

# A Scalable Method for Deductive Generalization in the Spreadsheet Paradigm

MARGARET BURNETT

Oregon State University

SHERRY YANG

Oregon Institute of Technology

and

JAY SUMMET

Oregon State University

---

In this paper, we present an efficient method for automatically generalizing programs written in spreadsheet languages. The strategy is to do generalization through incremental analysis of logical relationships among concrete program entities from the perspective of a particular computational goal. The method uses deductive dataflow analysis with algebraic back-substitution rather than inference with heuristics, and there is no need for generalization-related dialog with the user. We present the algorithms and their time complexities and show that, because the algorithms perform their analyses incrementally, on only the on-screen program elements rather than on the entire program, the method is scalable. Performance data is presented to help demonstrate the scalability.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*procedures, functions, and subroutines*; H.4.1 [**Information Systems Applications**]: Office Automation—*spreadsheets*; D.1.7 [**Programming Techniques**]: Visual Programming

General Terms: Human Factors, Languages

Additional Key Words and Phrases: Human-computer interaction, concrete programming, graphical programming, Forms/3, spreadsheet languages, generalization

---

## 1. INTRODUCTION

Like most researchers involved in end-user and graphical programming, we believe that concreteness, direct manipulation, and immediate visual feedback are critical characteristics for end-user and graphical programming languages. These characteristics were first made widely available to end

---

This research was supported in part by the National Science Foundation under awards CCR-9806821 and ITR-0082265, and by NASA grant NGT540022.

An early version of this approach was presented in Yang and Burnett [1994].

Authors' addresses: Margaret Burnett, Department of Computer Science, Oregon State University, Corvallis, OR 97331; email: burnett@cs.orst.edu; Sherry Yang, Computer Systems Engineering Technology, Oregon Institute of Technology, 3201 Campus Dr., Klamath Falls, OR 97601; email: yangs@oit.edu; Jay Summet (current address): College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280; email: summet@cc.gatech.edu.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2002 ACM 1073-0516/02/1200-0253 \$5.00

users in commercial spreadsheet systems, with great market success. Today, spreadsheet systems are the most widely used type of end-user programming language. In this paper we present a new generalization method that supports extended use of these characteristics in the spreadsheet paradigm.

The spreadsheet paradigm includes not only commercial spreadsheet systems, but also a number of research languages that extend the paradigm with features such as gestural formula specification [Burnett and Gottfried 1998; Leopold and Ambler 1997], graphical types [Burnett and Gottfried 1998; Wilde and Lewis 1990], visual matrix manipulation [Ambler 1999; Wang and Ambler 1996], high-quality visualizations of complex data [Chi et al. 1998], and specifying graphical user interfaces (GUIs) [Myers 1991]. Forms/3 [Burnett and Gottfried 1998; Burnett et al. 2001a] is the extended spreadsheet language in which we have prototyped our method. Forms/3 can be described as a “gentle slope” language [Myers et al. 1992; Myers et al. 2000], intended to allow end users to create spreadsheets with fewer limitations than exist in other spreadsheet languages, while at the same time allowing more sophisticated users with some programming background to create more powerful spreadsheets without having to leave the spreadsheet paradigm.

Our generalization method is compatible with traditional, single-grid spreadsheet languages, but also supports extended spreadsheet languages such as Forms/3 that relax several of the traditional restrictions. In solving the generalization problem in a way general enough to handle such extensions, we could not use the strictly spatial generalization approach based on physical relationships traditionally used by commercial spreadsheet systems. In these systems, when a user copies or “fills” a formula into other cells, the system generalizes any cell references in the copied formula based on the new cells’ number of rows and columns away from the original cell.

The commercial spreadsheet spatial approach fell short because, in the presence of features such as multiple grids and linked spreadsheet copies (such as multiple copies of a spreadsheet containing different data but mostly the same formulas), logical relationships—in addition to or instead of spatial relationships—must be used in defining the generalized meaning of the program. In considering alternatives to the spatial approach, we chose not to require the user to explicitly specify the intended generality in an abstract textual programming language, because such an approach would run counter to our goals of concrete programming via direct manipulation.

Many mechanisms to support automatic generalization of concrete programs have been devised for graphical programming languages [Ambler and Hsia 1993; Frank and Foley 1994; Kahn 1996; Kurlander and Feiner 1992; Lieberman 1993; McDaniel and Myers 1999; Myers 1993, 1998; Olsen 1996; Perrone and Repenning 1998; Repenning 1995; Sassin 1994; Smith 1975; Sugiura and Koseki 1996; Vander Zanden and Myers 1995; Wolber 1997]. Some of these use inference,<sup>1</sup> and some require the user to explicitly provide the

---

<sup>1</sup>The convention in literature about demonstrational programming languages is that the term *inference* means reasoning techniques employing guesswork. For example, in the context of programming by demonstration, Myers and Maulsby [1993] defined *inferencing* as the computer

generalized meaning. The two most significant ways in which our method differs from these are the following:

- *neither inferential nor user-assisted*: the generalization method presented in this paper uses deductive analysis with algebraic substitution 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.”<sup>2</sup> In contrast, other approaches to automatic generalization either use inference or require the user to provide additional information to the generalization mechanism;
- *scalable*: the method presented here specifically addresses scalability. Scalability is necessary to maintain immediate visual feedback when programs increase in size. The method processes a program in a lazy, incremental fashion, operating only on the portion of the program currently on the screen. Thus, its cost is dependent upon display size, not upon overall program size. Prior generalization research has not investigated scalability properties of generalization methods.

In developing our method, we imposed three design constraints:

- *orderlessness constraint*: the order in which a program is edited must not determine the final generalized form of the program. That is, if two edit sequences result in two concrete programs eventually becoming identical, then their generalizations must also be identical. This constraint is necessary to maintain the incremental, opportunistic, editing process that is usual in the spreadsheet paradigm;
- *modelessness constraint*: the method must not impose modes upon the user. By this we mean, there cannot be a “pregeneralization mode” that requires user actions or reasoning that are different from those of a “postgeneralization mode.” Modelessness is one of the key characteristics of the spreadsheet paradigm that distinguishes it from traditional programming methods. Hence, it was critical to guard against loss of this characteristic;
- *generality constraint*: The method must be flexible enough to handle all non-circular cell referencing and linked spreadsheet “design patterns,” including patterns different from the call-return pattern of traditional programs, such as patterns resembling coroutines and pipelines. These patterns are possible in spreadsheets because of nontraditional scope and lifetime rules for its cells: every “variable” (cell) has global scope and persists until the program ends, and thus even intermediate cells can be referenced from any “procedure” (spreadsheet) at any time. The generality constraint honors this referencing flexibility.

---

trying to determine the appropriate generalization using *heuristics*, which is in turn described as an alternative term for guessing. In this paper, we follow this convention.

<sup>2</sup>Of course, the system cannot read the user’s mind. If the user enters incorrect relationships or enters a concrete program that does not fulfill his or her goals, then the generalized version will not fulfill those goals either.

We begin our presentation with a discussion of related work in Section 2, followed by an introduction to the generalization method from the user's perspective in Section 3. Section 4 explains in detail how the generalization method works behind the scenes, including the algorithms, their worst-case time costs, and the resulting scalability of the method. Section 5 extends the method to grids, temporal sequences and animations, and user-defined types. Performance measurements are given in Section 6, and the paper concludes in Section 7. For additional perspectives on the method's correctness properties and costs, and for additional programming examples, see the supplementary information included in Appendices A through D (available online in ACM's Digital Library, <http://portal.acm.org/portal.cfm>) or the full technical report in Burnett et al. [2001b].

## 2. RELATED WORK

The program entities that need to be generalized in a graphical or nongraphical spreadsheet language are the cell references. When a user enters a formula referencing some cell, that particular reference is not a problem, but when the formula is copied, replicated, or otherwise reused, its references need to be somehow generalized so that they will refer to cells appropriate for the new contexts.

For spreadsheet languages based upon a single grid, including commercial spreadsheet languages, generalization of cell references has been based strictly upon spatial relationships. However, this spatial strategy is limited—it works only for a single grid and, even with this restriction, it relies on inference heuristics that sometimes guess wrong. For example, if an Excel user inserts a new row just before a total row in a grid, the system automatically adjusts previous ranges used in the sums to exclude the row being added: that is, it infers that the user does not want to include the new row in the sums. (This is the correct inference if the user is adding a new subtotal row but is the incorrect inference if the user is adding a new detail row.) The method presented in this paper does not guess about spatial relationships; rather, it focuses primarily on logical relationships, and hence does not restrict a spreadsheet to any particular number of grids—there can be multiple grids, and there can also be individual cells not part of any grid at all.

An earlier version of Forms/3 [Burnett and Ambler 1994] made a start at solving the generalization problem by contributing an internal textual notation to record the generality of a program, including its logical relationships in addition to its spatial relationships. However, no facility was present to deductively interpret a user's direct manipulations in order to produce the notation. Also, although this internal notation was powerful enough to support the standard structures found in traditional programming languages such as subroutine-like relationships, it was not powerful enough to support nontraditional relationships.

Because of Forms/3's extensive use of prototypical values for concreteness and direct manipulation, it shares with demonstrational languages some of the same difficulties in determining the generality intended by user-provided

concrete prototypical values. Programming-by-demonstration systems solve the generalization problem in one of two ways: by either (1) direct reliance on the user to explicitly provide generalization information, or (2) inference with user assistance and/or corrections. In both cases, the user assists the generalization process, by specifying which portions of the input should be generalized, by having to explicitly request generalization to achieve/maintain correctness, by selecting the most appropriate general form after the system has generated multiple possibilities, and/or by making corrections (e.g., via counterexamples) when the system has inferred an incorrect general program.

There are several ways a user can provide generalization information explicitly. For example, in Tinker [Lieberman 1993], the user explicitly provides the entire generalized meaning of a program. In PT [Ambler and Hsia 1993], the user explicitly differentiates between constants and generalized parameters. In ToonTalk [Kahn 1996] and in Topaz [Myers 1998], the user explicitly specifies the generalization parameters. In AgentSheets [Perrone and Repenning 1998; Repenning 1995], the user specifies any desired generalizations via analogies (“Cars move on roads like trains move on tracks”). In KidSim/Cocoa/Stagecast [Cypher and Smith 1995; Heger et al. 1998], users can generalize a graphical rule they have demonstrated by stating that it should abstract beyond the specific object type to, for example, all instances of a set of types.

