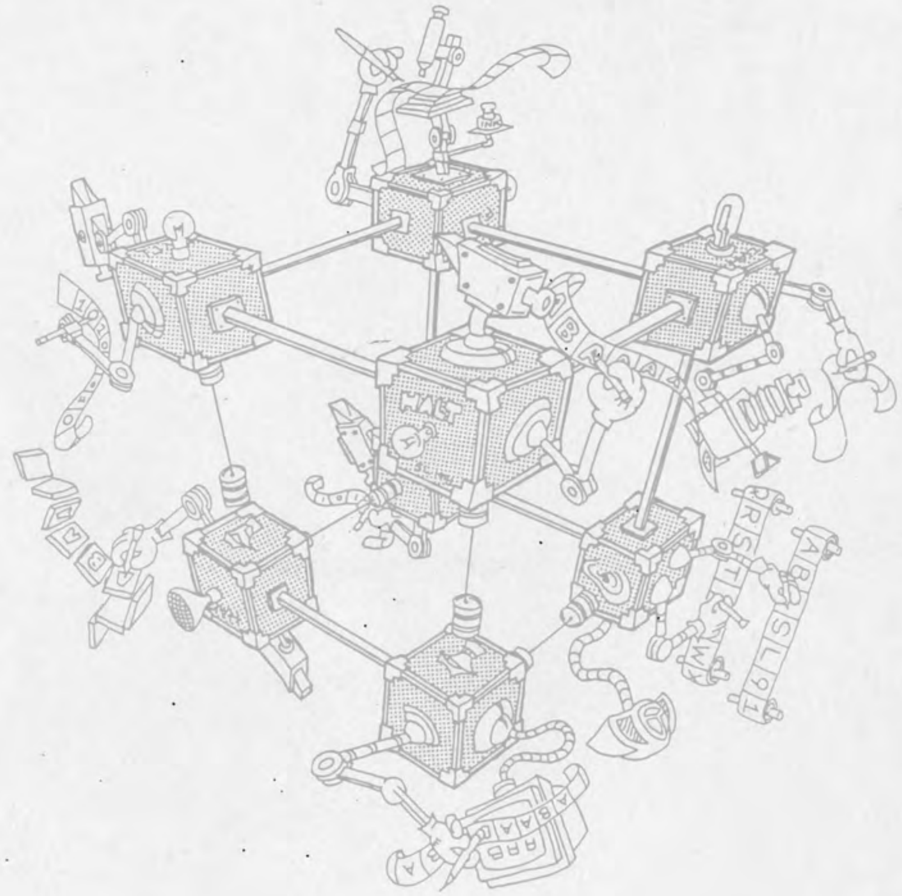


94-60-3

April 1994

FROM CONCRETE FORMS TO GENERALIZED ABSTRACTIONS THROUGH PERSPECTIVE-ORIENTED ANALYSIS OF LOGICAL RELATIONSHIPS

Sherry Yang
Margaret M. Burnett

Oregon State University
Department of Computer Science
Corvallis, OR 97331

From Concrete Forms to Generalized Abstractions through Perspective-Oriented Analysis Of Logical Relationships*

Sherry Yang and Margaret M. Burnett

Department of Computer Science, Oregon State University
Corvallis, Oregon 97331-3202 USA
E-mail: yang@research.cs.orst.edu, burnett@cs.orst.edu

Abstract

We believe concreteness, direct manipulation and responsiveness in a visual programming language increase its usefulness. However, these characteristics present a challenge in generalizing programs for reuse, especially when concrete examples are used as one way of achieving concreteness. In this paper, we present a technique to solve this problem by deriving generality automatically through the analysis of logical relationships among concrete program entities from the perspective of a particular computational goal. Use of this technique allows a fully general form-based program with reusable abstractions to be derived from one that was specified in terms of concrete examples and direct manipulation.

1: Introduction

We believe that concreteness, direct manipulation and responsiveness are among the most important advantages of working in a visual programming language. Toward this end, users of Forms/3 [Burnett 1991; Burnett and Ambler 1994] program very concretely, and receive continuous visual feedback throughout the process. But although they use direct manipulation and prototypical values extensively and without restriction during development, they do so with the expectation that the program they enter in such a concrete fashion will work the same way for any future values that might someday replace the prototypical values. The problem that we address in this paper is how to generalize the concrete program that was entered so that this expectation of generality can be fulfilled.

In solving this problem, we could not use the simple generalization approach based on physical relationships used by spreadsheets. Forms/3, while similar in some ways to spreadsheets, is more powerful than spreadsheets, in part because it supports reusable user-defined abstractions. Thus, unlike spreadsheets, logical relationships instead of physical relationships define the generalized meaning of the program. We chose not to use an abstract textual programming-language approach to allow the user to explicitly specify the intended generality, because such an approach would run counter to our goals of concreteness and programming with direct manipulation.

