change. Suppose that a cell formula in cell *x* which refers to cell *y* is created or edited. Then *x* enters into a "negotiation" with *y* to obtain *y*'s value. As part of this negotiation process, *y* adds *x* to its dependency list, if *x* is not already there.

This mechanism has four virtues. First, the DG is stored, and so is available for analysis. Second, it is stored in a distributed fashion, which is an advantage if a spreadsheet language desires to make use of large-scale parallelism that may be available. Third, it allows identification of the roots of the DG forest. In the event that a full recalculation must be done (upon loading a spreadsheet, for example), recalculating the roots of the forest guarantees that all cells of the spreadsheet will be notified, and hence recalculated, because all cells in the spreadsheet depend directly or indirectly upon the cells corresponding to the root nodes of the DG. Fourth and most important, a change in a single cell *x* does not induce a requirement to iterate over all the cells in the spreadsheet; instead, only the subgraph rooted at *x* needs to be recalculated.

Dependencies are represented as arrows, and cells as nodes, in a directed graph, as in Figure 3. The arrows point the opposite direction of the dependency. This means that the arrows point in the direction in which the data flows from node to node; this makes analysis of the DG more intuitive. For example, when a cell *y* changes, we know that the subgraph rooted at *y* must be recalculated, which in the figure is the node *z*.

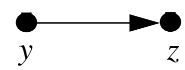

Figure 3 A dependency from *z* to *y*

Sometimes cells wish to refer to the values of other cells without being notified when the source value changes. An example is a random number generator, which might take its seed value from a clock cell[2], but does not wish to be notified forevermore when the clock ticks. This is done by use of the `copy` function, and induces what is termed a *copy dependency*. These are denoted with dashed arcs in the DG, as shown in Figure 3.

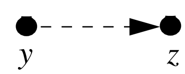

Figure 4 A copy dependency from *z* to *y*

2.6. Functions and Macros

Functions can be added to spreadsheet languages by simply allowing the definition of first-class functions in cell formulas; the resulting object simply becomes its containing cell's value. A reference to the cell, by name or address, then yields the function object, which is then applied to any arguments supplied by the caller.

It turns out, however, that functions as they are usually thought of are not always convenient for use in spreadsheet languages. This is because of relative cell addressing. A function whose purpose is to average together the values of the east and west neighbors of the calling cell, for example, will simply not work, because functions definitions are evaluated when they

1. What is actually meant by *x* "knowing about it," is that the runtime system becomes responsible for recalculating *x*.

2. Clock cells are special cells which change values at a specified interval. The authors' system adds a primitive

```
(every <milliseconds> <body>)
```

to their subset of Scheme to accomplish this.

	A	B	C
1	**10**	**(lambda ())**	**20**
		(lambda () (/ (+ A1 C1) 2)	
2	**100**	**15**	**200**
		(B1)	
3	**1**	**15**	**2**
		(B1)	

Figure 5 An E-W averaging function

are made (becoming lambda forms in Scheme, for example). An attempt to construct such a function will result in a function which averages the values to the east and west of the cell in which the function is defined. Figure 5, in which both values and formulas are shown (values are in boldface), illustrates the results of such an attempt. Cell `B1` defines a function which averages the values of its east and west neighbors. Cells `B2` and `B3` call this function, but the result is the average of `A1` and `C1` every time.

The solution to this problem is the use of macros, which use call-by-name semantics instead of call-by-value semantics as ordinary functions do. In concept,

	A	B	C
1	**10**	**(macro ())**	**20**
		(defmacro () (/ (+ A1 C1) 2)	
2	**100**	**150**	**200**
		(B1)	
3	**1**	**1.5**	**2**
		(B1)	

Figure 6 An E-W averaging macro

macros are imported into the calling cell and evaluated in place. This results in the correct and expected behavior, as shown in Figure 6.

It is possible to do everything with functions that one can do with macros. The difference between the two is their treatment of free variables—a function evaluates a free variable in the context of the cell in which it is defined, whereas a macro evaluates it in the context of the cell in which it is used. Therefore, if one is willing to eschew the use of free variables, macros are an unnecessary feature. It is a characteristic of many spreadsheets, however, that free variables, in the form of relative references to neighboring or near-neighboring cells, are a large and often-used convenience. If we desire to accommodate users who like this convenience, we must provide macros.

