Maintaining Liveness in a Spreadsheet with Logic Programming

Philip T. Cox

Technical Report CS-2009-03

June 18, 2009

Faculty of Computer Science
6050 University Ave., Halifax, Nova Scotia, B3H 1W5, Canada

Maintaining Liveness in a Spreadsheet with Logic Programming

Philip T. Cox
Dalhousie University,
Halifax, NS, Canada B3H 1W5
+1 902 494-6460
Philip.Cox@Dal.Ca

Abstract

Spreadsheets are among the most widely used programming tools, having been adopted almost universally for computing and tabulating financial information. They were not designed for use in strategic applications, however, since they lack all but the most rudimentary programming support, and are highly likely to contain errors.

In L-sheets, a recently proposed extension to spreadsheets, the standard data flow computational model is augmented with a form of visual logic programming in which term unification is replaced by array unification, providing both improved programmability and a means to specify the high-level structure of sheets.

One of the most important properties of spreadsheets is "level 3 liveness", the immediate recomputation of those cells in a sheet affected by an editing change. This is simple to achieve with data flow computation, but is complicated by the addition of logic programming. Here we address the problem of maintaining liveness in L-sheets.

Categories and Subject Descriptors D.1.7 [**Programming Techniques**]: Visual Programming. I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving - logic programming. H.4.1 [**Information Systems Applications**]: Office Automation - spreadsheets.

General Terms Reliability, Languages.

Keywords Spreadsheets; logic programming; arrays; liveness; visual programming.

1. Introduction

The spreadsheet, which first appeared in 1979 as VisiCalc, was one of the first programming tools for non-technical users [2]. The key to the success of the spreadsheet is that it employs a simple metaphor familiar to its target users, paper ledger sheets containing data in rows and columns, presented in a graphical user interface with which the user can directly interact.

Early spreadsheets were rudimentary at best, providing little more than the ability to enter values and formulae into cells. Nevertheless, they gave end-users, initially financial industry workers, the power to rapidly build programs for analysing and tabulating data, and forecasting, Needless to say, users began to build larger and more complex spreadsheet applications, and to use them to support key business decisions.

As spreadsheet use has grown, features have been added, such as multi-sheet workbooks, references between sheets and workbooks, macros, plug-ins, functions and access to external programming languages. Consequently, the spreadsheet has evolved from a simple, useful but limited programming tool, to an ad hoc collection of features, with little support for developing robust and reliable applications. As a result, although the spreadsheet is one of the most widely used programming tools [22], it is also one of the most error-prone. Various studies have shown that over 90% of all spreadsheets contain errors [16]. Many instances have been reported of spreadsheet errors leading to significant financial losses and other negative consequences [11].

Although spreadsheets have been an ingredient of many research projects, their deficiencies have lately become the centre of attention, at least partly motivated by legislation placing liability for financial misreporting on company directors and managers [16]. Some researchers are concentrating on minimising the risk of spreadsheet errors by auditing [8], or employing spreadsheet development methodologies and management practices [18, 21]. Others are focussing on spreadsheet technology itself, with a view to providing tools for testing and debugging [9], analysis of structure [1, 5, 14], high level design [10, 15], and programming [3, 17, 26].

Several research projects have investigated the intersection of spreadsheets and logic programming. In some proposals, a spreadsheet-like grid is used as an interface to a logic programming engine, the cells displaying the values of variables instantiated by a logic program separate from the sheet [23, 25]. In others, arrays are used to define predicates. For example, in NEXCEL an array can represent a predicate defined by a set of clauses, the bodies of which consist of literals that refer to other arrays representing predicates [4]. In LogiCalc, an array can represent a table defining a relation, and cells can contain queries, lists or other terms [13]. In XcelLog, cells contain expressions which are translated into clauses for execution by an underlying logic programming execution mechanism [19]. All of these systems present a sheet interface to a computational mechanism which is significantly different from the formula-in-cell data flow model, and therefore do not contribute to overcoming the deficiencies of standard spreadsheets. To our knowledge, there has been no combination of logic programming and spreadsheets that preserves the standard data flow model, aside from L-sheets [6, 7], described below.

Most of the cells in more complex spreadsheet applications are not unique, but have characteristics in common with their neighbours. For example, some contiguous cells may contain formulae of the same form, each referring to cells at the same vertical and horizontal offset. Although cells with such common characteristics form arrays within the sheet, spreadsheets provide only basic tools for setting up and managing these arrays, limited to dragging the corner of a cell to copy appropriate variants of its formula to the cells in a rectangle.

L-sheets is based on the observation that assembling a rectangular array of cells by attaching a column (or row) to another array is analogous to assembling a list with the | operator in Prolog. Arrays and lists are fundamentally different, of course, in that lists are recursive structures accessed at one end, while arrays are not. Hence the corresponding basic operation on arrays is juxtaposition, either vertical or horizontal, of two arrays where neither is restricted to be a column (or row). Pursuing the analogy, L-sheets replaces unification of terms by unification of arrays to obtain a form of logic programming in which the spreadsheet user can specify the structure of arrays, relationships between arrays and computations that add content to array cells. Since arrays and the juxtaposition of arrays can be visually represented, the resulting language is visual, and in keeping with the sheet metaphor familiar to spreadsheet users. In contrast to previous spreadsheet/LP combinations, L-sheets delivers the following benefits:

- A visual, high-level specification of spreadsheet structure.
- Enhanced programmability.
- The existing "formula-in-cell" model, familiar to current users, is retained.
- User-defined abstractions are built in the sheet interface, allowing for a smooth transition from novice to expert user.

If L-sheets is to succeed as a seamless extension of standard spreadsheets, it must preserve key properties, in particular, the feedback provided by immediate re-evaluation of cells following editing. Although this is easy to achieve in the data flow model of standard spreadsheets, it is more complicated when there are both data flow and logic programming computations in a sheet. A preliminary and inconclusive discussion of re-execution can be found in [6]. Experience with a prototype implementation resulted in some refinements to the language [7], and have provided further insights into re-execution, leading to the algorithm we report here which has yet to be incorporated into the prototype.

In the next section we informally describe L-sheets, and make observations about some characteristics of the language entailed by the requirement that L-sheets be as consistent with normal spreadsheets as possible. In Section 3, we develop the re-execution procedure, followed by concluding remarks and discussion in Section 4.

2. L-SHEETS

In this section, we describe L-sheets informally via examples, reproduced from [7] with minor changes. These examples were chosen because one illustrates what we see to be a typical use of L-sheets, while the other performs a computation one would not normally associate with spreadsheets. In addition, they have contrasting re-execution properties, relevant to the discussion of liveness in Section 3. A more formal description of the language can be found in [6].

An application in L-sheets consists of a set of *worksheets*, like those in an ordinary spreadsheet, together with *program sheets* which contain definitions as described in the examples below. Computations in a worksheet are defined in two ways. The user may insert formulae into cells in the usual way, or apply definitions from program sheets, which will also insert formulae into cells. Values for cells are computed according to the data flow defined by these formulae.

2.1. Spreadsheet Structure Specification

As we mentioned earlier, one approach to improving the reliability of spreadsheet applications is to provide tools for specifying the structure of sheets. This is illustrated in an example due to Erwig *et al.* [10], which deals with the structure of a budget worksheet, as shown in Figure 1. This worksheet is an example of a family of worksheets in which rows of budget items are repeated, and columns are repeated in blocks of three, each block containing data for one year. As the figure shows, formulae which occur in, or refer to, repeated cells, conform to a pattern which can be applied to a budget worksheet of any size.

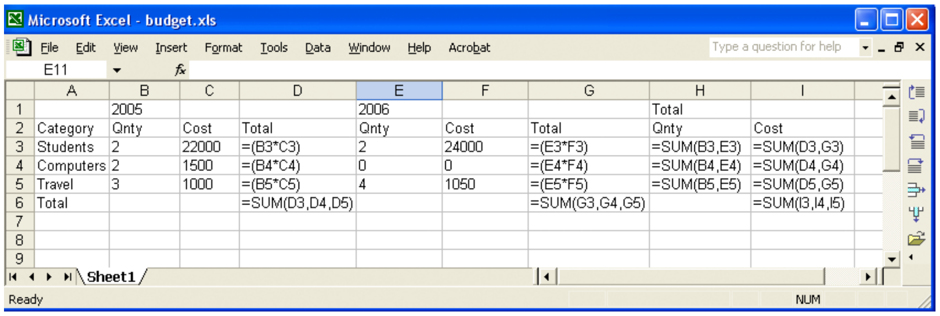
The family of budget worksheets of which the sheet in Figure 1 is an example, is specified by the definitions **budget** and **years** in the program sheet shown in Figure 2. The **budget** definition has a single *case*, the *head* of which is a *template* named **budget**, represented by a pale grey rectangle. The *body* of this case consists of a single template, named **years**. Body templates are usually dark grey. Templates contain *parameters*, which are arrays, drawn as grids similar to a worksheet grid. In our example, the **budget** template has a single parameter, while the **years** template in the body of the case of the **budget** definition has two.