Some programming-by-demonstration systems, such as Peridot [Myers 1993], Pavlov [Wolber 1997], and Gamut [McDaniel and Myers 1999], do not require as much explicit information from users and can rely on inference. This is possible because their limited problem domains (e.g., user interface construction, 2D board games) reduce the number of valid possibilities. Lapidary [Vander Zanden and Myers 1995], a one-way constraint demonstrational system for user interface construction, uses explicit user specification of some kinds of parameters but uses inference to generalize other kinds of parameters. Other inference-based approaches to generalization include inductive groups [Olsen 1996], in which the system generalizes properties of objects in a group explicitly selected by the user, and inferring the generalization of macros or functions from the user’s command history in applications such as in DemoOffice [Sugiura and Koseki 1996], ProDeGE+ [Sassin 1994], and Chimera [Kurlander and Feiner 1992].

Any inference technique can guess wrong, and systems based on inference therefore can generate incorrect programs if this possibility is not addressed. For example, Pavlov asks the user to constrain its inference via choices made through a dialog. When there is more than one possible generalization of a user’s history, DemoOffice uses a heuristic to choose the most likely, whereas ProDeGE+ and Chimera present the user with a dialog asking for clarification. One of the most advanced with respect to communicating about incorrect guesses is Gamut. When Gamut makes a mistake in generalization (either a failure to act or an inappropriate action) the user corrects the system via “Stop That” and “Do Something” buttons. Because of this, in Gamut, the user assists the system only when it fails.

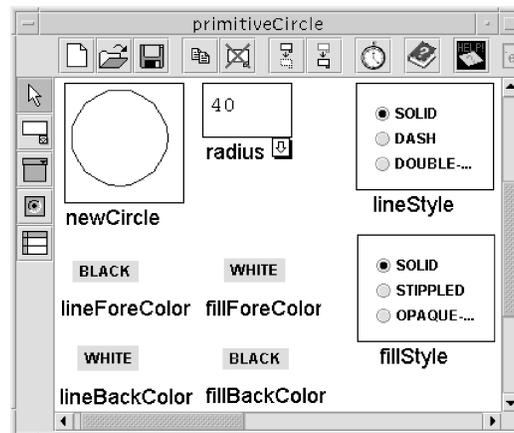


Fig. 1. A portion of a Forms/3 form (spreadsheet) that defines a circle. The attributes of the circle in cell `newCircle` are specified by the other cells. A user can view and edit a cell's formula by clicking on its tab, such as that attached to the bottom right of cell `radius`. Radio button sets (e.g., `lineStyle`) and popup menus (e.g., `fillForeColor`) are the equivalent of cells with constant formulas.

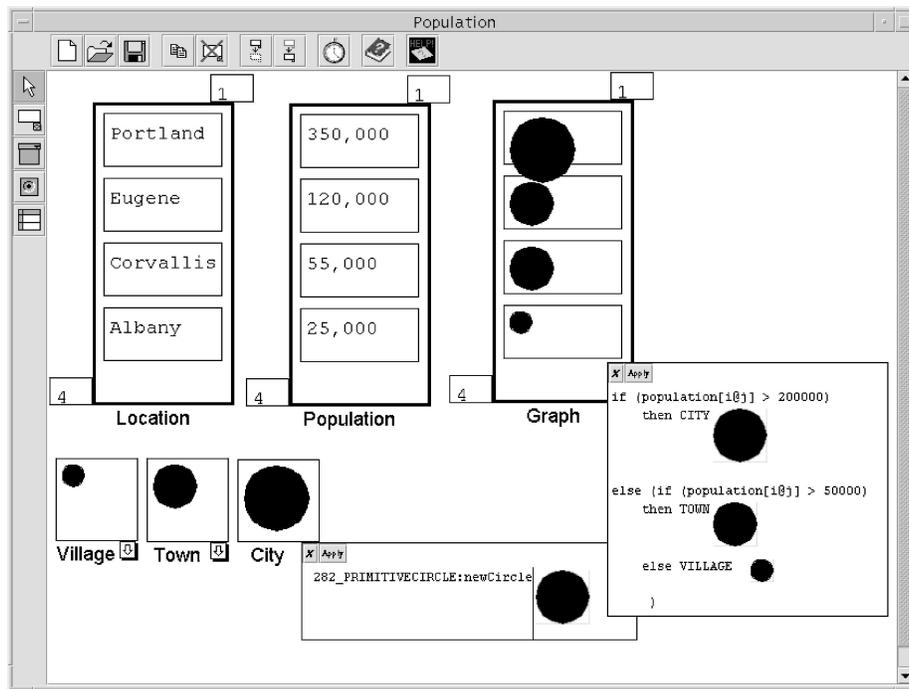
Forms/3's generalization method is different from the methods described in this section in that Forms/3 does not use inference heuristics, and in Forms/3 the user does not need to explicitly request, assist, participate in, or correct the generalization process.

### 3. THE GENERALIZATION METHOD FROM THE USER'S PERSPECTIVE

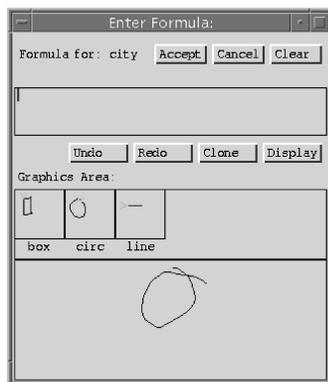
#### 3.1 Generalization Example: Population Visualization (User's View)

Suppose a user, such as a population analyst, would like to define a visual representation of data using domain-specific visualization rules that make use of the built-in `primitiveCircle` form (spreadsheet) of Figure 1. Figure 2 shows such a visualization in Forms/3. The program categorizes population data into cities, towns, and villages, and represents each with a differently sized black circle. If a referenced cell is on another form, the notation displayed in the formula is to precede the cell name with the form id and a colon, such as "282\_primitiveCircle:newCircle," as in Figure 2(a). In the example of Figure 2, the population analyst defined the formulas for cells `city`, `town`, and `village`, either by pointing at cells on three different copies of the system-defined `primitiveCircle` form, or by sketching the circles to automatically generate these form copies, and then modifying the `fillForeColor` cell on each to Black.

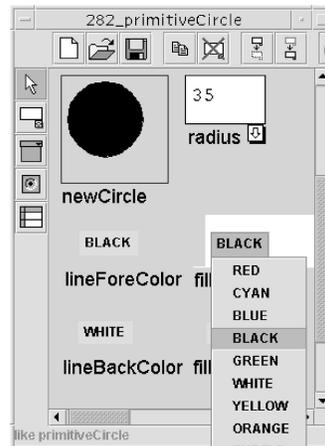
As Figure 2 shows, in Forms/3 it is possible to have multiple copies of a spreadsheet working together (an extension of the idea of the "linked spreadsheets" common in other spreadsheet languages). We will refer to an original as the *model* form and to the copies as the *instances*. The `primitiveCircle` form in Figure 1 is an example of a model form, and the form in Figure 2(c) is an instance of it. We will also use this terminology at the granularity of cells: cells on the model are termed *model cells*, and cells on instances are *instance cells*.



(a)

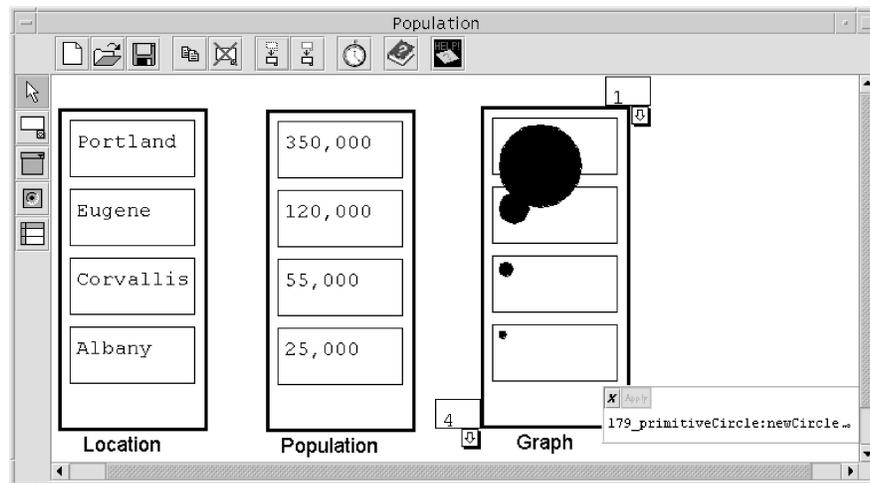


(b)

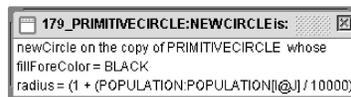


(c)

Fig. 2. (a) A form being developed to visualize population data. The formula shown at the right of the  $4 \times 1$  grid labeled graph is shared by all its cells. (The black circles in the formulas are drawings of the cells' current values, which can optionally be displayed in formulas.) (b) To define the formula for city, the population analyst first drew a circle in city's formula/gesture window. The analyst then pointed at the circle to display its defining form, a portion of which is shown in (c). Copied forms, such as this one, are gray to indicate that they are copies. However, white cells on such forms, such as radius and fillForeColor, are no longer copies—they have been edited with formulas different from the original. The population analyst specified the fillForeColor formula to be "BLACK" via the popup menu.



(a)



(b)

Fig. 3. (a) A more general version of the population program. Generalization has occurred, as is clear from the fact that there are different circles in graph's different cells. However, the concrete formula—i.e., the way the user programmed it—is the one shown. (b) The legend showing what 179\_primitiveCircle stands for.

In Figure 2, all of the cells in the graph grid officially share the displayed formula. Because of this, generalization is not needed for correctness of this application, although using a generalized notation saves storage space and time, as will be seen later.

Now suppose that, instead of referencing only the forms he or she manually generated while programming the city, town, and village cells, the population analyst would like for the circles to more closely reflect population differences, by defining each circle's radius to be a fraction of the corresponding population, such as in Figure 3. This is where the need for generalization becomes critical.

To create this more general version shown in Figure 3, the population analyst again sketches a circle, which creates another instance of the system-provided primitiveCircle form (e.g., 179\_primitiveCircle), and then edits the instance's cell fillForeColor to Black and its cell radius to be “(1 + (population:population[i@j]/10000))” to indicate a generic reference to an element of the population grid. The system needs to produce immediate feedback, so it computes and displays a sample result cell radius using population[1@1], and recomputes/redispays cell newCircle on that copy with the new radius. The analyst then refers in graph's shared formula to newCircle.

