Visual Languages and the Conflict Between Single Assignment and Iteration

Allen L. Ambler

Margaret M. Burnett

1. INTRODUCTION

In the quest for visual languages, one inherent natural expression is visual, rather than textual, single assignment is often a design objective. The reasons are both conceptual and practical. Conceptually, programs with single assignment involve fewer concepts, they are easier to understand, the flow of data is easier to visualize, and the notions of mathematics are better preserved. Practically, the single assignment rule says that once a variable has been bound to a value, it remains bound to that value. This notion is needed in many declarative programming paradigms, including dataflow, logic, and functional. It is also important in maximizing parallelism.

On the other hand, iteration is built on the idea of repeatedly executing a body of code, usually for the purpose of repeated modification of some named variable. Thus single assignment and iteration would seem contradictory. For language designers the choices are recursion only, or as William Ackerman characterized it: "a somewhat unusual notation for iteration"[1]. While recursion is computationally adequate, there are arguments for not totally abandoning iteration, including the argument that many algorithms are most naturally expressed iteratively. This paper looks at a variety of "unusual notations" for iteration, with particular attention to those used in dataflow and visual languages, and then offers another solution to this problem.

2. BACKGROUND

There are two distinct types of iteration. These are referred to with differing terminology in the literature. The term horizontally parallel iteration is used herein to denote iteration in which the outcome of one cycle does not affect the outcome of the next. Typical examples are: "do for all" elements in a set some operation (as in forall). In this form each iteration is completely independent, so there is no conflict with the notion of single assignment. This form of iteration is not dealt with further in this paper.

The other type of iteration, denoted herein as temporally-dependent iteration, refers to iteration where the outcome of a cycle depends on one or more of the previous cycles. For example, the following loop computes the nth Fibonacci number as the final value of fib.

```
fib := 1;
for count:= n downto 0 do
  for count:= 1 downto 0 do
    fib := fib+pred;
  end;
end;
```

In this loop, count, fib, and pred violate the notion of single assignment.

Traditionally, for iteration one needs to alter the values of one or more variables between successive iterations, but the single assignment rule prevents this. Some languages address this problem by allowing the single assignment rule in a very restricted way. The textual dataflow languages Val [2] and Id [4] allow a special form of multiple assignment restricted to specifically identified loop variables and allowed only between iteration cycles and only inside special iteration constructs. Thus the single assignment is still in force within any one cycle, but between cycles the loop variables can be adjusted. For example, in the Val program:

```
for count, fac, prev := N, 1, 0;
do if count < 0
  then fac := fac*prev;
ext count, fac, prev := count, fac*prev, fac;
end
```

the iter construct controls multiple assignment. Only variables specifically identified as loop variables can be changed repeatedly between iterations and they will be changed "simultaneously." Id uses a similar approach:

```
(initial count := N; fac:= 1; pred := 1
while count > 0
  new count := count-1;
  new fac := fac*pred;
  new pred := fac;
  return pred)
```

Here new is being used to control multiple assignment. Again multiple assignment is allowed only for loop variables and occurs simultaneously between iterations.

Sisal [7], a textual dataflow language, contains a similar temporally-dependent iteration construct that allows the immediately previous value of a loop variable to be referenced by using old in front of the variable name. An example is:

```
fib := for initial
  count := 1;
  pred := 1;
  fib := fib+pred;
while count < n
  repeat
    count := old count + 1;
    pred := old fib;
    fib := pred + old pred;
  return
  value of fib
end
```

While this Sisal program looks very similar to the prior two programs in Val and Id, Sisal treats multiple assignment as multiple assignment, but as temporal assignment. This is also the case for several other single assignment languages (considered below) and forms the basis for preserving single assignment.

The idea of temporal assignment is that a name is attached not to a single value, but to a stream of values generated over time. Typically the value of the name is then associated with just the most recent value in this stream. This suggests that if a name is bound to a stream of values, then one should be able to access more than just the most recent such value. In Sisal, the prior value can be referenced via old; in addition, the function can return not only the final value, value of fib, as above, it also can return array of fib, which is the entire stream of values assigned to fib.

