The Eductive Implementation of a Three-dimensional Spreadsheet*

Weichang Du and William W. Wadge

Abstract

This paper presents an implementation technique for a three-dimensional spreadsheet based on a dataflow computation model called eduction. In the evaluation of a spreadsheet, initiated by the user’s request, demands for values of various cells of the spreadsheet are sent out. The demands cause the operations computing the requested values to be performed if the values of their operands are available, otherwise new demands will in turn be sent out to other operations. Once all the demands are satisfied, the requested cell values are available. This technique shows potential not only for spreadsheet implementations but also for other software packages such as graphics and simulators.

Keywords: Spreadsheet, Intensional programming, Applicative programming, Demand-driven evaluation

1 Introduction

Computer spreadsheets, as calculating tools, have user-friendly interfaces and are available for a wide range of microcomputers. Conventional spreadsheets, however, are usually hard to program for complex problems, since they were designed mainly for business applications and for use by computer novices. They appear to be inadequate for wider applications, especially in science and engineering. This inadequacy limits the potential of spreadsheets as a programming paradigm.

To make up the deficiencies of the conventional spreadsheets, a new spreadsheet, called the intensional spreadsheet, has been designed. This spreadsheet is three-dimensional, involving two spatial dimensions and one temporal dimension. The spreadsheet is expected to be suitable for a wide range of applications and for various users, including those who have had conventional programming experience. The most important feature of the intensional spreadsheet is that it is based on a type of formal logic—intensional logic. Intensional logic is concerned with assertions and other expressions whose meaning depends on an implicit context. In the sense of intensional logic, the whole spreadsheet is considered as a single entity whose value consists of values of cells associated with spatial points and moments in time. A high-level programming language, Plane Lucid (which is an extension of the programming language Lucid [10]), is used for programming the spreadsheet, so that values of cells can be defined through intensional operations. In the next section a brief description of the intensional spreadsheet design is given, and readers may refer to References [6, 7] for more details.

The intensional spreadsheet has been implemented by a novel technique called eduction [8, 3]. Eduction is a tagged demand-driven computation model. In this model, operations are driven by demands for results of these operations and availability of their operands. Eduction is suitable for implementing not only the intensional spreadsheet but also other conventional spreadsheets. The idea of the eductive spreadsheet implementation is to let the evaluation of the spreadsheet be driven by demands of the user for values of various cells of the spreadsheet. To satisfy these demands, the tagged demands for the values of the cells are sent out, where the tags specify the spatial and temporal coordinates of the requested values. The demanded operations in the

definitions of these cells may in turn demand values in other cells, and are eventually performed when the values of their operands become available. The evaluation is completed when all demands are satisfied. Using eduction, unnecessary computation can be avoided, and operations can be performed in any order depending only on data dependencies among operations. This approach provides a great opportunity for large complex spreadsheets to be efficiently implemented in a parallel environment. Eduction can also help solve the problem of detecting dependencies among cells by building dependency lists for cells dynamically during evaluation.

2 The Intensional Spreadsheet

2.1 Programming language—Plane Lucid

The definition language of the intensional spreadsheet, *Plane Lucid*, is a spatial extension of the declarative programming language Lucid. Plane Lucid is used to define individual cells and global entities such as user-defined functions. The syntax of Plane Lucid allows an expression to be associated with a *where* clause in which variables local to the expression can be defined [10]. The expression associated with a *where* clause is also called the subject of the clause. For example, the following expression's *where* clause defines two local variables, a and b, and the expression has the constant value 3:

\[
\begin{align*}
  a + b \\
  \text{where} \\
  a = 1; \\
  b = 2; \\
  \text{end}
\end{align*}
\]

Plane Lucid provides a set of operators that permit the values of expressions to vary in space and time. These operators are called *intensional operators*. For the three dimensions (two spatial and one temporal), there are five kinds of intensional operators to switch context, i.e., spatial and temporal coordinates. The intensional operators for the horizontal dimension in space are *right*, *left*, *hsby* (for ‘horizontally succeeded by’), *hbf* (for ‘horizontally before’) and *side*. Similarly, the corresponding intensional operators for the vertical dimension in space are *up*, *down*, *vsby* (for ‘vertically succeeded by’), *vbf* (for ‘vertically before’) and *edge*. The corresponding intensional operators for the temporal dimension are *next*, *prev* (for ‘previous’), *fby* (for ‘followed by’), *before* and *first*. Table 1 lists the informal meanings of the intensional operators in Plane Lucid.

These intensional operators can also be classified into three groups according to how they switch context from one point to another and combine values from different contexts. A relative context-switching operator switches context from the current point to a neighboring point in a given dimension. An absolute context-switching operator switches context from the current point to a particular point in a given dimension whose coordinate in the dimension is 0. A dimensional context-switching operator can be used to define, iteratively in most cases, values of an expression at all points in a given dimension. For example, the definition:

\[
\text{sum} = 0 \text{hsby sum} + X
\]

defines all the partial sums of the vector \( X \) in the positive direction of the horizontal dimension.