The analyst's task is finished, but the system still needs to generalize further. If it did not generalize, all the cells in the graph grid would be the same

size, because they would all refer to `newCircle` on *the same* copy, such as `179_primitiveCircle`. After the system generalizes, using the method described later in the paper, each reference in `graph`'s formula will be to cell `newCircle` on *an appropriate* copy of `primitiveCircle`.

After generalization, overly specific instances in the formula, such as `179_primitiveCircle`, are treated as samples. The fact that they are now just samples is communicated to users by the tiny key icon next to them. If the user mouses over or clicks on the key icon, a legend is displayed, which can be transient or a subwindow as in Figure 3(b), to show what the sample stands for. Figure 3(b) says that the sample reference in the formula for the `Graph[i@j]` cells is to `newCircle` on a copy of `primitiveCircle` whose `fillForeColor` cell is `Black` and whose `radius` cell is calculated using `population:population[i@j]`.

This generalized formula is stored internally by the system in a notation both more terse and slightly more general than the user-viewable legend; this internal notation will be presented in detail later. But we use this example to point out two characteristics that are evident even in the legend viewable by the user: the generalized formula includes the name of the model form (namely, `primitiveCircle`) that was used to create the instance, and all the relevant cell relationships in which the instance differs from the model (namely, `fillForeColor` is `Black`, and `radius` is `population:Graph[i@j]`). These characteristics are needed so that the formula alone can provide enough information to enable the system (1) to find a form instance consistent with the generalized formula if such an instance exists, and (2) to create any needed form instance from its model form “just in time” at runtime, if such an instance does not already exist. Because of capability (1), if the user changes Albany's population to 55,000, the system can look up a copy of `primitiveCircle` whose `radius` is 55,000 and whose `fillForeColor` is `Black`, which will allow it to reuse the Corvallis version for Albany. (This capability, a variant of lazy memoization [Hughes 1985], is important for efficiency, but is not necessary for correctness.) Because of capability (2), if the user makes the grids bigger and adds a new city with a population of 40,000, the system will be able to construct a copy of `primitiveCircle` with the appropriate cell formulas, even though the user never manually created this copy. This capability is necessary for correctness.

### 3.2 Generalization Example: Recursive Fibonacci (User's View)

Since `Forms/3` is a gentle slope language, it supports sophisticated users with programming background who may wish to employ advanced techniques at the upper end of the slope, such as recursion. Figure 4 shows a recursive solution to the classic problem of computing the *N*th Fibonacci number. `Fibonacci` is a “toy” problem, but it is large enough to demonstrate generalization without being too large to walk through the method in detail.

Recursion is supported by copying a model form to additional instances whose cells can be referred to in formulas. Thus, the `Fibonacci` program involves three forms: a model to compute the Fibonacci number *N*, and two instances that calculate the previous two Fibonacci numbers. The prototypical value “5” has been given as the formula for cell *N* on (model) form `Fib` to allow concrete feedback.

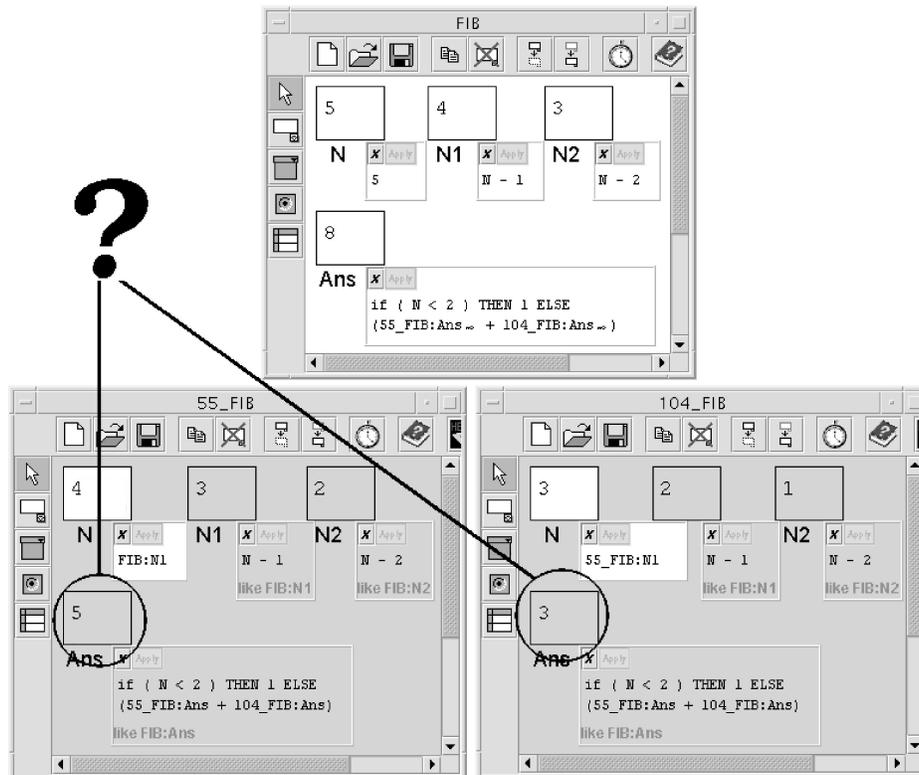


Fig. 4. One way to program Fibonacci in this language. This version includes nontraditional relationships that suggest it might have been programmed by a novice. For example, `104_Fib:N` refers to `55_Fib:N1` instead of following a more traditional hierarchy by referring to `Fib:N2`. (See Figure D-1 in Appendix D for the hierarchical version.) The concrete formulas are shown. The superimposed question mark points to values that cannot be calculated from such concrete formulas—they must be generalized first.

During program creation, the user created `Fib`, and then copied it to create forms `55_Fib` and `104_Fib` (instances of `Fib`). Instances inherit their model's cells and formulas unless the user explicitly edits the formula for an instance cell. Subsequent formula changes to the model are propagated to unedited formulas on the instances.

To express the computation for the Fibonacci program in the figure, the user entered the formulas in the cells that are white (white indicates that they have been edited). To produce immediate visual feedback, the system must display the resulting value as soon as a formula is entered. Generalization becomes necessary when the system needs to display the answer (8) as soon as the formula for `Fib's Ans` is entered.

The need for generalization at this point lies in concreteness. As entered by the user, part of the formula for `Fib's Ans` is the sum of the `Ans` cells on `55_Fib` and `104_Fib`. This is too concrete—without generalization, all future copies of `Fib`, regardless of how their `N` cells are changed, will sum the specific `Ans` cells on `55_Fib` and `104_Fib`. In fact, without generalization, `55_Fib's Ans` formula (which

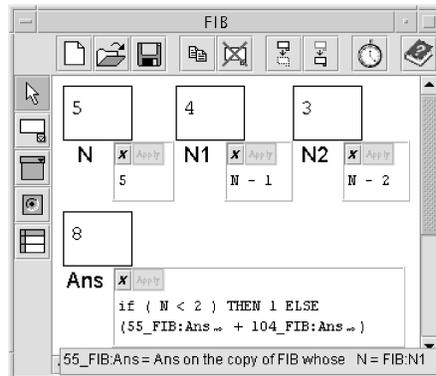


Fig. 5. The Fibonacci program after generalization. The user has moused over the leftmost tiny key icon in Ans's formula to view the legend (at the bottom) describing what 55\_Fib:Ans represents.

was inherited from Fib) is circular, because it refers to 55\_Fib's Ans (itself). This is also the case for 104\_Fib's Ans.

What the system needs to do is to analyze the program to find the relevant parts and deduce from them a generalized version. The details of how the generalized version is deduced are given in Section 4. After performing the generalization method, the model form appears as in Figure 5. As in the population example in Section 3.1, the tiny key icons to the right of the cell references in Ans's formula indicate that these concrete names are just samples representing generalized references. The legend, which is displayed at the bottom during a user mouse-over, says that the 55\_Fib:Ans reference stands for any Fib instance with the stated logical relationship to Ans through N1. Note that, as in the population example, the generalized formula is general enough to allow the system to find a suitable copy if one exists, or to create a new copy from the model if one does not already exist.

The Fibonacci example helps to demonstrate the effects of the orderlessness and modelessness constraints on the solution space of possible generalization mechanisms. If there are no pregeneralization and postgeneralization "modes," and if the user can enter formulas in any order, then the system cannot glean information based upon the order the user enters the formulas. For example, the user might enter the Ans formula first, thus referring to the return value Ans from 55\_Fib before providing information that there will be an "input" cell N. Because no information about the N parameter-like cell is yet known, the reference to 55\_Fib's Ans at this point appears to be an absolute reference to a fixed cell (analogous to a reference to a global variable or constant in a traditional language), whereas instead it will be the result of a parameterized subroutine-like call when the user eventually enters the other formulas. For the user, this means freedom to work on the problem in any order that seems natural, but to the system it means lack of information.

The generality constraint's effects are also illustrated by the Fibonacci example. This program does not follow the traditional call-return pattern, because 104\_Fib's N is not defined in terms of Fib's N2, but rather in terms of 55\_Fib's N1.

Translated to traditional terms, `Fib` does not “call” `104_Fib`, yet it refers directly to `104_Fib`’s “return value.” Such patterns of references are not unusual in spreadsheet languages, and they make a program’s structure hard to predict because it will not always fall into traditional patterns.

#### 4. BEHIND THE SCENES: THE GENERALIZATION ALGORITHMS

In Section 3, we showed how the generalization method looks to the user. Now we move behind the scenes to show the system’s view of generalization. In this section we describe generalization of single cells. Section 5 then describes the extensions that handle more complex structures, such as grids.

##### 4.1 Overview

The method consists of two steps: (1) incrementally tracking logical relationships and (2) lazily generalizing these relationships “just in time.” In the Fibonacci example, Step 1 consists of incrementally building a graph of the relationships among the `N`, `N1`, `N2`, and `Ans` cells on `Fib`, `55_Fib`, and `104_Fib`. This step is triggered whenever a user action establishes or changes a relationship. When the user enters the formula for `Ans`, Step 2 is triggered because of the apparent circularity `Ans` introduces. Step 2 focuses on only those relationships necessary to compute `Ans`. Sometimes such relationships are complex and nested, so the system completes its work in a bottom-up fashion. Starting with some overly concrete cell reference, the system deduces a generalized description from the graph of relationships it built during Step 1, and substitutes via algebraic back-substitution the generalized description into all relevant formulas that use the concrete reference. Generalization is complete when all the concrete instances involved in the computation being generalized (`Ans`, in this example) have finally been eliminated through these substitutions.