The generalization technique presented in this paper uses deductive analysis to derive a generalized program from a concrete one. Generalization is accomplished through the analysis of logical relationships among concrete program entities from the perspective of a particular computational goal. Because it does not use inference, there is no risk of "guessing wrong". This technique improves upon an earlier approach used by Forms/3 because it supports a modeless direct manipulation interface, and because it is flexible enough to handle all possible referencing and calling patterns, including some not commonly found in traditional programming languages. This allows the user to program concretely and flexibly, without unnecessary rules and restrictions.

1.1: Example of the problem

We will illustrate the generalization problem with an example. A Forms/3 program consists of cells on forms. A form is used to group related calculations, providing the functionality of a procedure in other programming languages. Figure 1 shows a solution to the n'th element of the Fibonacci sequence, which is the sum of the n-1'th and n-2'th Fibonacci numbers. The prototypical formula "5" has been specified for cell N on form FIB so that the user can receive concrete feedback. The solution involves three forms: one to compute the Fibonacci number for the desired N and two more to calculate N-1'th and N-2'th Fibonacci numbers. We term the original FIB the *model*, and FIB01 and FIB02 *instances* of FIB. In Forms/3, instances inherit their model's cells and formulas unless the user explicitly provides a different formula for a cell on

*This work was supported in part by the National Science Foundation under grants CCR-9215030/CCR-9396134 and CCR-9308649.

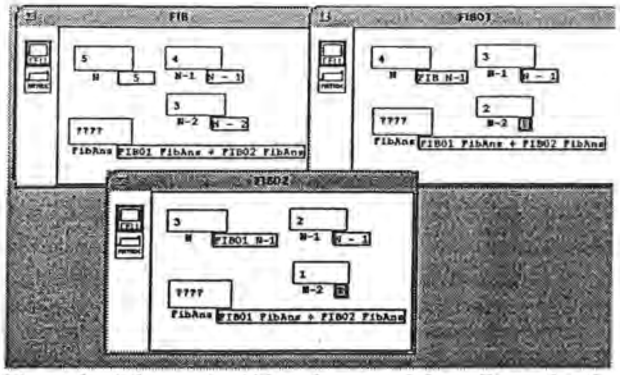


Figure 1: Fibonacci. Although none of the cells need to be named unless the user wants to name them, we've named them all here for clarity of discussion. Several of the concrete formulas are shown.

an instance. Changes in the model are propagated to instances.

To express the computation for the Fibonacci program, the following set of user actions is needed, but the user may enter them in any order. The only restriction is that FIB01 and FIB02 have to be created before the user can refer to them. Notice that formula references can be made by direct manipulation (clicking), or alternatively, the user can type cell names if they have names and are on the same form.

In Any Order

- Copy **FIB** to create **FIB01**
- Copy **FIB** to create **FIB02**
- User selects **FIB**'s **N** and types **5**
- User selects **FIB's N-1**, clicks in **FIB's N** and types **-1**
- User selects **FIB's N-2**, clicks in **FIB's N** and types **-2**
- User selects **FIB01's N** and clicks in **FIB's N-1**
- User selects **FIB02's N** and clicks in **FIB01's N-1**
- User selects **FIB's FibAns**, types **if N = 0 or N = 1 then 1 else**, clicks in **FIB01's FibAns**, types +, and clicks in **FIB02's FibAns**

If the formulas happen to be entered in the sequence listed above, the system will compute and display the value 5 as soon as the formula for FIB's N is entered. Likewise, the system will display the value 4 as soon as the formula for FIB's N-1 is entered, and so on for each formula, as shown in figure 1. The problem is how to enable the system to display the answer (8) as soon as the formula for FIB's FibAns is entered.

The problem lies in concreteness. As entered by the user, the recursive part of the formula for FIB's FibAns is the sum of the FibAns cells on FIB01 and FIB02. This is too concrete–without generalization, all future copies of FIB, regardless of how their N cells are changed, will sum the specific FibAns cells on FIB01 and FIB02 (which compute the fourth and third Fibonacci numbers). Because of this, without generalization, FIB01's FibAns formula (which was inherited from FIB) is circular, because it too refers to FIB01's FibAns and FIB02's FibAns. To solve this problem, the system needs to recognize and record the logical relationship between FIB and its instances from the perspective of computing FibAns, instead of recording the concrete program exactly as it was entered.

1.2: Factors involved in the problem

There are several factors that can interfere both with generalization and with immediate feedback of the forms entered in this fashion.