This suggests the possibility that in a dataflow paradigm, which already has the notion of a stream of values, iteration can be created by simply feeding back the output of a node (or group of nodes) into its own input stream. This approach creates a cyclic graph which as Sharp [11] points out, is open to general timing-related problems.

Lucid [12], a textual single-assignment language, avoids the multiple-assignment problem by making explicit use of the "time" dimension. In Lucid, each output of an expression bound to a name is the most recent value in its time sequence of values. Thus, for

```
n = 0 fby 1 + n
```
n is a sequence of numbers over time, starting at 0, followed by (fly) 1, 2, 3, and so on. Note that there is no special syntax needed to refer to the immediately previous value of n; however, earlier values must be accessed explicitly with the attime construct:

thirdvalue = n attime 3

To refer to a relative instance, such as the one before the last, is more difficult and generally leads to creating loops that specifically "time-delay" values. This can be seen in the Fibonacci example.

fibonacci(n) = f attime n
  where
    f = 1 fly f + g;
    g = 0 fly f;
  end

In this example g is used in a conventional manner as a variable to delay a value; alternatively, in Lucid this function can be written without using g:

fibonacci(n) = f attime n
  where
    f = 1 fly f + (0 fly f);
  end

The way to interpret this expression is: f is initially 1, followed by the sum of the value stream produced by f (delayed one time interval by the first fly) and the stream produced by the expression (0 fly f) produces 0 followed by f time-delayed yet again. That is f produces the stream 1 followed by 1,2,3,5,8,13,...; because (0 fly f) produces 0,1,1,2,3,5,... while f produces 1, 1, 2, 3, 5, 8, ... and thus f+(0 fly f) produces 1, 2, 3, 5, 8, 13,....

Creating time-delays can rapidly become confusing unless there is some more direct means of indicating relative references. What is desired is a prior f plus prior prior f, or f attime NOW(f)+1 plus f attime NOW(f)+2, where NOW(f) is the time index of the next value in the series for f. Unfortunately, NOW(f) does not exist, so we are forced to write the function as follows:

fibonacci(n) = f attime n
  where
    f=1 fly (f attime (1-i)) + (f attime (1-(i-1)));
    i=2 fly i+1;
  end;

where i is used to approximate NOW(f). This is almost the desired expression with the exception of the computation of i. The initialization of i to 2 is required to allow it to catch up with the built-in time delay induced by the expression.*

Show-and-Tell [5, 8] is an iconic dataflow language that takes a similar approach to temporarily-dependent iteration. In Show-and-Tell, the graph remains strictly acyclic. The heavy box in Figure 2.1 shows a temporarily-dependent iteration construct. When the iterative box is evaluated, it unfolds accordion style into multiple instances as is illustrated in Figure 2.2. The input values to the loop are inputs to the first instance, which produces outputs that are then inputs to the second, etc., as long as the iteration condition (shown in the dotted box) remains true. When the iteration condition finally fails, the outputs of the iteration prior to the failing instance are treated as the outputs of the iteration. Figure 2.2 shows the semantics of the iteration construct as it is used in the Fibonacci example. While this form of iteration treats loop variables as streams of values as well, only loop variables from the previous iteration are accessible in the current iteration, unless new variables are introduced for the purpose of saving older values, as is done in the example.

Of course, iteration might simply be eliminated entirely in favor of recursion. This decision is particularly prevalent for languages whose basic paradigms are derived from either functional or logic programming paradigms. It has likewise been suggested for dataflow paradigms [11] because a system with no iteration contains no cycles, hence avoiding most timing-related problems. However, such heavy reliance on recursion raises other issues [6]. When should the recursive instances of the function be created in order to maximize parallelism? How many instances should be created? When can some or all of them be destroyed? Fabrick [10], a visual
dataflow language which uses a bidirectional structure model**, is an example of a single-assignment system that uses recursion exclusively.

In preparation for looking at another form of iteration, a brief overview of the language Forms/2 is included. Forms/2 is based upon the earlier language Forms, described in [3].

Figure 2.1: Show-and-Tell's computation of the 10th Fibonacci Number, using the temporarily-dependent iteration construct (adapted from [3]).