Driving the generalization method at the top level and deciding when it is time to take one of these steps is an event loop that watches for events relevant to generalization and calls the appropriate algorithms. See Figure 6. As is evident in the figure, the method is lazy and incremental.

As Figure 6 also shows, the method is subtractive. Most approaches to program generalization view the problem as being to *generate* missing information. However, our approach views the problem as being the presence of too much information, and *removes* extraneous, overly concrete information in order to arrive at the generalized version. (This view is shared by *ToonTalk* [Kahn 1996], although the way generalization is actually accomplished is quite different.) For example, in the population program, the user explicitly stated the relationship (via cell references) between a representative population grid cell, a radius cell on an instance of `primitiveCircle`, and the circle to be displayed for the corresponding graph grid cell, so the system does not need to use heuristics to infer which cells are supposed to be related. The problem the system does have to solve is that the user stated the relationship using a concrete instance of `primitiveCircle`, which was overly concrete and needs to be eliminated from the program somehow (this will occur during Step 2). Subtracting rather than

```

Algorithm EventLoop
loop forever
  sleep until next event;
  if (event.type = "formula edited" or
      event.type = "cell moved onto screen") then
    //Step 1: Incrementally add to relationship information:
    for each on-screen reference in event.cell's formula:
      Insert (label, (event.cell, reference));
    for each on-screen reference deleted from
      event.cell's formula:
      Delete (label, (event.cell, reference));
    if Cycle? (event.cell) then
      event.type = "possible cycle";

    //Step 2: If can't delay it any longer, generalize:
    if NeedToGeneralize?(event.type) then
      RemoveICE;           //Subtracts out overly concrete information.
      Generalize;
  end loop;

```

Fig. 6. The top level event loop. “Label” is an edge label in the graph modeling cell relationships, as will be explained in Section 4.2.

adding is part of what avoids the use of inference, because there is no need to conjure up missing information.

The other part of what avoids inference is the fact that, in the spreadsheet paradigm, the user explicitly states the exact nature of the relationships (via the operators in the formulas), so these do not have to be inferred. For example, in the Fibonacci program, the user explicitly stated that the way the two instances of Ans need to be combined to compute the value of the third Ans is by adding them (via the + operator).

The method requires three properties of a language: first, the user must be required to be able to see a cell in order to edit its formula or to refer to it. This property is common in languages in which programming is done by direct manipulation. Second, the relationships (operators to use in combining related cells) must be given explicitly in the formulas, as is usual for spreadsheet languages. Third, the system must be able to statically enumerate the concrete cells directly affecting a cell, simply by perusing that cell’s formula. The latter prerequisite means that the method need not use dynamic information such as the current values of cells. It also makes the method incompatible with pointers and with nonformula programming devices such as state-modifying external macros.

#### 4.2 The Cell Reference Graph

The entity that ties Steps 1 and 2 together is the *cell reference graph*. As the user enters formulas textually or via sketches and direct manipulation, the system incrementally tracks the cell reference relationships in the cell reference graph. The cell reference graph is a model of relationships among on-screen cells, along with derived generalization-oriented information about them. An important aspect of the cell reference graph is the fact that its components are limited to only on-screen cells: this is a key to the scalability of the method.

The cell reference graph  $CG = (V, LE)$  is a directed multigraph of vertices  $V$  and labeled edges  $LE$ , where

$$\begin{aligned} V &= \{v \mid v \text{ is an on-screen cell in the program}\}, \\ LE &= \{\langle label(e), e \rangle \mid label(e) \in L \text{ and } e \in E\}, \\ E &= \{(u, v) \mid \text{cell } u \text{ is referred to in } v\text{'s formula, where } u, v \in V\}, \\ L &= \{dg, dc, ig, ic\}. \end{aligned}$$

Although we refer to them as sets,  $LE$  and  $E$  are actually bags (sets allowing duplicates), as are the subsets (subbags) of  $LE$  and  $E$ .

An edge  $(u, v)$  in  $E$  is said to be *direct* if the user entered the reference to  $u$  in  $v$ 's formula, whether by pointing at  $u$ , by typing its name explicitly into  $v$ 's formula, or by sketching/manipulating to generate a formula that explicitly references  $u$ . Edges that are not direct are *inherited*, and come about when the user copies a form, thereby causing cells on copied forms to inherit the same formulas as those on the original form. The terms *concrete* and *generalized* are used to describe a relationship's current generalization state. Each relationship is modeled by an edge: an edge is said to be *concrete* if it has not yet been generalized; otherwise it is *generalized*. Because edges model cell relationships, we also map our terminology of edges to cell references and to the cells (nodes) in those relationships. For example, given a concrete edge  $(u, v)$ , which models a reference  $u$  in  $v$ 's formula, we will refer to  $u$  as a *concrete reference*, and if  $v$  has concrete references in its formula then  $v$  is a *concrete cell* (node). We will employ this terminology mapping with the other terms describing edges as well: if  $(u, v)$  is a direct, generalized, or inherited edge, then  $u$  will be said to be a *direct*, *generalized*, or *inherited reference*, respectively, and so on.

Using the two pairs of opposing terms *direct/inherited* and *generalized/concrete*,  $E$  can be subdivided into four subsets: direct concrete edges (DCE), direct generalized edges (DGE), inherited concrete edges (ICE), and inherited generalized edges (IGE). Thus, DCE, DGE, ICE, and IGE are disjoint, and their union is  $E$ .  $Label(e) \in L$  for each labeled edge in  $LE$  names the subset to which each edge  $e$  belongs. The value of  $label(e)$  is *dg* if  $e \in DGE$ , *dc* if  $e \in DCE$ , *ig* if  $e \in IGE$ , and *ic* if  $e \in ICE$ . Figures 7 and 8 show the nodes and the direct edges (those in DGE and DCE) for the Fibonacci example. Additional diagrams of labeled edges in cell relationship graphs will be shown in later subsections.

Since  $CG$  nodes are cells,  $CG$  node information can be embedded in or linked together with whatever data structure is already employed by the spreadsheet language for quick retrieval of cells. In our implementation, this is a hash table, where the hash key is the cell's ID and the hash value is the cell.

In general, the following information is required for each node (cell) for generalization purposes:

- general information about the cell, including (but not limited to) *cellID*, *formula*, and whether the cell is a model or an instance;
- *ModelCell*: the model cell from which this cell was copied. (If this cell is itself a model cell, refers to itself);
- *Generalized?*: whether or not this cell's formula is already generalized;

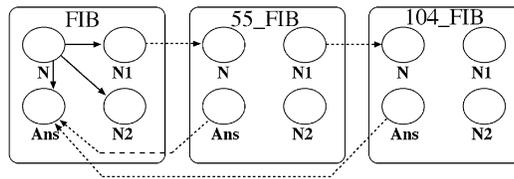


Fig. 7. Fibonacci's cell reference graph showing only direct edges. The direction of the edge indicates the direction of dataflow. For instance, the edge from N to N1 in FIB indicates that N1 contains a reference to N, i.e., the value of N flows into N1. Figure 8 explains the edge patterns.

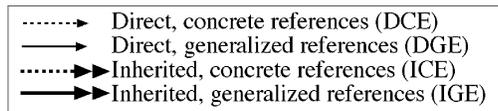


Fig. 8. Edge patterns for cell reference graph diagrams. Concrete edges are dashed lines; generalized edges are solid. Direct edges have one arrowhead; inherited edges have two.

— *Edges*: adjacency list representing labeled edges  $(label, (u, v))$  in LE from on-screen cells  $u$  to  $v$ , where cell  $u$  is referred to in  $v$ 's formula. *Edges* is stored for cell  $v$ ; hence, it is a list of in-edges, which is more efficient than out-edges for generalization processing. Each edge also includes a *counter* as an efficiency practicality to avoid separately storing duplicate edges to the same node.

Note that CG nodes represent “on-screen” cells. Since the size of this graph is what ultimately determines the cost of generalization, the fact that only on-screen cells are included is important to the scalability of the method. In the Forms/3 implementation of the method, on-screen forms are forms that are at least partially visible. However, the method is not sensitive to the particular definition of “on-screen” used, because changes in status between off-screen and on-screen trigger generalization updates, regardless of the particular definition used to define these statuses. The restriction in the cell reference graph to on-screen elements of the program relies upon the prerequisite that a cell must be on the screen for the user to directly edit its formula or to directly reference it. As long as this property is carefully maintained, and if a cell is generalized when it is moved off the screen, then it cannot require further generalization as long as it remains off-screen.

#### 4.3 Step 1: Incrementally Processing Relationships Among On-Screen Cells

4.3.1 *Incremental Processing of the User's Actions.* The cell reference graph is built incrementally. A node is added when the user creates a new cell or displays one that was previously off-screen. A node is deleted if the user removes its cell from the screen; deletion of these nodes triggers formula generalization if the remaining nodes' formulas have dependencies on the node being deleted. Edges are added if the user edits a formula to include cell references that were not in the formula before. Conversely, edges are deleted if their corresponding cell references are removed from a formula.

Edge maintenance is done using common algorithms for graph edge maintenance we'll refer to as Insert, Delete, and Find; these simple algorithms can

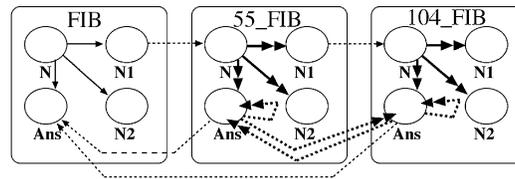


Fig. 9. The cell reference graph of Figure 7, with all edges shown.

be seen in Appendix A. Insert and Delete call Find if their callers have not provided necessary location information. Node (cell) maintenance and lookup are not explicitly described here, since they are already part of any spreadsheet language, regardless of whether there is a generalization facility. As each edge is added to the cell reference graph, it is labeled. The most common case of adding edges is when the user edits a formula. The other case is when a cell is moved onto the screen. Both cases are handled the same, so we can describe the method's operations concretely as having been triggered by a formula edit without loss of generality.

When the user first enters a formula, all its edges are direct, as was depicted in Figure 7. Thus, each edge representing the formula references must either be in DGE or DCE, depending upon whether or not the edge is already general enough. First, consider an edge  $(u, v)$  in  $E$  such that cells  $u$  and  $v$  are on the same form, as with the edge (Fib:N, Fib:N1) in Figure 7. The relationship about which copy of the Fib form is needed for N1's reference to N is simple: the same copy for N as for N1. As this illustrates, edges among cells on the same form do not need further processing to become generalized, and the edge is an element of DGE. Edges that span multiple forms, such as (Fib:N1, 55\_Fib:N), may need further processing, and they are elements of DCE.