A definition is analogous to a set of clauses defining a predicate in Prolog. In particular, a case corresponds to a clause, a template to a literal, and the parameters of a template to the list of terms from which a literal is composed.

The structure of an array in a template is indicated by the colour of its grid lines. A vertical grey line indicates that the width of the subarray in which the line lies is variable. For example, consider the array in the **budget** template. Although the subarray with the heavy outline labelled **A** is drawn with two columns, the line separating these columns is grey, so the subarray **A** is variable in width. Similarly, grey horizontal lines indicate variable height. With this is mind, we see that the array in the **budget** template has one column, followed by a variable number of columns, followed by two columns. It also has two rows, followed by a variable number of rows, followed by one row. Values of subarrays are displayed when they are known. For example, the value of the 1×1 subarray at the left end of the second row is the string "Category". The annotation *F1* on the 1×1 subarray at the bottom right corner is not part of the program. It has been added to indicate that the content of the subarray is as shown in the legend.

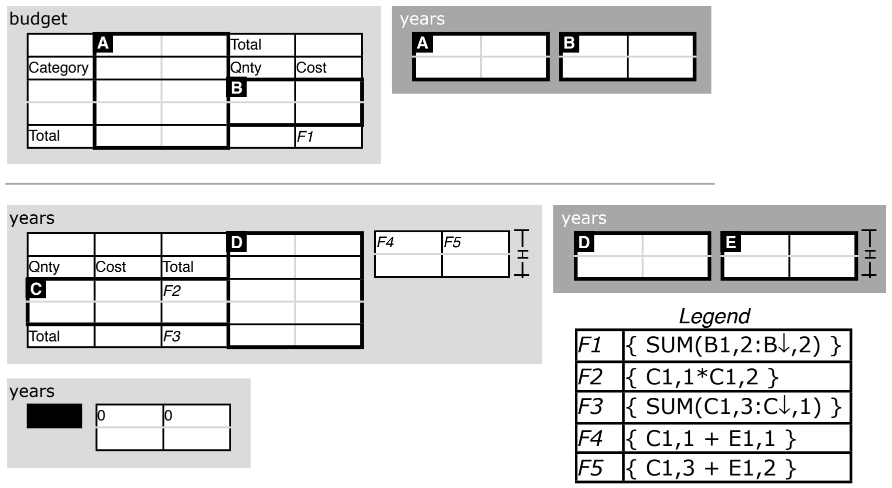
The definition **budget** specifies that an array is an instance of a budget worksheet if it has the structure described in the previous paragraph and contents as shown, and the subarrays named **A** and **B** have the structure and content specified by the definition **years**.

To apply this definition to a worksheet, the user selects a rectangular array of cells in the sheet to correspond to the parameter (Figure 3a). A (virtual) *goal template* named **budget** is created with the selected array as its parameter. As in Prolog execution, this goal template is unified with the head of the first (and only) case for **budget**. This involves unifying the selected array with the parameter of the head template, using the algorithm described in [7], which matches the structure of the two arrays and adds the



	A	B	C	D	E	F	G	H	I
1		2005			2006			Total	
2	Category	Qnty	Cost	Total	Qnty	Cost	Total	Qnty	Cost
3	Students	2	22000	=(B3*C3)	2	24000	=(E3*F3)	=SUM(B3,E3)	=SUM(D3,G3)
4	Computers	2	1500	=(B4*C4)	0	0	=(E4*F4)	=SUM(B4,E4)	=SUM(D4,G4)
5	Travel	3	1000	=(B5*C5)	4	1050	=(E5*F5)	=SUM(B5,E5)	=SUM(D5,G5)
6	Total			=SUM(D3,D4,D5)			=SUM(G3,G4,G5)		=SUM(I3,I4,I5)
7									
8									
9									

Figure 1. Budget worksheet displaying formulae (from [10] p.299)



F1	{ SUM(B1,2:B↓,2) }
F2	{ C1,1*C1,2 }
F3	{ SUM(C1,3:C↓,1) }
F4	{ C1,1 + E1,1 }
F5	{ C1,3 + E1,2 }

Figure 2. An L-sheets program defining a family of budget spreadsheets. The annotations *F1*, *F2*,… are not part of the program, but indicate that the content of the annotated subarray is as shown in the legend. (after [6]).

contents of the subarrays of the parameter array of the head template to the corresponding cells of the goal parameter array (Figure 3b). In particular, the formula *F1* is rewritten to refer to the appropriate cells in the worksheet, then added to the content of the bottom-right cell of the rectangular array selected in the worksheet. Since the second column of the subarray **B** corresponds to column I in the sheet and stretches from row 3 to row 5, *F1* is rewritten to SUM(I3:I5). Note that the syntax of formulae in a program sheet differs from that of formulae in a worksheet. In particular, a reference in a program sheet formula consists of a subarray name followed by row and column indices, while in a worksheet a reference consists of the name of a column and the number of a row in the sheet. The ↓ in the formula SUM(B1,2:B↓,2) denotes the last row of the referenced subarray **B**.

Conceptually, the above execution step replaces the initial goal template with a **years** goal template, which is derived from the body of the applied case, and contains the two rectangular arrays from the worksheet, indicated by shading, to which subarrays **A** and **B** were bound by the unification. Execution then proceeds in a fashion analogous to Prolog.

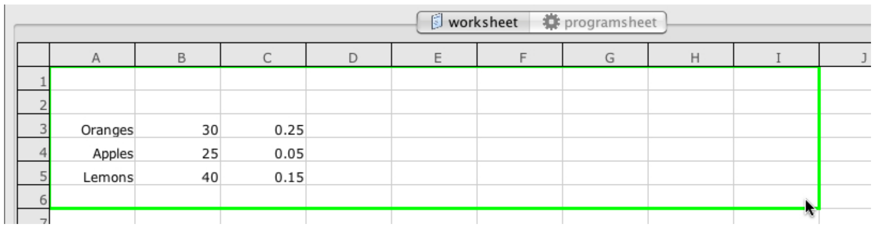
The first case of **years** contains two *rulers*, used here to indicate that the second parameter of the head templates and the array **E** in the body template both have height H. The black rectangle in the head of the second case of the **years** definition represents an array which is 0 in at least one dimension. Also, if a subarray which is variable in one or both dimensions contains a formula, then when the subarray is unified with an array of fixed size, the formula is extended in the same way that a formula in Excel is extended; that is, by appropriately incrementing the indices of references in the formula.

We leave it to the reader to complete the example, verifying that the final state of the worksheet is as shown in Figure 3c. Note that execution proceeds according to the normal Prolog execution order: that is, cases of a definition are tried in the order in which they are listed, and failure causes backtracking. In this example, failure can occur only if two arrays cannot be unified because their sizes cannot be matched.

2.2. Improved Programmability

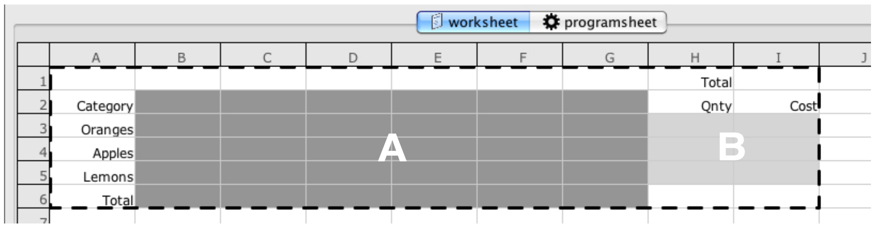
Although solving simultaneous linear equations is not a typical spreadsheet use, the L-sheets implementation of Gaussian elimination with partial pivoting, depicted in Figure 4, illustrates the improvement in programmability which has been achieved. The array labelled **A** in the head of the first case of **gauss** must be bound to an $n \times n+1$ array containing the coefficients and right-hand sides of the equations to be solved, while **C** must be bound to a $1 \times n$ array for the solution vector.

The first goal template of the single case of **gauss**, with a striped background, no name, and arrays containing only Boolean-valued expressions, is an example of a *guard*. During execution, the arrays in the guard must be instantiated to arrays of fixed size, and their formulae extended as described above. The guard succeeds if all the formulae in each of its arrays produce **true**. Note that $ in a program sheet formula has the same meaning as in a worksheet formula: that is, it indicates that the following row or column index is not

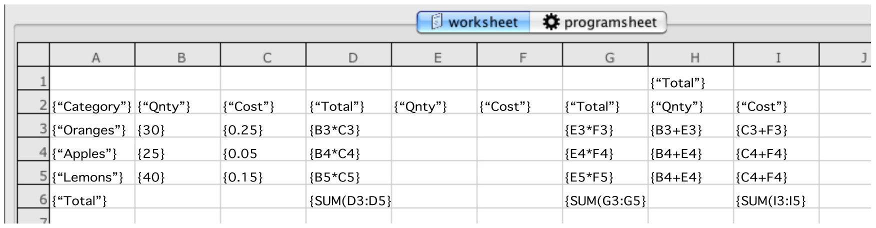


	A	B	C	D	E	F	G	H	I	J
1										
2										
3	Oranges	30	0.25							
4	Apples	25	0.05							
5	Lemons	40	0.15							
6										

(a) Selecting the parameter



(b) After unification. The dotted outline and shaded areas have been added for explanatory purposes.