Figure 2.2: This shows the semantics of the 10th Fibonacci Number shown in Figure 2.1.

3. GENERAL DESCRIPTION OF FORMS/2 AND DEPENDENCY-RESOLUTION PROGRAMMING

A form corresponds to a piece of paper (potentially a quite large piece of paper) on which can be pasted a variety of objects. Each object on a form can be referenced in one of two ways: by pointing at it (the most common way) or by name.

*In discussions of Lucid, the is current concept is often mentioned in association with iteration. This concept is in our opinion not really so much involved with iteration as it is with the Lucid concept of a function invocation. Because Lucid does not visualize a function invocation as different from a named expression, difficulties arise in holding the values of parameters and sequencing results. is current allows a mechanism for holding parameters. The interested reader is encouraged to see [12].

**The structure model is a less widely-used form of the dataflow paradigm in which a single data structure is constructed on each arc by means of execution of the program, and remains there for the duration of the program. Unlike the token model, there is no notion of consumption of data nor of history.
To construct a form, each object is selected from an object template menu, dragged to the appropriate position within the form, and stretched to the desired dimensions. The selection, placement, sizing, and aesthetic values, such as font, etc., are all handled in a fashion similar to MacDraw or any other object-oriented drawing program.

Objects can be either static or computed. Examples of static objects are text and graphical images that may be permanently placed within the form for aesthetic effects. Computed objects are composed of cells that take on values. Each cell can contain only a single value. Cells may be either individual (i.e., not part of a larger structure) or members of a larger structure (primarily arrays).

Programming is performed by arranging input, intermediate, and output cells visually to create a form; specifying for each cell either a value or some computational expression which can be evaluated to yield a value; and then evaluating cell expressions until all output cells have values.

Cell expressions are mathematical expressions. They are composed of literal values, references to other cells, and primitive operations and user-defined functions. A cell expression can reference any cell (or any object in any object within the containing form or within other forms, subject only to the restriction that the resulting derived evaluation must not be made to be circular. Since order of evaluation is updated after each new cell formula is entered, any circularity that might be introduced is detected immediately.

There are two categories of objects: bounded and unbounded. Bounded objects are static with fixed, known dimensions. Unbounded objects may have at least one dimension that is unknown when the computation is specified. During evaluation all objects must have their dimensions fixed prior to evaluation. The usual means by which an unbounded object will have its bound fixed is by mapping that object onto another object whose dimensions are already known. In this case the known dimensions are inherited by the unbounded object. This is the case when an unbounded formal parameter is matched to its actual parameter, which must have already been evaluated, and therefore has fixed dimensions.

A cell expression is evaluated based on the dependencies of its corresponding input cells. A cell expression within the current iteration of the form can reference any cell of the prior iteration as long as the corresponding input cells have already been evaluated. This allows the possibility that not all instances will actually be retained, but requires that any discarded instance be reconfigurable without potential loss of data, i.e., results must be retained for a given computation.
Evaluated as follows: in the first (0th) iteration the <first expression> is evaluated repeatedly referencing prior instances. The <second expression> is evaluated as the value of the cell, i.e., the <nth expression> is evaluated repeatedly after all others are exhausted.

5. COMPARISON OF FORMS/2'S ITERATION WITH RECURSION

Substantial differences exist between Forms/2's versions of iteration and recursion. First, recursion is a pure function call; that is, it accepts a set of values, performs some computation on these values, and then returns some set of values. In the process, it may utilize local variables, but such local variables are local to the particular invocation of the function and cannot be referenced from one instance to another as in the manner of referencing prior instances used in Forms/2's version of iteration. It might be argued that such prior referenced local variables in iterative invocations are simply additional parameters to the recursive call, but to account for arbitrary back references, all instances of such variables would be required. In addition, the conventional mechanism for recursive calls proceeds as follows: if function F has inputs I and outputs O, then for the i'th iteration of F its inputs are I[i] and its outputs are O[i], which are then used to compute O[i + 1], etc. In Forms/2 iteration, the outputs of the last iteration O[i] are directly mapped to the outputs of the entire form iteration, i.e., they are directly mapped to the outputs out. Thus there is no return to prior iterations with the possibility of additional computation.

