A Scalable Method for Deductive Generalization
in the Spreadsheet Paradigm

Margaret Burnett*, Sherry Yang**, and Jay Summet*

*Oregon State University
Corvallis, Oregon
{burnett, summet}@cs.orst.edu
**Oregon Institute of Technology
Klamath Falls, Oregon
yangs@oit.edu

Abstract
Concreteness, direct manipulation, and immediate visual feedback are widely-used characteristics in graphical and end-user programming languages. However, these characteristics present a challenge in automatically generalizing programs for reuse, especially when examples are used as a way of achieving concreteness. Perhaps the most widely used of all such languages are spreadsheet languages, which include commercial spreadsheet systems as well as various graphical languages with roots in the spreadsheet paradigm.

In this paper, we present an efficient method for automatically generalizing programs written in such languages, through the incremental analysis of logical relationships among concrete program entities from the perspective of a particular computational goal. The method uses deductive analysis rather than inference, 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 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.

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

1 Prior publication history: Part of the user’s view of generalization and short summaries of the strategy have been included in previous journal papers [Burnett and Gottfried 1998; Burnett et al. 2001]. One early version of the approach was published in a conference [Yang and Burnett 1994]. These papers do not include in-depth information such as algorithms, complexity analysis, or performance data.
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 GUIs [Myers 1991]. In this paper, we use the term spreadsheet languages to describe all such systems following the spreadsheet paradigm and the term extended spreadsheet languages to describe spreadsheet languages, such as those in the previous sentence, with features beyond those of widely-used commercial spreadsheet systems.

Forms/3 [Burnett and Gottfried 1998; Burnett et al. 2001] is the extended spreadsheet language in which we 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 to do so. In keeping with the characteristics of concreteness, direct manipulation, and immediate visual feedback, users of Forms/3 program very concretely, and receive continuous visual feedback throughout the process. But although they use direct manipulation and prototypical values extensively and flexibly 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 we address in this paper is how to generalize the concrete program that was entered so that this expectation of generality can be fulfilled.

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. The traditional approach fell short because, in the presence of features such as multiple grids and linked spreadsheet copies, 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 traditional 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 other 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; Myers 1998; Olsen 1996; Perrone and Repenning 1998; Repenning 1995; Sassin 1994; Smith 1975; Sugiura and Koeski 1996; Wolber 1997]. Some of these use inference, and some require the user to explicitly provide the generalized meaning. The two most significant ways in which our method differs from these are:

- Neither inferential nor user-assisted: The generalization method 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\(^1\), there is no risk of “guessing wrong.” In contrast to

---

\(^1\) In much of the artificial intelligence literature, the term inference includes both sound reasoning techniques such as deduction, and techniques employing guesswork. However, in literature about demonstrational programming languages, the term is normally used to mean only reasoning techniques employing guesswork. In this paper, we follow the latter convention.
this, 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 dependant upon display size, not upon overall program size. Prior generalization research has not investigated scalability properties of generalization methods.

In developing our method, we also 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 edits 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 “pre-generalization mode” that requires user actions or reasoning that are different from those of a “post-generalization mode.” Modelessness is one of the key characteristics of the spreadsheet paradigm that distinguishes it from traditional ways of programming. Hence, we felt 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 co-routines and pipelines. This constraint is necessary for the referencing flexibility that is customary in the spreadsheet paradigm.

We begin our presentation in Section 2 with an overview of related works and their relationships to the generalization method presented here. Section 3 then introduces the generalization method from the user’s perspective. 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 shows how the generalization method supports several design patterns possible in spreadsheet-based programs. Section 6 extends the method to grids, temporal sequences and animations, and user-defined types. After a presentation of performance measurements in Section 7, the paper concludes.

2: RELATED WORK

As a graphical spreadsheet language, Forms/3 shares many features with traditional spreadsheet languages. Also, 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.

The generalization approach used by other spreadsheet languages has always been based on spatial rather than logical relationships. Programming-by-demonstration systems, on the other hand, have contributed a much greater variety of approaches, generalizing on various relationships including spatial and logical. They solve the generalization problem in one of two ways: either (1) direct reliance on the user for generalization information, or (2) inference with user assistance and/or corrections. In both cases however, the user assists the generalization process, specifying which portions of their input should be generalized and/or assisting the
system by requesting generalization, 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. Forms/3’s generalization method is different from these systems in that in Forms/3, the user does not explicitly request, assist, participate in, or correct the generalization process.

2.1: Generalization in Demonstrational Systems.

Some by-demonstration systems that avoid inference do not actually do any generalization automatically. Instead, the user simply types in the generalized meaning manually in a language such as Lisp or Smalltalk. One example is the Tinker system [Lieberman 1993]. Other by-demonstration languages avoid inference by requiring some user-contributed generalization information, which is combined with deduction to complete the generalized program. In PT [Ambler and Hsia 1993], a general-purpose by-demonstration language, useful generalization-related information is provided by the user as part of the 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. ToonTalk [Kahn 1996] supports generalization by allowing the user to specify which operation parameters can be generalized by “erasing” detail from the concrete example used during programming with a special eraser tool. Topaz [Myers 1998] is another system in which the user explicitly specifies the generalization operation’s parameters via dialogs. In contrast to these systems, in our approach, the user does not converse with the system about generalization parameters.

An example of a deductive approach in which the user must trigger generalization is AgentSheets [Perrone and Repenning 1998; Repenning 1995] which includes graphical rule analogies that allow users to generalize a behavior from one object to another (“Cars move on roads like trains move on tracks”). However, the system will not generalize the program (apply the analogy rules) without an explicit request from the user.

AgentSheets is representative of graphical rule-based languages, which are based on before/after examples. Another system in this family, Inference Bear [Frank and Foley 1994], also uses deductive reasoning as a way of generalizing based upon “before” and “after” examples. Inference Bear’s deductive reasoning system reproduces the correct “after” example when given the corresponding “before” situation. However the deductive reasoning system does not know about the state of the entire system, and there is no guarantee that the generalized program will take all relevant factors into account to produce output for a new situation or input.

KidSim/Cocoa/Stagecast [Cypher and Smith 1995; Heger et al. 1998] also allows behavior to be specified with graphical “before” and “after” situations. This system does not perform automatic generalization per se. Instead, the before/after examples are spatially matched as in the example. Users can also explicitly generalize a rule further by stating that it should abstract beyond the specific instance to, for example, all instances of a set of types. Additional, non-graphical attributes are also allowed to a limited extent.