Besides the intensional operators, other ordinary operators, such as +, −, *, / and *if–then–else*, in Plane Lucid are pointwise. A pointwise operator does not change the underlying context. When a pointwise operation is performed at a spatial point at a moment in time, its operands are always evaluated in the same context.
Table 1: The meanings of Plane Lucid intensional operators.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Evaluation results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Relative context-switching operators:</td>
<td></td>
</tr>
<tr>
<td>right $x$</td>
<td>$x$ at the point to the right of the evaluation point in space</td>
</tr>
<tr>
<td>up $x$</td>
<td>$x$ at the point above the evaluation point in space</td>
</tr>
<tr>
<td>next $x$</td>
<td>$x$ at the point next to the evaluation point in space</td>
</tr>
<tr>
<td>left $x$</td>
<td>$x$ at the point to the left of the evaluation point in space</td>
</tr>
<tr>
<td>down $x$</td>
<td>$x$ at the point below the evaluation point in space</td>
</tr>
<tr>
<td>prev $x$</td>
<td>$x$ at the point before the evaluation point in space</td>
</tr>
<tr>
<td>2. Absolute context-switching operators:</td>
<td></td>
</tr>
<tr>
<td>side $x$</td>
<td>$x$ at the point with horizontal coordinate 0 and the current vertical and temporal coordinates</td>
</tr>
<tr>
<td>edge $x$</td>
<td>$x$ at the point with vertical coordinate 0 and the current horizontal and temporal coordinates</td>
</tr>
<tr>
<td>first $x$</td>
<td>$x$ at time 0 and the current spatial position</td>
</tr>
<tr>
<td>3. Dimensional context-switching operators:</td>
<td></td>
</tr>
<tr>
<td>$x$ hsby $y$</td>
<td>$(\text{left } y)$ if the current horizontal coordinate &gt; 0 else $x$</td>
</tr>
<tr>
<td>$x$ vsby $y$</td>
<td>$(\text{down } y)$ if the current vertical coordinate &gt; 0 else $x$</td>
</tr>
<tr>
<td>$x$ fbby $y$</td>
<td>$(\text{prev } y)$ if the current time &gt; 0 else $x$</td>
</tr>
<tr>
<td>$x$ hbf $y$</td>
<td>$(\text{right } x)$ if the current horizontal coordinate &lt; 0 else $y$</td>
</tr>
<tr>
<td>$x$ vbby $y$</td>
<td>$(\text{up } y)$ if the current vertical coordinate &lt; 0 else $y$</td>
</tr>
<tr>
<td>$x$ before $y$</td>
<td>$(\text{next } x)$ if the current time &lt; 0 else $y$</td>
</tr>
</tbody>
</table>

2.2 Description

The intensional spreadsheet, like conventional spreadsheets, is a display-oriented calculation tool. Table 2 shows what a part of the spreadsheet looks like on the screen. Name corresponds to a user-defined spreadsheet variable name, and the label Time at the left bottom corner indicates the current moment in time.

In the display of spatial coordinates in Table 2, U means ‘up’, D means ‘down’, L means ‘left’ and R means ‘right’ with respect to the origin.

The intensional spreadsheet is considered as a single entity called a spreadsheet variable. The value of the spreadsheet variable varies in a three-dimensional space including two spatial dimensions and a temporal dimension, called the spreadsheet space. A cell of the spreadsheet variable is associated with a point in the spreadsheet space. The value of a cell can be defined by a Plane Lucid expression or formula. The value of a cell without a defining expression is undefined. The definition of the spreadsheet variable, therefore, consists of the definitions of all its cells. In other words, the definition of a cell at a particular point in the spreadsheet space defines the value of the spreadsheet variable at that point. When the value of the spreadsheet variable is demanded at a point, the definition of the cell at that point is evaluated.

In the defining expression of a cell, the intensional operators can be used to refer to other cells intensionally. For example, let right edge next $S$ be the definition of a cell at a point ($h,v,t$)
Table 2: A window of the intensional spreadsheet.

<table>
<thead>
<tr>
<th>Name</th>
<th>L3</th>
<th>L2</th>
<th>L1</th>
<th>0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>U3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

in the spreadsheet space, where $S$ is the spreadsheet variable, and $h$, $v$ and $t$ are the coordinates of the horizontal, vertical and time dimensions, respectively. The result of evaluating the cell is the value of the cell in the spreadsheet $S$ at the point $(h + 1, 0, t + 1)$. There are some significant differences between the intensional and the conventional ways of referring to values of other cells in the definition of a cell in a spreadsheet. The former is the result of the intensional operators being applied to the spreadsheet variable, whereas the latter are usually just references to individual cells. Generally speaking, since an intensional operator can be applied to any Plane Lucid expression, more complex calculations involving values of other cells can be defined intensionally in a cell’s formula. By contrast, the conventional references are used only to access values of particular cells.