When the user makes a new instance of a form, IGE and ICE edges are added to the cell reference graph. Each in-edge to a model cell will be copied to a new in-edge to the instance cell. Since DGE edges to the model are fully generalized, later copies of these edges are also, by definition, fully generalized, and are members of IGE. Inherited edges that are copies of those in DCE have not yet been generalized, and are classified as ICE edges. See Figure 9.

At this point, the cell reference graph has some similarities to a dataflow graph, but it contains anomalies due to concreteness. In Figure 9, for example, the edges leading into and out of Ans in 55\_Fib and 104\_Fib are too concrete, because instead of the circular references to themselves and each other, the Ans cells on 55\_Fib and 104\_Fib's Ans should reflect the general roles that they have in computing the  $N - 1$ st and  $N - 2$ nd numbers in the sequence. This type of anomaly is one of the generalization triggers.

*Costs:* The costs of the edge maintenance algorithms depend on the length of the adjacency list, which is bounded by the length of the cell's formula, which is in turn constant bounded in most spreadsheet languages. Thus, the worst-case cost of Find is the cost of finding the edge list and then traversing it, totaling  $O(|V|)$  assuming a hash-table-based implementation and doubly linked edge lists. Since there are cases in which

```

//Detects possible cycles
Algorithm Cycle? (aCell)
  initialize marks;
  depth-first search aCell.edges, marking to
    prevent duplicate subtraversals;
  if cycle detected by attempting to re-visit aCell
    then return true
  else return false;

```

Fig. 10. Algorithm `Cycle?` is called to detect the presence of possible cycles, which if found will trigger generalization. Using a marking approach, such as by marking with the system clock value at the time the search is invoked, avoids the need to reinitialize the data structure at function invocation time.

Insert and Delete call Find, the costs of Insert, Delete, and Find are all  $O(|V|)$ .

**4.3.2 Cycle Detection to Trigger Formula Generalization.** Generalization is lazy, being performed only when necessary and only on the cells requiring it. Concrete circular dependencies can be generated by copying, as just illustrated, which makes immediate visual feedback impossible until the relationships behind the concrete forms are analyzed to generalize the concrete formulas. Thus, one case that triggers generalization is entry of a formula that causes a “possible cycle” in the cell reference graph.

Whenever edges are added to the cell reference graph, the graph is analyzed to find out if a cycle has been formed. If a cycle exists that includes at least one edge in ICE, i.e., an inherited concrete edge, generalization might be able to remove the cycle. We will refer to such cycles as *possible cycles*. The presence of a possible cycle constitutes an emergency: generalization must occur right away to try to remove the cycle, because until the cycle is removed (or other corrective action is taken), incorrect or nonterminating calculations may be generated.

Possible cycles are detected via Figure 10’s `Cycle?`, which uses a depth-first search on the on-screen cells. Figure 9 includes examples of possible cycles involving cell `Ans` on `55_Fib` and `104_Fib`. Searching only on-screen cells does not detect all cycles in the program, but it does detect all cycles involving ICE edges, which can only exist among on-screen cells.

If, on the other hand, the user has entered a *true cycle*—i.e., if every edge in the cycle is either a direct edge or has already been generalized—a language-specific response occurs. Although some spreadsheet languages allow true cycles and some do not, all take some kind of special action if a true cycle occurs. Such actions include rejecting the most recent formula entry as in `Forms/3`, or asking the user to specify some maximum number of iterations as in `Excel`.

*Costs:* `Cycle?` costs only  $O(|E|)$  because it does not necessarily search all vertices—only those in the connected subgraph (paths of edges) rooted at the cell just edited. (Recall that  $E$  is the set of edges in CG, which means  $|E|$  is the number of direct and inherited references among on-screen cells.) Note that this cost is overshadowed by another task that takes place after a formula edit, even without a generalization subsystem: spreadsheet systems must do some form of cycle detection on all cell references involved, regardless of whether

they are on the screen, as explained in the preceding paragraph. These costs are not part of the cost of generalization, because they are necessary even in systems that do not support generalization.

**4.3.3 Other Triggers.** The top-level algorithm `EventLoop` refers to an  $O(1)$  predicate, `NeedToGeneralize?(event)`, which decides when formula generalization cannot be delayed any longer. Detecting cycles in the cell reference graph as described above is one event for which `NeedToGeneralize?` returns true, triggering formula generalization.

The other events are:

- (1) *Saving a model form.* Any generalized formula contains complete enough specifications of the logical relationships to automatically generate any needed form instances that do not already exist. A side benefit of this completeness of information is that the system can omit storing concrete form instances permanently, which significantly reduces space and time of the “save” operation. Thus, generalization must be triggered when saving; otherwise the formulas in the (to be saved) model form could contain references to concrete form instances that will not be saved.
- (2) *Making a new instance by copying a form.* Reusing a form in this way requires the model form to be generalized first. All the model’s instances automatically become generalized at the same time, by virtue of their “copied” formulas—which are really just pointers to the model’s now generalized formulas. If generalization did not happen at this point, the new instance could refer to an old instance that was too concrete to be appropriate for the new instance.
- (3) *Editing an instance cell that affects the generalized meaning of a previously generalized (on-screen) cell.* As will become clear from the definition of the generalized notation in Section 4.4, this means the affected cell(s) must be regenerated. This can only occur with on-screen cells, because of the next case. It is already necessary for spreadsheet evaluation engines, both lazy and eager, to do some updating of affected cells when an edit occurs [Burnett et al. 1998]; thus the only additional cost due to generalization is looking to see if those cells have been generalized.
- (4) *A form is removed from the screen.* The departing form’s cells must be generalized and their cell reference dependencies removed from the cell reference graph. This is important to the method’s scalability because it keeps costs bounded by the number of on-screen cells, not by the number of cells in the program. (If the departing form is a model, its instances must depart too, to prevent later editing of the instances in a way that changes the generalized formulas of the off-screen cells.) Also, any on-screen cells that refer to the departing form’s cells must be generalized, before the departing cells are deleted from the cell reference graph.

These triggers help the approach to maintain an important invariant: all saved forms are generalized. For this reason, and because of the algebraic

```

//Removes ICE edges from the relevant subgraph.
Algorithm RemoveICE
  for each cell that triggered generalization:
    RemoveICEEdges (cell);

RemoveICEEdges (v)
  if v has been visited then return;
  mark v visited;
  for each le ∈ v.edges:
    if le.label = "ic" then Delete(le); //removes ICE in-edges to v
    RemoveICEEdges (le.u) // and in paths leading to v.
    
```

Fig. 11. Algorithm to remove ICE edges from the relevant subgraph.

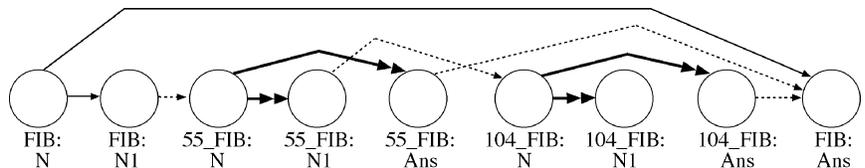


Fig. 12. SSCG rooted at cell Ans, after removal of the ICE edges.

substitution basis of Step 2, the outcome of generalization is independent of the order users edit or perform other operations that may trigger generalization—by the time a form is saved, it is guaranteed to be generalized, and the results will always be the same.

#### 4.4 Step 2: Generalizing the Formula Relationships

Once formula generalization has been triggered, a subgraph of the cell reference graph is mapped to a compact textual notation. This notation is then processed to eliminate the concrete instances from the formula being generalized. This is done using algebraic back-substitution of relationships in place of concrete instances. The result is combined with the original formula to produce the generalized version of the formula. The details of these tasks are as follows.

**4.4.1 The Subgraph of Interest.** If the generalization trigger was one of those given in Section 4.3.3, then the entire cell reference graph is of interest. However, if the trigger was a formula edit generating a possible cycle, the cell containing the newly edited formula can be viewed as the root of a smaller subgraph to be generalized, and only cells with paths to that root need be considered. We will refer to the subgraph of interest as *SSCG* (selected subgraph of CG).

Our method is subtractive: it removes edges from SSCG that are elements of ICE (inherited concrete edges), using the algorithm in Figure 11. The reason for removing ICE edges is that they are the ones that lead to anomalies such as those in the Fibonacci example. It is possible to remove them without losing important information because they are inherited: hence the same information can always be found in the original entry. In Figure 12, notice that only those cells relevant to the computation of the cell Ans are in SSCG. For example, the

N2 cells are not present because there is no path in the cell reference graph from these cells to Ans.

*Costs:* Because the size of the CG is bounded by the number of cells on the screen, the size of SSCG, the subgraph being generalized, is also bounded by this number. Since RemoveICEedges walks through the triggering cells' edge lists in depth-first order, the cost of RemoveICE is  $O(|E|)$  in the worst case. Marking is used to avoid duplicate visits. The call to Delete is  $O(1)$  in this context, because the locations of the labeled edges passed to it have already been discovered.

**4.4.2 How SSCG Is Used to Generate the Generalized Formulas.** At this point, the relationships remaining in SSCG are combined with the original formulas to generate the generalized version of the formulas. Generalized formulas are lazily demanded starting with SSCG's root(s). That is, first the root is attempted, and if its generalization requires generalization of cells on a path to it, their generalizations are then attempted, and so on.

The generalized formulas take relationships and context into account. The context of a cell reference being generalized is which particular copy of the form it resides on. To describe this, an internal notation is needed that maps the relationships modeled in SSCG back into the formulas. (This internal notation is for use by the system, not the user.) This notation is described in detail Sections 4.4.3 and 4.4.4.

As we pointed out in the examples of Section 3, two requirements for this internal notation are that it must provide enough information for the system (1) to find a form instance consistent with the generalized formula if such an instance exists, and (2) to create the needed form instance from its model form if such an instance does not exist. If these requirements were not met, the only way a form could be reused *during execution* would be for the user to manually create a new instance by copying from the model and modifying it, just as he/she did while programming it originally. Further, the modelessness constraint—maintaining the user's pregeneralization capabilities, such as editability of any (visible) formula even after generalization of its concrete references, displayability of concrete instances even after generalization replaces them, and so on—is possible largely because of these two requirements.