There is an interesting correspondence between the role of macros and functions in a spreadsheet language, and those of functions and RPC calls in parallel and distributed systems. In spreadsheet languages which provide such things, macros are rather more often used than functions, because relative addresses are more often used than absolute addresses. Since macros are effectively imported and evaluated in place, while functions can be evaluated remotely, macros end up corresponding to a local function call in a distributed system, while functions correspond to RPC calls, which are usually executed on other processors.

2.7. Iteration

Iteration can be added to a spreadsheet language in two fundamental ways. First, it can be provided by the in-cell language; this method has been used in prior work [YC 94]. Second, it can be emulated using circular dependencies; this is truer to the functional, declarative nature of spreadsheets, and better preserves opportunities for parallelism.

Iteration via circular dependencies is best explained with an example. Consider the following dependency graph, corresponding to the C loop

```
for (B1=A1; B1<A2; B1++) {
   B3();
}
```

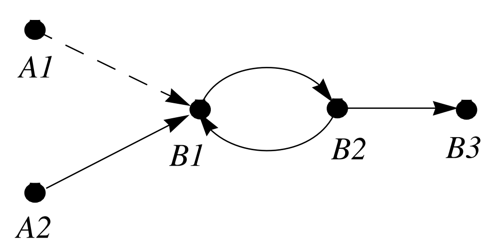

Figure 7 Iteration via circular dependency (DG representation)

Here, B1 is the loop counter, A1 its beginning value, A2 its limit, and B3 the loop body. The extra node

	A	B
1	**11**	**11**
		(if (< B2 A2) (+ B2 1)))
2	**100**	**11**
		B1
3		**<result>**
		(begin B2 <arbitrary code>))

Figure 8 Iteration via circular dependency

in the DG, B2, exists to create the circular dependency that drives the loop.

The cells shown in Figure 8, implement this loop in a spreadsheet; values are shown in boldface and formulas in courier type. The key features of the program are

- The "index variable" B1 updates itself upon a change to its companion B2.
- B2 changes each time B1 changes
- The loop body updates itself each time B2 changes

There are other ways of arranging the counters and dependencies to achieve the same results; the differences are not material, however. Some language systems may allow cells to make self-circular references; in our prototype language, however, all self-references are copy dependencies, so it is our habit to use pairs of cells such as the `B1`/`B2` pair for index variables.

3. Misfits

Not all general-purpose language structures fit well within the spreadsheet metaphor. Recursion and side-effecting assignment are the foremost of these.

3.1. Recursion

It is easy enough to outfit the in-cell language with recursion; an inter-cellular way of representing it is harder. The difficulty lies in the spreadsheet's immediate feedback. In the iterative loop illustrated in Figure 8, for example, the user can observe changes in cell values as the loop progresses. This is particularly true if the spreadsheet system provides a way of stepping execution.

A recursive program, however, works by creating new values on a stack. Representing a dynamically growing and shrinking stack visually, in the kind of intuitive way that other values are represented in a spreadsheet, remains an unsolved problem. Representing the way that recursion causes values to flow from one level of the stack to the next seems particularly hard.

Our goal here is to map existing general-purpose language structures and concepts to spreadsheets, so for completeness recursion should be included. But since iteration can be provided for, and recursive program structures can be mapped to iterative ones, the lack of a suitable representation for recursion is not a serious one.

3.2. Side-effecting assignment

Side-effecting assignment is a very common feature of general-purpose languages. Although the functional language community holds that it is unnecessary, it is natural to the way that people think, and useful for modeling events in the real world.

When a bullet breaks a bottle, for example, we do not say that the bottle broke itself in response to the bullet's approach. But this is what functional languages seem to want us to do. Modeling this event in a functional language makes it appear that a bottle is a collection of forces which can be activated by the approach of a bullet in such a way as to cause dissolution of the bottle. But this is backward. It is quite clear, rather, that an outside agent has caused damage to the bottle; force has been applied to a point on the bottle's surface, and propagation of that force has shattered the bottle. Languages with side-effecting assignment allow the expression of this physical intuition quite naturally.

It is possible to add side-effecting assignment to spreadsheets, with certain restrictions. We allow cells to set the values of other cells, constrained by some set of access controls. Additionally, setting the value of a cell erases any formula it might have had. The assignment operation induces a third type of dependency in the DG; in addition to ordinary dependencies and copy dependencies, we now have *write dependencies.*

The cost of doing this is twofold. First, analysis of the dependency graph becomes more difficult. Without write dependencies, it can be proven that programs which do not contain dependency chains that meet before a cycle in the DG are always deterministic, and even those in which this condition holds can be evaluated in such a way as to guarantee determinism. With write dependencies, this assurance is not available; write dependencies introduce enough new opportunities for non-determinism that it becomes difficult to write significant programs which do not stray into the realm of non-determinism.