Cells of a spreadsheet variable can be defined at four levels: local, dimensional, planar and global. Given a cell in the spreadsheet space, a local definition, which is a Plane Lucid expression, defines the value of the spreadsheet variable at that particular cell. In a local definition, besides ordinary pointwise operators, only relative and absolute context-switching intensional operators are normally used to communicate with other cells. Given a dimension and two coordinates of the other two dimensions, a dimensional definition defines values of the cells in the whole given dimension that have the same coordinates as the given ones for the other two dimensions. In other words, all these cells in the given dimension have the same dimensional definition. For example, let $S$ be the spreadsheet variable. At the D2 row, (vertical coordinate $-2$) and time 0, the dimensional definition

$$(S - 1) \ hbf \ 0 \ haby \ (S + 1)$$

defines the value of every cell in $S$ in the horizontal dimension at row D2 and time 0 to be its column number. Similarly, given two dimensions and a coordinate of the other dimension, a planar definition defines values of the cells on the whole plane consisting of the two given dimensions at the given coordinate of the other dimension. For example, at time 0 the planar definition

$$H + V$$

where

$$H = (H - 1) \ hbf \ 0 \ hsb y \ (H + 1);$$

$$V = (V - 1) \ vbf \ 0 \ vsb y \ (V + 1);$$

end

defines the value of every cell of $S$ in the horizontal–vertical plane at time 0 to be the sum of its column number and row number. A global definition defines all cells of a spreadsheet variable in
a single expression. For example, the global definition

\[
H + V + T
\]

where

\[
H = (H - 1) \text{hsby} (H + 1);
V = (V - 1) \text{vsby} (V + 1);
T = (T - 1) \text{fby} (T + 1);
\]

defines the value of each cell of \( S \) to be the sum of its horizontal, vertical and time coordinates. When we say that a certain expression serves as the definition of a region (such as a line or a plane), we mean simply that each cell in the region has the given expression, unaltered, as its definition.

A spreadsheet variable can have at most one global definition. When a cell has two definitions that are defined at different levels, one definition is superior to the other one in the order: local, dimensional, planar and global. When there are two different definitions at the same level relevant to a particular cell, it is required that the two definitions yield the same value at the point in question. Our implementation enforces this requirement at runtime. Both definitions are evaluated. If both values agree, the common result is taken as the final value of the cell. Otherwise the cell's value is error.

The spreadsheet allows a user to define as many spreadsheet variables as he/she needs. In this sense, the intensional spreadsheet is truly a multi-spreadsheet, i.e., in a single intensional spreadsheet program several spreadsheets (with different names) can be programmed and calculated. Two spreadsheets can be consolidated by referring to a spreadsheet variable in the definition of a cell of the other spreadsheet variable. From the practical point of view, however, only few of these variables denote values the user wants to see on the screen. Other spreadsheet variables are auxiliary.

The spreadsheet design also allows user-defined functions. A user-defined function can only be defined globally. It can be called in the definitions of cells of any spreadsheet variables or of other user-defined functions. When a function is called in the defining expression of a cell during evaluation, the intensional operations in the function are based on the context—the point in the spreadsheet space that the cell is associated with. For example, let the function be defined as

\[
\text{UpRight}(x) = \text{up right } x
\]

and the function call \( \text{UpRight}(S) \) is in the definition of the cell at the point \((h, v, t)\). In evaluating the function call, the spreadsheet variable \( S \) is evaluated at the point \((h + 1, v + 1, t)\).

The following is an example showing some features of the intensional spreadsheet. It is an intensional spreadsheet program to play the game of life, which may briefly be described as follows. The game is played on a two-dimensional grid. Each point of the grid may contain an organism. Every gridpoint is considered adjacent to eight other gridpoints. A function \( \text{occ}(p) \) represents the number of the adjacent gridpoints of the gridpoint \( p \) that are occupied by organisms. Using two simple rules, a new organism configuration can be formed from the previous one:

1. If \( 2 \leq \text{occ}(p) \leq 3 \), the organism at the point \( p \) will survive, otherwise it will die.
2. If \( \text{occ}(p) \equiv 3 \), a new organism will be born at point \( p \) if \( p \) is currently empty.

Programming the intensional spreadsheet to solve this problem is very simple, because many features of the spreadsheet are well suited to the problem. In this example, the size of the game of life is a \( 5 \times 5 \) array that initially (at time 0) has five organisms, and after every fifty consecutive configurations the game goes back to the initial configuration.