	A	B	C	D	E	F	G	H	I	J
1								{“Total”}		
2	{“Category”}	{“Qnty”}	{“Cost”}	{“Total”}	{“Qnty”}	{“Cost”}	{“Total”}	{“Qnty”}	{“Cost”}	
3	{“Oranges”}	{30}	{0.25}	{B3*C3}			{E3*F3}	{B3+E3}	{C3+F3}	
4	{“Apples”}	{25}	{0.05	{B4*C4}			{E4*F4}	{B4+E4}	{C4+F4}	
5	{“Lemons”}	{40}	{0.15}	{B5*C5}			{E5*F5}	{B4+E4}	{C4+F4}	
6	{“Total”}			{SUM(D3:D5}			{SUM(G3:G5}		{SUM(I3:I5}	

(c) Spreadsheet after execution, displaying formulae rather than values.

Figure 3. Applying the budget definition to a sheet.

incremented when the formula is extended. The guard in this case specifies that the absolute value of the leading coefficient of the equation represented by subarray **B** is greater than or equal to the absolute value of the leading coefficient of each of the other equations, and is not zero. If the guard succeeds, then **B** is the pivot row.

This example illustrates that array unification, unlike term unification, is not unique. If the array to be unified with the first array in the head of the second case of **triangularise** has *n* rows, then there are *n* ways of unifying the two arrays, so during execution of this case, unification generates successive matches until one is approved by the guard template. Array unification is, in this respect, similar to the unification of lists containing segment variables in LISTLOG [12].

2.3. Observations

The motivation for L-sheets was not to invent a new programming mechanism, but to enhance the reliability of spreadsheets by improving programmability and providing a means to specify spreadsheet structure, as discussed in Section 1. Consequently, L-sheets is designed to exhibit behaviour that is as consistent with standard spreadsheet behaviour as possible. In the remainder of this section, we discuss some of the features of L-sheets that result from this design principle.

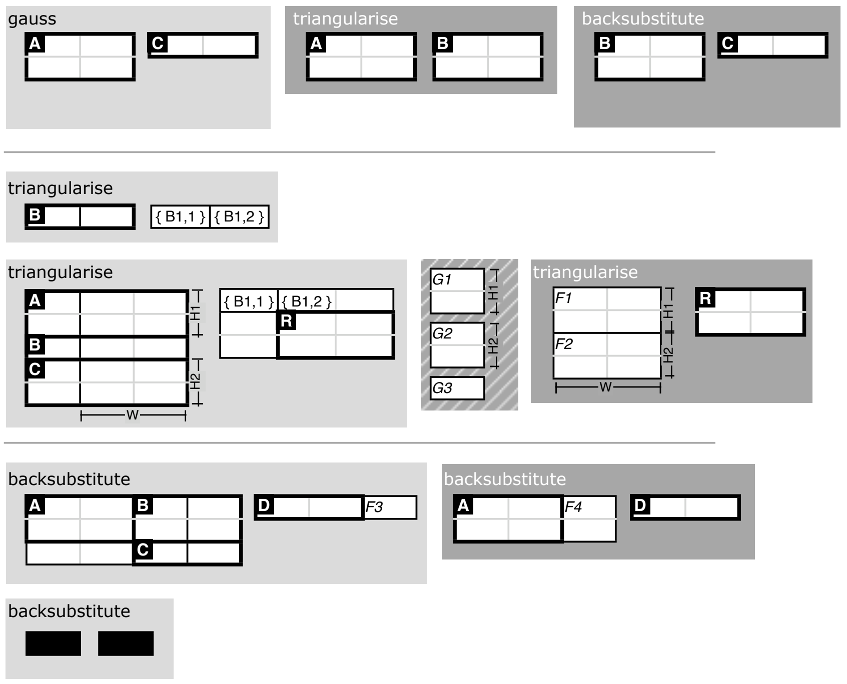
2.3.1. Occurrences of subarrays

An array that occurs in a template of a definition is "virtual" in the sense that it has no physical manifestation, unlike an array in a goal template which has a fixed position in the worksheet and includes some number of actual worksheet cells. Suppose the head of a case of a definition D includes two occurrences of an array **A**, and that a goal template is created by applying this definition to a worksheet in such a way that the two occurrences of **A** are unified with different arrays **B** and **C** in the worksheet. Each cell in **B** would become identical to the corresponding cell in **C**, in effect providing two points of access to one set of formulae. Although there may be some value to this, it violates the "ledger sheet" metaphor on which spreadsheets are based. No subarray, therefore, can occur more than once in a head template.

2.3.2. Cell content

In the example in Section 2.1, the array selected in the worksheet to be unified with the parameter of **budget** includes cells which already contain formulae, and cells to which the execution of **budget** will add formulae. Clearly a non-empty worksheet cell could quite possibly have a formula added to it by execution of a goal template. The likelihood of this happening is even greater if one of the arrays in a goal template is of variable size. For example, consider the definition **unique**, depicted in Figure 5, which produces an output array by removing duplicate values from an input array. The array selected in the worksheet for output must be of variable length, since its actual length will depend on the content of the input array, so it may well extend across cells which already have content. There are three ways of dealing with this eventuality, as follows.

- *A cell can contain at most one formula*: Execution of a goal template fails if it violates this condition. This is contrary to the behaviour of standard spreadsheets in which a new formula added to a cell replaces the current content.
- *New content replaces existing content*, as in standard spreadsheets. This solution has several undesirable consequences. First, since the new content is generated automatically by execution of a goal template, the user is not directly responsible for the change, and may find it mysterious. Second, it could occur unexpectedly as a consequence of a seemingly unrelated action, for example, as a result of editing a cell the value of which is used to compute the value of one of the cells of the input array in the **unique** example, thereby changing the size of the output array. Third, since execution of a template can produce new formulae for many cells, there could be many simultaneous changes. Finally, execution of the template may replace some cell contents resulting from a previously executed goal template T but not others, resulting in an anomalous situation in which the content of cells in the scope of X no longer corresponds to the execution of X.
- *A cell contains a set of formulae*. If all formulae compute the same value, this is the value of the cell, otherwise a conflict error occurs. This solution has the following advantages. First, it includes the standard spreadsheet model in which a cell contains one formula or is empty. Second, it does not signal errors unnecessarily. Third, it retains all formulae so the user can decide how to resolve conflict errors. For these reasons, this solution was adopted for L-sheets.



Legend

F1	{ A1,2–A1,$1*B$1,2/B$1,$1 }		*G1*	abs(B$1,1) = abs(A1,1)
F2	{ C1,2–C1,$1*B$1,2/B$1,$1 }		*G2*	abs(B$1,1) = abs(C1,1)
F3	{ C1,2 / C1,1 }		*G3*	abs(B1,1) > 0
F4	{ B1,2 – B1,1* C1,2 / C1,1 }			

Figure 4. Gaussian elimination.

Note that cells in guard templates each contain one formula.

2.3.3. *Unification*

In logic programming, unification accomplishes several tasks, which in other languages are distributed across distinct mechanisms:

(a) composing and decomposing structured data;
(b) driving conditional execution by generating failure or success; and
(c) recording computed data by variable binding.

In spreadsheets, (a) has no equivalent; (b) Boolean expressions are evaluated to select alternative formulae; and (c), values are assigned to cells via formula evaluation. To preserve familiar spreadsheet functionality, unification in L-sheets focusses on the mechanisms new to spreadsheet users: (a) the composition and decomposition of structured data, and the success or failure that results (part of (b)). The remainder of (b) is dealt with by Boolean expressions in guard templates. Aside from adding formulae to cells, unification plays no part in (c).

2.3.4. *Failure and backtracking*

In standard spreadsheets, although formulae can refer to externally defined functions, computations are primarily built by placing formulae in cells to create data flow. To preserve this view, definitions in L-sheets should, to the extent possible, behave like functions. Hence, computational elements in a worksheet cannot interact via failure-induced backtracking and re-execution, actions which are confined to the execution of individual goal templates.

2.3.5. *Variable-size arrays*

In L-sheets, variable-size arrays are combined into compound arrays, creating patterns that specify the structure of families of fixed-size arrays. Analogously, Prolog terms containing variables are patterns that specify the structure of families of ground terms. However, although it is acceptable for variables to occur in the answers to Prolog queries, it makes no sense for a worksheet to contain a variable-size array, so execution must fix the sizes of the arrays of a query template. Note, however, that it is necessary to allow variable-size arrays to be selected in a worksheet when applying a definition, for example, the definition **unique** in Figure 5.