The second cost is psychological. Side-effecting assignment removes from a spreadsheet language the declarative character that is in part responsible for the wide popularity of spreadsheets. Although conventional spreadsheets do use assignment in a limited kind of way—in "macros," generalizing to allow this in all spreadsheet formulas invites confusion as non-determinism proliferates through the spreadsheet.

In spite of these costs, finding a "good" way to introduce assignment may be a desirable objective for a spreadsheet language designer. Array operations are an example of a seemingly good use for side-effecting assignment. Consider the array outer-product operation:

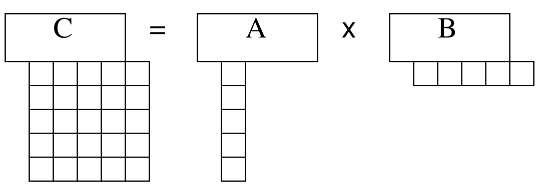

Figure 9 An array operation

Cells A, B and C contain blocks of other cells. We wish to specify, in a single cell formula in C, what the results of such an operation should be. Doing so is quite easy using relative addressing. But this means that the cell C must (at least indirectly) assign values to its child cells. The same problem pertains with almost all array operations. This is a shame, because there is, on the surface, a simple and intuitive mapping from the constructs of APL and array theory [Mor 73] to the spreadsheet metaphor.

We seem to be in a bit of a quandary. On the one hand, side-effecting assignment is evil. On the other hand, it seems to help us to provide a valuable abstraction (array operations), and so it is good.

We can neatly sidestep this problem, with respect to array operations, by allowing cells to inherit their formulas from other cells. Most array operations can be accommodated by allowing cells to inherit their formula only from their parent[1]. Consider the array operation illustrated in Figure 9. Set `C`'s formula to the following

```
(* A[-,0] B[0,-])
```

and arrange for all of `C`'s cells to inherit their formulas from `C` (we can neglect the mechanism for this for this paper's purposes). For any given cell, the formula multiplies the same-rowed element of `A`'s zero'th column and the same-column'th element of `B`'s zero'th row. The desired array operation is in place.

Can a function or macro be written to perform the outer product? Yes. Let the function cell OP contain the formula

```
(defmacro (A B) (* A[-,0] B[0,-]))
```

and cell C contain

```
(OP A B)
```

The desired effect happens; the macro OP is evaluated in place, in each of the cells of C, so the relative addressing of the original formula is preserved.

There may be other reasons for including side-effecting assignment in a spreadsheet language. We may, for example, want some way of resetting the loop indices in the looping program in Figure 8. The available evidence suggests, however, that adding assignment should be done carefully and in a limited way, if at all, and that alternative structures should be used, when they do not obscure the action of the program more than assignments would.

4. Addressing

General-purpose languages which provide arrays always provide some means of addressing various elements within an array. Spreadsheets, however, intro-

1. Allowing other cells to be the source of a formula evolves the spreadsheet language toward the concept of delegation [Lie 86]. It also opens up the door for a multi-level delegation scheme, in which a cell inherits first from its row, or if that is empty, its column, or finally its parent block. This idea has been proposed by Bill Wadge in another context.

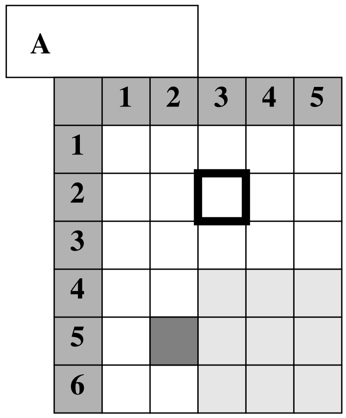

Figure 10 Cell offsets

duce new addressing modes which are missing in almost all mainstream general-purpose languages. Addressing in spreadsheets, both conventional and of the GSM persuasion, can be separated into three categories: *absolute*, *relative* and *range*. Additionally, when nested blocks are introduced, an additional degree of freedom results in *explicit* and *implicit* addressing forms. These correspond to the way names are used in object-oriented languages.

In this section, we will depart from the conventional spreadsheet addressing constructs which have been used so far in this paper, and use a mathematically-based notation. The hope is that this will make the concepts clearer to the reader. Mappings from standard spreadsheet notation to the mathematical notation used here are not difficult. They do little to clarify the present discussion, however, and so will not be presented.