To define this game-of-life spreadsheet program, we need two auxiliary spreadsheet variables, \( \text{Bound} \) and \( \text{Init} \), that specify the boundary condition and the initial configuration, respectively. In
defining \emph{Bound}, four planar definitions and a global definition are used. The four planar definitions specify 0 as the values of the cells on the planes consisting of the horizontal and time dimensions at row 0 and D6, and the planes consisting of the vertical and time dimensions at column 0 and R6, respectively. The global definition defines values of the cells of \emph{Bound} elsewhere as 1. Table 3 shows the values of the spreadsheet \emph{Bound} at an arbitrary point. \emph{Bound} does not vary in the time dimension.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\emph{Bound} & 0 & R1 & R2 & R3 & R4 & R5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
D1 & 0 & 1 & 1 & 1 & 1 & 0 \\
D2 & 0 & 1 & 1 & 1 & 1 & 0 \\
D3 & 0 & 1 & 1 & 1 & 1 & 0 \\
D4 & 0 & 1 & 1 & 1 & 1 & 0 \\
D5 & 0 & 1 & 1 & 1 & 1 & 0 \\
D6 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{tabular}
\caption{The auxiliary spreadsheet \emph{Bound} of the game-of-life spreadsheet.}
\end{table}

In defining \emph{Init}, we need five local definitions to specify the five gridpoints, D2R3, D3R2, D3R3, D3R4, and D4R3, that are initially occupied by organisms, and a global definition to specify that all other gridpoints are initially empty. Table 4 shows the values of the spreadsheet \emph{Init} at time 0.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
\emph{Init} & 0 & R1 & R2 & R3 & R4 & R5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
D1 & 0 & 0 & 0 & 0 & 0 & 0 \\
D2 & 0 & 0 & 0 & 1 & 0 & 0 \\
D3 & 0 & 0 & 1 & 1 & 1 & 0 \\
D4 & 0 & 0 & 0 & 1 & 0 & 0 \\
D5 & 0 & 0 & 0 & 0 & 0 & 0 \\
D6 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{tabular}
\caption{The auxiliary spreadsheet \emph{Init} of the game-of-life spreadsheet.}
\end{table}

The user-defined function \emph{occ}(p) is defined globally as follows:

\[ \text{occ}(x) = \text{left}\ p + \text{right}\ p + \text{up}\ p + \text{down}\ p + \text{up left}\ p + \text{up right}\ p + \text{down left}\ p + \text{down right}\ p; \]

Finally, the principal spreadsheet variable \emph{Life} can be defined by the following global definition:

\[ \text{Init \ fby (Bound \ s if Life then survive else birth fi)} \]

\begin{verbatim}
where
survive = if occ(Life) \equiv 2 \lor occ(Life) \equiv 3 then 1 else 0 fi;
birth = if occ(Life) \equiv 3 then 1 else 0 fi;
end
\end{verbatim}

and a planar definition for the plane consisting of the horizontal and vertical dimensions at time point 50.

\textit{first Life.}
In the above spreadsheet variable, the value, 1 or 0, of a cell denotes that the corresponding gridpoint contains a live organism or is empty at a time point. The spatial relationship between a gridpoint and its adjacent points is specified by the function $occ$. During evaluation, when $occ(Life)$ is evaluated in a cell’s definition it returns the number of live organisms among the eight adjacent gridpoints of the point that the cell corresponds to. The temporal relationship of a gridpoint with its adjacent points and itself is specified by the intensional operation $fby$ in the global definition of $Life$. At a time point, this operation causes the function call $occ(Life)$, one of the local variables $birth$ and $survive$ as well as the spreadsheet variable $Life$ in the definition to be evaluated one moment before the current time, so that the action of ‘survive’ or ‘birth’ can be done according to the previous situations of the neighbors of a gridpoint and itself. It is easy to see that, as time increases, the organism configurations represented by the spreadsheet exactly follow from the rules introduced. Table 5 shows the configuration at time point 1.

Table 5: The game-of-life spreadsheet configuration at time 1.

<table>
<thead>
<tr>
<th></th>
<th>Life</th>
<th>0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$Time = 1$

3 The Implementation of the Intensional Spreadsheet

The implementation of the intensional spreadsheet consists of two relatively independent parts: the user interface and the spreadsheet interpreter. This paper concentrates on the spreadsheet interpreter.

The interface between the user interface and the spreadsheet interpreter is made as simple as possible to keep them independent of each other. The user interface passes the following information to the interpreter: (1) requests to enter or delete definitions of spreadsheet variables and user-defined functions; (2) requests to evaluate cells of spreadsheet variables; and (3) input values when they are needed. The spreadsheet interpreter passes the following messages to the user interface: (1) returned values after evaluation or reevaluation; (2) requests for input; and (3) error messages.

3.1 Eductive computation

The interpreter of the intensional spreadsheet is implemented by a sequential version of the dataflow computation model called eduction. Eduction is the tagged demand-driven computation model.

Dataflow computation can basically be classified into two different models, data-driven and demand-driven, according to how the computation is controlled. In the data-driven model [5, 2], an operation is performed as soon as all its operands, which are data items, are available either as input or as the result of already completed operations. In this model, there is a one-way traffic in the computation: data flows in one direction from the producers to the consumers. In the demand-driven model, an operation is performed as soon as there is a demand for the result of the operation and all the operands that are needed are available. If there is a demand for an operation that needs an operand whose value is not available, then a demand for that operand

7
is made to the operation that produces it. When, as a result of such demands, there is a value available for each of the needed operands, the operation is performed, and its result is sent out to the operations that need this result. In this model, there is in fact a two-way traffic in the computation: data flows in one direction from the producers to the consumers and demands flow in the other direction from the consumers to the producers.