On the other hand, many by-demonstration systems use inference to generalize a program. While inference is not powerful enough for general-purpose programming, it can be very effective in a limited problem domain. For example, inference was used in the by-demonstration system Peidot [Myers 1993], which is a language specifically for user interface specification. Building upon this work, more sophisticated inference algorithms such as plan recognition and decision tree learning are used by Gamut [McDaniel and Myers 1999], an integrated language environment for building interactive software.
Any inference technique can guess wrong, and systems based upon inference therefore can generate incorrect programs if this possibility is not addressed. Pavlov [Wolber 1997] is an interface builder that uses inference rules on a user-provided example to generate a stimulus-response behavior. In the process of generalizing, the Pavlov system generates various possible conditions that limit its stimulus-response behavior, and asks the user to select the appropriate condition(s) with a dialog. In inference-based generalization approaches, it is common for systems to use this kind of technique to require that the user approve a guess before it is used.

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. Most other generalization methods that use inference rely on the user to proactively assist at the generalization stage, by triggering generalization and/or picking the correct option from a list of possible choices generated by the inference engine.

Other 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 the generalization of imperative macros or functions in applications such as DemoOffice [Sugiura and Koeski 1996], ProDeGE+ [Sassin 1994] and Chimera [Kurlander and Feiner 1992] from the user’s command history. In these examples, when there is more than one possible generalization of a user’s history, DemoOffice uses a heuristic to choose the most likely, while ProDeGE+ and Chimera present the user with a dialog asking for clarification. These approaches use inference and require user intervention in the generalization process.

2.2: Generalization in Spreadsheet Languages

As a graphical spreadsheet language, Forms/3 follows the path initiated by the spreadsheet languages NoPumpG [Lewis 1990] and NoPumpII [Wilde and Lewis 1990] which extend the spreadsheet paradigm to support interactive graphics. However, the NoPump languages include no facility for generalization.

The program entities that need to be generalized in a 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 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, and these adjustments exclude the row being added—which is correct if the user is adding a new subtotal row and incorrect if the user is adding a new detail row. The method presented in this paper considers logical relationships rather than relying solely upon spatial 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. However, no facility was present to deductively interpret a user’s direct manipulations in order to produce the notation. Also, in this earlier version, the textual notation described each copy of a form by enumerating exactly how it differed from the original, a notation powerful enough to support the standard structures found in traditional programming languages such as
global references and subroutine-like relationships, but not powerful enough to support many non-traditional relationships.

3: THE GENERALIZATION METHOD FROM THE USER’S PERSPECTIVE

Two HCI-related design goals that have had a particularly strong influence on graphical languages are directness and immediate visual feedback. Directness means employing a vocabulary (whether graphical or textual) directly related to the task at hand. Directness is illustrated in Forms/3 by the ability to specify graphical values such as the value of cell `newCircle` in Figure 1 by sketching, manipulating, and gesturing with circles [Burnett and Gottfried 1998]. Supporting directness in such ways leads to a need for more advanced generalization capabilities than have previously been available in commercial spreadsheet languages. Supporting immediate visual feedback constrains the solution space to approaches that can support quick response.

![Forms/3 form (spreadsheet) that defines a circle.](image)

**Figure 1:** A portion of a Forms/3 form (spreadsheet) that defines a circle. The circle in cell `newCircle` is specified by the other cells, which define its attributes. 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.

3.1: Generalization Example: Population Visualization (User’s View)

A Forms/3 spreadsheet is called a *form*. 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 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. In the example of Figure 2, the population analyst defines 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. Also, in Forms/3 (and in some other spreadsheet languages), several cells can be defined to officially share a single formula. In Figure 2, all the cells in the *graph* grid share the displayed formula. Because of these capabilities, generalization is not needed for correctness of
this application, although using a generalized notation saves storage space and time, as will be seen later.

Figure 2: (a) A form under development to visualize population data. The formula shown at the right of the 4x1 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). The form is gray because it is a copy, but white cells have formulas different from the original. Finally, the population analyst specified the fillForeColor formula to be “BLACK” via a popup menu.
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” of some other spreadsheet languages). We will refer to an original as the *model form* and to the copies as the *instances*. The 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*. When editing a formula in Forms/3, a user can refer to any of these cells by pointing at them. 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).

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. This is where the need for generalization becomes critical.

To create this more general version of the program, the population analyst again sketches a circle, which creates another instance of the system-provided *primitiveCircle* form (say, 179PrimitiveCircle), 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/redispalyes 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 179PrimitiveCircle. 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`.

Figure 3: 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 in this figure. (The user can also see a more general view of the formula, as will be shown in the next example.)
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 recursive 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 for walking through the method in detail.

Recursion is supported by copying a model form to additional instances that can be referred to in other 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. 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 enters the formulas in the white cells. To produce immediate visual feedback, as soon as any formula is entered, the system must display the resulting value. 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

![Figure 4: One way to program Fibonacci in this language. This version includes non-traditional 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 also Appendix A 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.](image)
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 (which compute the N-1st and N-2’nd Fibonacci numbers). In fact, without generalization, 55_Fib’s Ans formula (which 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. To solve this problem, the system recognizes and records the logical relationships among Fib and its instances, which allows it to compute and display the answers for all three forms’ Ans cells.

After generalization, the model form appears as in Figure 5. Note the tiny key icons to the right of the concrete cell references in Ans’s formula. This indicates that these concrete names are just samples representing generalized references. Moving the mouse over a key brings up a legend explaining what the concrete reference represents, as shown at the very bottom of the figure, and this is the way a user can view the generalized meaning. The other direction is possible too: a user can view an example of a generalized reference if it is not already on display by simply clicking on the reference, at which time a concrete example of that form will spring into view.

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 pre-generalization and post-generalization “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.

Figure 5: The Fibonacci program after generalization. The user has moved the mouse over the leftmost tiny key icon in Ans’s formula to view the legend (at the bottom) describing what 55_FIB:Ans represents. This generalized formula is used in context by all copies of FIB; thus 55_FIB:Ans’s formula no longer refers to itself, but rather to Ans on a copy of FIB whose N=55_FIB:N-1.
In the previous section, we showed what the generalization method looks like to the user. Now we move behind the scenes to show how it looks to the system. From this point until Section 6, we describe generalization of single cells. Single cells that do not serve as part of a grid or group of related cells require generalization when they are copied\(^1\). Concentrating on these cells allows illustration of the basic approach without detouring into the details of more complex collections of cells. In Section 6 we then add the straightforward extensions needed to handle the more complex structures, including the handling of grids with shared formulas among groups of cells.