Modelessness: The first is the orderless nature of the input syntax. To encourage the user to concentrate on problem-solving rather than the computer's requirements, the user is entirely unrestricted in how s/he enters a program. Input is modeless, and formulas can be entered in any order. All cell references can be made by pointing at a cell, with no distinction in the way global references are made from the way parameter-passing is accomplished or from the way return values are referenced. To the user this means freedom to concentrate on the problem, but to the system it means lack of information. For example, in the set of formula entries given above, the user may enter the last formula first, thus referring to the return value FibAns from FIB01 before providing information that there will be an input parameter N. In this case, the reference to FIB01's FibAns appears to be a global (absolute) reference when in fact it is intended as the result of a parameterized subroutine-like call.

Flexibility: Second, there is no restriction on the patterns of references that can be made. This allows the user to modularize and re-use calculations with complete flexibility, including ways that are not supported in many programming languages. Thus, familiar program structures such as global references and subroutine-like uses of forms are possible, but less traditional referencing patterns such as mutually dependent modules (forms) and pipeline-like referencing are also possible. With this flexibility, a program's structure is hard to predict because it will not always fall into traditional patterns.

Circularity: Third, because of the support for recursion, a form must be able to be copied and re-used before its definition is completed. This can cause circular dependencies to be generated by the copying, as in the example above, making immediate feedback impossible until the relationships behind the concrete forms are analyzed to derive a generalized version of the recursive form.

2: Related work

Although Forms/3 does not use programming by example or programming by demonstration, because of its extensive use of prototypical values for concreteness and direct manipulation, Forms/3 shares with that family of VPLs some of the same difficulties in determining the generality intended by concrete prototypical values. Many by-demonstration systems use inference to solve this problem. Inference is most effective in a limited problem domain; for example, inference was used in the by-demonstration system Peridot [Myers 1993], which is a language specifically for user-interface specification.

PT [Ambler and Hsia 1993] is a general-purpose by-demonstration VPL that does not use inference. The feature of PT that makes this possible is the fact that generalized information is specified by the user as part of programming. For example, it is possible in PT to select an object by pointing at it, and to inform the system (using direct manipulation and formula-like operations) what attribute of the object caused it to be selected. An example of such an attribute might be the minimum-valued object in the selected group. Although Forms/3 does use examples, it is not a by-demonstration system. Another difference between PT and Forms/3 that is specific to the generalization problem is that in PT the user explicitly identifies the values that are examples versus those intended as constants.

The distinction between concrete and generalized versions of a form-based program is similar to the difference between absolute and relative reference found in traditional spreadsheets, which are among the earliest examples of the form-based approach to programming. However, in spreadsheets the generalization problem is binary; either a cell reference is absolute, or it is defined by a specific physical relationship that can be expressed by an integer offset. In Forms/3 the generalization problem is deriving reusable abstractions, which requires detecting logical relationships among cells that reference each other. For example, in Forms/3 if cell A references cell B on a different form, A could be similar to a formal parameter with B as the actual parameter, or B could be similar to a "return value" from another form's calculations, or B could be filling a role similar to a global constant.

The form-based systems NoPumpG [Lewis 1990] and NoPumpII [Wilde and Lewis 1990] are extensions to spreadsheets that support interactive graphics, but unlike traditional spreadsheets, the physical positions of cells do not determine a program's meaning. Because the NoPump systems are not intended to support general-purpose programming, there is no facility for generalized abstractions.

C32 [Myers 1991] is part of the Garnet user interface development system. It uses a spreadsheet model to allow users to construct constraints, which are relationships among graphical objects. In C32, prototypical values are explicitly designated by the user. C32 does not detect program structure based on the formula references. Instead the user explicitly provides formulas in LISP while referencing the prototypical objects by direct manipulation, and the user can later instruct C32 to substitute formal parameter variables for prototypical values in these LISP formulas.

An earlier version of Forms/3 [Burnett 1991; Burnett and Ambler 1994] used an internal textual notation to record the generality of a program, but the earlier version did not contain a facility for interpreting direct manipulations in order to produce the notation. In the earlier version, the internal textual notation described each copy of a form by enumerating exactly how it differed from the model. The notation supported the standard structures found in programming languages such as global references and subroutine-like relationships, but did not support some non-traditional structures because of the circularity they introduced into the notation. In this paper, we add the facility for interpreting direct manipulations to produce the notation, and revise the notation to support all program structures.

3: The approach

The approach to produce reusable generalized abstractions from concrete form-based programs is based on two key features:

(1) Deductive analysis of the relationship between concrete program entities to derive a generalized program: This frees user from having to explicitly identify prototypical values, parameters, variables, and the like. It also allows the user to program flexibly, without being restricted to any particular kind of program structure.

(2) Use of a calculation's *perspective* during the deductive analysis: Since the program structure is not explicitly specified by the user, perspective allows the system to locate the relationships that compute the intended results. Section 3.2 details the importance of perspective for this purpose.