6. EXAMPLE OF FORMS/2 ITERATION: FIBONACCI NUMBERS

In this section, a function to find the Nth Fibonacci number is constructed using Forms/2. The construction is described in detail in the captions associated with the figures.

The process begins with a blank new form that will be used to define the generic instance of Fibonacci, a function that given N will return the Nth Fibonacci number. Within the form are placed three single cell variables, denoted N, T, and F. These will be used in the computation. N is to be the input value; F is to be the output value; and T will be used as a temporary to compute the i'th Fibonacci number beginning at 0 and ending at N.

The cell formula for T will be entered first. It will be expressed in terms of two prior instances of T. To create images of these prior instances the "iteration walking" icon in the upper right corner is clicked twice (once in the current instance and once in the prior instance [-1]). Notice that these instances are indicated by appending to their form name a negative offset enclosed in square brackets, see Figure 6.1. They are indicated relative to the current (1st) instance.

Next the cell formula itself is input by selecting the cell T in the current instance. This creates an expression window that displays the cell formula in two ways: textually and iconographically. The cell formula is entered by typing "+" indicating the addition function (all functions are displayed in functional notation), and then by clicking on the cells that correspond to T in the instance [-1] and then in the instance [-2]. The preferred means of entering cell expressions is expected to be a combination of typing simple operations and constants and pointing at cells to generate cell references. Other possibilities are typing completely the entire cell expression including specific cell references; or using purely graphical means of selecting operators from templates and drawing lines to connect corresponding inputs and outputs. Of these latter two options, the former is thought to be tedious because the cell reference syntax requires a fair degree of precision, while the latter is thought to be tedious because a few keystrokes will often suffice for numerous selections, placements, and connecting.

In the iconographical cell expression correspondsences between the cells as they appear in the expression and where they are located in the forms are indicated by colored (black and white patterns in the figures) borders. The use of color is to indicate that the images of two cells are really the same cell. This approach was also used in Alex [9]. It seems adequate given that most cell expressions use a limited number of terms and that only a single cell expression is displayed at any time. This approach however does not readily transfer to monochrome (as can be seen from the figures). An option the authors have resisted to avoid clutter is connecting corresponding cells with some kind of line; however, this is possibly desirable as an additional option.

The formula for T is entered in Figure 6.1 in terms of the i'th iteration. When the formula is completed, i.e., the close box in the upper left corner is clicked, the system attempts to spread the formula across all iterations and functional notation.

EXAMPLE OF FORMS/2 ITERATION: FIBONACCI NUMBERS

In this section, a function to find the Nth Fibonacci number is constructed using Forms/2. The construction is described in detail in the captions associated with the figures.

The process begins with a blank new form that will be used to define the generic instance of Fibonacci, a function that given N will return the Nth Fibonacci number. Within the form are placed three single cell variables, denoted N, T, and F. These will be used in the computation. N is to be the input value; F is to be the output value; and T will be used as a temporary to compute the i'th Fibonacci number beginning at 0 and ending at N.

The cell formula for T will be entered first. It will be expressed in terms of two prior instances of T. To create images of these prior instances the "iteration walking" icon in the upper right corner is clicked twice (once in the current instance and once in the prior instance [-1]). Notice that these instances are indicated by appending to their form name a negative offset enclosed in square brackets, see Figure 6.1. They are indicated relative to the current (1st) instance.

Next the cell formula itself is input by selecting the cell T in the current instance. This creates an expression window that displays the cell formula in two ways: textually and iconographically. The cell formula is entered by typing "+" indicating the addition function (all functions are displayed in functional notation), and then by clicking on the cells that correspond to T in the instance [-1] and then in the instance [-2]. The preferred means of entering cell expressions is expected to be a combination of typing simple operations and constants and pointing at cells to generate cell references. Other possibilities are typing completely the entire cell expression including specific cell references; or using purely graphical means of selecting operators from templates and drawing lines to connect corresponding inputs and outputs. Of these latter two options, the former is thought to be tedious because the cell reference syntax requires a fair degree of precision, while the latter is thought to be tedious because a few keystrokes will often suffice for numerous selections, placements, and connectings.