From the system’s perspective, the method consists of two steps: incrementally tracking logical relationships (step 1) and lazily generalizing these relationships “just in time” (step 2). 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 relies upon the strategies of incrementalism and laziness. These two strategies support each other in a mutually dependent way. For example, by making small amounts of incremental progress frequently (i.e., after each user action), it is possible to delay other portions of the work. On the other hand, because the system lazily processes only the work that cannot be delayed any longer, it is possible to do only a small increment of work at any one time—the rest can be delayed. The method’s scalability without loss of responsiveness is due to the way incrementalism and laziness are combined.

Another strategy used by our method is subtractiveness. Most approaches to program generalization view the problem as being to \textit{generate} missing information. However, our approach views the problem as being the presence of too much information, and \textit{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.) Subtracting rather than adding is part of what avoids the use of inference, because there is no need to conjure up missing information.

\begin{algorithm}
\begin{algorithmic}
\Function{EventLoop}{ }
\State loop forever
\State sleep until next event;
\If { (event.type = “formula edited” or event.type = ”cell moved onto screen”) } 
\State \textit{//Step 1: Incrementally add to relationship information:}
\State for each on-screen reference in event.cell’s formula Insert (label, (event.cell, reference));
\State for each on-screen reference deleted from event.cell’s formula Delete (label, (event.cell, reference));
\State if Cycle? (event.cell) then event.type = “possible cycle”;
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\begin{algorithmic}
\If { NeedToGeneralize?(event.type) } 
\State RemoveICE;
\State Generalize;
\EndIf
\end{algorithmic}
\end{algorithm}

Figure 6: The top level event loop.

\[^1\] There is one very easy case to generalize: A formula may be a constant with no references to other cells. This case is already fully generalized and there is nothing further to do; thus, we will not discuss it further.
4.1: 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 simply 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 efficiency of the method.

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

\[
V = \{ v | v \text{ is an on-screen cell in the program} \}
\]

\[
LE = \{ (label(e), e) | \text{label(e)} \in L \text{ and } e \in E \}
\]

\[
E = \{ (u,v) | \text{cell u is referred to in v's formula, where } u, v \in V \}
\]

\[
L = \{dg, dc, ig, ic\}
\]

Although we refer to them as sets, \( LE \) and \( E \) are actually bags (sets allowing duplicates), and are the subsets (sub-bags) 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 on those of 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. When the generalization algorithm generalizes an edge (a relationship), it generalizes all concrete edges leading to the same node at once; because of this fact, the terms "concrete" and "generalized" can also be used to describe a node's (cell's) generalization state.

Using the 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 \( e \in ICE \). (Several diagrams of these 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. (An alternative organization such as a heap binary tree would show better worst-case time, but in our experience, not better time in practice.)

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.
- \( Edges= \) 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, even if they are iconified or obscured by other forms, and on-screen cells are cells on on-screen forms. 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 following property: a cell must be on the screen for the user to edit its formula or to reference it. (This property is common in graphical languages.) 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 the screen.

4.2: Step 1: Incrementally Processing Relationships Among On-Screen Cells

4.2.1: 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 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 the simple algorithms Insert, Delete, and Find, shown in Figure 7. (Node 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. In our implementation, these are standard hash table operations.) Assuming an appropriate data structure for the adjacency lists, such as a doubly-linked list, the most expensive line in the figure is Find’s line 2, which requires a linear search through the edge list. The length of the adjacency list is bounded by the length of the cell’s formula, which has a constant bound in most spreadsheet languages. Thus, the worst-case cost of Find is the sum of the cost of traversing the edge list and the cost of finding the list in the first place, which we count as O(|V|) worst

```plaintext
//Add labeledEdge le = (label,(u,v))
Algorithm Insert(le)
(1) foundLE = Find(le);
(2) if foundLE = notFound then add le to le.v.edges
(3) else foundLE.counter = foundLE.counter + 1;

//Delete labeledEdge le = (label(e),(u,v))
Algorithm Delete(le)
(1) if le’s location is known then foundLE = le else foundLE = Find(le);
(2) if foundLE.counter = 1 then remove foundLE from foundLE.v.edges
(3) else foundLE.counter = foundLE.counter - 1;

//Find labeledEdge le = (label(e),(u,v))
Algorithm Find(le)
(1) let v = le.v  //v is the cell where le should be stored.
(2) if le is in v.edges then return le
(3) else return notFound;
```

Figure 7: Basic algorithms for edge maintenance.
case, assuming a hash-table-based implementation and doubly-linked edge lists. In fact, even if formula length were not constant-bounded, the length of edges would not exceed $|V|$, given the presence of the counter optimization for edges. Thus, the cost of Insert, Delete, and Find are all $O(|V|)$.

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. The latter is handled the same as the first case, so we can describe the method’s operations concretely as triggered by a formula edit without loss of generality.

When the user first enters a formula, all its edges are direct, as shown in Figure 8. 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 $(\text{Fib:N,Fib:N1})$ in Figure 8. Obviously, the relationship about which copy of the Fib form is needed for $N_1$’s reference to $N$ is simple: the same copy for $N$ as for $N_1$. 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 $(\text{Fib:N1, 55_Fib:N})$ may need further processing, and they are elements of DCE.

When the user makes a new instance of a form for reuse, 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 been generalized yet, and are classified as ICE edges. See Figure 10.

At this point, the cell reference graph has some similarities to a dataflow graph, but it contains anomalies due to concreteness. In Figure 10, 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.

This type of anomaly is easy to see in recursive examples like the one above, but it also can arise in non-recursive programs. Consider two linked spreadsheets $S1$ and $S2$ that were copied from the same original, but a cell on $S1$ now refers to one on $S2$ and a cell on $S2$ now refers to one on $S1$. An example of such a case might be sales spreadsheets for adjacent departments that have some shared cash registers, hence requiring both departments’ shared register subtotals to be computed using part of the adjacent department’s data. Recalling that many users developing spreadsheets are not computer scientists, it is quite likely that referencing patterns will not all go the same direction, and apparent dataflow anomalies like these will arise.