3.1: Step 1: Recognizing relationships among program entities

As formulas are defined via direct manipulation in Forms/3, a *cell reference graph*. The cell reference graph is used to store the relationships and to trigger formula generalization. Informally, a cell reference graph is simply a representation of the cell references and the derived generalization information about them. There are

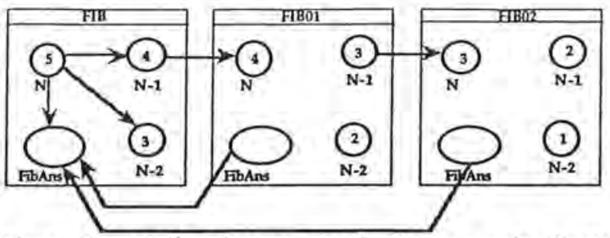


Figure 2: Cell reference graph showing only direct references (denoted by dotted edges). The direction of the edge indicates the direction of dataflow. For instance, the edge from N to N-1 in FIB indicated that N-1 is a reference to N, i.e., the value of N flows into N-1.

three different kinds of references in a cell reference graph: direct references created by the user via direct manipulation, inherited references, and fully generalized ones. More formally, the cell reference graph CG = (V, E) is a directed graph where

V= {u | u is a cell in the program}
E = {(u,v) | cell v makes a reference to cell u}
and a function f: E–>L that assigns a label to each edge according to the origin of the reference, where:

$$f(E) = \begin{cases} 0\text{: Direct reference by the user} \\ 1\text{: Inherited reference} \\ 2\text{: Fully generalized reference} \end{cases}$$

We define 3 subsets of E:

$$DE \subseteq E \quad = \{(u, v) \mid f(u, v) = 0\}$$
$$IE \subseteq E \quad = \{(u, v) \mid f(u, v) = 1\}$$
$$GE \subseteq E \quad = \{(u, v) \mid f(u, v) = 2\}$$

DE, IE and GE are disjoint sets and their union is E.

3.1.1: Incremental processing of the user's actions

The cell reference graph is built incrementally, and this allows incremental analysis. Incremental analysis of the direct references occurs as soon as the user refers to a cell by clicking on it. At first, each direct reference is considered to be an element of DE. This is depicted in figure 2, which shows only the edges in DE in the cell reference graph for the Fibonacci example.

Some of the direct edges shown in figure 2, such as the edge from N to N-1 in FIB, are internal references. An *internal reference* is an edge (u,v) in E such that cell u and cell v are on the same form. Internal references do not need further processing to become general, because the relationship between cells on the same form is made clear by the fact that they are encapsulated in one form, and will be reusable on new copies of the form without further generalization. Therefore, whenever a direct edge is added that is an internal reference, it is immediately considered to be a generalized edge and is defined as an element of GE instead of DE.

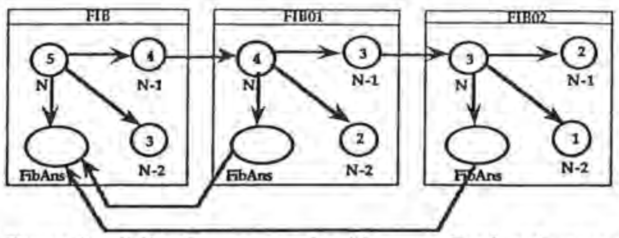


Figure 3: Cell reference graph with generalized references (denoted by solid edges) propagated to instances.

When the user copies a form for reuse, inherited edges are added to the cell reference graph. If the original edge was in GE, its inherited version is also placed in GE because it is by definition already fully general. Figure 3 shows that the internal references were generalized, and that this is reflected in the inherited references in form instances.

Inherited edges that did not originate in GE, i.e. those that have not been generalized yet, are placed in IE. This is depicted in the edges connected to the three FibAns nodes in figure 4.

At this point, the cell reference graph has some similarities to a dataflow graph, but it contains anomalies and information about relationships that need to be analyzed before it can be reduced to a true dataflow graph. In figure 4, for example, the formula for FibAns in FIB01 and FIB02 is incorrect, because instead of the circular references to themselves and each other, the FibAns cells on FIB01 and FIB02's FibAns should reflect the general roles that they have in computing the N-1'th and N-2'th numbers in the sequence. This type of anomaly triggers generalization, as is discussed in the next section.

3.1.2: Cycle detection to trigger formula generalization

Whenever edges are added, the graph is analyzed to find out if a cycle has been formed. Detection of a cycle will either result in an error message or will trigger generalization.