Many people favor the data-driven model in parallel processing because of two reasons related to the time efficiency of the computation. First, just in propagation of demands, work must be performed before any operations actually get executed. Secondly, in the demand-driven model, the amount of parallel activity seems less than in the data-driven model, because operations are only executed after it has been determined that their results are definitely needed. However, the demand-driven model does have some advantages over the data-driven counterpart. One problem of the data-driven model is dealing with infinite data objects, because it is obviously impossible to compute all elements of an infinite object even if all the operands of the operations computing these elements are available. By contrast, in the demand-driven model, only when an element of an infinite data object is needed by some operations or as the final result of the program will the operation computing the element be performed. In the case of the intensional spreadsheet, a spreadsheet variable is in fact an infinite data object that theoretically consists of an infinite number of cells. It is obviously impossible to calculate all the values of all the cells. The other problem with the data-driven model, related to the previous one, is that many unnecessary operations may be performed during the computation. The data-driven model can gain speed of computation at the risk of wasting a large amount of resources, because some intermediate computing results take resources but may be never needed. For example, in the intensional spreadsheet, only the values of the cells that are requested by the user actually need be computed.

There is another way in which various dataflow computation models differ. Some use piped dataflow [5] and some use tagged dataflow [2]. In piped dataflow, the path between an operation or producer and another operation or consumer is considered to be a pipe along which data items flow, in a first-in, first-out manner. In other words, there is a queue acting as a buffer between the two operations. In tagged dataflow, the path between the operations simply indicates the routes that data items must take. The buffer between the operations is not a queue; it is a set. However, some discipline must be imposed on the order in which an operation takes the data item from the incoming path. It is achieved by associating the data items and demands with tags so that the operation looks for data items and responds to demands with appropriate tags. Notice that the tag does not indicate the order in which data items are added to a set. Rather, the tag indicates the position of the data item in the conceptual sequence of items emitted by an operation. For example, in a Plane Lucid expression, a pointwise binary operation receives data items from two other operations. These items must be associated with appropriate tags indicating the coordinates in the three-dimensional space, so that the operation can only be performed on the items from the two producers whose tags are the same.

Tagged demand-driven dataflow is especially suitable for handling non-strict operations. A strict operation is one that requires values for all its arguments in order to produce a result. A typical non-strict operation is if-then-else. This operation is essential in any model of computation, but in any non-tagged dataflow its effect must be achieved by the use of two separate operations to select and distribute data items [5]. With tagged-token dataflow if-then-else can be a ternary operation. It is not very efficient in the data-driven model, because all three operands of the operation will be evaluated, even though only two will be used. With the demand-driven model, however, the if-then-else operation is quite efficient, because only the operands that are needed are evaluated.

Another surprising property of eduction is that it does not require any static memory. The program is never changed during computation. As a result, values that have been computed but not saved can be recomputed from scratch if they are needed again.
3.2 An eductive interpreter

The intensional spreadsheet interpreter is eductive. The principle behind the eductive approach of the interpreter may be briefly described as follows. The entire computation is driven by demands of the user who wants to evaluate one or more cells of some spreadsheet variables after he/she has defined the spreadsheet. In computing the value of a spreadsheet variable at some point in the spreadsheet space, the interpreter is led to compute values of the spreadsheet variable and other spreadsheet variables at possibly different points. When the value of a spreadsheet variable at a point is required, the interpreter consults the relevant definition of the variable. It uses the formal semantics—intensional semantics—to interpret the operations constituting the definition and compute the required value from the values of the operands that are possibly associated with other points in the spreadsheet space.

Although during computation, according to the eduction model, all evaluated values do not need to be stored and can be reevaluated when they are required later on, a memoryless interpreter is hopelessly inefficient. The spreadsheet interpreter avoids recomputations by storing computed values in a large so-called value warehouse, which is an associative memory. Each time a new demand is generated, the interpreter first checks the value warehouse to see if the value is already available. A value in the value warehouse is labeled by the spreadsheet variable that the value belongs to and the tag consisting of the three coordinates of the point that the value is associated with (as well as a place parameter, which is discussed later). Values in the value warehouse are efficiently accessed by their labels using hashing.

Besides the value warehouse, the interpreter also includes a definition warehouse, again an associative memory, to store definitions of spreadsheet variables and user-defined functions. Each time a new demand for the value of a spreadsheet variable at a point is generated, if the interpreter cannot find the needed value of the spreadsheet variable at the point in the value warehouse, it fetches a compiled form of the definition of the cell of the spreadsheet variable at that point and evaluates the cell by running the definition. Then the resulting value is stored in the value warehouse for later use. Definitions in the definition warehouse are tagged differently depending on the ways in which they are defined. The spreadsheet interpreter uses a special data value all to indicate that a coordinate of a dimension in the tag in fact represents all the coordinates in the dimension. Thus the global definition of a spreadsheet variable or a user-defined function is tagged by three all coordinates. A planar definition of a spreadsheet variable is tagged by an actual coordinate and two all coordinates. A dimensional definition of a spreadsheet variable is tagged by two actual coordinates and an all coordinate. And a local definition of a spreadsheet variable is tagged by three actual coordinates.