2.3.6. *Errors*

In spreadsheets, a *cycle error* occurs if a data flow cycle is detected, and a *value error* occurs if a formula cannot be evaluated for some reason. In L-sheets, the presence of goal templates creates dependencies in addition to data dependencies, creating a graph of which the data flow graph is a subgraph (see Section 3.3). This extended digraph must also be acyclic. Other error types are required to deal with goal templates. A *conflict error* arises if two formulae in the content of a cell return different values, as discussed above. If execution of a goal template fails, or does not fix the sizes of the arrays of the template, a *goal error* is produced. A *guard error* indicates that a guard cannot be evaluated as a result of a value error in one of its arrays.

2.3.7. *Liveness*

One of the most significant features of spreadsheets is their immediate reaction to changes. If the content of a cell is changed, the values of cells affected by the change are recalculated, providing instant feedback, a property termed “level 3 liveness” by Tanimoto [24]. This behaviour is easily achieved by following data flow links to locate and recompute affected cells. Preserving this property in L-sheets is complicated by the presence of goal templates, which may or may not need to be re-executed. Furthermore, if a template is re-executed, it may change the contents of cells, altering the data flow. The next section examines liveness in detail.

3. Liveness

The examples in Section 2 consider only the logic programming aspects of L-sheets, concentrating on the execution of a single goal template in a worksheet, the level of functionality achieved in the current prototype. In general, however, a worksheet may contain several goal templates, the arrays of which may overlap, as well as formulae placed in cells by the user in the normal way. To achieve the liveness that spreadsheet users expect, we must determine how these computational elements should react when a change is made to the sheet. We first establish the conditions that trigger re-execution of a goal template; then investigate the interaction of worksheet elements as they re-execute.

3.1. Execution Triggers, Virtual Cells, And Data Flow

The factors that affect execution of a goal template are success or failure of unification, and success or failure of guards. The former depends on the dimensions of parameter arrays, so if a dimension of a parameter is changed, the goal template must be re-executed. The outcome of evaluating a guard depends on the values of some cells. In the second case of **gauss** in the example in Section 2.2, formulae in cells that are referenced by the guard template must be evaluated during the execution, while formulae constructed for the result array can be returned unevaluated to the worksheet, then evaluated in the usual way. This implies that if the values of cells in the equation array change, the **gauss** goal template must be re-executed, changing the formulae deposited in the cells of the result array. In the example in Section 2.1, however, no formulae are evaluated during execution of a **budget** goal template, so re-execution following changes in cells is unnecessary.

Figure 6 illustrates the creation of virtual cells, the data flow between them, and triggers attached to cells to initiate re-execution. In this example, **gauss** is applied to a 3×4 array containing three equations, and a result array of height 1 and variable width, as shown in the grey band at the top. Each of the following bands corresponds to a goal template with parameters from the band immediately above. For example the **gauss (1)** band corresponds

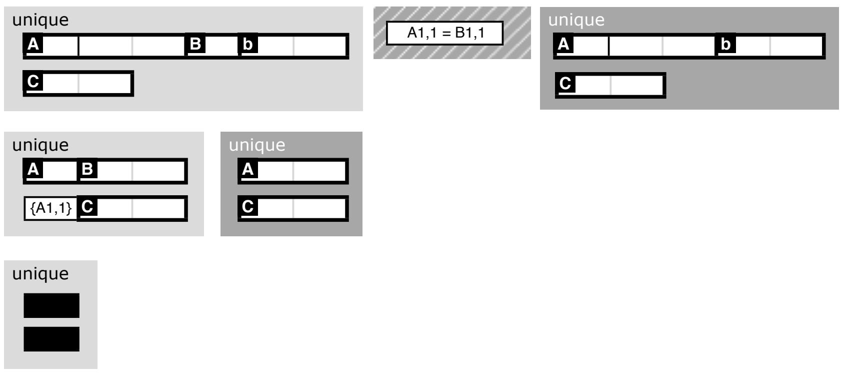


Figure 5. Removing repeated elements from an array.

to the goal template **gauss**, the parameters of which are the two arrays in the **worksheet (0)** band. The arrays in a band, reading left-to-right and top-to-bottom, correspond to the arrays in the non-guard templates of the case that has been applied to the associated goal template, in the order that they appear in the case.

An array or subarray allocated during the execution of the corresponding template is indicated by a solid border, while an array or subarray with a dashed border is a reference to an array or subarray allocated previously, indicated by a dashed arrow. For example, the first array in the band labelled **gauss (1)**, is a reference to the array in the worksheet containing the equations to be solved. The dark grey shading in the latter marks out the subarray to which the former refers, the entire array in this instance. The second array in the **gauss (1)** band is a reference to the result array in the worksheet, and the third is an array of variable height and width, allocated by this invocation of **gauss**, as indicated by the label ➊. The third array corresponds to the array named **B** in the body of **gauss**, and consists of cells which do not lie in the sheet, called virtual cells.

The band labelled **triangularise (2)** shows the arrays involved when the second case of **triangularise** is applied, the parameters in the head of the first case having failed to unify with the given arrays. Note that the fourth array is a reference to a subarray of the second, and that the third consists of 6 virtual cells allocated by **triangularise (2)**. Since the guard template and unification algorithm negotiate to select the pivot equation in the first parameter of the head template, as explained in Section 2.2, the values of cells in the first column of the equation array will determine the composition of the formulae deposited in the second and third arrays, so if any of them subsequently change, **triangularise (2)** must be re-executed. Hence *triggers* are attached to these cells, indicated by the annotation 2*. This invocation of **triangularise** adds formulae to the contents of certain cells, indicated by the annotation 2 in the cells, and the solid arrows labelled 2, each denoting the presence of a number of data flow links. For example, the formula in the top left cell of the third array refers to the first and second cells of the first row in the first array, and the first

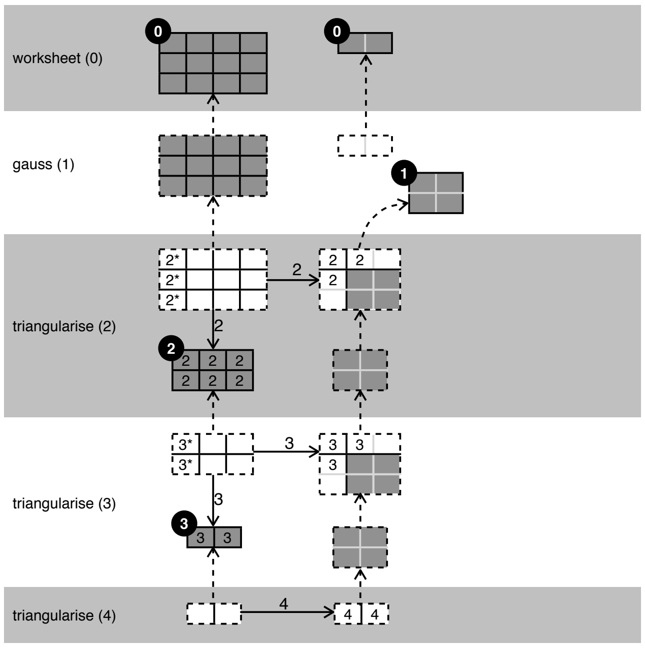


Figure 6. Creating virtual cells, data flow and triggers.

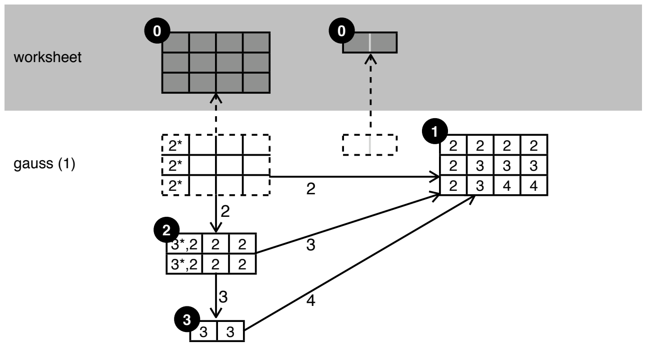


Figure 7. Structure produced by the execution of **triangularise**.

and second cells of the pivot equation row in the first array. Hence there are four data flow links to this cell from cells of the first array, and in total, the solid arrow from the first to the third array represents 24 data flow links. The reader is invited to work through the construction of the rest of this diagram. Note that the dimensions of the third array in the **gauss (1)** band are finally determined when the base case of **triangularise** is executed, as shown in the last band.

Figure 7 is obtained by deleting the bands corresponding to goal templates which have been completely executed, removing array references in those bands and appropriately reassigning data flow links incident on array references. Except for the array references in the **gauss (1)** band, which will be removed later, and a further pruning of the structure discussed below, the resulting diagram depicts the structure of data at this point.

The diagram in Figure 8 shows the arrays involved in the execution of **backsubstitute**. Some of these consist of a combination of virtual cells and references: for example, the cells in the last column of the third array in **backsubstitute (5)** are virtual, while the rest of this array is a reference to the shaded subarray in the first array in this band.