An error message is generated by a cycle formed by direct and/or generalized references only (DE ∪ GE),

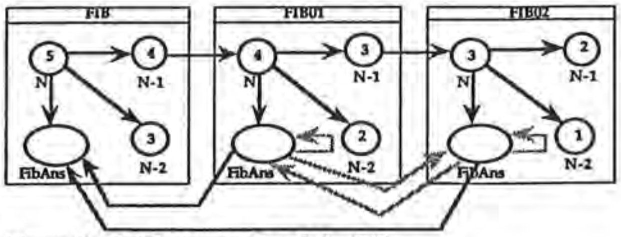


Figure 4: Cell Reference Graph for FIB

⟶	*Direct references ∈ DE*
⇢	*Inherited references ∈ IE*
⟶	*Generalized references ∈ GE*

because circular formulas are not allowed in Forms/3. This type of cycle is formed by direct references entered by the user. Since the cycle detection is done incrementally, the user's most recent reference created the cycle, and therefore this last reference must be illegal. The user will be warned about the error and the last reference will be rejected. Figure 5 shows an example of an illegal circular reference formed by edges in DE.

However, if a cycle is formed that includes at least one edge in IE, i.e., an inherited reference, generalization must occur right away to remove the cycle because the cycle prevents responsiveness–the computation in its concrete form would be non-terminating. Figure 4 included an example of such a cycle formed by the inherited references of the cell FibAns.

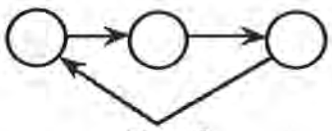


Figure 5

3.1.3: Other triggers

Detecting cycles in the cell reference graph as described above is one way to trigger formula generalization. The other ways that trigger formula generalization are:

(a) Saving or closing a form: Generalization must be done when saving or closing a form because otherwise the system might be saving references that are not reusable.

(b) Copying a form: Reusing a form in this way requires the form to be generalized first.

(c) Deleting an instance: Once an instance has been deleted, the system will not be able to use it later in deriving a generalization.

Each time generalization is triggered, it is sufficient to only generalize those forms that are currently on the screen, even though they may contain references to forms previously created but no longer displayed. This is because all the other forms were already generalized when they were saved or removed from the screen. This fact reduces the time complexity of the generalization technique to be proportional to the amount of the program currently displayed on the screen.

3.2: Step 2: Generalizing the relationship

The goal of generalization is to reduce the cell reference graph to a dataflow graph that contains only references that are fully general (GE). Generalization is done by first performing a modified topological sort on the cell reference graph minus its IE edges to discern the flow of the logical relationships. The formulas are then generalized and recorded by describing the relevant references from the perspective of each goal cell. When all of the references have been generalized, the cell reference graph is the same as a general dataflow graph.

3.2.1: Identifying and using perspective

First, a modified topological sort is performed to identify the logical relationships of the referenced cells from the perspective of the goal cells. The topological sort is modified in that it preserves the edges in the graph. In the topologically-sorted CG = (V, E), the vertices in V are ordered such that if CG contains an edge (u,v), then vertex u appears before vertex v in V. The ultimate goals of the computations are the last node(s), i.e. the sink(s), in the sorted graph. Topologically sorting the graph locates all the sinks and also locates all the sources. Figure 6 shows the topologically-sorted cell reference graph for the Fibonacci example.

It is from the perspective of each goal that the generalized references to the other cells contributing to it take place. Finding this perspective is important because (1) it makes known the beginnings and ends of the dependencies, and (2) in mutually dependent forms, as in the case of co-routines and other non-traditional structures, perspective allows the system to deal with each individual cell's computation path separately to avoid generating circular expressions of dependency.

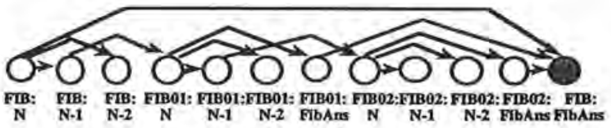


Figure 6

3.2.2: Recording the generalized relationships

After the relationships have been located and sorted out by the topologically-sorted cell reference graph, the relevant portions of these relationships are recorded in the fully generalized formulas. This recording is done using an internal textual notation. It is important for this notation to provide enough information for the system (1) to recognize the needed form instance if it exists, and (2) to create the needed instance from its model form if such an instance doesn't exist. This is important because if the system were not given enough information to automatically locate and create these instances, the only way a form could be reused *during execution* would be for the programmer to manually copy it from the model and modify it, just as he or she did while programming it originally. (Since recording this information makes the internal description rather long and involved, it is never seen or used by the user.)

To accomplish this, in the formula for a cell X, each reference to a cell on another form instance is described by enumerating each way the form instance is different from the model, if that difference is relevant to the goal cell X. If a form instance is different from the model in ways that are not relevant to computing the goal cell X, then that difference is recorded as *don't care*. The *don't care* differences are not used in the computation, but they provide information necessary for the system to recognize the needed form instance if it already exists.