Eduction is a very simple technique when the tag involved is composed of horizontal, vertical and time coordinates, i.e., essentially small integers for indicating a particular point in the spreadsheet space. However, complications arise when the method is extended to handle some basic features of the intensional spreadsheet, such as user-defined functions. For example, the value of a local variable such as $M$ in

\[
\begin{align*}
  f(x) &= M \\
  \text{where} \\
  M &= x \hsby M + \text{right } x; \\
  \text{end}
\end{align*}
\]

depends on more than just time and space; it depends on the particular call or invocation of the function $f$. Operationally, the definition of $f$ is simply a template, and a use of $f$ is a separate copy with its own separate internal storage. As a result, at any given point in the spreadsheet space, there may be many different individual values of the variable $M$. The spreadsheet interpreter incorporates a simple solution to the problem. It assigns unique ‘coordinates’ to the calls of user-defined functions in a spreadsheet program. A place parameter is set for each user-defined function. When a user-defined function call is found in the definition of a cell of a spreadsheet variable or another function during compilation, the place parameter corresponding to the called function is inserted into the proper object code for that function call, and the place parameter is increased.
by 1. As a result, during evaluation, every function call has a unique place number for the called function. If a function is defined nestedly or recursively, all the nested or recursive calls to the function share one place number, but the value evaluated in every call is given a different place number derived from the shared place number when it is stored in the value warehouse. Thus, the horizontal, vertical, time and place parameters together are enough to determine individual values of variables such as $M$.

### 3.3 The implementation

The spreadsheet interpreter is built on a software tool called the Popshop [9]. The Popshop is an attempt to use sound software engineering principles to facilitate the production of prototype implementations for experimental non-procedural languages. The Popshop is intended solely for languages that are based on the data objects, operations and syntax of Pop-2 (a Lisp-like AI language [4]). The data types of Pop-2 are integers and reals, strings of ASCII characters, words as identifiers and sequences of ‘signs’, and lists. The data types of Pop-2 are also the data types used in intensional spreadsheet programs. At present, the actual programming of the Popshop is done in C. Figure 1 is the scheme of the spreadsheet interpreter.

![Figure 1: The structure of the spreadsheet interpreter.](image)

The basis of the implementation of the spreadsheet interpreter is the Popmachine [9], which works like the ‘CPU’ in conventional computers. The Popmachine is centered around the value stack of Pop-2 data items. Some basic operations can be performed on the top elements of the stack. The operations include ones to read from and write onto the stack, to pop the stack, to swap the top elements, to cycle the top elements, and so on. The basic Popmachine has no storage other than its stack. There are no fetch or store operations in the core instruction set.

The assembly/machine language of the Popmachine is called Popcode, which is similar to Postscript. A Popcode program is a Pop-2 data object. It is a list of items each of which is either an operand to be placed on the stack or an operation to be executed. Items that are lists, strings or numbers are taken to be operands; items that are words are taken to be operations. The Popmachine is a stack machine, i.e., all the Popmachine operations take their arguments from the stack, and there are no address fields in Popcode. For example, for a simple expression $x + y$, the compiled Popcode is 

```
["x eval "y eval +
```

where the operations "x and "y push the identifiers $x$ and $y$ onto the stack, the operation eval evaluates the variable whose name is at the top of the stack and leaves the result on the top of the stack, and the operation $+$ replaces the two data items on the top of the stack by their sum.

The spreadsheet interpreter is an expansion of the Popmachine designed to run preprocessed spreadsheet programs, extended Popcode programs that are compiled from Plane Lucid source. The interpreter is constructed by adding some ‘boards’ to the Popmachine. These boards provide
the time register, the horizontal register, the vertical register and the place register, as well as the value and definition warehouses.

The horizontal, vertical and time registers hold horizontal, vertical and time coordinates, respectively. They enable commands to increase and decrease the corresponding dimension parameters, to set them to any integers and to save or restore their contents. During evaluation, the contents of the registers are always equal to the ‘current’ horizontal, vertical and time coordinates, respectively.

The place register holds a single place coordinate that is a small natural number in most cases or an encoding of a finite sequence of numbers, when function nesting takes place. The register enables various commands to set, save and restore it. The place register contains the place number of the current function call.

3.4 How the spreadsheet interpreter works

3.4.1 Entering definitions

The user interface passes a definition as a string of characters to the spreadsheet interpreter associated with the name of the defined spreadsheet variable or user function and the coordinates of the point(s) in the spreadsheet space at which the definition is defined. After the interpreter receives the definition, it takes two steps to complete the task: compiling the source definition and storing the compiled object definition(s) into the definition warehouse.

3.4.2 Compilation of definitions

Compiling a definition has three phases: generating a parse tree, renaming intermediate code and generating Popcode. The spreadsheet interpreter must compile definitions incrementally. It cannot do global analysis of the whole program, because definitions are entered separately. This sometimes causes difficulties, especially when user-defined function calls are involved in the source definitions.