4.2.2: 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 the following type of 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. Detecting 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 non-terminating calculations may be generated.

Possible cycles are detected via Figure 11’s `Cycle?`, which uses a depth-first search on the on-screen cells. `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. Figure 10 includes examples of possible cycles involving cell `Ans` on `55_Fib` and `104_Fib`.

**Figure 8:** A portion of 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 9 explains the edge patterns.

**Figure 9:** 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.

**Figure 10:** The cell reference graph of Figure 8, with all edges shown.

```cpp
//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;
```

**Figure 11:** 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.
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. Note that, since cycle detection is done incrementally after each formula edit, the user’s most recent formula edit is the one that created the cycle. In Forms/3, the language-specific response is to produce an error message and to reject the formula, because circular formulas are not allowed in Forms/3. Although some spreadsheet languages allow true cycles and some do not, all take some kind of special action if a true cycle occurs, such as rejecting the most recent formula entry as in Forms/3, asking the user to specify some maximum number of iterations as in Excel, etc. Hence, even without generalization, such systems must do some form of cycle detection on all cell references involved, regardless of whether they are on the screen. Our implementation does this during the evaluation process via exception handling if a stack overflow occurs (which simply displays an error value in cells whose values cannot be computed and then returns to the event loop). Another approach we have used in the past is depth-first traversal of the dataflow graph rooted at the new formula’s cell. These costs are not part of the cost of generalization, because they are necessary even in systems that do not support generalization.

4.2.3: 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. As Figure 12 shows, 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 generate automatically 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, and this 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, and this automatically generalizes all instances of the form as well (through pointers referring copied formulas on form instances to the model’s formulas). If generalization did not happen at this point, the new instance could refer to an old instance that is 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 4.3, this means the affected cell(s) must be re-generalized. This can only occur with on-screen cells, because of the next case. The cost of detecting this case is still counted as O(1), because it does not add new traversals for generalization purposes that are not already present in the course of spreadsheet evaluation. In other words, 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 cell or form is removed from the screen: Cell reference dependencies are removed from the cell reference graph when a cell is removed from the screen. 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. Because of this strategy, removing a cell from the screen also
removes its information from use during later generalizations. This strategy also requires removal of any instance cells from the screen whose formulas affect the generalized cell, to prevent them from later being edited in a way that changes the generalized formulas of the off-screen cell. Also, generalization is required for on-screen cells relying upon a cell that is being removed from the screen, before the departing cell’s information is deleted from the cell reference graph.

```csharp
// Is formula generalization needed yet?
Algorithm NeedToGeneralize? (event)
    if event.type = "possible cycle" then return true;
    if event.type = "save" then return true;
    if event.type = "copy" then return true;
    if event.type = "formula edited" and
        event.cell affects an on-screen generalized cell then return true;
    if event.type = "remove" then return true;
    return false;
```

Figure 12: Events requiring immediate formula generalization.

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

Once formula generalization has been triggered, a subgraph of the cell reference graph is then mapped to a compact textual notation, which is combined with the original formula to produce the generalized version of the formula. The details of these tasks are as follows.

#### 4.3.1: The Subgraph of Interest

If the generalization trigger was one of those given in Section 4.2.3, 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). Because the size of the cell reference graph is bounded by the number of cells on the screen, the size of SSCG, the subgraph being generalized, is also bounded by this number.

Our method is subtractive in that it makes some of its progress by removing information: it removes edges from SSCG that are elements of ICE (inherited concrete edges), using the algorithm in Figure 13. 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 have already been discovered of the labeled edges passed to it.

The advantage to 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, although they were necessary for computing concrete formulas, once generalization is complete they are superfluous. This is because they are inherited: hence the same information can always be found in the original entry.

Traversals through SSCG happen in backwards topological order. For example, Figure 14 shows SSCG for the Fibonacci example, laid out in reverse traversal order (i.e., traversal is from right to left, against the dataflow arrows). \texttt{Ans} is the topological root of SSCG. Notice that only those cells relevant to the computation of the cell \texttt{Ans} are in SSCG. For example, the \texttt{N2} cells are not present because there is no path in the cell reference graph from these cells to \texttt{Ans}. 

- 17 -
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 all ICE in-edges to v
        RemoveICEedges(le.u) // and in paths leading to v.

Figure 13: Algorithm to remove ICE edges from the relevant subgraph. (Refer again to Algorithm Cycle? for a brief description of one possible marking approach.)

Figure 14: SSCG rooted at cell \textbf{Ans}, after removal of the ICE edges.

4.3.2: 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, and hence is not displayed or used by the user.) This notation is described in detail in the next two subsections.

Two requirements for this internal notation are that it must provide enough information for the system (1) to recognize 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 or she did while programming it originally.

4.3.3: Notation 1: A First Approximation

As a first approximation of such a notation, let \( F \) be a form, let \( F_i \) be a copy of \( F \) instantiated directly by a user performing a \texttt{copy} action, and let \( \text{DefSet}_i \) be a set of elements of format “\( Y\circ\phi \)” 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:
\[ F_i = F(\text{DefSet}_i) \]  

(Notation 1)

Given \( F_i \), the generalized description \( F(\text{DefSet}_i) \) 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 “\textit{self}” in \( F(\text{DefSet}_i) \) for references to the same form copy and 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 \( \text{Fib} \) needed to support computation of \( 55\_\text{Fib:Ans} \) and \( 104\_\text{Fib:Ans} \).

For example, the generalized formula for \( \text{Fib:Ans} \) expressed using this notation is:

\[
\begin{align*}
\text{if } \text{self:N} &< 2 \text{ then 1} \\
\text{else } &\text{generalized description of } 55\_\text{Fib:Ans} + \\
&\text{generalized description of } 104\_\text{Fib:Ans}
\end{align*}
\]

(1)

where \text{generalized description of } 55\_\text{Fib} from the perspective of \( \text{Fib:Ans} \) (the cell currently being generalized) is:

\[ \text{Fib(N }\equiv \text{ self:N1)} \]

(2)

and \text{generalized description of } 104\_\text{Fib} from the perspective of \( \text{Fib:Ans} \) is:

\[ \text{Fib(N }\equiv \text{ generalized description of } 55\_\text{Fib:N1)} \]

which reduces, by substituting in (2) above, to:

\[ \text{Fib(N }\equiv \text{ Fib(N} = \text{self:N1):N1)} \]

Self’s meaning is context-dependent, where context is which cell’s formula is being generalized. Stated another way, \textit{self} in the formula for cell \( X \) on form \( F_i \) means \( F_i \). This states relationships in a way that causes each to be executed in the appropriate context, allowing even \( 55\_\text{Fib:Ans} \) and \( 104\_\text{Fib:Ans} \) to compute using the same kind of relationships as in \( \text{Fib:Ans} \). Self’s meaning is static, and it never varies within any one formula. For example, all three \textit{self}s above mean cell \( \text{Ans} \)’s form instance, not other copies of \( \text{Fib} \).