This notation generalizes the Fibonacci problem posed at the beginning of this paper as follows. As shown in figure 6, the FibAns cell on FIB is the goal. The formula for FibAns makes references to three cells, FIB's N, FIB01's FibAns and FIB02's FibAns, and they are recorded as:

FIB's N: this is an internal reference. This reference is recorded as "self:N". (The use of "self" denotes the fact that this reference is on the same form as cell FibAns, the goal cell.)

FIB01's FibAns: This is a reference to another cell FibAns on a form instance that, when viewed in figure 6 from the perspective of the goal cell FibAns, is the instance of FIB in which N is defined as FIB's N-1. FIB is "self" from the perspective of the goal, so this reference is recorded as "FIB(N ☞self: N-1):FibAns".

FIB02's FibAns: FIB02's FibAns refers to N which refers to FIB01's N-1. FIB01's N-1 refers to its N which refers to FIB's N-1. Since this path leads back to the form FIB, ("self"), the reference is generalized as:

"FIB(N ☞FIB(N ☞self: N-1):N-1):FibAns"

Putting these three references together gives the complete generalized formula for FIB's FibAns:

```
"If self:N = 0 or self:N = 1 then 1 else
      FIB(N ☞ self: N-1):FibAns +
      FIB(N ☞ FIB(N ☞self: N-1):N-1):FibAns"
```

Several additional examples are given in section 4, which discusses the generality of the approach.

The Fibonacci example contains many relative (i.e. non-absolute) relationships. Examples of non-absolute relationships in other languages are those created by parameter-passing in traditional languages, by relative referencing in commercial spreadsheets, and referencing patterns such as the networks that can be built up in dataflow languages.

Absolute relationships, the functional equivalent of absolute references in commercial spreadsheets and of references to global constants in traditional languages, are unchanged by generalization. For example, if a cell X on some form F referred to cell FibAns on an instance of FIB in which N's formula was 5+2, X's formula would be "FIB(N ☞ 5+2)". The FIB(N ☞ TaxTable:Z) describes an instance of FIB in which N's formula is an absolute reference to cell Z on the (model) form TaxTable.

3.2.3: Reduction to a generalized dataflow graph.

As each cell's formula is generalized and recorded, the edges involved are relabelled as generalized edges. When the last edge is processed, the resulting cell reference graph is the same as a dataflow graph. After generalization, it is possible to discover that cycles remain that were formed by at least one of the generalized edges. Since such a cycle would be illegal, the dataflow graph is analyzed for cycles after generalization is complete.

4: Generality of the approach

This approach can be used to generalize all legal program structures. We will illustrate the generality of the approach through the use of *form-collapsed multi-graphs*, a diagram we introduce solely for the purpose of demonstrating the generality of the approach. (Form-collapsed multi-graphs are not part of the approach itself). Each node in the form-collapsed multi-graph is a form and all the edges from the fully-generalized dataflow graph are retained at the form level. Thus each form has the same number of incoming and outgoing edges as in the cell reference graph. Figure 7 shows the resulting form-collapsed multi-graph for the Fibonacci program.

Program structures are reflected by these form-collapsed multi-graphs. There are only four possible patterns of these graphs. An example of each is given in figure 8. For each of the patterns, we will show an example and illustrate how generalization is performed.

Pattern 1, no cycles: This pattern reflects abstractions with no parameter-passing, i.e. all references are absolute references, like global variables in other languages. Figure 9 shows the generalized dataflow graph of such a program. This program defines a screen saver with a floating image. The location of the floating image is computed and updated based on the system clock. Because all the references are absolute, they don't change with generalization. The formula for cell FloatingImage thus needs only to reflect the dataflow path in absolute terms in recording how referenced cell Image's form differs from the model in ways that are relevant to cell FloatingImage: "Picture(X ☞ SystemClock:Sec; Y ☞ SystemClock: Min):Image"

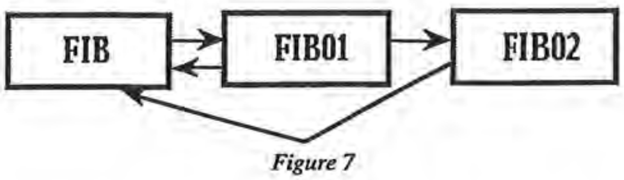


Figure 7

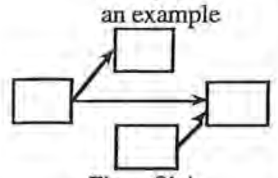


Figure8(a)

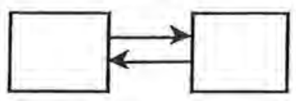


Figure 8(b)

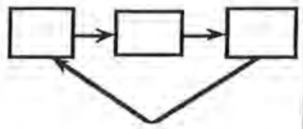


Figure 8(c)