**4.4.3 Notation 1: A Form-Oriented Notation.** Let  $F$  be a form, let  $F_i$  be a copy of  $F$  instantiated directly by a user performing a copy action, and let  $DefSet_i$  be a set of elements of format " $Y \equiv \phi$ ," where each  $Y$  is a cell on  $F_i$  whose formula has been edited to be some arbitrary formula  $\phi$ . Thus, it is possible to abstractly specify copy  $F_i$  by enumerating how its cell relationships differ from those in  $F$  using the following notation:

$$F_i = F(DefSet_i) \quad (\text{Notation 1})$$

This will allow references to the specific copies to be eliminated from the generalized formula, because given  $F$ , the generalized description  $F(DefSet_i)$  is sufficient for the system to automatically generate copies exactly the same as  $F_i$  at future runtimes. More important, by substituting the pseudo-ID "self"

in  $F(DefSet_i)$  for references to the same form copy and back-substituting generalized descriptions of the other form copies referenced, this description is sufficient to generate additional, computationally similar, copies of  $F$  such as the additional copies of `Fib` needed to support computation of `55.Fib:Ans` and `104.Fib:Ans`.

For example, the generalized formula for `Fib:Ans` expressed using this notation is

```
if self:N<2 then 1
else generalized description of 55.Fib : Ans +
     generalized description of 104.Fib:Ans. (1)
```

where *generalized description of 55.Fib* from the perspective of `Fib:Ans` (the cell currently being generalized) is

```
Fib(N ≡ self:N1) (2)
```

and *generalized description of 104.Fib* from the perspective of `Fib:Ans` is

```
Fib(N ≡ generalized description of 55.Fib:N1),
```

which reduces, by making the algebraic back-substitutions illustrated by the arrows from boxed expressions to underlined expressions, to

```
if self : N<2 then 1
else Fib(N ≡ self : N1):Ans +
     Fib(N ≡ Fib(N ≡ self : N1) : N1) : Ans.
```

As this example shows, what Notation 1 does is to substitute every concrete reference by a description of a logical relationship. In this example, it substituted `55.Fib` by a description of its relationship to `Fib` (via `N1`), and similarly every concrete reference to `104.Fib` by a description of its relationship to `Fib` (via its relationship to `55.Fib:N1`).

Note that `self`'s meaning is context-dependent, where context is which cell's formula is being generalized. For example, if the cell being generalized is on form  $F_i$ , then `self` means  $F_i$  until generalization for that cell is entirely complete. This states relationships in a way that causes each to be executed in the appropriate context, allowing even `55.Fib:Ans` and `104.Fib:Ans` to compute using the same kind of relationships as in `Fib:Ans`. `Self`'s meaning is static, and it never varies within any one formula. For example, all three `self`s above mean the model form `Fib`, not other copies of `Fib`, because `Fib:Ans` is the cell being generalized.

4.4.4 *Notation 2: Improving Notation 1 through Better Use of Granularity and Perspective.* Because Notation 1 works at the granularity of forms, it suffices for supporting the traditional call-return hierarchical structure of one function invocation calling another, as found in traditional applicative languages, allowing even recursive programs to be programmed concretely and then generalized correctly. In fact, the above example even included a “pipeline” of values through the  $N$  cells, which the notation was also able to handle. If the

technique were intended only for programmers, this amount of support might be adequate.

However, as the user sets up spreadsheets whose cells reference one another, he or she could introduce not only hierarchical and pipelined referencing patterns, but any arbitrary noncircular cell referencing pattern. Support for these other noncircular referencing patterns seems necessary as well, since it does not seem reasonable to expect end users to structure their programs in only the ways commonly used by professional programmers.

For example, suppose the first factor in Ans's addition needed simply to be 1 more than the second factor. (To differentiate this changed version from the original example, at this point we now change the form names to NotFib.) One way to program this would be for the user to change 55\_NotFib's Ans cell's formula to

$$1 + 104\_NotFib:Ans.$$

Unfortunately, this change would prevent the formula for NotFib:Ans given in (1) above from being generalized as easily as before because, rather than (2), now the generalized description of 55\_NotFib needs to include 104\_NotFib's generalized description:

$$\begin{aligned} \text{NotFib}(N \equiv \text{self} : N1, \\ \text{Ans} \equiv 1 + \underline{\text{generalized description of } 104\_NotFib:Ans}) \end{aligned} \quad (3)$$

while 104\_NotFib's generalized description still needs to include 55\_NotFib's:

$$\text{NotFib}(N \equiv \underline{\text{generalized description of } 55\_NotFib:N1}) \quad (4)$$

One reason for the above apparent circularity comes from granularity: Notation 1 has been describing relationships at the granularity of entire forms rather than cells. The other reason is that perspective has not been fully considered—which cell relationships are actually relevant in referring to some cell  $Z$  in a cell  $X$ 's formula.

Subgraph SSCG's reduced membership to only cells relevant to the cell(s) being generalized, along with its topological ordering, provide ways to overcome both of these difficulties. Using SSCG, we change what is being described from forms (Notation 1) to cells, and make more use of perspective: only cells relevant to the cell  $X$  being described are included in  $X$ 's description. More precisely, every cell included on the left-hand side of a " $\equiv$ " in a  $DefSet_i$  in  $X$ 's formula must exist in SSCG and thus have a path to  $X$ .

If  $X$  is the cell whose formula is currently being generalized and its formula contains a concrete reference to  $F_i : Z$ , then let  $AffectsSet_x$  be  $\{Y \equiv \phi \mid \exists \text{ a path from } Y \text{ to } X \text{ in SSCG}\}$ , where  $\phi$  is any formula. The strategy is to generalize  $X$ 's reference to  $F_i : Z$  using only cells that are in  $AffectsSet_x$ , the set of  $Y$ s that directly or transitively affect  $X$ . Using this strategy, we modify the description of a generalized version of some concrete reference  $F_i : Z$  in  $X$ 's formula to be

$$F_i : Z = F(DefSet_i \cap AffectsSet_x) : Z \quad (\text{Notation 2})$$

For example, the generalized formula for NotFib:Ans expressed using Notation 2 is

```

if self:N < 2 then 1
else → generalized description of 55_NotFib:Ans +
       generalized description of 104_NotFib:Ans (1')
    
```

where the generalized description of 55\_NotFib:Ans from the perspective of NotFib:Ans (the cell currently being generalized) is

```

NotFib(N ≡ self:N1,
  Ans ≡ 1 + generalized description of 104_NotFib:Ans) : Ans (3')
    
```

and the generalized description of 104\_NotFib:Ans that pertains to both the perspective of NotFib:Ans in (1') and that of 55\_NotFib:Ans in (3') is

```

NotFib(N ≡ generalized description of 55_NotFib:N1) : Ans (4')
    
```

The use of cell granularity and perspective in (3') and (4') avoids the cycles of Notation 1's (3) and (4): 104\_NotFib : Ans is not in  $AffectsSet_{55\_NotFib:N1}$ . Thus it will not be included in Notation 2's generalized description of 55\_NotFib:N1, which becomes

```

NotFib(N ≡ self : N1) : N1 (5')
    
```

Finally, the algebraic back-substitutions illustrated above result in the following final generalized formula for NotFib:Ans:

```

if self : N < 2 then 1
else NotFib(N ≡ self : N1,
  Ans ≡ 1 +
  NotFib(N ≡ NotFib(N ≡ self : N1) : N1) : Ans) : Ans +
  NotFib(N ≡ NotFib(N ≡ self : N1) : N1) : Ans
    
```

**4.4.5 The Formula Generalization Algorithm.** Figure 13 gives the algorithm that implements this notation. It is called when the top-level algorithm's call to `NeedToGeneralize?` has returned true. The generalized formulas are recorded via calls to `GenFormula` for the cell(s) related to `u`, whose event triggered the call to `Generalize` by causing `NeedToGeneralize?` to return true. For example, if `Generalize` is triggered by detection of a possible cycle, then the cell `u` whose formula introduced the possible cycle is the only one traversed by the loop, but these calls may in turn generate calls to `GenFormula` for other cells affecting `u`. Since unmodified instance cells share formulas with model cells via pointers, an instance cell simply initiates generalization of its model version and the result is automatically shared. Instance cells that have overridden the model formulas (such as `55_NotFib:Ans`) will eventually be generalized if they are encountered in the  $DefSet \cap AffectsSet$  described in Section 4.4.4, and otherwise do not affect any model and can be ignored (since after generalization, instance forms can be discarded).

```

//Lazily records the necessary generalized formula relationships using Notation 2.
Algorithm Generalize
  for each cell u that triggered generalization:
    if u is an instance cell
      then GenFormula(u.modelCell, u.modelCell)
    else GenFormula(u,u);

//Generalizes one cell (currentCell), passing along origCell for perspective.
GenFormula(currentCell, origCell)
  if not currentCell.generalized? then
    for each ungeneralized (ref, currentCell)
      ∈ currentCell.edges:
        replace ref in currentCell.formula with
          GenRef (ref, currentCell, origCell);
    currentCell.generalized? = true;
    return currentCell.formula;

//Expresses a reference in currentCell's formula using Notation 2.
GenRef(ref, currentCell, perspective)
  case label(ref, currentCell) of
    dg or ig: return (formID(ref, perspective) ++
      ":" ++ ref.CellID);
    dc: case ref of
      model: return (formID(ref, perspective) ++
        ":" ++ ref.CellID);
      instance: return (formID(ref, perspective) ++
        "(" ++
          Def∩Affects(ref, perspective) ++
        ")" ++ ref.CellID);

//Computes the DefSet ∩ AffectsSet part of Notation 2.
Def∩Affects(ref, perspective)
  result = { };
  //This is AffectsSet:
  for each affects ∈ SSCG rooted at perspective: .
    if (affects is on the same form as ref and
      affects.formula has been edited) then //affects ∈ DefSet.
      result = result ∪
        (affects.cellID ++
          "≡" ++ GenFormula(affects, perspective));
  return result;

```

Fig. 13. Algorithm Generalize. Operator ++ denotes string concatenation.

GenFormula works its way through the in-edges modeling the on-screen references in cell `currentCell`'s formula, calling GenRef for every cell reference. GenRef translates a single reference in `currentCell`'s formula to Notation 2. The calls to formID in this function return "self" if ref and perspective are on the same form; otherwise it is the form name of ref's model form, such as NotFib in the example above. References to cells on model forms can be referred to directly, since there are no relationships needed to describe the relationship of a model form to itself. The algorithm handles instances (copies) by calling Def∩Affects. Def∩Affects performs the intersection. It is able to do this by walking through the *AffectsSet* for cell perspective, checking the form location and modification status of each cell encountered.