4.3.4: Notation 2: Better Use of Granularity and Perspective

Because Notation 1 works at the granularity of entire 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 in fact any arbitrary non-circular cell referencing pattern. Support for these other non-circular 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 \( \text{Ans} \)’s addition needed simply to be 1 more than the second factor. (To clearly differentiate this changed version from the original example, at this point we will now change the form names to \( \text{NotFib} \).) One way to program this would be for the user to change \( 55\_\text{NotFib} \)’s \( \text{Ans} \) cell’s formula to:

\[ 1 + 104\_\text{NotFib:Ans} \]
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:

\[
\text{NotFib}(N \equiv \text{self:N1}, \\
\text{Ans} \equiv 1 + \text{generalized description of} \ 104\_\text{NotFib}:\text{Ans})
\]  

(3)

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

\[
\text{NotFib}(N \equiv \text{generalized description of} \ 55\_\text{NotFib}:\text{N1})
\]  

(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. Use of perspective is another example of our subtractive strategy: perspective allows some information to be filtered out, and this prevents the generalization algorithm from getting caught in apparent cycles (when viewed at the granularity of forms) that do not really exist at the granularity of cells.

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 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 “=” in a DefSet in X’s formula must exist in SSCG and thus have a path to X.

Because the notation maps SSCG edges to cell references, we will also map our terminology of edges to cell references. 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. 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.

Employing this terminology, if X is the cell whose formula is currently being generalized and its formula contains a concrete reference to \(F_i:Z\), then let \(\text{AffectsSet}_X\) 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 \(\text{AffectsSet}_X\), the set of Ys 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:

\[
F_i:Z = F(\text{DefSet}_i \cap \text{AffectsSet}_X):Z
\]  

(Notation 2)

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

\[
\text{if self:N<2 then 1} \\
\text{else} \quad \text{generalized description of } 55\_\text{NotFib}:\text{Ans} \ + \\
\text{generalized description of } 104\_\text{NotFib}:\text{Ans}
\]  

(1’)

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

\[
\text{NotFib}(N \equiv \text{self:N1}, \\
\text{Ans} \equiv 1 + \text{generalized description of} \ 104\_\text{NotFib}:\text{Ans}:\text{Ans}
\]  

(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:

\[
\text{NotFib}(N \equiv \text{generalized description of} \ 55\_\text{NotFib}:\text{N1}):\text{Ans}
\]  

(4’)
Note how the use of cell granularity and perspective in (3’) and (4’) avoids the cycles of Notation 1’s (3) and (4): 104_NotFib:Ans does not affect 55_NotFib:N1, and hence the description of 55_NotFib:N1 does not need to include 104_NotFib:Ans, which removes the circularity of Notation 1’s lines (3) and (4). More formally, 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:

$$\text{NotFib}(N\equiv\text{self}:N1):N1$$

(5)

Substituting (5) back into (4’) completes the generalized description of 104_NotFib:Ans needed:

$$\text{NotFib}(N\equiv\text{NotFib}(N\equiv\text{self}:N1):N1):\text{Ans}$$

(4’’)

Substituting (4’’) back into (3’) completes the generalized description of 55_NotFib:Ans:

$$\text{NotFib}(N \equiv \text{self}:N1, \\
\text{Ans} \equiv 1 + \text{NotFib}(N\equiv\text{NotFib}(N\equiv\text{self}:N1):N1):\text{Ans}):\text{Ans}$$

(3’’)

Finally, substituting (3’’) and (4’’) back into (1’) results in the following final generalized formula for NotFib:Ans:

$$\begin{align*}
\text{if self:N} & < 2 \text{ then 1} \\
\text{else } \text{NotFib}(N \equiv \text{self}:N1, \\
\quad \text{Ans} & \equiv 1 + \text{NotFib}(N\equiv\text{NotFib}(N\equiv\text{self}:N1):N1):\text{Ans} + \\
\quad \text{NotFib}(N\equiv\text{NotFib}(N\equiv\text{self}:N1):N1):\text{Ans}
\end{align*}$$

4.3.5: The Formula Generalization Algorithm

Figure 15 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 ∩ AffectsSet described in the previous subsection, and otherwise do not affect any model and can be ignored (since after generalization, instance forms can be discarded). The maximum number of times through the loop is, of course, O(|V|).

GenFormula works its way through the in-edges modeling the on-screen references in cell currentCell’s formula, calling GenRef for every cell reference. As we have pointed out elsewhere, the maximum of length of any one formula is constant-bounded in most spreadsheet languages, and we assume that here.
Except for the calls to \texttt{Def\textperiodcentered Affects}, \texttt{GenRef} is a constant-time function. It translates a single reference in \texttt{currentCell}'s formula to Notation 2. The calls to \texttt{formID} in this function return “self” if \texttt{ref} and \texttt{perspective} are on the same form; otherwise it is the form name of \texttt{ref}’s model form, such as \texttt{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 \texttt{Def\text_period Affects}. The last line is just for completeness; edges labeled \texttt{ic} have already been removed by the \texttt{RemoveICE} algorithm presented earlier.

\texttt{Def\text_period Affects} performs the intersection. It is able to do this by walking through the \texttt{AffectsSet} for cell \texttt{perspective}, checking the form location and modification status of each cell encountered. The union is really just an append, since each element is encountered uniquely. Thus, \texttt{Def\text_period 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 \texttt{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

```
// 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 \texttt{GenFormula}(\(u\).modelCell, \(u\).modelCell)
    else \texttt{GenFormula}(\(u\), \(u\));

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

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

// Computes the \texttt{DefSet} \textperiodcentered \texttt{AffectsSet} part of Notation 2.
Def\text_period Affects(\texttt{ref}, \texttt{perspective})
  result = { }; // This is AffectsSet.
  for each \(affects\) \texttt{\in} SCCG rooted at \texttt{perspective} // affects \texttt{\in} DefSet.
    if (\(affects\) is on the same form as \texttt{ref} and
      \(affects\).formula has been edited) then
      result = result \textcup (\texttt{affects}.cellID ++ "=" ++ \texttt{genFormula}(\texttt{affects}, \texttt{perspective}));
  return result;
```

Figure 15: Algorithm \texttt{Generalize}. Operator ++ denotes string concatenation.
removing a form from the screen).