In the iconographical cell expression correspondsences between the cells as they appear in the expression and where they are located in the forms are indicated by colored (black and white patterns in the figures) borders. The use of color is to indicate that the images of two cells are really the same cell. This approach was also used in Alex [9]. It seems adequate given that most cell expressions use a limited number of terms and that only a single cell expression is displayed at any time. This approach however does not readily transfer to monochrome (as can be seen from the figures). An option the authors have resisted to avoid clutter is connecting corresponding cells with some kind of line; however, this is possibly desirable as an additional option.

The formula for T is entered in Figure 6.1 in terms of the i'th iteration. When this formula is completed, i.e., the close box in the upper left corner is clicked, the system attempts to spread the formula across all iterations and functional notation.

In this section, a function to find the Nth Fibonacci number is constructed using Forms/2. The construction is described in detail in the captions associated with the figures.

The process begins with a blank new form that will be used to define the generic instance of Fibonacci, a function that given N will return the Nth Fibonacci number. Within the form are placed three single cell variables, denoted N, T, and F. These will be used in the computation. N is to be the input value; F is to be the output value; and T will be used as a temporary to compute the i'th Fibonacci number beginning at 0 and ending at N.

The cell formula for T will be entered first. It will be expressed in terms of two prior instances of T. To create images of these prior instances the "iteration walking" icon in the upper right corner is clicked twice (once in the current instance and once in the prior instance [-1]). Notice that these instances are indicated by appending to their form name a negative offset enclosed in square brackets, see Figure 6.1. They are indicated relative to the current (1st) instance.

Next the cell formula itself is input by selecting the cell T in the current instance. This creates an expression window that displays the cell formula in two ways: textually and iconographically. The cell formula is entered by typing "+" indicating the addition function (all functions are displayed in functional notation), and then by clicking on the cells that correspond to T in the instance [-1] and then in the instance [-2]. The preferred means of entering cell expressions is expected to be a combination of typing simple operations and constants and pointing at cells to generate cell references. Other possibilities are typing completely the entire cell expression including specific cell references; or using purely graphical means of selecting operators from templates and drawing lines to connect corresponding inputs and outputs. Of these latter two options, the former is thought to be tedious because the cell reference syntax requires a fair degree of precision, while the latter is thought to be tedious because a few keystrokes will often suffice for numerous selections, placements, and connectings.

In the iconographical cell expression correspondsences between the cells as they appear in the expression and where they are located in the forms are indicated by colored (black and white patterns in the figures) borders. The use of color is to indicate that the images of two cells are really the same cell. This approach was also used in Alex [9]. It seems adequate given that most cell expressions use a limited number of terms and that only a single cell expression is displayed at any time. This approach however does not readily transfer to monochrome (as can be seen from the figures). An option the authors have resisted to avoid clutter is connecting corresponding cells with some kind of line; however, this is possibly desirable as an additional option.

The formula for T is entered in Figure 6.1 in terms of the i'th iteration. When this formula is completed, i.e., the close box in the upper left corner is clicked, the system attempts to spread the formula across all iterations and functional notation.
Figure 6.4: Finally we note the formula for $F$. It is expressed as when $C$ is reduced to 0; then $F$ is the value of $T$. When will produce no value, represented as $T$, until its condition is met. As long as $F$ produces no value and $F$'s value is needed, the system is forced to try another iteration.

Figure 6.5: The definition complies. we check up and document to some value and Result $F$ will be as long as $F$ is still not evaluated, which is correctly displayed as 1. Also in Figure 6.5, the form has been cleaned up and documented. The cells have been rearranged and text has been added. (Test and graphics can be added for effect anywhere within a form using a MacDraw-like editor.) Finally, a short iconic representation of the form is created in the upper left corner. This form shows only the input and output cells. When future instances of the Fibonacci form are created, this abbreviated form will be displayed rather than the larger form used to define Fibonacci. This has the effect of hiding the implementation details and reducing the variables to just those that function as inputs and outputs and that must be mapped in any form call. Cells with corresponding names are the same cells. At any point the iconic form can be expanded to see the larger full form.