4.1. Absolute Addressing

Let cell A contain a block of cells, as in Figure 10. The outlined and dark-shaded cells are denoted, respectively, as `A[3,2]` and `A[2,5]`. This is absolute addressing, and corresponds exactly to usage in general-purpose languages.

4.2. Relative addressing

Relative cell addressing is another matter. The relative address of the shaded cell, when referenced from the outlined cell `A[3,2]` is `A[-1,+3]`. The relative form of the address refers to the cell one column to the left and three rows down from the cell making the reference.

Relative addresses are extremely common in spreadsheets. Normal procedure, when building a spreadsheet, is to carefully code up a few representative cell formulas, then to copy and paste them into other cells to complete the worksheet. A spreadsheet for computing student grades, for example, usually is set up so that each student's records are on a single row. Individual scores appear in various cells of the row, and to the right of these, somewhere, one or more cells make computations that result in an overall grade for that student. Building such a worksheet involves getting these formulas right for one student, then copying and pasting them to the cells in the same relative positions in other students' rows. Relative addressing does the rest.

4.3. Range Addressing

Another common addressing mode in spreadsheets is range addressing. The nine lightly-shaded cells at the bottom right of block `A` in Figure 10 can be denoted as `A[3..5,4..6]`. Range addressing is used to denote ranges, i.e. subblocks, and is particularly useful in conjunction with array operations such as those discussed in section 3.2, if they are provided.

4.4. Explicit and Implicit addressing

A spreadsheet language that features nested blocks can benefit from the addition of implicit addressing. The outlined cell in Figure 10, `A[3,2]`, can refer to the shaded cell as `A[2,5]` or just `[2,5]`. These are the explicit and implicit addressing forms. What is referred to explicitly and implicitly, respectively, is the *parent* of the cell making the reference.

The distinction between explicit and implicit addressing has the same relationship to block nesting as absolute and relative addressing have to block layout. In other words, implicit addressing is just relative addressing applied to containment level. We might say that implicit addressing is the same thing as *relative scoping*.

Object-oriented (OO) languages use something that looks like relative scoping; methods within an object need not qualify the names of instance variables of that object. This is mainly syntactic sugar, though, and although relative scoping does provide that function, there is more going on in a spreadsheet language than that. Formulas are copied and pasted from one block to another quite readily in a spreadsheet language, whereas methods can seldom be copied and pasted from one class to another. Even when they are, say via some inheritance mechanism, the same results do not pertain, because OO languages permit access to data members by name only, not by position.

The geometric nature of spreadsheets is in fundamental contrast to the entirely algebraic nature of most modern programming languages. It is this which gives implicit and relative addressing in spreadsheets their distinctive character. A spreadsheet cell can refer to another cell two cells to the left, or use its position in its parent block to address another cell in the same position in an entirely different block. Equivalent operations in an algebraic language make sense only in the context of arrays. Otherwise one is forced to make assumptions about how objects are laid out in memory; algebraic languages are usually designed to hide this level of detail.

4.5. Addressing and general-purpose languages

No general-purpose language the authors are aware of provides the variety of addressing modes that spreadsheet languages do. It's an open question as to whether they should. After all, the addressing modes in spreadsheets arise from the tabular layout enforced by the metaphor; conventional languages do not have this restriction.

In the event that one did wish to add, say, relative addressing to a conventional language, what would be the approach? Relative addressing is only valid if the language provides some notion of context, and conventional general-purpose languages lack this, at least in the geometric sense.

It may be possible to take some inspiration from Lucid [AFJ 91], an intensional dataflow language which does provide a notion of context. In Lucid, variables represent infinite temporal sequences of values. These sequences represent the *intension*, the union of all possible contexts for the variable. Sequences are built with operators such as `fby` ("followed by") as in the following statement:

```
evens = 2 fby evens + 2
```

which builds the infinite sequence containing all the positive even numbers. A sequence is intensional because the result of an expression (the *extension*) varies over the indexing dimension *time*. This is not very much different, however, from saying that an array in a conventional programming language is intentional because the result of an indexing operation varies with the value of the index. Multidimensional variants of Lucid [FJ 93] allow for sequences of multiple dimensionalities; the foregoing objection seems to hold for these as well.

Other languages that feature lazy evaluation, such as SASL [Kam 90], are capable of building infinite sequences without a notion of context. A sequence is evaluated to whatever degree is needed; the unneeded portion of a sequence is represented by a *closure*, which encapsulates the last known value and a function for computing the next one.