4.4: Computational Equivalence between Concrete and Generalized Formulas

We will say that the generalized version of a reference in cell X’s formula is *computationally equivalent* to the concrete version if replacing every concrete reference $F_i;Z$ with the generalized reference results in the same value in cell X as with the concrete reference, provided that the concrete version terminates. In this section, we show that the generalization method maintains this property. The notion of computational equivalence is basically correctness of the method under the Modelessness constraint given in the introduction, which requires that there cannot be a “pre-generalization mode” that requires user actions or reasoning that are different from those of a “post-generalization mode.”

For now, assume the cell reference graph has information about all relationships in the program, i.e., that there are no off-screen cells, and that no ICE edges are present in the concrete version. (We will relax these assumptions in the next paragraph.) Under these assumptions, as we have pointed out before, replacing every reference $F_i;Z$ with $F(DefSet_i);Z$ using Notation 1 would have maintained this property, since $F(DefSet_i)$ completely describes $F_i$ by enumerating every difference between $F_i$ and $F$. Despite its omission of information from Notation 1, Notation 2 does not lose this property, because a cell not in AffectsSet $x$ cannot possibly have any effect on X’s value; hence a reference to $F_i;Z = F(DefSet_i \cap AffectsSet_x);Z$ must produce the same result in X as a reference to $F(DefSet_i);Z$ would produce. Hence the generalized version of the formula under both Notations 1 and 2 are computationally equivalent to the concrete version under these assumptions, provided that the concrete version terminates.

We now relax the above assumptions. First, we will allow some of the relationships in the program to be off-screen (and thus not in the cell reference graph). Because a trigger for generalizing is removal of a cell from the screen, we know that all relationships involving off-screen cells were generalized at the time the cell was removed. We have already explained that it is neither possible to change the formulas of existing off-screen cells, nor to add new references to these off-screen cells without bringing them back onto the screen. Because of this property, all relationships involving an off-screen cell have already been generalized in a way that cannot be affected by the user’s manipulations and edits of on-screen information, and the cell reference graph can safely omit information about them. Even when off-screen cell formulas are generated via gestures, the generated formulas are created in pre-generalized form; after their creation the only way they can be edited in a way that requires new generalization activity is for them to be brought onto the screen.

Second, we will allow ICE edges to be present in the concrete version. Although these edges are removed from the cell reference graph, this does not result in loss of information, because the inherited references on form instances that these edges model are, by definition, simply duplicates of references that also exist in the model. Thus, the model form’s edges include the same information as the ICE edges and it is not necessary to include the latter’s duplicated information in order to generalize. It is also not necessary to actually generalize the inherited formulas modeled by the ICE edges, because cells with “inherited formulas” can just copy or use the formulas in the model version after generalization is complete.

Thus, the method satisfies the computational equivalence property for concrete forms that are able to terminate. Further, if the generalized program itself terminates (which means there are no remaining cycles), it can be said to be computationally equivalent in some sense to the non-terminating concrete version. Specifically, when all non-terminating concrete instance calculations are replaced by the generalized instance calculations (which do terminate), then the generalized model is obviously computationally equivalent to the concrete model, since both
models combine the instances’ results using the same operators and since there are no remaining cycles.

### 4.5: Time Complexities Case by Case

From the cost of each algorithm, the total cost of the generalization method can be derived by considering the possible user actions relevant to generalization and which algorithms they trigger:

- **New formula/cell:** A new formula is entered or brought onto the screen, or an existing formula is modified. There are two possible sub-cases: Either the new formula caused a possible cycle or it did not.
- **Several new formulas/cells:** Several existing cells are moved onto the screen, such as when an entire form is displayed. This is the same case as the previous one except for its volume.
- **A form is saved or chosen for removal from the screen or memory:** At least all the cells on the form must be generalized; in some cases all the cells on the screen must be generalized, as was discussed earlier.
- **A new instance is created by copying from a model.**

Table 1 recapitulates the costs presented in the previous sections of each algorithm, and Table 2 uses these costs to show the time complexity for each of the above user actions. As Table 2 shows, the worst-case costs are in every case based on the size of the on-screen portion of the program as opposed to the size of the entire program. We have mentioned that in the Forms/3 implementation of the method, some cells that are not actually visible are classified as “on-screen,” such as cells on obscured or in iconified windows. However, the method does not require this or any other particular definition of “on-screen” to succeed. Any reasonable definition will suffice, because taking a cell out of this status is one of the events triggering generalization. For example, to precisely bound generalization cost by the number of pixels on the screen, a variation that we have considered is for the definition of “on-screen” to exclude obscured cells and cells on iconified windows. Under such a change, whenever the user iconified a form or obscured a cell, generalization would be triggered (for the affected cells). This change would generate more calls to the generalization algorithm than presently occur, but each call would potentially process a smaller SSCG, since fewer cells would be considered to be on-screen. However, there has been no reason to make this change, as generalization time in our implementation is already fast enough to not introduce noticeable delays, as will be demonstrated in Section 7.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Description</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert(le)</td>
<td>Add labeled edge le.</td>
<td>O(</td>
</tr>
<tr>
<td>Delete(le)</td>
<td>Delete labeled edge le.</td>
<td>O(</td>
</tr>
<tr>
<td>Find(le)</td>
<td>Find labeled edge le.</td>
<td>O(</td>
</tr>
<tr>
<td>Cycle?(aCell)</td>
<td>Detect possible cycles in subgraph rooted at cell aCell.</td>
<td>O(</td>
</tr>
<tr>
<td>RemoveICE</td>
<td>Remove ICE edges.</td>
<td>O(</td>
</tr>
<tr>
<td>Generalize</td>
<td>Record generalized formula relationships</td>
<td>O(</td>
</tr>
<tr>
<td></td>
<td>(cost shown is if triggered by editing one cell’s formula).</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Summary of algorithm costs.
<table>
<thead>
<tr>
<th>User Action</th>
<th>Algorithms Invoked (from Table 1)</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>New formula/cell on the screen, possible cycle</td>
<td>Insert, Delete, Find: called for each reference in (constant-length) formula. Cycle?: called once. RemoveICE: called once. Generalize: Called on one cell.</td>
<td>$O(</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>$O(</td>
</tr>
<tr>
<td>New formula/cell on the screen, no cycle</td>
<td>Insert, Delete, Find: called for each reference in (constant-length) formula. Cycle?: called once.</td>
<td>$O(</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>$O(</td>
</tr>
<tr>
<td>Several new formulas/cells on the screen.</td>
<td>Same as new formula (above), but repeat for each cell.</td>
<td>$O(</td>
</tr>
<tr>
<td>Any action requiring generalization of a single entire form</td>
<td>RemoveICE: called once. Generalize: Called on multiple cells. Delete, Find: In cases where cells are being removed from the screen, up to $</td>
<td>E</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>$O(</td>
</tr>
<tr>
<td>Any action requiring generalization of the entire screen</td>
<td>Same as row above.</td>
<td>$O(</td>
</tr>
<tr>
<td>New instance (copy)</td>
<td>Cost of generalization of single form. Insert, Find: Add edges for up to $</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>$O(</td>
</tr>
</tbody>
</table>