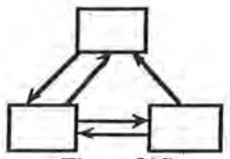


Figure 8(d)

<u>*Pattern 2, a graph with a 2-node cycle*</u>: A 2-node cycle models the functionality of a traditional subroutine with parameters. (Notice that such a cycle is at the granularity of forms, not cells. As previously discussed, circular cell references are not allowed after generalization is complete.) Figure 10 shows the generalized dataflow graph of a factorial program. Factorial of a number N is defined as N! = N * (N-1)! The topological sort determined that cell Ans on form Fact is the goal of the computation. In defining Ans's formula, those cells contributing to the computation and also on the same form as the Ans, namely N and N-1, are recorded as generalized references. The generalized formula for Ans becomes "If self:N = 1 then 1 else self:N * Fact(N ☞ self:N-1):Ans".

<u>*Pattern 3, a cycle with more than 2 nodes*</u>: This pattern corresponds to a program structure that is not commonly found in most textual programming languages, because the values are passed forward, and only travel back to the original caller through a circular path. Figure 11 shows the generalized dataflow graph of a program that computes the average of a list of numbers. The generalized formula for Average reflects the path of the parameters as they travel through the forms: "ComputeAvg(ΣX ☞ Summation(List ☞ Counter(List ☞ self:List):List):ΣX; n ☞ Counter (List ☞ self:List):n): $\overline{X}$".

<u>*Pattern 4, any combination of the above patterns*</u>: Form FIB was one example of pattern 4, because it included both a 2-node cycle and a 3-node cycle. We have shown how each of the basic three patterns can be handled. Any combination of the above patterns can also be handled, because each cell's formula is recorded by isolating information relevant from that particular cell's perspective. The granularity of describing the form instances is at the cell level as it relates to one cell's goal, which means that regardless of how many ways the basic three patterns are combined, no circularity in the notation can arise, because circularity at the cell level is not present.

Another example of pattern 4 is depicted in figure 12, a dataflow graph containing two 2-node cycles. This is a program to compute the permutation of N,K, defined by P(N,K) = N! / (N-K)!. The generalized formula for P(N,K) is "self:N! / self:N-K!" The formulas for cells N! and N-K! are subsequently generalized as "Fact(N ☞ self:N):Ans" and "Fact(N ☞ self:N-K):Ans" respectively. Two mutually-dependent co-routines would also be modeled by pattern 4 as a group of 2-node patterns.

5: Efficiency

The generalization technique developed in this paper is amenable to scaling up because its time complexity is bounded by V, the number of cells currently on the screen. Thus the complexity does not grow with the size of the program, but only with the amount of the program on the screen. The incremental cost to add edges to the graph is O(1) when a direct reference is made, or $O(|V_F|)$ when a form F is copied (where $|V_F|$ denotes the number of cells on F). Cycle detection for the cell reference graph CG is O(|V|), using the standard DFS cycle-detection approach, and the topological sorting is θ(|V|+|E|) using the standard algorithm. The implementation of the technique is currently being added to the Forms/3 system.

6: Conclusion

The approach presented in this paper allows a fully general form-based program to be derived from one that whose formulas were specified with concrete examples and direct manipulation. This is accomplished through recognizing and recording the logical relationships among the concrete data, from the perspective of the computational goals of the program fragment currently on the screen. The key benefits of the technique are:

(1) It supports a visual style of general-purpose programming which incorporates extensive use of concrete examples, direct manipulation, and responsiveness.

(2) It removes any order requirements from the user's program entry process. This frees the user to concentrate on problem-solving, rather than having to concentrate on providing information to the computer in the order the computer wants it. Since declarative languages strive to be solely dependent on definitions rather than on the order a program is specified, this is an especially important attribute for declarative VPLs.

(3) It is scalable, because its performance is bounded by the number of program entities currently on the screen, not by the number of entities in the program.

(4) It does not use inference. Hence there is no restriction to domain-specific programming tasks, because there is no risk of "guessing wrong" in the generalization process.

Acknowledgments

We would like to thank Marla Baker for her assistance in editing this paper and Carisa Bohus and Pieter van Zee for their helpful suggestions on earlier drafts.

References

[Ambler and Hsia 1993] Allen Ambler and Yen-Teh Hsia, "Generalizing Selection in By-Demonstration Programming," *Journal of Visual Languages and Computing*, 4(3), 283-300, September 1993.

[Burnett 1991] Margaret Burnett, "Abstraction in the Demand-Driven, Temporal-Assignment, Visual Language Model", Ph.D. Thesis, Department of Computer Science, University of Kansas, 1991.

[Burnett and Ambler 1994] Margaret Burnett and Allen Ambler, "Interactive Visual Data Abstraction in a Declarative Visual Programming Language", *Journal of Visual Languages and Computing*, March 1994.