Plane Lucid [DW 90] is a variant of Lucid in which additional primitives are present for dealing with objects in a plane. These primitives, such as right, left, up, down and so on, do give this "conventional" language a notion of geometric context. But this language has been primarily used to implement a spreadsheet!

While Lucid and its variants are interesting, and do confer a notion of context onto algebraic variables, they are difficult for most users. It is hard to get used to the idea that all variables are infinite sequences. When one extends this to the multi-dimensional case, it gets even harder. This is in stark contrast to most users' experiences with spreadsheets, which they find immediately intuitive and easy to understand.

The conclusion seems to be that spreadsheets differ in a fundamental way from general-purpose languages, and that this difference is rooted in the concept of geometric context, which arises naturally out of the graphical interface offered to the user. Furthermore, it is hard to think of a way to add geometric context to algebraic languages without resorting to making memory layout programmer-visible. Yet spreadsheet languages do not reveal anything at all to the programmer about memory layout.

5. Interactivity and Dependency Graphs

The dependencies between cells in a spreadsheet, taken all together, yield a dependency graph (DG). Thanks to the way we represent it, the correspondence between a dependency graph and a dataflow graph seems quite obvious. But while it is true that there is a correspondence, it would be a mistake to suppose that it is exact. The reason lies in the interactive nature of

	A
1	**5**
	5
2	**13**
	15
3	**20**
	(+ A1 A2)

Figure 11 A simple spreadsheet

spreadsheets.

Consider the spreadsheet in Figure 11, in which both values and formulas are shown (we are back to using conventional spreadsheet addressing now) This corresponds to the dependency graph in Figure 12.

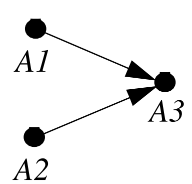

Figure 12 Dependency graph for Figure 11

The crucial difference between a spreadsheet and a dataflow system is that in dataflow, nodes are not computed until all their inputs are available [AU 91]. This means that if the user changes the value in cell `A1`, a dataflow system would not recompute `A3` until the user also changed the value of `A2`. This is clearly not what we want a spreadsheet system to do, and so the analogy between dataflow and spreadsheets breaks down.

This means that the extensive theoretical work on dataflow, while it may provide guidance and insight, is not of direct help when analyzing spreadsheets. Nor can we simply consider spreadsheets to be members of the family of dataflow languages.

6. Conclusion

Spreadsheets belong to a family of languages in which geometry plays a fundamental role. We might call this the *geometric family* of languages. Mainstream general-purpose languages, on the other hand, have a more algebraic character and belong to a family of languages we can term the *algebraic family*.

Such a classification leads us easily into the idea that there may be some fairly fundamental dissimilarities between spreadsheets and general-purpose languages. There are a number of minor, but only two major points of difference, however. Relative addressing modes are foreign to general-purpose languages, because they lack the notion of a geometric context; this context is provided by the geometric UI of spreadsheets. And side-effecting assignment is simply a bad idea in spreadsheets under most circumstances, destroying their declarative nature and forcing them to be something that they are not.

Our conclusion is that while spreadsheets and general-purpose languages can learn from each other, and indeed have more in common than not, they still represent fundamentally different paradigms. Having different paradigms is useful, though; it gives us more than one point of view when approaching a problem, and so we simply say "Vive la difference!"

7. References

[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.

[BC 91] J. A. Brown and H. P. Crowder. APL2: Getting Started. In *IBM Systems Journal* 30(4):433-445, 1991.

[DW 90] Weiching Du and W. W. Wadge. The Eductive Implementation of a Three-dimensional Spreadsheet. In *Software—Practice and Experience* 20(11): 1097-1114. November 1990.

[FJ 93] A.A. Faustini and R. Jagannathan. Multidimensional Problem Solving in Lucid. From the authors.

[KBB 96] P. Kogge et al, Pursuing a Petaflop: Point Designes for 100 TF Computers Using PIM Technologies. In *Frontiers of Massively Parallel Computation*, 1996.

[Kam 90] Sam Kamin. Programming Languages, an Interpreter-based Approach. Addison-Wesley, 1990.

[Lie 86] Henry Liebermann. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings,* pp 214-223, 1986. Published as SIGPLAN Notice 21(11), November 1986.

[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.

[RS 95] Repenning, A. and T. Sumner, "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," In *IEEE Computer*, Vol. 28, pp. 17-25, 1995

[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.