Table 2: Costs by user action.

5: GENERALIZATION EXAMPLES IN FOUR DESIGN PATTERNS

Recall the Generality constraint, requiring that the generalization method be flexible enough to handle cell referencing and linked spreadsheet “design patterns” beyond the ubiquitous call-return pattern of traditional programs. The notion of design patterns has become widely used in understanding program structures in traditional programming paradigms. To add intuition at a high level about the generalization method, here we show four spreadsheet design patterns—patterns of relationships among linked spreadsheets—to illustrate how the generalization method supports them. We will present these design patterns through the use of *form-collapsed multi-graphs*, a diagram we introduce solely for the purpose of this discussion. (Form-collapsed multi-graphs are not part of the generalization method itself.)

The form-collapsed multi-graph is an abbreviated version of the cell reference graph in that nodes from the post-generalization cell reference graph representing cells on the same form are collapsed into a single node, and only inter-form edges from the cell reference graph are included. Figure 16 shows the form-collapsed multi-graph for the Fibonacci program. Relationship patterns are emphasized by these form-collapsed multi-graphs. Examples of four design patterns supported by spreadsheet languages are given in Figure 17 using these graphs.
Figure 16: A form-collapsed multi-graph of the Fibonacci program. The relationship between \texttt{Fib} and \texttt{55\_Fib} is the same as would occur in a call-return relationship in conventional languages, but the relationship between \texttt{Fib} and \texttt{104\_Fib} is less conventional.

Figure 17: Four design patterns. (a) No cycles. (b) A 2-node cycle. (c) Multiple 2-node cycles between the same 2 nodes. (d) A cycle among more than 2 nodes.

5.1: Acyclic Pattern Example: Globals and Pipelines

In spreadsheet languages, there are two types of references. An \textit{absolute} reference is to the exact cell indicated, even when the formula is later copied elsewhere, similar to references to global variables or constants in traditional languages. A \textit{relative reference} is functionally the same as an absolute reference if the formula is never copied. However, if the formula is copied, a relative reference must be translated to a new absolute reference that has the same relationship to the new referencing cell as the original reference had to the original referencing cell. The acyclic pattern shown in Figure 17(a) contains only absolute references. This pattern is useful both for referring to global values and for setting up a pipeline of computations.

The cell reference graph in Figure 18 models a program that uses absolute references for both these purposes. This program defines a screen saver with a floating image, whose location is computed through a pipeline of computations that originates at the (global) system clock.

Absolute references do not require generalization for correctness. However, generalizing in this pattern reduces storage requirements, because the generalized formula notation eliminates the need to permanently store copies, provided that the model form itself is stored. For example, \texttt{ScreenSaver:FloatingImage}'s formula reflects how referenced cell \texttt{Image}'s copy of \texttt{Picture} differs from the model in ways that are relevant to cell \texttt{FloatingImage}:

\begin{verbatim}
Picture(X = SystemClock:Sec; Y = SystemClock: Min):Image
\end{verbatim}

Figure 18: The cell reference graph of the screen saver program from the perspective of cell \texttt{FloatingImage} (shaded for emphasis), laid out inside its form-collapsed multigraph. This is an example of the acyclic pattern.
5.2: 2-Node Cycle Example: Subroutine Functionality

A 2-node cycle, such as depicted in Figure 17(b), models the functionality of a traditional subroutine with parameters. (Note that in a form-collapsed multi-graph, such a cycle is at the granularity of forms, not cells.) When such relationships involve relative references, they must be generalized. The relationship between \( \text{Fib} \) and \( 55_{\text{Fib}} \) in Figure 16 follows this pattern. As discussed earlier, the part of \( \text{Fib:Ans} \)’s formula that makes use of this relationship:

\[
\text{if (self:N < 2) then 1 else 55_{\text{Fib}}:\text{Ans} + \ldots}
\]

becomes the following after generalization:

\[
\text{if (self:N < 2) then 1 else Fib(N \equiv \text{self:N1}):\text{Ans} + \ldots}
\]

5.3: Example of Multiple 2-Node Cycles Involving Only 2 Nodes: Co-routines

This pattern models what is known as “co-routines” in some programming languages that support concurrency. In co-routines, two procedures pass information back and forth as they work in parallel. In spreadsheet languages, this pattern is exhibited by two spreadsheets, each of which contains more than one cell referenced by the other. Figure 17(c) shows this pattern.