*Costs:* The maximum number of times through `Generalize`'s loop, which calls the other algorithms in Figure 13, is  $O(|V|)$ . `GenFormula` is bounded by the maximum length of any one formula, which is constant-bounded in most spreadsheet languages. Except for the calls to `Def∩Affects`, `GenRef` is also a constant-time function. The union in `Def∩Affects`'s last statement is really just an append, since each element is encountered uniquely. Thus, `Def∩Affects`'s worst-case cost is dominated by the number of times through the loop,  $O(|E|)$ .

Multiplying these factors together gives a total for `Generalize` of  $O(|V|*|E|)$  if triggered by a cell completing a possible cycle, or  $O(|V|^2*|E|)$  if triggered by the other events such as removing a form from the screen. A further discussion of how the cost of this algorithm combines with the other algorithms when triggered by various user actions is provided in Appendix C (available in ACM's Digital Library, <http://portal.acm.org/portal.cfm>). The most important aspect of the cost of this algorithm and the method's other algorithms lies in the fact that they all depend on  $|V|$  and  $|E|$ , which are in turn bounded by the number of cells that fit on a screen—no matter how large the rest of the program is. (Appendix C also contains a discussion of cost implications of different definitions of *on-screen*.) The method's incremental strategy that allows the method to process only on-screen cells is the essence of the method's scalability.

## 5. BEYOND SIMPLE CELLS

In this section we show how the generalization method can support more sophisticated spreadsheet entities than simple cells.

### 5.1 Grids

Arrays and matrices in traditional programming languages are ways to group data elements of similar attributes so that their elements can be processed using the same code. In a spreadsheet language, such grouping is done with grids, as in the population example shown earlier in Figure 3.

A grid has rows and columns. In `Forms/3`, grids are dynamically sized, and the number of rows and columns are determined dynamically by evaluating a distinguished cell for each, known as *size cells*. For example, Figure 3 shows the size cells for grid graph; formulas for these cells are entered in the usual way. The remaining cells in the grid (called the *element cells*) reside in regions. A *region* is a mechanism to define a shared formula for all the cells within a contiguous, rectangular group of cells<sup>3</sup>; for example, grid graph in Figure 3 consists of one region comprised of all the element cells in the grid. In `Forms/3`'s region formulas, *i* represents “my row” and *j* represents “my column.” At runtime, the evaluation engine substitutes a cell's actual row and column location for the *i* and *j* in computing that particular cell's value.

Since the region formulas make formula sharing explicit based on spatial relationships, it is not necessary to generalize further for that type of relationship. (Also, because the system does not need to adjust references (since they

<sup>3</sup>Explicit sharing of this nature is also found in some other spreadsheet languages, such as Lotus and Formulate [Ambler 1999].

are already expressed generally), incorrect guesses such as the one triggered by the row insertion in Excel described in Section 2 do not arise.) However, in order to add the generality based on logical relationships shown earlier for simple cells, it is necessary to combine the generalization method presented earlier with the spatial relationship information already present in the shared formulas.

For example, recall from Section 3 that in the population example, the user began creating the formula for grid graph's region in Figure 3 by drawing a circle then clicking on it to bring up an instance of the circle form, just as before in Figure 2. The user then brought up the instance of the circle form so as to specify two changes, changing the formula for `fillForeColor` to `BLACK` and the formula for `radius` to `population[i@j]`, which says that eventually every element of the population grid is to be referred to in this way. To produce immediate feedback upon entry of this formula, the sample display value for `radius` is the grid element at row 1, column 1, which is `population[1@1]` in this example.

The entry of this formula triggers algorithm `Generalize`, which does the work of generalizing not only `radius`'s formula, but also the region formula in grid graph because of its dataflow relationship leading to `radius`. The resulting generalized region formula for the graph grid in Notation 2 becomes

```
primitiveCircle(fillForeColor ≡ BLACK,
                radius ≡ 1 + population[i@j]/10000) : newCircle.
```

When processed by the evaluation engine, the contents of an element cell such as `graph[3@1]` will be computed as the result of cell `newCircle` in a copy of `primitiveCircle` in which `fillForeColor`'s formula is `BLACK` and `radius`'s formula refers to `1 + population[3@1]/10000`.

**5.1.1 Impacts on the Generalization Method.** To add support for grids to the generalization method, the following straightforward additions were made to the algorithms of Section 4. First, there are new, implicit dependencies introduced by grids that needed to be maintained in the cell reference graph: the dependencies among the elements of a grid region, the grid's size cells, and the grid as a whole. These dependencies are maintained in the cell reference graph automatically whenever a new formula is entered for a grid cell or region. These implicit dependencies are added to the cell reference graph as though they were explicit cell references made by the user (except that all cells in a region are represented by a single node in the graph to conserve space and time). As stated above, a new generalization trigger was added, which triggers generalization when an `i@j` grid subscript reference is entered by the user. Generalization is also triggered if a cell with an `i@j` grid subscript is present in the *AffectsSet* of a cell being edited. Finally, the front end of the system was made to use the sample display value of `[1@1]` when grid references are edited, i.e., prior to generalization. These were the only modifications needed, and they do not change the asymptotic cost of the algorithms; the cost still depends on the number of cells on the screen.

## 5.2 Animated Cell Values and the Time Dimension

Some graphical and end-user programming languages support temporal programming, the ability to explicitly define temporal relationships (e.g., McDaniel and Myers [1999]; Wolber [1997]). This is useful for graphical animations, for example. In spreadsheet languages, temporal programming and animation can be supported if formulas for cells are viewed as defining a vector of values along an explicit time dimension (a  $t$ -axis), rather than just an atomic value.

In Forms/3's support of animation [Burnett et al. 2000], formulas can reference cells' values at earlier moments in time. For example, on the circle form of Figure 1, cell *radius*'s formula could be changed so that, after an initial value at time 1, it refers to its own earlier value as in "radius<t-1>+1" (adds 1 to the value of *radius* at  $t$ -position "now - 1"), which would cause *newCircle* to expand in size over time. In general, when cell *A*'s formula references cell *B*'s earlier value in time, where *A* and *B* are not necessarily distinct, a special temporal label is attached to the edge connecting *B* to *A* in the cell reference graph. The temporal label is needed to distinguish formulas referencing earlier values from truly circular references in the cycle detection routine. Without the temporal flag set, this apparent self-reference would have been falsely detected as a cycle. The temporal label is only used in the cycle detection routine. It does not otherwise change the generalization algorithms, and does not affect costs.

## 5.3 User-Defined Data Types

Forms/3 supports user-defined abstract data types. The basic approach to data abstraction in Forms/3 is described elsewhere [Burnett and Gottfried 1998; Burnett and Ambler 1994]. To briefly summarize the aspects of it relevant to generalization, *type definition forms* are used to define new types. This is another extension to the idea of linked spreadsheets. Type definition forms are similar to ordinary forms, but also contain a special type of cell called an *abstraction box*. An abstraction box defines the composition of the type, and its value is an instance of the type.

For example, Figure 14 defines a data type *Mortality*. Using an abstraction box, the user has specified that the value of *Mortality*, an instance of this new type, consists of the *Age*, *Ht*, and *Weight* cells' values. Because of this information, the abstraction box itself does not need a formula in this program.

In order for the generalization method to support user-defined data types, the implicit dependencies between an abstraction box and its interior cells are added to the cell reference graph as though they were explicit cell references made by the user. This was the only modification needed to extend generalization to support user-defined data types. This modification does not change the asymptotic cost of the algorithms.

## 6. PERFORMANCE DATA

To demonstrate scalability in a real implementation, we gathered performance data for the following scenario: A user incrementally creates a program that contains four groups of linked spreadsheets that require generalization. After

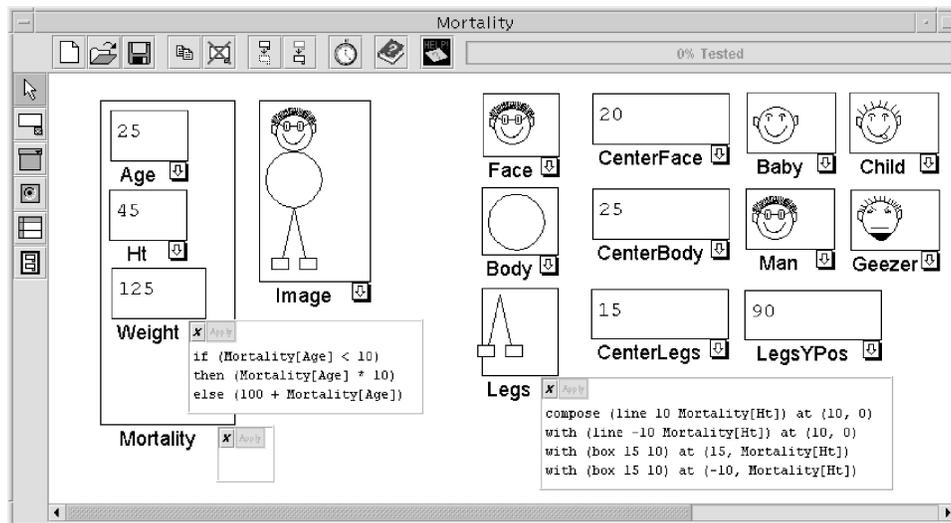


Fig. 14. A user-defined abstract data type.

he or she is finished programming and debugging each group—which triggers generalization several times per group—the user moves it off-screen and begins another group. Since screen size is limited, spreadsheet users are essentially required to follow this kind of development paradigm.

For this experiment, we chose two spreadsheet groups that employ the basic method as described in Sections 3 and 4, and two that employ extensions described in Section 5. The first two are computation-oriented programs: Factorial (given in Appendix D's Figure D-2, available in ACM's Digital Library, <http://portal.acm.org/portal.cfm>) and the version of Fibonacci (given in Appendix D's Figure D-1, also available at ACM's Digital Library). The third, Stock With Colors, is a business graphics program that makes use of the temporal extensions of Section 5, generating an animated bar chart of stock prices that are updated over time. It is given in Appendix D's Figure D-3. The fourth, Population, is the visualization-oriented example of Figure 3, which employs the grid extensions described in Section 5.

We did all timings in a “lower-end” environment that does not have particularly optimal hardware or software. We used a single-user Sun Ultra-1 (143 Mhz CPU) under Solaris 8, using Lucid Common Lisp V5.0. Dynamic garbage collection was disabled to isolate its time, most of which is unrelated to the cost of the generalization algorithm. The combination of a relatively slow CPU and the Lisp computing environment falls at the lower end of the performance options available to today's users, and is an acid test of the viability of our method in real-world computing environments.