The first phase, generating a parse tree, is common in most compilers. In this phase, the parser produces a parse tree from the source definition.

The second phase, renaming intermediate code, does the following work on the parse tree produced by the previous phase.

1. If the entered definition is for a user-defined function or involves user-defined function calls, it gives the user-defined function and its formal parameters new names to avoid name conflicts, because eventually their object definitions will be stored in the definition warehouse in the same way as other variables.

2. If the entered definition includes a where clause, it flattens the where clause, i.e., it converts local variables of any nested where clause to globals that have unique names in the spreadsheet program. In other words, after this phase, each local variable in the where clause becomes an auxiliary spreadsheet variable with only a global definition.

All new names produced in this phase are unique in the spreadsheet program.

The last phase, generating Popcode, is more complicated. It not only translates the source definition from the renamed parse tree into the appropriate object definition(s) in Popcode, but also performs some other tasks for the storage step and later evaluations.

One of these tasks is that if the source definition involves a where clause, apart from producing an object definition for the subject of the where clause, it has to produce an object definition for each local variable that has become a ‘global’ variable after the renaming phase.

Another task of this phase is to produce a proper place number for each function call. It keeps a function-call count, which is initially one, for each user-defined function during the compilation of the spreadsheet program. Whenever a user-defined function call appears in the definition, the current function-call count for that function is added into the code of the object definition as the referenced place number, and then the count is increased by 1. Eventually, for every particular call
to a user-defined function, a unique place number is assigned. Meanwhile, the object definition for every actual parameter in a user-defined function call is produced.

3.4.3 Storage of object definitions

In this step, the object definition(s) produced in the previous step are stored in the definition warehouse with proper labels. A label for an object definition includes the name of its owner, which has one of the following types: a spreadsheet variable, a renamed local variable, a renamed user-defined function or a renamed formal parameter. The label also includes the tag consisting of the three coordinates to indicate the cell that the definition is associated with, and the place number. Notice that the object definition of an actual parameter is stored into the definition warehouse with the same name as its corresponding formal parameter’s but the place number corresponding to the particular function call that it is associated with.

3.4.4 Evaluation

The evaluation of a spreadsheet variable is pointwise, i.e., every time only one value of the spreadsheet variable at one point in the spreadsheet space is evaluated. It is impossible to evaluate a spreadsheet variable at all points because the spreadsheet variable varies in an infinitely large three-dimensional space. The evaluation is eductive.

The evaluation begins when the interpreter injects a demand for a value of a spreadsheet variable at a point in the spreadsheet space. During the evaluation, the interpreter first checks if the demanded value has already been produced by accessing the value warehouse with the label consisting of the variable’s name, the current horizontal, vertical and time coordinates in the corresponding registers, and the current place number in the place register. If the value has been produced, then it is returned. Otherwise, one or more proper definitions of the variable have to be found in the definition warehouse according to the label. If the variable at the given point has more than one definition, only the definition(s) with highest priority is chosen. It is possible that more than one dimensional or planar definition could be chosen. In this case, each fetched definition is evaluated, and the values are compared. If they are the same, the value is pushed on the stack and also stored in the value warehouse labeled with the variable’s name and the tag consisting of the current coordinates as well as the place number in the corresponding registers.

If the definition warehouse does not contain any definition of the demanded variable at the given point, the interpreter sends a request to the user interface asking the user to input a value or a definition for the variable at the point. If a single value is input, it is stored into the value warehouse labeled with the name of the variable and the tag consisting of the three coordinates of the point as well as the place number. Otherwise the input definition is stored into the definition warehouse with the label. After the needed input is entered, the evaluation continues.

3.4.5 Detection of dependencies

It is obvious that if one does not keep track of dependencies among cells of spreadsheet variables and user-defined functions in a spreadsheet program, after the user deletes or modifies even a single definition, the reevaluation of the spreadsheet program will be very inefficient. The reason is that all values of the defined cells of spreadsheet variables have to be recalculated, since it is unknown which cells are affected by the modification. The dependency problem is one of the main efficiency considerations in spreadsheet design [1].

The presence of intensional operators, user-defined functions and possibly multiple definitions of spreadsheet variables makes the dependency problem in the intensional spreadsheet even more complicated than in conventional spreadsheets. Multi-level and nested applications of intensional operators may have such complicated references to spreadsheet variables at various points in the spreadsheet space that it is very difficult to determine the dependencies textually. In addition, since a user-defined function can be called in any cell definition, the dependencies among the cells, therefore, are often hidden, which makes things even more difficult.
Fortunately, with the eductive approach, one does not need to detect the dependencies by analyzing the defined spreadsheet program textually. Instead, one can build the dependency list for each cell of a spreadsheet variable and each user-defined function dynamically during evaluation of the spreadsheet. The strategy here is based on the following consideration. When the value of a cell of a spreadsheet variable or a user-defined function call is evaluated because of some demand, those variables and functions that demanded this value directly or indirectly are the dependents of the cell or the function being evaluated. On the other hand, when demands are sent out to evaluate the value of a cell of a spreadsheet variable or a user-defined function, those cells of variables and functions whose evaluations are caused by these demands directly or indirectly are depended on by the cell or the function being evaluated. Here when an object whose value depends on values of another object, the former is called a dependent of the latter and the latter is called a depender of the former. The following is a sketch of the algorithm to build dependency lists of a cell of a variable or a user-defined function dynamically. In the algorithm, there are two lists, the dependent list and the depender list, for each definition of a cell or user-defined function. If two cells have the same dimensional, planar or global definition, the dependency list for the definition is shared by these cells. Whenever the value of a cell of a variable $X$ or the definition of a function $X$ is fetched by demand, the following algorithm is applied:

**Dependency detection algorithm**

- Add the labels of all objects waiting for responses of their demands to the dependent lists of $X$ and all members of $X$’s depender lists.
- Add $X$’s label and all members of $X$’s depender list to the depender lists of all objects waiting for responses of demands.

### 3.4.6 Deletion

The user can delete or modify a local, dimensional, planar or global definition of a spreadsheet variable as well as the definition of a user-defined function at any time during programming the spreadsheet.

The deletion of a source definition may result in (1) deleting more than one object definition in the definition warehouse if the source definition contains a `where` clause and/or user-defined function call(s); (2) removing those stored values from the value warehouse whose owners depend, directly or indirectly, on the deleted definition(s), because those values are no longer valid.

To achieve the above, the following algorithm is applied during the deletion. Let $X$ be the cell(s) or user-defined function whose definition the user wants to delete:

1. Remove from the value warehouse $X$’s value and those values whose owners are in $X$’s dependent list.
2. (a) For each renamed local variable defined in $X$’s source definition, remove its object definition from the definition warehouse;
   (b) For each user-defined function call appearing in $X$’s source definition, find its place number and then remove its formal parameters’ object definitions that have the same place number from the definition warehouse;
   (c) Delete $X$’s object definition from the definition warehouse.
3. Remove $X$ from those objects’ dependent and depender lists whose names are in $X$’s depender and dependent lists, respectively.

After these three steps, there is no trace of the deleted definition.
3.4.7 Modification

The modification of a definition of a spreadsheet variable or a user-defined function can be thought of as two separate steps: deleting the existing definition and entering a new definition. In this case, however, the original dependent list associated with the modified definition must be retained, while its depender list has to be rebuilt from scratch.

3.4.8 Reevaluation

When a definition of a spreadsheet variable or a user-defined function is modified, reevaluation for the cells of some variables that directly or indirectly depend on the modified definition has to be performed, because their values displayed on the screen are no longer valid and have to be updated. To achieve this, all cells in the modified object’s dependent list are reevaluated, whereas other cells’ values are not affected by the modification.

3.5 About storage management

The size of the intensional spreadsheet depends mainly on the size of the value warehouse because the number of values it stores increases rapidly during evaluation. Appropriate methods must be used to restrict the value warehouse to a reasonable size, because not all values computed by the interpreter will be used again and any value can be recomputed even if it was thrown away prematurely.

If every value ever computed were stored in the value warehouse and nothing were ever removed, it would be very time efficient because nothing would ever be recomputed, but it would be incredibly wasteful of space. Therefore, it is necessary to remove some values that will not be further used from the value warehouse before the warehouse becomes too large. The authors have designed a heuristic to reduce the size of the value warehouse by removing the least recently used values.

4 Conclusions

The intensional spreadsheet has been show to be a new approach to spreadsheet design incorporating recent developments in high-level programming languages. In a sense, the intensional spreadsheet is in fact a programming environment for Plane Lucid. The intensional spreadsheet allows interactive and incremental Plane-Lucid programming. The eductive approach to implementing the intensional spreadsheet also has a potential implementation for software packages such as spreadsheets, graphics or simulators that involve space and time changes or other context changes.

The eductive approach has advantages over conventional approaches. In the conventional approaches, the evaluation order of a spreadsheet is predetermined, so the user has to be careful of spatial positions when defining relations among cells, because evaluations of their values may be affected by different cell layouts. In the eductive approach, since evaluation is driven by demands, the evaluation order is not predetermined and it only depends on data dependencies among definitions of cells, so that spatial positions of definitions of cells are not important. Furthermore, the dataflow style of implementation offers an opportunity for large spreadsheets to be evaluated in parallel, as all operations that are demanded and whose operands are available can be performed simultaneously.

Dynamic detection of dependencies among cells of spreadsheet variables through demands is another advantage of the eductive approach. There is, however, a tradeoff between static analysis and dynamic analysis of spreadsheet programs since the latter is also time-consuming. The storage of evaluated values with tags may reduce most of the detecting time, as during evaluation variables and functions send demands out only if their values are never evaluated, otherwise their values can be obtained from the value warehouse directly.
Acknowledgements

The research described in this article was sponsored in part by the Natural Sciences and Engineering Research Council of Canada. Weichang Du is also partially supported by a B.C. Advanced Systems Foundation Graduate Research Scholarship.

References