The structure remaining after execution of **gauss** is shown in Figure 9. Bearing in mind that this diagram does not show the details of data flow present in the actual structure, we note the following. If a value in the third or fourth column of the equation array changes, no re-execution is necessary. If a value in the second column of the equation array changes, then a value in the first column of the first reduced array will change, triggering re-execution of **triangularise (3)**, and the dependent invocation **triangularise (4)**. First, any virtual cells labelled ➌ or ➍, and formulae indicated by the cell annotations 3 and 4 are deleted, **triangularise (3)** is re-executed, and consequent changes in cell values are propagated in the usual way. Since the shape of the triangular array is unchanged and there are no 5* triggers, **backsubstitute (5)** is not re-executed.

The structure in Figure 9 can be further pruned, resulting in the diagram in Figure 10, as follows. Since the array of virtual cells labelled ➌ will be deleted prior to any re-execution, its only function is to contain intermediate formulae which are referenced by formulae in cells marked 4 in the triangular array. This array can therefore be deleted, and the formulae it contains embedded directly in the formulae that refer to them. Similar reasoning leads to the deletion of the arrays of virtual cells labelled ➎ and ➏. We could also remove the triangular array, and embed the formulae it contains in the formulae of the result array in the worksheet: this, however, would necessitate re-execution of **backsubstitute (5)** following any re-execution of **triangularise (2)** or **triangularise (3)**, even though **backsubstitute (5)** is sensitive only to the size of

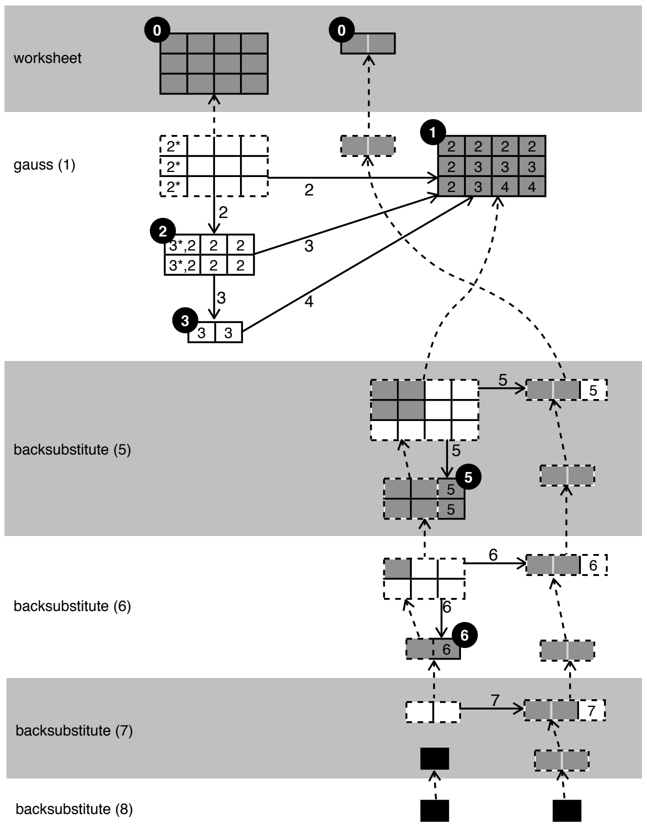


Figure 8. Structure produced by execution of backsubstitute.

the triangular array, not its content. Note that the goal templates that accompany this structure are **gauss(1)**, **triangularise(2)** and **triangularise(3)**.

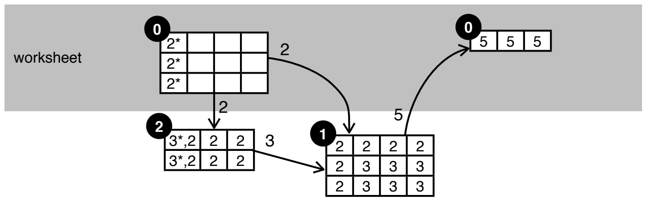


Figure 10: Final pruned structure of virtual cells.

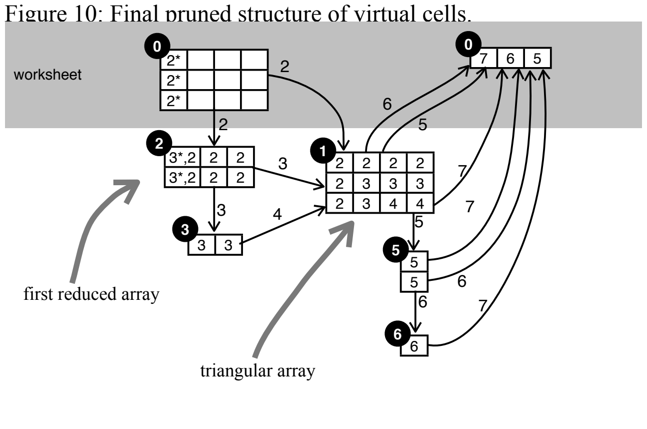


Figure 9. Virtual cells, data flow and triggers after execution of **gauss**.

The final, pruned structure in Figure 10 contains the smallest set of virtual cells required to avoid rebuilding formulae unnecessarily during re-execution initiated by changes in the values of triggered cells. Note that a virtual cell pruned away in the final step could be referenced by more than one formula, in which case its formula would be embedded in more than one other formula, and computed more than once during any subsequent evaluation. Such virtual cells could be retained to achieve more economical computation. At the other extreme, all virtual cells and associated goals could be removed, in which case any trigger associated with a virtual cell would be propagated to worksheet cells referenced by the virtual cell's formula, resulting, in our example, in the structure in Figure 11. Note that when a trigger is propagated during removal of the virtual cell that contains it, it is changed to refer to the goal which is the nearest remaining ancestor of the goal to which it originally referred.

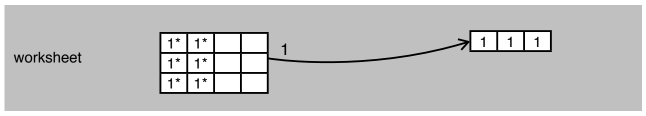


Figure 11. Structure with no virtual cells.

Clearly, applying the above analysis to execution of a **budget** goal template will produce a structure, analogous to the diagram in Figure 10, with no virtual cells or triggers.

The above example does not illustrate some subtleties of the process of attaching triggers to cells. Consider some definition **A** with three cases, which, by the Prolog analogy mentioned in Section 2.1, we can represent as follows:

```
A(X) :- G1(X), ...
A(Y) :- B(Y,W),G2(W), ...
A(Z) :- ...
```

where **G1** is a guard that refers to cell **X** occurring in the head of the first case, and **G2** is a guard that refers to a cell **W**, the content of which is determined during execution of the template **B**, which in turn refers to cell **Y** in the head of the second case. Suppose that the execution that ensues when this definition is applied to a goal template in the worksheet terminates with the second case of **A**. In the process, a trigger will be attached to the worksheet cell, $\mathbf{X_0}$, corresponding to **X** in the head of the first case, and to the virtual cell, $\mathbf{W_0}$, corresponding to **W** in the second case. Hence, triggers accumulate as execution advances through the cases of a definition.

Now suppose the value of $\mathbf{W_0}$ changes, triggering re-execution, which terminates with the third case of **A**. Since backtracking out of the second case will delete associated structures including the virtual cell $\mathbf{W_0}$, the trigger on this cell must first be propagated back to any cells to which the formula in $\mathbf{W_0}$ refers.

Finally, if the definition of **B** includes guards, the structure resulting from the initial execution described above will contain virtual cells allocated during execution of the **B(Y,W)** goal, with triggers arising from these guards. During backtracking out of the second case of **A**, these triggers must be propagated back to referring cells as described above.

3.2. Execution Graph Of A Query

Regardless of the degree of pruning, the resulting structure of goals, cells, data flow links, triggers and dependencies is called the *execution graph* of the query. An execution graph has five types of vertices, four of which are depicted in Figure 12 which corresponds to the final pruned structure in Figure 10. The four types of vertices in this figure are *worksheet cells*, *virtual cells* and *goals*, represented by black squares, grey squares and diamonds, respectively, and *arrays*, each represented by a grey box enclosing the vertices corresponding to the cells of which the corresponding array is composed. This arrangement is meant to make the diagram readable, not to imply a relationship between an array vertex and the vertices it contains. As discussed below, the function of an array vertex is simply to provide a reference to the dimensions of the corresponding array.

In Figure 13, we have added *formula* vertices (the fifth type), represented by circles, together with associated *data flow* edges, represented by plain arrows, and *dependency* edges, represented by dotted arrows. To avoid clutter, we have shown only some of the formula vertices and dependency edges that occur in our example. A formula vertex corresponds to one of the formulae in the content of one cell: hence there is exactly one data flow edge leaving a formula vertex, and any number of data flow edges entering a cell vertex. A dependency edge connects a goal vertex to another vertex which exists because of the execution of the goal: hence there is no dependency edge entering a worksheet cell, and exactly one dependency edge entering each other vertex. We have assumed that the last equation turns out to be the pivot at each level of triangularisation.