To utilize this form, another form is created in Figure 6.6. It has two cells: TestN which will be set to some value and Result $F$ which will be the corresponding result value of Fibonacci. A new instance of Fibonacci is created. Parameters are set by setting the Result $F$ cell of CallFibonacci to $F$ of this new instance (Figure 6.6) and then setting N of this new instance of Fibonacci to the value of TestN in CallFibonacci (Figure 6.7). It is important to realize that this change affects only this instance of Fibonacci. This completes specifying the call linkage. When a trial value of TestN is specified, as in Figure 6.7, the system immediately recomputes all dependent cells.

The key to understanding the evaluation of Fibonacci is to trace backwards from the required output values, which in this case are all values of form CallFibonacci. TestN is a constant and thus already computed. Result $F$ demands the value of $F$ from the form Fibonacci, so the system focuses on Fibonacci to compute $F$.

To compute $F$, requires first computing the when condition $=(C, 0)$. This forces evaluation of $C$, which attempts to evaluate $fby(N, - (\text{Fibonacci}[-1]:C, 1))$. Since this is the first instance of Fibonacci, $fby(N, - (\text{Fibonacci}[-1]:C, 1))$ evaluates to $N$, which in turn forces the evaluation of $N$, which is mapped to TestN, which is 5. Thus $N$ and the initial value of $C$ are also 5. Now $=(C, 0)$ fails. This means that the when expression demands the value of $F$, and forces another iteration of Fibonacci in an attempt to compute $F$. If this iteration can produce a value for $F$, then it will be returned to Result $F$ as the value of $F$ for the evaluation of Fibonacci.

The next attempt proceeds similarly, again to evaluate $F$. It forces evaluation of $C$; this time computed from the prior value of $C$. The new value of $C$, 4, is still not 0, so the when fails forcing a new iteration. This process continues, until in the sixth iteration finally $C$ is 0. Now, the when expression demands the value of $T$, that is, the sixth. To compute $T$ for the sixth iteration the last expression of $fby(1, 1, + (\text{Fibonacci}[-1]:T, \text{Fibonacci}[-2]:T))$ is used, or $+(\text{Fibonacci}[-1]:T, \text{Fibonacci}[-2]:T)$. This calls for the evaluation of $T$ for the prior two iterations, the fifth and the fourth. And so on for the whole sequence of iterations back to the 0th; after which finally successive values of $T$ can be computed. Thus, $F$ is computed in the sixth iteration and its value becomes that of Result $F$.

The values of both $F$ and Result $F$ changed to correspond to the change in TestN. In the process Fibonacci was forced to expand to 6 iterations which the user has caused to be shown as in Figure 6.8. The display of Fibonacci is in its iconic representation and hence there is no
In Figure 6.9, a particular instance of Fibonacci is expanded, showing the cell values of all cells, both internal and external. Also in Figure 6.9, a particular cell F of instance Fibonacci[1], is expanded showing the cell formula with computed values.

7. SUMMARY

Forms/2 treats iteration in a form as a temporal sequence of instances of the form. The effect is that every variable of the form can be viewed as a temporal sequence. Within a particular instance of the form, a variable (cell) can be assigned at most once; however, all previous corresponding cell instances can be referenced as negative offsets relative to the current instance. This provides a natural means of accessing prior results without timing-related problems associated with cyclic feedback. Inputs to the iterative form are inputs to all instances of the form; outputs of the iterative form are only those of the last instance of the form.

Termination is controlled rather uniquely in Forms/2 as a direct result of the dependency-resolution evaluation mechanism. When all outputs can be computed, the computation terminates. When any cell cannot be computed because a when condition for evaluation is not met, then a new iteration is forced. This is somewhat analogous to looping until the conjunction of a set of conditions is met.

Technical issues aside, the Forms/2 solution seems to provide an intuitive approach to iterative problems. For the Fibonacci solution, allowing access to all prior instances makes it possible to express a solution without an intermediate variable whose function it is to retain (and time-delay) the value of the iteration two instances ago. This simplifies the expression of the problem.

8. BIBLIOGRAPHY


evidence of its internal operation here, only the external values of N and F.