Table I displays the performance data for each spreadsheet group, broken out for user actions that triggered formula generalization. The second column shows the number of entries in the cell reference graph (CG) at the time generalization was triggered, as compared with the running total of the number of cells in the program in the third column. These columns' data reflect the

Table I. Generalization Response Time Costs for Actions that Triggered Generalization While Incrementally Creating a Program Consisting of 727 Cells (Average of 10 Runs in a Lower-End Computing Environment)

User Actions	Entries in CG	(Running) Total Cells in Program	Generalization Time (seconds)
Factorial:			
Copy	22	24	0.010
Edit	24	45	0.029
Off-screen	24	45	0.040
Fibonacci:			
Copy	25	53	0.008
Copy	28	57	0.030
Edit	30	161	0.021
Off-screen	30	161	0.061
Stock with Colors:			
Copy	89	283	0.053
Copy	120	318	0.099
Copy	151	353	0.140
Off-screen	61	356	0.114
Population:			
Copy	129	448	0.049
Edit	131	727	0.072
Off-screen	131	727	0.298

fact that the number of cells in the cell reference graph is bounded by screen size, even as the total number of cells in the program grows. The generalization times (fourth column) are the amount that the generalization method added to response time costs. Standard deviations were very low, and thus the means reported are representative of the raw times. As this data shows, in most cases response time was affected by the generalization algorithm by less than a tenth of a second.

## 7. CONCLUSION

The method presented in this paper allows a general spreadsheet program to be derived from one 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.

There are two primary contributions of this method:

- It deductively reasons without the user having to provide generalization-oriented information. Other generalization methods, whether deductive or inferential, have required user assistance either before generalization (such as to identify the differences between sample values and constants) or after generalization (to correct faulty inferences).
- It is the first generalization method to emphasize scalability. The costs of its generalization algorithms are bounded by the number of cells currently on the screen, not by the size of the program.

The combination of these two contributions with the design characteristics of orderlessness, modelessness, and generality leads to additional advantages. The combination of orderlessness and modelessness removes order requirements from the user's programming process. The method is general in that all noncircular referencing patterns are supported, including those not commonly found in traditional programming languages. These characteristics allow the generalization method presented here to support a graphical style of spreadsheet programming that incorporates extensive use of concrete examples and direct manipulation, maintaining—even for large programs—the immediate visual feedback expected of spreadsheet environments.

#### ACKNOWLEDGMENTS

We thank the anonymous reviewers and the editor for their constructive feedback, which was very helpful in improving the quality of this paper. The population, mortality, and stocks examples were written by Herkimer Gottfried, Shikha Gottfried, and Miguel Arredondo-Castro.

#### REFERENCES

- AMBLER, A. 1999. The Formulate visual programming language. *Dr. Dobb's Journal* 24, 8, 21–28.
- AMBLER, A. AND HSIA, Y. 1993. Generalizing selection in by-demonstration programming. *J. Vis. Lang. Comput.* 4, 3, 283–300.
- BURNETT, M. AND AMBLER, A. 1994. Interactive visual data abstraction in a declarative visual programming language. *J. Vis. Lang. Comput.* 5, 1, 29–60.
- BURNETT, M., ATWOOD, J., AND WELCH, Z. 1998. Implementing level 4 liveness in declarative visual programming languages. In *Proceedings of the IEEE Symposium on Visual Languages* (Halifax, N. S., Canada, Sept. 1–4, 1998). IEEE Computer Society Press, Los Alamitos, Calif., 126–133.
- BURNETT, M., ATWOOD, J., DJANG, R., GOTTFRIED, H., REICHWEIN, J., AND YANG, S. 2001a. Forms/3: a first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Funct. Program.* 11, 2, 155–206.
- BURNETT, M., CAO, N., AND ATWOOD, J. 2000. Time in grid-oriented VPLs: just another dimension? In *Proceedings of the IEEE Symposium on Visual Languages* (Seattle, Wash., Sept. 10–13, 2000). IEEE Computer Society Press, Los Alamitos, Calif., 137–144.
- BURNETT, M. AND GOTTFRIED, H. 1998. Graphical definitions: expanding spreadsheet languages through direct manipulation and gestures. *ACM Trans. Comput.-Hum. Interact.* 5, 1, 1–33.
- BURNETT, M., YANG, S., AND SUMMET, J. 2001b. *A Scalable Method for Deductive Generalization in the Spreadsheet Paradigm*. Technical Rep. TR 01-60-02, Oregon State University, Corvallis, Ore; <http://www.cs.orst.edu/~burnett/reprints.html>.
- CHI, E., RIEDL, J., BARRY, P., AND KONSTAN, J. 1998. Principles for information visualization spreadsheets. *IEEE Comput. Graph. Appl.* 18, 4, 30–38.
- CYPHER, A. AND SMITH, D. 1995. KidSim: end user programming of simulations. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (Denver, Colo., May 7–11, 1995). ACM Press, New York, 27–34.
- FRANK, M. AND FOLEY, J. 1994. A pure reasoning engine for programming by demonstration. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (Marina del Rey, Calif., Nov. 2–4, 1994). ACM Press, New York, N.Y., 95–101.
- HEGER, N., CYPHER, A., AND SMITH, D. 1998. Cocoa at the Visual Programming Challenge 1997. *J. Vis. Lang. Comput.* 9, 2, 151–169.
- HUGHES, J. 1985. Lazy memo-functions. In *Proceedings of Functional Programming and Computer Architecture* (Nancy, France, Sept. 16–19, 1985). J. Jouannaud, Ed. Lecture Notes in Computer Science, vol. 201. Springer-Verlag, Bielefeld, Germany, 129–146.

- KAHN, K. 1996. ToonTalk—An animated programming environment for children. *J. Vis. Lang. Comput.* 7, 2, 197–217.
- KURLANDER, D. AND FEINER, S. 1992. A history-based macro by example system. In *Proceedings of the ACM the Symposium on User Interface Software and Technology* (Monterey, Calif., Nov. 15–18, 1992). ACM Press, New York, N.Y., 99–106.
- LEOPOLD, J. AND AMBLER, A. 1997. Keyboardless visual programming using voice, handwriting, and gesture. In *Proceedings of the IEEE Symposium on Visual Languages* (Capri, Italy, Sept. 23–26, 1997). IEEE Computer Society Press, Los Alamitos, Calif., 28–35.
- LIEBERMAN, H. 1993. Tinker: a programming by demonstration system for beginning programmers. In *Watch What I Do: Programming by Demonstration*, A. Cypher, Ed. MIT Press, Cambridge, Mass., 48–64.
- MCDANIEL, R. AND MYERS, B. 1999. Getting more out of programming by demonstration. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, Pittsburgh, Penn., May 15–20, 1999). ACM Press, New York, N.Y., 442–449.
- MYERS, B. 1991. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (New Orleans, La., April 28–May 2, 1991). ACM Press, New York, N.Y., 243–249.
- MYERS, B. 1993. Peridot: creating user interfaces by demonstration. In *Watch What I Do: Programming by Demonstration*, A. Cypher, Ed. MIT Press, Cambridge, Mass., 124–153.
- MYERS, B. 1998. Scripting graphical applications by demonstration. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (Los Angeles, Calif., April 18–23, 1998). ACM Press, New York, N.Y., 534–541.
- MYERS, B., HUDSON, S., AND PAUSCH, R. 2000. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.* 7, 1, 3–28.
- MYERS, B. AND MAULSBY, D. 1993. Glossary. In *Watch What I Do: Programming by Demonstration*, A. Cypher, Ed. MIT Press, Cambridge, Mass., 593–603.
- MYERS, B., SMITH, D., AND HORN, B. 1992. Report of the ‘End-User Programming’ Working Group. In *Languages for Developing User Interfaces*, B. Myers, Ed. A. K. Peters, Ltd., Wellesley, Mass., 343–366.
- OLSEN, JR., D. 1996. Inductive groups. In *Proceedings of the ACM Symposium on User Interface Software and Technology* (Seattle, Wash., Nov. 6–8, 1996). ACM Press, New York, N.Y., 193–199.
- PERRONE, C. AND REPENNING, A. 1998. Graphical rewrite rule analogies: avoiding the inherit or copy & paste reuse dilemma. In *Proceedings of the IEEE Symposium on Visual Languages* (Halifax, N.S., Canada, Sept. 1–4, 1998). IEEE Computer Society Press, Los Alamitos, Calif., 40–46.
- REPENNING, A. 1995. Bending the rules: steps toward semantically enriched graphical rewrite rules. In *Proceedings of the IEEE Symposium on Visual Languages* (Darmstadt, Germany, Sept. 5–9, 1995). IEEE Computer Society Press, Los Alamitos, Calif., 226–233.
- SASSIN, M. 1994. Creating user-intended programs with programming by demonstration. In *Proceedings of the IEEE Symposium on Visual Languages* (St. Louis, Mo, Oct. 4–7, 1994). IEEE Computer Society Press, Los Alamitos, Calif., 153–160.
- SMITH, D. 1975. Pygmalion: a computer program to model and stimulate creative thought, Ph.D. dissertation, Stanford University, Stanford Calif. 1975. (A more accessible summary of this work can be found in Cypher, A. 1993. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, Mass.)
- SUGIURA, A. AND KOSEKI, Y. 1996. Simplifying macro definition in programming by demonstration. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, Seattle, Wash., Nov. 6–8, 1996). ACM Press, New York, N.Y., 173–182.
- VANDER ZANDEN, B. AND MYERS, B. 1995. Demonstrational and constraint-based techniques for pictorially specifying application objects and behaviors. *ACM Trans. Comput.-Hum. Interact.* 2, 4, 308–356.
- WANG, G. AND AMBLER, A. 1996. Solving display-based problems. In *Proceedings of the IEEE Symposium on Visual Languages* (Boulder, Colo., Sept. 3–6, 1996). IEEE Computer Society Press, Los Alamitos, Calif., 122–129.

- WILDE, N. AND LEWIS, C. 1990. Spreadsheet-based interactive graphics: from prototype to tool. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (April 1990). ACM Press, New York, N.Y., 153–159.
- WOLBER, D. 1997. An interface builder for designing animated interfaces. *ACM Trans. Comput.-Hum. Interact.* 4, 4, 347–386.
- YANG, S. AND BURNETT, M. 1994. From concrete forms to generalized abstractions through perspective-oriented analysis of logical relationships. In *Proceedings of the IEEE Symposium on Visual Languages* (St. Louis, Mo., Oct. 4–7, 1994). IEEE Computer Society Press, Los Alamitos, Calif., 6–14.

Received December 2000; revised January 2002 and March 2002; accepted April 2002