Finally, in Figure 14 we have added the *trigger* edges, represented by arrows with diamond-shaped heads. A trigger edge enters a goal vertex, and indicates the need to re-execute the goal if the value of the vertex (a cell or array) at the tail of the edge changes. Note that although goal 3 should be re-executed if the dimension of the array represented by the lower left array vertex changes, it is unnecessary to include a trigger edge indicating this, since the dimension can change only if goal 2 is re-executed, in which case goal 3 will be re-executed because of the dependency edge from 2 to 3.

Let D be the domain over which formulae are constructed, including *c*, *v*, *x*, *q*, *g* which respectively denote cycle, value, conflict, goal and guard errors, and ⊗, denoting "no value". A value is associated with each vertex in the execution graph, as illustrated by the following examples.

- *Successful execution:* Suppose formulae in the cells of the first parameter array in the above example produce the values displayed in the corresponding cell vertices in Figure 15. After execution, the other vertices have the values shown in or beside them. All goal vertices have the value ⊗, indicating success.
- *Execution with evaluation errors:* Now suppose the formulae in the cells of the first parameter array in the above example produce the values shown in Figure 16. The values of the upper two vertices in the last column indicate errors arising from the evaluation of the formulae in the corresponding cells. The value ⊗ in the bottom vertex of this column indicates that the corresponding cell is empty. Neither the errors nor the ⊗ value cause execution of the query to fail, so the structure of the execution graph is the same. The values, however, are as shown, assuming the Excel convention that ⊗ is equivalent to 0 when evaluating arithmetic expressions. Although the for-

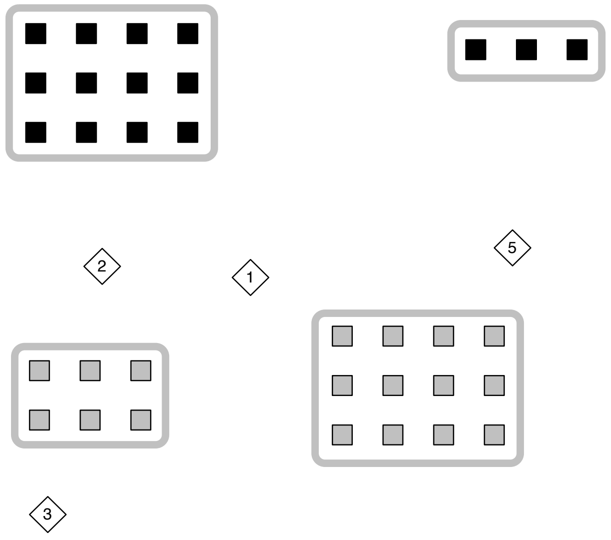


Figure 12. Cell, array and goal vertices of an execution graph of the **gauss** query.

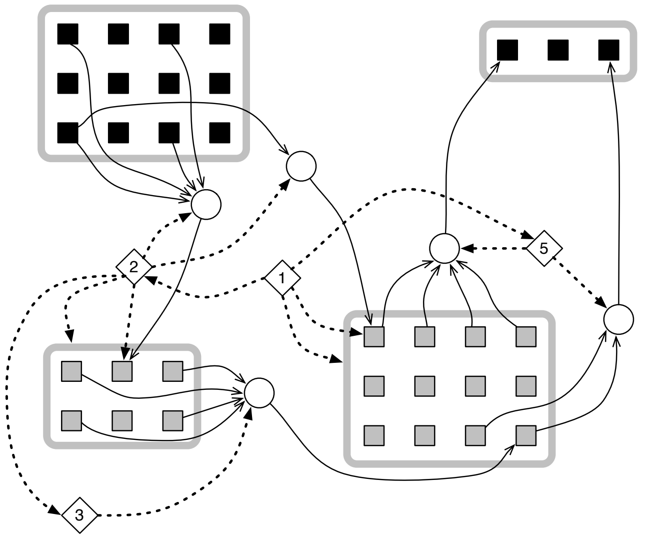


Figure 13. An execution graph of the **gauss** query, including some of the formula vertices and associated data flow edges, and some of the dependency edges.

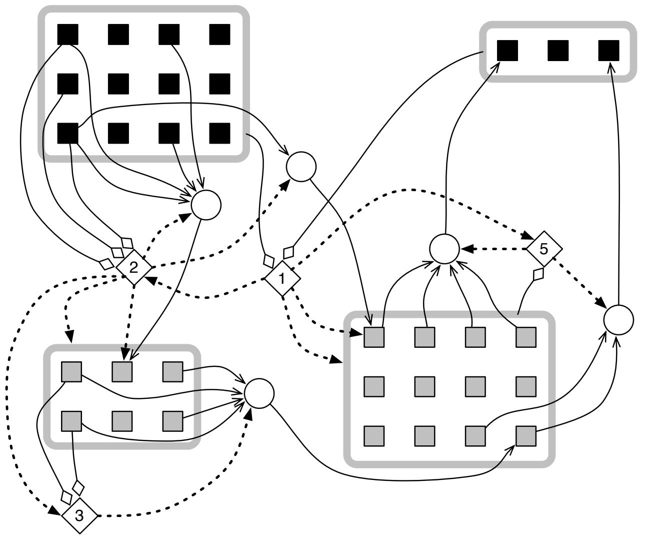


Figure 14. An execution graph of the **gauss** query, including the trigger edges.

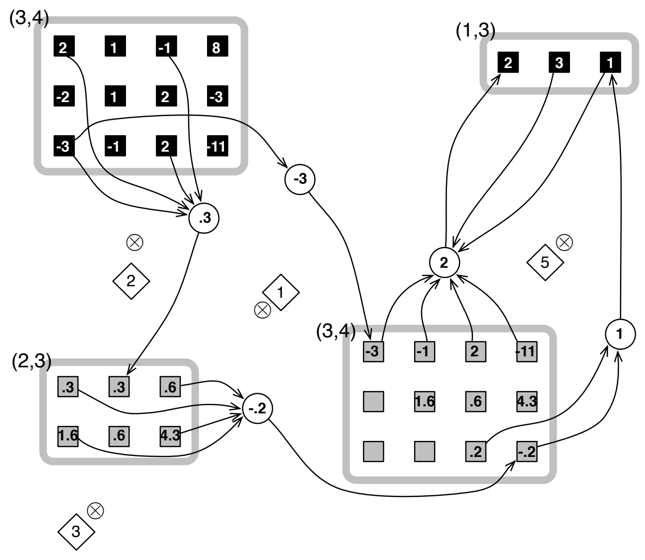


Figure 15. Values for vertices of an execution graph of the **gauss** query, following successful execution.

mulae introduced by the application of **gauss** propagate the *v* and *x* errors, the errors are inherent in the worksheet: hence the goal vertices in the graph have the value ⊗.

- *Execution with cycle errors:* Figure 17 shows the values that result if the third cell in the first row of the first parameter array contains a formula dependent on the first cell of the second parameter array. In contrast to the previous example, the cycle error is not solely due to the worksheet the formulae introduced by the application of **gauss**, hence the cycle error is propagated from formula vertices to the goal vertices on which they depend. The cycle in this example involves only data flow edges, but in general, cycles can include edges of all three kinds.

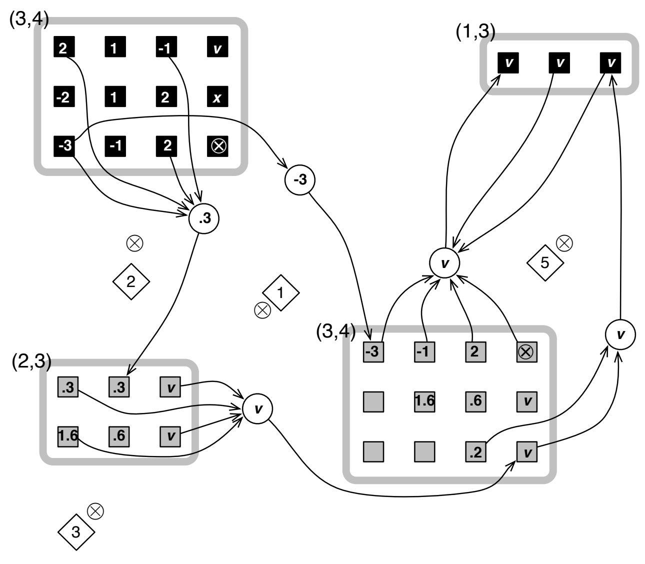


Figure 16. Values of execution graph vertices following execution with evaluation errors.