[Lewis 1990] Clayton Lewis, "NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery", in Visual *Programming Environments: Paradigms and Systems*, (E. Glinert, editor), IEEE CS Press, Los Alamitos, California, 526-546, 1990.

[Myers 1991] Brad Myers, "Graphical Techniques in a Spreadsheet for Specifying User Interfaces", ACM Sigplan Notices/Proceedings CHI '91, New Orleans, Louisiana, 243-249, Apr. 27-May 2, 1991.

[Myers 1993] Brad Myers, "Peridot: Creating User Interfaces by Demonstration", *Watch What I Do: Programming by Demonstration* (A. Cypher, editor), The MIT Press, Cambridge, Massachusetts, 125-154, 1993.

[Wilde and Lewis 1990] Nicholas Wilde and Clayton Lewis, "Spreadsheet-based Interactive Graphics: from Prototype to Tool," ACM Sigplan Notices/Proceedings CHI '90, Seattle, Washington, 153-159, April 1990.

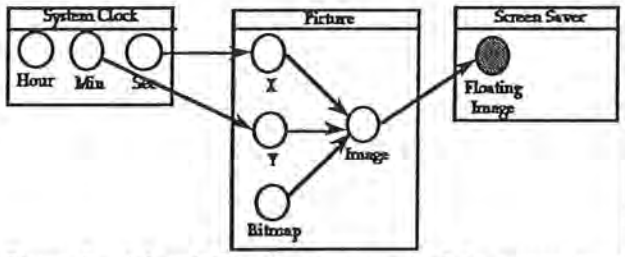
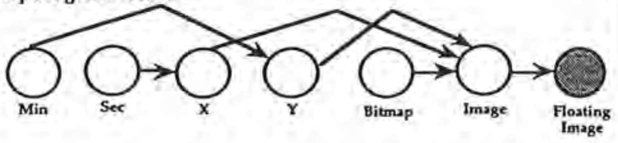


Figure 9: (above) The dataflow graph of the screen saver program, superimposed on the form-collapsed multi-graph. This is an example of the first pattern , in which there are no cycles. The goal is shown shaded. (below) The result of the topological sort.



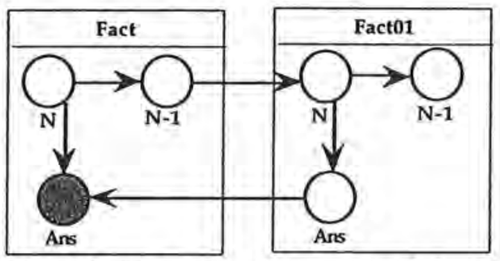
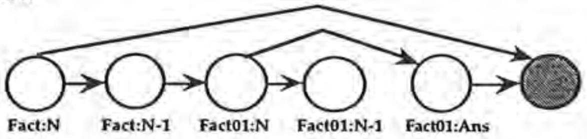


Figure 10: (above) The dataflow graph of the factorial program, superimposed on the form-collapsed multi-graph. This is an example of the second pattern, a 2-form cycle. The goal is shown shaded. (below) The result of the topological sort.



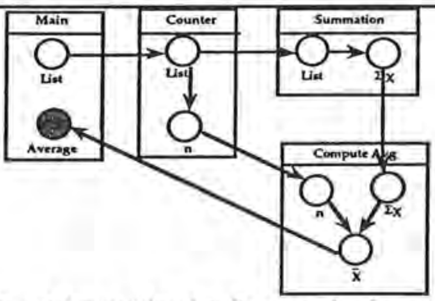
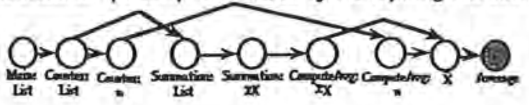


Figure 11 : (above) The dataflow graph of a program to compute the average of a list of numbers, superimposed on the form-collapsed multi-graph. This is an example of the third pattern , a cycle involving more than 2 forms. The goal is shown shaded. (below) The result of the topological sort.



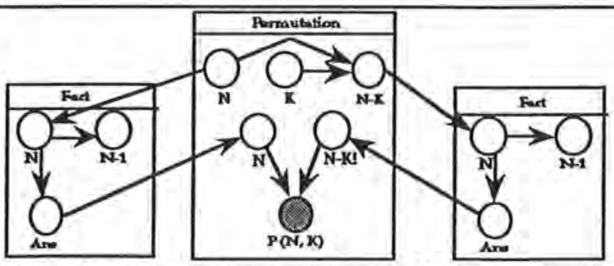


Figure 12: (above) The dataflow graph of a permutation program, superimposed on the form-collapsed multi-graph. This is an example of the fourth pattern, which is any combination of the other three patterns. The goal is shown shaded. (below) The result of the topological sort.