One use of this pattern is for more than one subroutine-like relationship between the same two forms in a logical extension of the previous pattern. For example, a user might add, via a new cell \( \text{Tree} \), to the Fibonacci program, a graphical diagram of the Fibonacci “call tree,” defining the diagram recursively in the same manner as cell \( \text{Ans} \). This addition adds another cycle to the same forms, but would result in the same generalized formula for cell \( \text{Ans} \) as before, since \( \text{Ans} \) does not depend on \( \text{Tree} \). Likewise, \( \text{Tree} \)’s formula does not depend on \( \text{Ans} \). The concrete version of \( \text{Tree} \)’s formula is quite similar to that given for \( \text{Ans} \) above:

\[
\text{if (self:N < 2) then 1 else compose 55_{\text{Fib}}:\text{Tree with \ldots}}
\]

which becomes the following after generalization:

\[
\text{if (self:N < 2) then 1 else compose Fib(N \equiv \text{self:N1}):\text{Tree with \ldots}}
\]

5.4: Example of Cycles with 3 or More Nodes: Return-to-Start Pipelines

This pattern, depicted by Figure 17(d), is not commonly supported in most programming languages. Here values are passed forward in a pipeline, and at the end travel back to the original caller without making intermediate stops at the intermediate callers. This was the relationship between \( \text{Fib} \) and \( 104_{\text{Fib}} \) in Figure 16. The relevant part of \( \text{Fib:Ans} \)’s formula for this pattern was:

\[
\text{if (self:N < 2) then 1 else \ldots + 104_{\text{Fib}}:\text{Ans}}
\]

which was shown earlier to generalize to:

\[
\text{if (self:N < 2) then 1 else \ldots + Fib(N \equiv Fib(N \equiv \text{self:N1}):N1):\text{N1}}:\text{Ans}
\]

5.5: Patterns in Perspective

Although we have expressed patterns in terms of form-collapsed multi-graphs, which are at the granularity of forms, recall that the generalization method works at the granularity of cells. Since true cycles at the granularity of cells in spreadsheet languages are either rejected or translated by the language to non-cyclic computations, the different cycle patterns shown in this section are not present at Notation 2’s granularity. Thus, the presence or absence of cycles in form-collapsed multi-graph patterns does not affect the generalization method’s ability to
succeed. However, these patterns do help demonstrate the wide variety of spreadsheet design patterns that the generalization method can handle.

6: BEYOND SIMPLE CELLS

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

6.1: 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 through the tabs. 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\(^1\); 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. 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 is always 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:

\[
\text{primitiveCircle( fillForeColor } \equiv \text{ BLACK,} \\
\text{ radius } \equiv \text{1 + population}[i@j]/10000) \text{: newCircle}
\]

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

\(^1\) Explicit sharing of this nature is also found in some other spreadsheet languages, such as Lotus and Formulate [Ambler 1999].
primitiveCircle in which fillForeColor’s formula is BLACK and radius’s formula refers to 1 + population\[3@1\]/10000.

6.1.1: 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 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. 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.

6.2: 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., [Burnett et al. 2000; 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, 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.

6.3: 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 19 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.

7: PERFORMANCE RESULTS

We have discussed scalability in terms of theoretical time costs. 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 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 3-5, and two that employ extensions described in Section 6. The first two are computation-oriented programs: Factorial (given in Appendix A’s Figure A-2) and the version of Fibonacci given in Figure A-1. The third, Stock With Colors, is a business graphics program that makes use of the temporal extensions of Section 6, using formulas that make use of Forms/3’s approach to time to generate an animated bar chart of stock prices that are updated over time. See Figure A-3. The fourth, Population, is the visualization-oriented example of Figure 3, which employs the grid extensions described in Section 6.

We chose to do all timings in a “lower-end” environment that does not have particularly optimal hardware or software. We used a single user Sun Ultra-1 (143MHz CPU) under Solaris 8, using Lucid Common Lisp V5.0. 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 3 displays the performance data for each spreadsheet group, broken out for user actions that triggered formula generalization. The actions labeled “copy” represent the user making a copy. Each action labeled “edit” represents a formula edit that triggered generalization. Each action labeled “off-screen” represents moving a spreadsheet into off-screen status (if it triggered generalization). A possible optimization for the latter trigger would be to generalize spreadsheets moving off the screen in a low-priority thread running in the background, to prevent this processing from affecting response time. However, as the data show, it has not been necessary to implement this optimization.

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 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. Some of the cells were created by the user, and others were generated automatically by the system as a result of the generalized descriptions (to support new copies for recursion, etc.). Of course, cells generated by the system are off-screen unless the user requests that they be displayed, thus affecting neither the size of the cell reference graph nor the cost of generalization. The counts also include the cells on the Forms/3 standard “System” spreadsheet, which is usually on the screen, and thus was kept on the screen throughout this experiment.

The generalization times (fourth column) were averaged over 10 runs. Thus, the times are “generalization response times”, i.e., the amount that the generalization method added to response time costs. Standard deviations were very low, and thus the means reported are quite representative of the raw times. As these data show, in most cases response time was affected by the generalization algorithm by less than a tenth of a second, even in this lower-end computing environment.

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

Table 3: 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).

1 We have also run the same performance tests on a Sun Ultra-5 (UltraSPARC-Ii CPU at 400mhz) which more closely represents modern desktop computers, and the times are less than half those shown here.
8: CONCLUSION

The method presented in this paper allows a fully 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 lead to additional advantages. The combination of orderlessness and modelessness removes order requirements from the user’s programming 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. Further, the method is general in that all non-circular referencing patterns are supported, even those not commonly found in traditional programming languages. Other approaches to generalization either require the program to fit in a single graphical module or support only specific patterns, such as particular spatial relationships or traditional call-return relationships.

Because of these characteristics, the generalization method presented here supports a graphical style of spreadsheet programming that incorporates extensive use of concrete examples and direct manipulation. In particular, the method’s scalability maintains—even for large programs—the immediate visual feedback expected of spreadsheet environments.

ACKNOWLEDGMENTS

We thank John Atwood and the other members of the Visual Programming Research Group at Oregon State University for their feedback and help with the implementation. Herkimer Gottfried wrote the population example program, Shikha Gottfried wrote the mortality example program, and Miguel Arredondo-Castro wrote the stocks example program. This work was supported in part by the National Science Foundation under awards CCR-9457473, CCR-9806821, and ITR-0082265, and by the Oregon Space Grant Program under NASA grant NGT5 40022.

REFERENCES


- 33 -


APPENDIX A: ADDITIONAL PROGRAMS

Figure A-1: A version of Fibonacci as it might be programmed by a more traditional programmer than the one who created Figure 4. This version is hierarchical—its relationships are those of a recursive “call tree”—which makes it much more traditional than Figure 4’s. However, it still requires generalization for the same reasons as for Figure 4.
Figure A-2: The factorial program from Section 7. The user has used the key icon to view the generalized meaning (at bottom left) of the “sample” reference to $53_{\text{Fact:Ans}}$.

Figure A-3: The stocks example program from Section 7. This is an investment visualization application. The rightmost cells in the \texttt{StockwColor} window form a horizontal color bar graph comparing different stock prices.