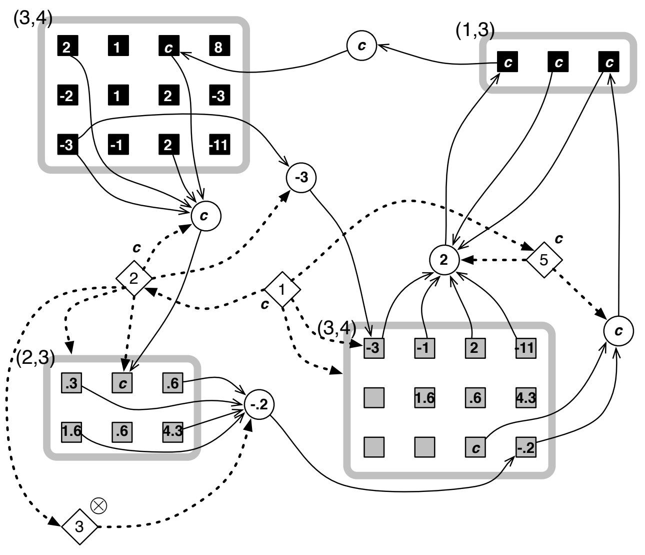


Figure 17. Values of execution graph vertices following execution with cycle errors.

- *Failure:* If a definition is applied to arrays of inappropriate size, unification will fail, resulting in a goal error, illustrated in Figure 18.

3.3. Interaction Of Worksheet Elements

In this section, we examine the re-execution that follows editing of a worksheet. First we formally define the structure of execution graphs, introduced in the above examples, omitting array vertices, since our main focus is re-execution following a change in cell contents.

If W is a worksheet, let $\mathcal{C}$ denote the set consisting of the cells in W and all virtual cells, $\mathcal{G}$ the set of goal templates in W and $\mathcal{F}$ the set of all occurrences of formulae in $\mathcal{C}$. The *computation graph* of W is a directed graph $\mathcal{E}$=(V,E) where V = $\mathcal{G} \cup \mathcal{C} \cup \mathcal{F}$, and E = $\mathcal{D} \cup \mathcal{X} \cup \mathcal{T}$ where

$\mathcal{D}$ = { (c,f) | c ∈ $\mathcal{C}$, f ∈ $\mathcal{F}$ and f contains a reference to c } ∪
{ (f,c) | c ∈ $\mathcal{C}$, f ∈ $\mathcal{F}$ and f is in the content of c }

$\mathcal{X}$ = { (g,f) | g ∈ $\mathcal{G}$, f ∈ $\mathcal{F}$ and f is created by the execution of g } ∪
{ (g_1,g_2) | g_1,g_2 ∈ $\mathcal{G}$ and g_2 is created by the execution of g_1 }

$\mathcal{T}$ = { (c,g) | c ∈ $\mathcal{C}$, g ∈ $\mathcal{G}$ and a trigger for g is associated with c }

$\mathcal{D}$, $\mathcal{X}$ and $\mathcal{T}$ are the sets of *dataflow*, *dependency* and *trigger* edges, respectively.

Note that "worksheet" and "computation graph" are actually synonymous: the latter is simply a structural description of the former.

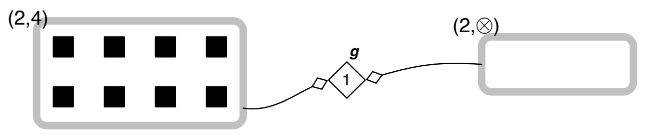


Figure 18. Values of execution graph vertices following an execution with goal error.

DEFINITION. If $\mathcal{E}$ is a computation graph and $C_1 \subseteq C \subseteq \mathcal{C}$ then:

- If $g \in \mathcal{G}$, then g is *independent* of C iff for every $c \in C$, there is no walk from c to g.
- If $G \subseteq \mathcal{G}$ and $f \in \mathcal{F}$, f is a *barrier for* C_1 *in* C[G] iff there is a walk from some $c \in C - C_1$ to f which includes at least one trigger edge incident on some $g \notin G$.
- If $G \subseteq \mathcal{G}$ and $g \in \mathcal{G}$, g is *partially controlled by* C_1 *in* C[G] iff for every $c \in C_1$ there exists at least one walk from c to g, and there is no walk from c to g that includes a barrier for C_1 in C[G]. g is *fully controlled by* C_1 *in* C[G] iff there is no C_2 such that g is partially controlled by C_2 in C[G], and $C_1 \subset C_2 \subseteq C$. If $g \in \mathcal{G}$ is fully controlled by C_1 in C[G], C_1 is called the *activator set for* g *in* C[G].

LEMMA. If $\mathcal{E}$ is acyclic, $g \in \mathcal{G}$, $G \subseteq \mathcal{G}$ and $C \subseteq \mathcal{C}$, then either g is independent of C or there exists a unique subset C_1 of C such that g is fully controlled by C_1 in C[G]. In the latter case, let $\mathcal{G}_C$ be the set of all goals fully controlled by subsets of C, then there exists at least one $g \in \mathcal{G}_C - G$ such that for all $g_1 \in \mathcal{G}_C$, $(g_1,g) \notin \mathcal{X}$. g is said to be a *candidate* for C[G].

Simply put, the above lemma guarantees that, when the value of every cell in some set C of an acyclic worksheet changes, either there are no goals that need to be re-executed, in which case the changes can simply be propagated by data-driven data flow in the usual way; or among the goals that must be re-executed, there is at least one which does not rely on values produced by re-execution of some others. It is important to note, however, the role of the suffix [G] in the definition. In practice, when the content of a cell changes, its value may not, so we cannot assume that a goal must be re-executed if the content of a cell in its activator set changes. Hence, if a candidate g is selected, but evaluating its trigger cells does not change their values, g should not be re-executed, and should be excluded when deciding which formulae are barriers while searching for another candidate.

This leads to the algorithm in Figure 19 for updating the worksheet following a change in the content of each cell in some set C. In line 9, "vertices dependent on g" means all vertices reachable from g by a walk consisting of dependency edges. "Data flow descendents" in line 13 means vertices reachable by walks consisting of dataflow edges. Execution of g in line 10 will halt if a guard or cycle error occurs.

Other changes that may be made to a worksheet are handled as follows.

a) *A goal template* g *is added to the sheet:* Let C=∅ and run the re-execution algorithm from line 10.

b) *A goal template* g *is removed:* Delete all dependent vertices, let C be the set of remaining cells the content of which has changed. Run the re-execution algorithm.

c) *A parameter array of a goal template is resized:* The template is removed (b) and a new template added (a).

4. Concluding remarks

Many of the features of spreadsheets that have contributed to their runaway popularity with end-users also make them highly error-prone and therefore unsuitable for the strategically important applications for which they are routinely used. Primitive programming facilities, lack of high-level structural abstractions, and lack of debugging and testing tools all contribute to the errors that occur in the majority of spreadsheet applications, with well known negative consequences.

```
 1  repeat {
 2    G = ∅;
 3    found = false;
 4    while (there is a candidate for C[G] and not found)
      {
 5      g = some candidate for C[G];
 6      evaluate trigger cells for g;
 7      if (values have changed) found = true;
      }
 8    if (found) {
 9      delete all vertices dependent on g;
10      execute g;
11      add cells with changed content to C;
     }
12  until (not found) ;
13  evaluate all data flow descendents of cells in C ;
```

Figure 19. Worksheet re-execution.

To rectify these problems, L-sheets augments the simple data flow model of spreadsheets with a form of logic programming in which the user can draw rectangular arrays of cells, specify how such arrays are composed of smaller arrays, and how they are to be filled with formulae. The resulting visual language provides a mechanism for describing the structure of a spreadsheet, as well as programming facilities superior to the simple data flow of formulae in cells. Although logic programming may be beyond the grasp of the majority of spreadsheet users, the L-sheets version, which deals with visually represented arrays rather than abstract terms, may be more palatable. Furthermore, beginners can start with the standard data flow model, incorporate pre-built definitions, then gradually move towards building their own.

In this paper, we have considered the problem of maintaining in L-sheets one of the most important features of spreadsheets, the immediate re-execution of those cells affected by an edit. This involved studying the interaction between computational elements in a worksheet, that is, goal templates and formulae in cells, which can be entered by the user or deposited by execution of goal templates. We noted that the behaviour of the logic programming extension should be as consistent as possible with standard spreadsheet behaviour, and that this principle has certain consequences for the design of the language, in turn affecting the way changes propagate through the computational elements in a sheet. These considerations led to a re-execution algorithm.

Issues we are currently investigating or intending to investigate are as follows.

- *Indicating or enforcing correct parameter size.* In the current prototype, when applying a definition to a sheet the user is given some help in choosing arrays of acceptable size. For example, as the user drags to mark out a rectangle of cells when applying **budget**, the outline of the rectangle is coloured green if the rectangle has 3*n* columns for some non-zero *n*, and at least 3 rows, and red otherwise. This is accomplished by repeating a simplified version of execution as the drag occurs, a brute force approach with significant limitations. We are at present exploring a more sophisticated method that relies on an analysis of the size constraints in the definitions.
- *Computing formats.* Arrays in program sheets will contain formats such as font, cell colours and borders, which will be transferred to the worksheet during unification. Such formats will become another part of a cell's content, so it will be necessary to determine the rules for combining formats and re-

solving format conflicts, and whether changes in formats should trigger re-execution.

- *Re-execution.* The present prototype of L-sheets implements the standard spreadsheet model, including re-execution of data flow, the building and editing of program sheets, and the application of definitions to worksheets. We have yet to incorporate the full re-execution algorithm described above.
- *Debugging facilities.* The errors that can occur during execution are discussed above. Appropriate debugging tools must be devised for tracking and fixing each of them. In addition, various debugging tools and methodologies have been proposed for standard spreadsheets [20]. Research is required to determine the extent to which they can be incorporated into L-sheets.
- *Language enhancements.* In the current version of the language, nesting of named arrays is not allowed, primarily to avoid complexity in the visual notation. Some programs, however, are more simply expressed with nested named arrays. We will investigate ways of incorporating them without complicating the interface.
- *More powerful array structures.* In the current version of L-sheets, an array of variable width is homogeneous horizontally: that is, it represents a repetition of similar columns. An array made up of repeated blocks of dissimilar columns must be specified by a definition, for example, the **years** definition in Figure 2. The same applies to arrays of repeated blocks of dissimilar rows. We are looking into extending the definition of arrays, and array unification, to allow such repetition of blocks.
- *Deducing definitions.* Several researchers have described mechanisms for analysing and generalising patterns in the content of spreadsheets to generate structure specifications [10, 14]. Spreadsheets such as those defined by **budget**, for example, are amenable to such analysis. We will explore the possibility of using these or other mechanisms to automatically generate L-sheets definitions. In particular, we are interested to see how far such automatic generation can be pushed beyond simple examples such as **budget**.

5. Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada via Discovery Grant OGP0000124.

6. References

[1] Abraham, R. and Erwig, M. 2006. Inferring templates from spreadsheets. In *Proceedings of the 28th International Conference on Software engineering* (Shanghai, China, May 20 - 28, 2006). ACM, New York, NY, USA, 182-191. DOI= http://doi.acm.org/10.1145/1134285.1134312

[2] Bricklin, D. and Frankston, R. *VisiCalc: Information from its creators.* http://www.bricklin.com/visicalc.htm. Accessed April 9, 2009

[3] Burnett, M., Atwood, J., Djang, R.W., Reichwein, J., Gottfried, H. and Yang, S. 2001. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Funct. Program.*, 11, 2 (March 2001), 155-206.

[4] Cervesato, I. 2007. NEXCEL, a deductive spreadsheet. *The Knowledge Engineering Review*, 22, 03 2007), 221-236. DOI= http://dx.doi.org/10.1017/S0269888907001142

[5] Chambers, C. and Erwig, M. 2008. Dimension inference in spreadsheets. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, 2008* (Herrsching am Ammersee, Germany, September 19-21, 2008). IEEE Computer Society, Piscataway, NJ, USA, 123-130. DOI= http://dx.doi.org/10.1109/VLHCC.2008.4639072

[6] Cox, P.T. 2007. Enhancing the Programmability of Spreadsheets with Logic Programming. In *Proceedings of the Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (Coeur d'Alène, ID, USA, September 23-27, 2007). IEEE Computer Society, Piscataway, NJ, USA, 87-94. DOI= http://dx.doi.org/10.1109/VLHCC.2007.18

[7] Cox, P.T. and Nicholson, P. 2008. Unification of Arrays in Spreadsheets with Logic Programming. In *Proceedings of the Practical Aspects of Declarative Languages* (San Francisco CA, 2008). Lecture Notes in Computer Science 4902. Springer, Berlin, 100-115. DOI= http://dx.doi.org/10.1007/978-3-540-77442-6_8

[8] Croll, G.J. 2008. In Pursuit of Spreadsheet Excellence. In *Proceedings of the European Spreadsheet Risks Interest Group* (Greenwich, UK, July, 2008).

[9] Erwig, M. and Burnett, M.M. 2002. Adding Apples and Oranges. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages* (Portland, OR, USA, January 19-20, 2002). Lecture Notes in Computer Science 2257. Springer, Berlin, 173-191. DOI= http://dx.doi.org/10.1007/3-540-45587-6

[10] Erwig, M., Abraham, R., Kollmansberger, S. and Cooperstein, I. 2006. Gencel: a program generator for correct spreadsheets. *Journal of Functional Programming*, 16, 3 (May 2006), 293-325. DOI= http://dx.doi.org/10.1017/S0956796805005794

[11] EuSpRIG. European Spreadsheet Risks Interest Group. *Spreadsheet mistakes - news stories.* http://www.eusprig.org/stories.htm. Accessed July 2008

[12] Farkas, Z. 1987. LISTLOG - A Prolog Extension for List Processing. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development* (Pisa, Italy, March 23-27, 1987). Lecture Notes in Computer Science 250. Springer-Verlag, Berlin, 82-95. DOI= http://dx.doi.org/10.1007/BFb0014968

[13] Gupta, G. and Akhter, S.F. 2000. Knowledgesheet: A Graphical Spreadsheet Interface for Interactively Developing a Class of Constraint Programs. In *Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages* (Boston, MA, USA, 2000). Lecture Notes in Computer Science 1753. Springer-Verlag, Berlin, 308-323. DOI= http://dx.doi.org/10.1007/3-540-46584-7

[14] Paine, J. 2004. Spreadsheet Structure Discovery with Logic Programming. In *Proceedings of the European Spreadsheet Risks Interest Group* (Greenwich, UK, 2004). 121-133.

[15] Paine, J. 2005. Excelsior: bringing the benefits of modularisation to Excel. In *Proceedings of the European Spreadsheet Risks Interest Group* (Greenwich, UK, July, 2005). 173-184.

[16] Panko, R.R. 2006. Spreadsheets and Sarbanes-Oxley: Regulations, Risks, and Control Frameworks. *Communications of the Association for Information Systems*, 17, Article 29 (2006), 647-676. Available at: http://aisel.aisnet.org/cais/vol17/iss1/29

[17] Peyton Jones, S., Blackwell, A. and Burnett, M. 2003. A user-centred approach to functions in Excel. In *Proceedings of the Proceedings of the eighth ACM SIGPLAN international conference on Functional programming* (Uppsala, Sweden, June, 2003). ACM, New York, NY, USA, 165-176. DOI= http://doi.acm.org/10.1145/944705.944721

[18] Rajalingham, K., Chadwick, D. and Knight, B. 2001. An Evaluation of a Structured Spreadsheet Development Methodology. In *Proceedings of the European Spreadsheet Risks Interest Group* (Greenwich UK, July, 2001). 39-59.

[19] Ramakrishnan, C., Ramakrishnan, I. and Warren, D. 2006. Deductive Spreadsheets Using Tabled Logic Programming. In *Proceedings of the 22nd International Conference on Logic Programming* (Seattle, WA, USA, August 10 - 22, 2006). Lecture Notes in Computer Science 4079. Springer, Berlin, 391-405. DOI= http://dx.doi.org/10.1007/11799573_29

[20] Ruthruff, J.R., Prabhakararao, S., Reichwein, J., Cook, C., Creswick, E. and Burnett, M. 2005. Interactive, visual fault localization support for end-user programmers. *Journal of Visual Languages and Computing*, 16, 1-2 (April 2005), 3-40. DOI= http://dx.doi.org/10.1016/j.jvlc.2004.07.001

[21] Saadat, S. 2001. Managing Critical Spreadsheets in a Compliant Environment. In *Proceedings of the European Spreadsheet Risks Interest Group* (Greenwich, UK, July, 2001). 21-24.

[22] Scaffidi, C., Shaw, M. and Myers, B. 2005. Estimating the numbers of end users and end user programmers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* (Dallas, TX, USA, September 20-24, 2005). IEEE Computer Society, Piscataway, NJ, USA, 207-214. DOI= http://dx.doi.org/10.1109/VLHCC.2005.34

[23] Spenke, M. and Beilken, C. 1989. A spreadsheet interface for logic programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Wings for the mind* (Austin, TX, USA, April 30 - May 4, 1989). ACM, New York, NY, USA, 75-80. DOI= http://doi.acm.org/10.1145/67449.67466

[24] Tanimoto, S.L. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages and Computing*, 1, 2 (June 1990), 127-139. DOI= http://dx.doi.org/10.1016/S1045-926X(05)80012-6

[25] van Emden, M.H., Ohki, M. and Takeuchi, A. 1986. Spreadsheets with incremental queries as a user interface for logic programming. *New Gen. Comput.*, 4, 3 (October 1986), 287-304.

[26] Wilson, S. 1997. Building a Visual Programming Language. *MacTech*, 13, 4 (1997). Available at: http://www.mactech.com/articles/mactech/Vol.13/13.04/Spreadsheet2000/